

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2790

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Harald Kosch László Böszörményi
Hermann Hellwagner (Eds.)

Euro-Par 2003

Parallel Processing

9th International Euro-Par Conference
Klagenfurt, Austria, August 26-29, 2003
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Harald Kosch
László Böszörményi
Hermann Hellwagner
University Klagenfurt, Institute for Information Technology
Universitätsstr. 65-67, 9020 Klagenfurt, Austria
E-mail: {harald.kosch, laszlo, hermann.hellwagner}@itec.uni-klu.ac.at

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): C.1-4, D.1-4, F.1-3, G.1-2, H.2

ISSN 0302-9743

ISBN 3-540-40788-X Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10931837 06/3142 5 4 3 2 1 0

Preface

Euro-Par Conference Series

The European Conference on Parallel Computing (Euro-Par) is an international conference series dedicated to the promotion and advancement of all aspects of parallel and distributed computing. The major themes fall into the categories of hardware, software, algorithms, and applications. This year, new and interesting topics were introduced, like *Peer-to-Peer Computing*, *Distributed Multimedia Systems*, and *Mobile and Ubiquitous Computing*. For the first time, we organized a *Demo Session* showing many challenging applications.

The general objective of Euro-Par is to provide a forum promoting the development of parallel and distributed computing both as an industrial technique and an academic discipline, extending the frontiers of both the state of the art and the state of the practice. The industrial importance of parallel and distributed computing is supported this year by a special *Industrial Session* as well as a vendors' exhibition. This is particularly important as currently parallel and distributed computing is evolving into a globally important technology; the buzzword

Grid Computing clearly expresses this move. In addition, the trend to a mobile world is clearly visible in this year's Euro-Par.

The main audience for and participants at Euro-Par are researchers in academic departments, industrial organizations, and government laboratories. Euro-Par aims to become the primary choice of such professionals for the presentation of new results in their specific areas.

Euro-Par has its own Internet domain with a permanent Web site where the history of the conference series is described: <http://www.euro-par.org>. The Euro-Par conference series is sponsored by the Association for Computer Machinery (ACM) and the International Federation for Information Processing (IFIP).

Euro-Par 2003 at Klagenfurt, Austria

Euro-Par 2003 was organized by the Institute of Information Technology, University of Klagenfurt, Austria. The conference location was the University of Klagenfurt which provided a convenient and stimulating environment for the presentation and discussion of recent research results.

A number of tutorials and invited talks extended the regular scientific program. Euro-Par 2003 invited five tutorials: *Project JXTA: An Open P2P Platform Architecture* (Bernard Traversat, Sun Microsystems); *Grid Computing with Jini* (Mark Baker, University of Portsmouth, and Zoltan Juhasz, University of Veszprem); *Pervasive Computing* (Alois Ferscha, University of Linz); *Carrier Grade Linux Platforms* (Ibrahim Haddad, Ericsson Research); and *A Family of*

Multimedia Representation Standards: MPEG-4/7/21 (Fernando Pereira, University of Technology Lisbon, and Hermann Hellwagner, University of Klagenfurt).

Invited talks were given by C.A.R. Hoare (Microsoft Research and Oxford University) on *The Verifying Compiler*; Jim Miller (Microsoft Research) on *Lessons from .NET*; Stefan Dessloch (Kaiserslautern University of Technology) on *Databases, Web Services, and Grid Computing*; and Henri E. Bal (Vrije Universiteit, Amsterdam) on *Ibis: A Java-Based Grid Programming Environment*. The first two invited speeches were in common with the co-located *Fifth Joint Modular Languages Conference (JMLC 2003)*, the main track of which took place prior to Euro-Par 2003 at the same venue.

The co-location of both conferences motivated us to organize a special “event” in the conference week: a memorial panel and an exhibition in honor of the recently deceased great computer scientists Ole-Johan Dahl, Edsger W. Dijkstra, and Kristen Nygaard. The virtual part of the exhibition has been made available for everybody via the Euro-Par 2003 Web site:

<http://europar-itec.uni-klu.ac.at/>.

Euro-Par 2003 Statistics

The format of Euro-Par 2003 followed that of the previous editions of the conference and consisted of a number of topics, each of them monitored by a committee of four members. In this year’s conference, there were 19 topics, four of which were included for the first time: Mobile and Ubiquitous Computing (Topic 15), Distributed Systems and Distributed Multimedia (Topic 16), Peer-to-Peer Computing (Topic 18), and a Demo Session (Topic 19) for the presentation of applications. The call for papers attracted 338 submissions, of which 159 were accepted. 103 were selected as regular papers and 52 as research notes. It is worth mentioning that four of the accepted papers were considered to be distinguished papers by the program committee. In total, 1233 review reports were collected, an average of 3.72 per paper.

Submissions were received from 43 countries (based on the corresponding author’s country), 29 of which were represented at the conference. The principal contributors by country were USA (25 accepted papers), Germany and Spain (each 21 accepted papers), and France (15 accepted papers).

Acknowledgments

A number of institutions and many individuals, in widely different respects, contributed to Euro-Par 2003. We thank for their generous support of the University of Klagenfurt; the Carinthian Economic Fund (KWF); the Carinthian International Campus for Science and Technology (Lakeside Park); the City of Klagenfurt; the Austrian Ministry of Education, Science and Culture (bm:bwk); the Austrian Ministry of Transportation, Innovation and Technology (bmvit); and

the Austrian Computer Society (OCG). The sponsor companies, Microsoft Research, Hewlett-Packard, Quant-X, Uniquare, IBM, ParTec, Sun Microsystems and the Verein der Freunde der Informatik@University of Klagenfurt provided the financial background required for the organization of a major conference. Finally, we are grateful to Springer-Verlag for publishing this proceedings.

We owe special thanks to all the authors for their contributions, members of the topics committee (more than 70 persons), and the numerous reviewers for their excellent work, ensuring the high quality of the conference.

We are especially grateful to Christian Lengauer, the chair of the Euro-Par steering committee, who gave us the benefit of his experience in the 18 months leading up to the conference.

Last, but not least, we are deeply indebted to the local organization team for their enthusiastic work, especially Martina Steinbacher, Mario Döller, Mulugeta Libsie, Angelika Rossak and the technical staff of our institute.

We hope that all participants had a very enjoyable experience here in Klagenfurt, Austria, at Euro-Par 2003!

Klagenfurt, June 2003

Harald Kosch
László Böszörményi
Hermann Hellwagner

Euro-Par Steering Committee

Chair

Christian Lengauer University of Passau, Germany

Vice Chair

Luc Bougé ENS Cachan, France

European Representatives

Marco Danelutto University of Pisa, Italy
Michel Daydé INP Toulouse, France
Rainer Feldmann University of Paderborn, Germany
Christos Kaklamanis Computer Technology Institute, Greece
Paul Kelly Imperial College, London, UK
Thomas Ludwig University of Heidelberg, Germany
Luc Moreau University of Southampton, UK
Rizos Sakellariou University of Manchester, UK
Henk Sips Technical University, Delft, The Netherlands

Non-European Representatives

Jack Dongarra University of Tennessee at Knoxville, USA
Shinji Tomita Kyoto University, Japan

Honorary Members

Ron Perrott Queen's University Belfast, UK
Karl Dieter Reinartz University of Erlangen-Nuremberg, Germany

Euro-Par 2003 Local Organization

Euro-Par 2003 was organized by the University of Klagenfurt.

Conference Chair

Harald Kosch
László Böszörményi
Hermann Hellwagner

Committee

Martina Steinbacher	Mario Döller	Mulugeta Libsie
Angelika Rossak	Andreas Griesser	Ronald Sowa
Peter Schojer	Remigiusz Górecki in Topic 8	

Euro-Par 2003 Programme Committee

Topic 1: Support Tools and Environments

Global Chair

Helmar Burkhart	Institut für Informatik, University of Basel, Switzerland

Local Chair

Thomas Ludwig	Institut für Informatik, Ruprecht-Karls- Universität, Heidelberg, Germany

Vice Chairs

Rudolf Eigenmann	School of Electrical and Computer Engineering, Purdue University, USA
Tomàs Margalef	Computer Science Department, Universitat Autònoma de Barcelona, Spain

Topic 2: Performance Evaluation and Prediction

Global Chair

Jeff Hollingsworth	Computer Science Department, University of Maryland, USA

Local Chair

Thomas Fahringer	Institute for Software Science, University of Vienna, Austria

Vice Chairs

Allen D. Malony	Department of Computer and Information Science, University of Oregon, USA
Jesús Labarta	European Center for Parallelism of Barce- lona, Technical University of Catalonia, Spain

Topic 3: Scheduling and Load Balancing

Global Chair

Yves Robert	Lab. de l'Informatique du Parallélisme, ENS Lyon, France

Local Chair

Dieter Kranzlmüller	GUP Linz, Johannes Kepler University, Linz, Austria

Vice Chairs

A.J.C. van Gemund	Delft University of Technology, The Netherlands
Henri Casanova	San Diego Supercomputing Center, USA

Topic 4: Compilers for High Performance

Global Chair

Michael Gerndt	Institut für Informatik, Technische Universität München, Ger- many

Local Chair

Markus Schordan	Lawrence Livermore National Laboratory, Livermore, USA

Vice Chairs

Chau-Wen Tseng	University of Maryland, College Park, USA
Michael O'Boyle	University of Edinburgh, UK

Topic 5: Parallel and Distributed Databases, Data Mining and Knowledge Discovery

Global Chair

Bernhard Mitschang	Institute of Parallel and Distributed Systems, Universität Stuttgart, Germany

Local Chair

Domenico Talia	Dipartimento di Elettronica Informatica e Sistemistica, University of Calabria, Italy

Vice Chairs

David Skillicorn	Queen's University, Kingston, Canada
Philippe Bonnet	Datalogisk Institut, Københavns Universitet, Denmark

Topic 6: Grid Computing and Middleware Systems

Global Chair

Henri Bal	Department of Mathematics and Computer Science, Vrije Universiteit, The Netherlands

Local Chair

Peter Kacsuk	Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary

Vice Chairs

Domenico LaForenza	Information Science and Technologies Institute, Italian National Research Coun- cil (CNR), Pisa, Italy
Thierry Priol	INRIA Rennes Research Unit, France

Topic 7: Applications on High-Performance Computers

Global Chair

Jacek Kitowski	Institute of Computer Science and ACC CYFRONET UMM, University of Mining and Metallurgy, Cracow, Poland

Local Chair

Peter Luksch	Institut für Informatik, Technische Universität München, Germany

Vice Chairs

Boleslaw K. Szymanski	Department of Computer Science, Rensselaer Polytechnic Institute, USA
Andrzej M. Goscinski	School of Information Technology, Deakin University, Australia

Topic 8: Parallel Computer Architecture and Instruction Level Parallelism

Global Chair

Stamatis Vassiliadis	Computer Engineering Laboratory, Delft University of Technology, The Netherlands

Local Chair

Arndt Bode	Institut für Informatik, Technische Universität München, Germany

Vice Chairs

Nikitas J. Dimopoulos	Electrical and Computer Engineering, University of Victoria, Canada
Jean-François Collard	HP Labs 3U, Hewlett-Packard, US

Topic 9: Distributed Algorithms

Global Chair

Jayadev Misra	Department of Computer Sciences, University of Texas at Austin, USA

Local Chair

Laurent Lefèvre	RESO/LIP, École Normale Supérieure de Lyon, France

Vice Chairs

Wolfgang Reisig	Institut für Informatik, Humboldt- Universität zu Berlin, Germany
Michael Schöttner	Abteilung Verteilte Systeme, Universität Ulm, Germany

Topic 13: Theory and Algorithms for Parallel Computation

Global Chair

Christos Kaklamanis

Computer Technology Institute and
Department of Computer Engineering
and Informatics, University of Patras,
Greece

Local Chair

Michael Kaufmann

Wilhelm-Schickard-Institut für
Informatik, Universität Tübingen,
Germany

Vice Chairs

Danny Krizanc

Computer Science Group, Mathematics
Department, Wesleyan University, USA
Laboratoire de Recherche en
Informatique, Université Paris-Sud,
France

Pierre Fraigniaud

**Topic 14: Routing and Communication
in Interconnection Networks**

Global Chair

José Duato Technical University of Valencia, Spain

Local Chair

Hermann Hellwagner Institute of Information Technology,
University of Klagenfurt, Austria

Vice Chairs

Olav Lysne Simula Research Lab and University of
Oslo, Norway
Timothy Pinkston University of Southern California, USA

Topic 15: Mobile and Ubiquitous Computing

Global Chair

Max Mühlhäuser FG Telekooperation, TU Darmstadt,
Germany

Local Chair

Alois Ferscha Institut für Praktische Informatik, Gruppe
Software, Johannes Kepler Universität,
Linz, Austria

Vice Chairs

Azzedine Boukerche University of Ottawa, Canada
Karin Hummel Institute for Computer Science and
Business Informatics, University of Vienna,
Austria

Topic 16: Distributed Systems and Distributed Multimedia

Global Chair

Fernando Pereira Electrical and Computers Department,
Instituto Superior Técnico, Lisboa, Portu-
gal

Local Chair

László Böszörményi Institute of Information Technology,
University of Klagenfurt, Austria

Vice Chairs

Abdulmotaleb El Saddik School of Information Technology and
Engineering (SITE), University of Ottawa,
Canada
Roy Friedman Department of Computer Science,
Technion – Israel Institute of Technology,
Haifa, Israel

Topic 17: High-Performance Object-Oriented and Middleware Systems

Global Chair

Geoffrey Fox	Community Grids Laboratory, Indiana University, USA

Local Chair

Michael Philippsen	Institut für Informatik, Universität Erlangen-Nürnberg, Germany

Vice Chairs

Mark Bull	Edinburgh Parallel Computing Centre (EPCC), University of Edinburgh, UK
Andrew Wendelborn	Department of Computer Science, University of Adelaide, Australia

Topic 18: Peer-to-Peer Computing

Global Chair

Luc Bougé	IRISA, ENS Cachan, Brittany Extension, Rennes, France

Local Chair

Franck Cappello	CNRS, LRI-Université Paris-Sud, France

Vice Chairs

Bernard Traversat	Project JXTA, Sun Microsystems, Santa Clara, USA
Omer Rana	Department of Computer Science, Cardiff University, UK

Topic 19: Demonstrations of Parallel and Distributed Computing

Global Chair

Ron Perrott	School of Computer Science, Queen's University Belfast, UK

Local Chair

Michael Kropfberger	Institute of Information Technology, University of Klagenfurt, Austria

Vice Chairs

Henk Sips	Faculty of Information Technology and Systems, Technical University of Delft, The Netherlands
Jarek Nabrzyski	Poznan Supercomputing and Networking Center, Poznan, Poland

Euro-Par 2003 Referees

(not including members of the programme or organization committees)

Afsahi, Ahmad	Bystroff, Chris
Alda, Witold	Byun, Tae-Young
Aldinucci, M.	Caarls, Wouter
Alexandru, Jugravu	Cabillic, Gilbert
Allcock, Bill	Cafaro, Massimo
Alt, Martin	Cai, Jianfei
Amodio, Pierluigi	Cai, Xing
Antochi, Iosif	Campadello, Stefano
Antoniou, Gabriel	Cannataro, Mario
Armstrong, Brian	Caragiannis, Ioannis
Ashby, Tom	Cardinale, Yudith
Attiya, Hagit	Caromel, Denis
Aumage, Olivier	Caron, Eddy
Austaller, Gerhard	Carter, Larry
Balatan, Zoltan	Casado, Rafael
Badia, Rosa M.	Catthoor, Francky
Bahi, Jacques	Chang, Chuan-Hua
Bajard, Jean-Claude	Chatterjee, Mainak
Banczúr, András	Cheresiz, Dmitry
Baniasadi, Amirali	Chiola, Giovanni
Baraglia, Ranieri	Chrysos, George
Barthou, Denis	Chun, B.N.
Basumallik, Ayon	Chung, I-hsin
Baude, Francoise	Cintra, Marcelo
Beaumont, Olivier	Coddington, Paul
Beck, Micah	Cole, Murray
Bellosa, Frank	Contes, Arnaud
Birnbaum, Adam	Coppola, Massimo
Bischof, Holger	Cortés, Ana
Bivens, Alan	Costa, Vitor Santos
Boavida, Fernando	Cramp, Anthony
Bodin, Francois	Crispo, Bruno
Boudet, Vincent	César, Eduardo
Braun, Elmar	Da Costa, Carlos
Breimer, Eric	Dail, Holly
Breton, Vincent	Dayde, Michel
Bretschneider, Timo Rolf	De Castro Dutra, Ines
Bubak, Marian	Deelman, Ewa
Buchholz, Peter	Denis, Alexandre
Buck, Bryan	Denneulin, Yves
Buyya, Rajkumar	Desprez, Frederic

Dhaenens, Clarisse
 Di Cosmo, Roberto
 Di Serafino, Daniela
 Dias, Artur Miguel
 Diessel, Oliver
 Dimakopoulos, Vassilos
 Do, Tai
 Dobrucky, Miroslav
 Dolev, Shlomi
 Dou, Jialin
 Drach-Temam, Nathalie
 Ducourthial, Bertrand
 Durr, C.
 Dutot, Pierre-Francois
 Dzwinel, Witold
 Eijkhout, Victor
 El Khatib, Khalil
 Ekaterinides, Yannis
 Emmanuel, S.
 Espinosa, Antonio
 Faber, Peter
 Fabrega, Josep
 Fagni, Tiziano
 Falkner, Katrina E.
 Farcy, Alexandre
 Feng, W.
 Ferragina, Paola
 Ferrante, Jeanne
 Fink, Torsten
 Fisher, Steve
 Fleury, Eric
 Folino, Gianluigi
 Ford, Rupert
 Fowler, Rob
 Franco, Daniel
 Franke, Bjoern
 Frenz, Stefan
 Frigo, Matteo
 Frohner, Akos
 Funika, Wlodzimierz
 Furfaro, Filippo
 Fursin, Grigori
 Förlinger, Karl
 Gansterer, Wilfried
 Garrido, Antonio

Gautama, Hasyim
 Gaydadjiev, G.N.
 Geist, Al
 Gelas, Jean-Patrick
 Getov, Vladimir
 Geuzebroek, Jeroen
 Gibbins, Nick
 Gjessing, Stein
 Glossner, John
 Gombas, Gabor
 Gorlatch, Sergei
 Goyeneche, Ariel
 Gratton, Serge
 Guermouche, Abdou
 Gupta, Amitava
 Haase, Gundolf
 Hammond, Kevin
 Hartl, Andreas
 Haumacher, Bernhard
 Hauswirth, Manfred
 Heinemann, Andreas
 Heinrich, Mark A.
 Hermenegildo, Manuel V.
 Hernández, Porfidio
 Heymann, Elisa
 Hlavacs, Helmut
 Hluchy, Ladislav
 Hopkins, T.R.
 Horn, Geir
 Hoschek, Wolfgang
 Hotop, Ewald
 Houda, Lamehamed
 Hu, Zhenjiang
 Hutchison, David
 Hyon, Emmanuel
 Iqbal, M. Ashraf
 Isaila, Florin
 Jegou, Yvon
 Jeitner, Jürgen
 Jin, Hai
 Johnson, Troy A.
 Jorba, Josep
 Jouhaud, Jen-Christophe
 Jouppi, Norman P.
 Ju, Roy

XVIII Organization

Juhasz, Zoltan
Juurlink, Ben
Kaeli, David
Kagstrom, Bo
Kalantery, Nasser
Karl, Wolfgang
Karp, Alan
Kat, Ronen
Keahey, Kate
Kelly, Paul
Kereku, Edmond
Kesavan, Ram
Khunjush, Farshad
Kielmann, Thilo
Kindermann, Stephan
Kleinjung, Jens
Kohn, Scott
Kondo, Derrick
Kotsis, Gabriele
Kowarschik, Markus
Krishnamurthy, Arvind
Kuchen, Herbert
Kumar, Sanjeev
Kunszt, Peter
Kuzmanov, Georgi
L'Excellent, Jean-Yves
Legendijk, R.
Langer, Ulrich
Lanteri, Stephane
Lauff, Markus
Lavenier, Dominique
Layuan, Li
Lee, Jack
Lisi, Francesca
Liu, Jane W.S.
Lopez, Pedro
Lourenco, Joao
Luque, Emilio
Luszczek, Piotr
Mairandres, Martin
Maman, Nathan
Manco, Giuseppe
Marcos, Aderito Fernandes
Markatos, Evangelos
Marques, Osni

Marques, Rui
Martinaitis, Paul
Mastroianni, Carlo
Matyska, Ludek
Mayrhofer, Rene
Mazzia, Francesca
McCance, Gavin
Medeiros, Pedro
Meier, Harald
Merzky, Andre
Michaelson, Greg
Midkiff, Sam
Min, Seung Jai
Miron, Pavlus
Molnos, Anca
Monteiro, Edmundo
Moreau, Luc
Moro, Gianluca
Moscu, Elena
Moshovos, Andreas
Moure, Juan Carlos
Muller, Jens
Muthukumar, Kalyan
Namyst, Raymond
Nandy, Sagnik
Napolitano, Jim
Nawarecki, Edward
Newhall, Tia
Nieminen, Risto
Nikoletseas, Sotiris
Nolte, Tina
Notare, Mirela Sechi
O'Donnell, John
Ohsumi, Toshiro
Orban, Dominique
Orduna, Juan Manuel
Orlando, Salvatore
Ortega, Julio
Ould-Khaoua, Mohamed
Overeinder, Benno J.
Paar, Alexander
Pallickara, Shrideep
Palmerini, Paolo
Pan, Zhelong
Park, Yong Woon

Peinl, Peter
 Peng, Jufeng
 Perego, Raffaele
 Perez, Christian
 Petitot, Antoine
 Petrini, Fabrizio
 Pham, Congduc
 Pichler, Mario
 Pierce, Evelyn
 Pllana, Sabri
 Podlipnig, Stefan
 Poetzsch-Heffter, Arnd
 Pommer, Andreas
 Poplavko, Peter
 Pralet, Stepahe
 Pramanick, Ira
 Prodan, Radu
 Pugliese, Andrea
 Puliafito, Antonio
 Quinson, Martin
 Radulescu, Andrei
 Rakhmatov, Daler N.
 Rantza, Ralf
 Rathmayer, Sabine
 Regin, Jean-Charles
 Reinemo, Sven-Arne
 Renambot, Luc
 Resch, Michael
 Ripoll, Ana
 Roe, Paul
 Ruiz, Daniel
 Saffre, Fabrice
 Safwat, Ahmed
 Saha, Debashis
 Sanders, Beverly
 Santos, Luis Paulo
 Sartori, Claudio
 Sasaki, Galen
 Schillo, Michael
 Schimmler, Manfred
 Schintke, Florian
 Schlansker, Michael
 Schojer, Peter
 Schreiber, Rob
 Schulz, Martin

Schuster, Assaf
 Schwarz, Holger
 Seitz, Christian
 Senar, Miquel Angel
 Sens, Pierre
 Seragiotto, Clovis, Jr.
 Sethumadhavan, Simha
 Shankar, Udaya A.
 Siciliano, Bruno
 Silva, Luis Moura
 Silvestri, Fabrizio
 Sima, Mihai
 Simpson, Steven
 Sion, Radu
 Skeie, Tor
 Sommeijer, Ben
 Sorensen, Dan
 Spriestersbach, Axel
 Srinivasan, Srikanth T.
 Stamatakis, Alexandros
 Stathis, Pyrrhos
 Stefan, Peter
 Stiles, Gardiner S.
 Stricker, Thomas M.
 Su, Alan
 Sulistio, Anthony
 Suppi, Remo
 Suter, Frederic
 Szeberenyi, Imre
 Sérot, Jocelyn
 Tao, Jie
 Taylor, Ian
 Tchernykh, Andrei
 Teich, Jürgen
 Temam, Olivier
 Teresco, Jim
 Terstyanszky, Gabor
 Theiss, Ingebjorg Thelin
 Thottethodi, Mithuna
 Todorova, Petia
 Tolia, Sovrin
 Tolksdorf, Robert
 Tonello, Nicola
 Tran, Viet
 Trinitis, Carsten

Trobec, Roman
Trunfio, Paolo
Truong, Hong-Linh
Tudruj, Marek
Turner, S.J.
Tusch, Roland
Ueberhuber, Christoph
Unger, Shelley
Utard, Gil
Vajtersic, Marian
Van Gijzen, Martin
Van der Vorst, Henk
Varela, Carlos
Varga, Laszlo Z.
Varshney, Upkar
Veldema, Ronald
Vivien, Frederic
Vogels, Werner
Vogl, Simon
Volker, Christian
Volkert, Jens
Von Laszewski, Gregor
Walter, Max

Wang, Dajin
Wasniewski, Jerzy
Weidendorfer, Josef
Welzl, Michael
Wismüller, Roland
Wong, Stephan
Woodcock, Jim
Wyrzykowski, Roman
Xiao, Li
Yan, Ken Qing
Yang, Yang
Yeo, Chai Kiat
Yi, Qing
Yoo, Chuck
Yuksel, Murat
Zambonelli, Franco
Zhang, Ming
Zheng, Yili
Zhou, Xiaobo
Zoccolo, C.
Zottl, Joachim

Table of Contents

Invited Talks

The Verifying Compiler: A Grand Challenge for Computing Research	1
<i>C.A.R. Hoare</i>	
Evolving a Multi-language Object-Oriented Framework: Lessons from .NET	2
<i>Jim Miller</i>	
Databases, Web Services, and Grid Computing – Standards and Directions	3
<i>Stefan Dessloch</i>	
Ibis: A Java-Based Grid Programming Environment	4
<i>Henri E. Bal</i>	

Topic 1: Support Tools and Environments

Topic Introduction	5
<i>Topic Chairs</i>	
A Hardware Counters Based Tool for System Monitoring	7
<i>Tiago C. Ferreto, Luiz DeRose, César A.F. De Rose</i>	
<i>ParaProf</i> : A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis	17
<i>Robert Bell, Allen D. Malony, Sameer Shende</i>	
On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications	27
<i>Hong-Linh Truong, Thomas Fahringer</i>	
Flexible Performance Debugging of Parallel and Distributed Applications	38
<i>Jacques Chassin de Kergommeaux, Cyril Guilloud, B. de Oliveira Stein</i>	
EventSpace – Exposing and Observing Communication Behavior of Parallel Cluster Applications	47
<i>Lars Ailo Bongo, Otto J. Anshus, John Markus Bjørndalen</i>	
A Race Detection Mechanism Embedded in a Conceptual Model for the Debugging of Message-Passing Distributed Programs	57
<i>Ana Paula Cláudio, João Duarte Cunha</i>	

DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications	66
<i>Hua Liu, Manish Parashar</i>	
DeWiz – A Modular Tool Architecture for Parallel Program Analysis	74
<i>Dieter Kranzlmüller, Michael Scarpa, Jens Volkert</i>	
Why Not Use a Pattern-Based Parallel Programming System?	81
<i>John Anvik, Jonathan Schaeffer, Duane Szafron, Kai Tan</i>	
Topic 2: Performance Evaluation and Prediction	
Topic Introduction	87
<i>Topic Chairs</i>	
Symbolic Performance Prediction of Speculative Parallel Programs	88
<i>Hasyim Gautama, Arjan J.C. van Gemund</i>	
A Reconfigurable Monitoring System for Large-Scale Network Computing	98
<i>Rajesh Subramanyan, José Miguel-Alonso, José A.B Fortes</i>	
Obtaining Hardware Performance Metrics for the BlueGene/L Supercomputer	109
<i>Pedro Mindlin, José R. Brunheroto, Luiz DeRose, José E. Moreira</i>	
Presentation and Analysis of Grid Performance Data	119
<i>Norbert Podhorszki, Peter Kacsuk</i>	
Distributed Application Monitoring for Clustered SMP Architectures	127
<i>Karl Förlinger, Michael Gerndt</i>	
An Emulation System for Predicting Master/Slave Program Performance	135
<i>Yasuharu Mizutani, Fumihiko Ino, Kenichi Hagihara</i>	
POETRIES: Performance Oriented Environment for Transparent Resource-Management, Implementing End-User Parallel/Distributed Applications	141
<i>Eduardo Cesar, J.G. Mesa, Joan Sorribes, Emilio Luque</i>	

Topic 3: Scheduling and Load Balancing

Topic Introduction	147
<i>Topic Chairs</i>	

Static Load-Balancing Techniques for Iterative Computations on Heterogeneous Clusters	148
<i>Hélène Renard, Yves Robert, Frédéric Vivien</i>	
Impact of Job Allocation Strategies on Communication-Driven Coscheduling in Clusters	160
<i>Gyu Sang Choi, Saurabh Agarwal, Jin-Ha Kim, Anydy B. Yoo, Chita R. Das</i>	
Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids.....	169
<i>Daniel Paranhos da Silva, Walfredo Cirne, Francisco Vilar Brasileiro</i>	
Dynamic Load Partitioning Strategies for Managing Data of Space and Time Heterogeneity in Parallel SAMR Applications	181
<i>Xiaolin Li, Manish Parashar</i>	
An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm	189
<i>Henan Zhao, Rizos Sakellariou</i>	
Performance-Based Dynamic Scheduling of Hybrid Real-Time Applications on a Cluster of Heterogeneous Workstations	195
<i>Ligang He, Stephen A. Jarvis, Daniel P. Spooner, Graham R. Nudd</i>	
Recursive Refinement of Lower Bounds in the Multiprocessor Scheduling Problem	201
<i>Satoshi Fujita, Masayuki Masukawa, Shigeaki Tagashira</i>	
Efficient Dynamic Load Balancing Strategies for Parallel Active Set Optimization Methods	206
<i>I. Pardines, Francisco F. Rivera</i>	
Cooperating Coscheduling in a Non-dedicated Cluster	212
<i>Francesc Giné, Francesc Solsona, Porfidio Hernández, Emilio Luque</i>	
Predicting the Best Mapping for Efficient Exploitation of Task and Data Parallelism	218
<i>Fernando Guirado, Ana Ripoll, Concepció Roig, Xiao Yuan, Emilio Luque</i>	
Dynamic Load Balancing for I/O- and Memory-Intensive Workload in Clusters Using a Feedback Control Mechanism.....	224
<i>Xiao Qin, Hong Jiang, Yifeng Zhu, David R. Swanson</i>	

An Experimental Study of k -Splittable Scheduling for DNS-Based Traffic Allocation	230
<i>Amit Agarwal, Tarun Agarwal, Sumit Chopra*, Anja Feldmann, Nils Kammenhuber, Piotr Krysta, Berthold Vöcking</i>	

Scheduling Strategies of Divisible Loads in DIN Networks	236
<i>Ligang Dong, Lek Heng Ngoh, Joo Geok Tan</i>	

Topic 4: Compilers for High Performance

Topic Introduction	241
<i>Topic Chairs</i>	

Partial Redundancy Elimination with Predication Techniques	242
<i>Bernhard Scholz, Eduard Mehofer, Nigel Horspool</i>	

SIMD Vectorization of Straight Line FFT Code	251
<i>Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber</i>	

Branch Elimination via Multi-variable Condition Merging	261
<i>William Krehling, David Whalley, Mark Bailey, Xin Yuan, Gang-Ryung Uh, Robert van Engelen</i>	

Exploiting On-Chip Data Transfers for Improving Performance of Chip-Scale Multiprocessors	271
<i>G. Chen, M. Kandemir, I. Kolcu, A. Choudhary</i>	

An Energy-Oriented Evaluation of Communication Optimizations for Microsensor Networks	279
<i>I. Kadayif, M. Kandemir, A. Choudhary, M. Karakoy</i>	

Increasing the Parallelism of Irregular Loops with Dependencies	287
<i>David E. Singh, María J. Martín, Francisco F. Rivera</i>	

Finding Free Schedules for Non-uniform Loops	297
<i>Volodymyr Beletsky, Krzysztof Siedlecki</i>	

Replicated Placements in the Polyhedron Model	303
<i>Peter Faber, Martin Griebl, Christian Lengauer</i>	

Topic 5: Parallel and Distributed Databases, Data Mining, and Knowledge Discovery

Topic Introduction	309
<i>Topic Chairs</i>	

A Parallel Algorithm for Incremental Compact Clustering	310
<i>Reynaldo Gil-García, José M. Badía-Contelles, Aurora Pons-Porrata</i>	

Preventive Multi-master Replication in a Cluster of Autonomous Databases	318
<i>Esther Pacitti, M. Tamer Özsu, Cédric Coulon</i>	
Pushing Down Bit Filters in the Pipelined Execution of Large Queries	328
<i>Josep Aguilar-Saborit, Victor Muntés-Mulero, Josep-L. Larriba-Pey</i>	
Suffix Arrays in Parallel	338
<i>Mauricio Marín, Gonzalo Navarro</i>	
Revisiting Join Site Selection in Distributed Database Systems	342
<i>Haiwei Ye, Brigitte Kerhervé, Gregor v. Bochmann</i>	
SCINTRA: A Model for Quantifying Inconsistencies in Grid-Organized Sensor Database Systems	348
<i>Lutz Schlesinger, Wolfgang Lehner</i>	
Topic 6: Grid Computing and Middleware Systems	
Topic Introduction	356
<i>Topic Chairs</i>	
Implementation of a Grid Computation Toolkit for Design Optimisation with Matlab and Condor	357
<i>Gang Xue, Matthew J. Fairman, Graeme E. Pound, Simon J. Cox</i>	
Grid Resource Selection for Opportunistic Job Migration	366
<i>Rubén S. Montero, Eduardo Huedo, Ignacio M. Llorente</i>	
Semantic Access Control for Medical Applications in Grid Environments	374
<i>Ludwig Seitz, Jean-Marc Pierson, Lionel Brunie</i>	
Automated Negotiation for Grid Notification Services	384
<i>Richard Lawley, Keith Decker, Michael Luck, Terry Payne, Luc Moreau</i>	
GrADSolve – RPC for High Performance Computing on the Grid	394
<i>Sathish Vadhiyar, Jack Dongarra, Asim YarKhan</i>	
Resource and Job Monitoring in the Grid	404
<i>Zoltán Balaton, Gábor Gombás</i>	
Delivering Data Management for Engineers on the Grid	412
<i>Jasmin Wason, Marc Molinari, Zhuoan Jiao, Simon J. Cox</i>	
A Resource Accounting and Charging System in Condor Environment . . .	417
<i>Csongor Somogyi, Zoltán László, Imre Szeberényi</i>	

Secure Web Services with Globus GSI and gSOAP	421
<i>Giovanni Aloisio, Massimo Cafaro, Daniele Lezzi, Robert Van Engelen</i>	

Future-Based RMI: Optimizing Compositions of Remote Method Calls on the Grid	427
<i>Martin Alt, Sergei Gorlatch</i>	

Topic 7: Applications on High-Performance Computers

Topic Introduction	431
<i>Topic Chairs</i>	

CAD Grid: Corporate-Wide Resource Sharing for Parameter Studies	433
<i>Ed Wheelhouse, Carsten Trinitis, Martin Schulz</i>	

Cache Performance Optimizations for Parallel Lattice Boltzmann Codes	441
<i>Jens Wilke, Thomas Pohl, Markus Kowarschik, Ulrich Rde</i>	

Effectiveness of Parallelizing the ILOG-CPLEX Mixed Integer Optimizer in the PUBB2 Framework	451
<i>Yuji Shinano, Tetsuya Fujie, Yuusuke Kounoike</i>	

Improving Performance of Hypermatrix Cholesky Factorization	461
<i>Jos R. Herrero, Juan J. Navarro</i>	

Parallel Agent-Based Simulation on a Cluster of Workstations	470
<i>Konstantin Popov, Vladimir Vlassov, Mahmoud Rafea, Fredrik Holmgren, Per Brand, Seif Haridi</i>	

Low Level Parallelization of Nonlinear Diffusion Filtering Algorithms for Cluster Computing Environments	481
<i>David Slogsnat, Markus Fischer, Andrs Bruhn, Joachim Weickert, Ulrich Brning</i>	

Implementation of Adaptive Control Algorithms in Robot Manipulators Using Parallel Computing	491
<i>Juan C. Fernndez, Vicente Hernndez, Lourdes Pealver</i>	

Interactive Ray Tracing on Commodity PC Clusters	499
<i>Ingo Wald, Carsten Benthin, Andreas Dietrich, Philipp Slusallek</i>	

Toward Automatic Management of Embarrassingly Parallel Applications	509
<i>Ins Dutra, David Page, Vitor Santos Costa, Jude Shavlik, Michael Waddell</i>	

Comparing Two Long Biological Sequences Using a DSM System	517
<i>Renata Cristina F. Melo, Maria Emilia Telles Walter, Alba Cristina Magalhaes Alves Melo, Rodolfo Batista, Marcelo Nardelli, Thelmo Martins, Tiago Fonseca</i>	
Two Dimensional Airfoil Optimisation Using CFD in a Grid Computing Environment	525
<i>Wenbin Song, Andy Keane, Hakki Eres, Graeme Pound, Simon Cox</i>	
Applied Grid Computing: Optimisation of Photonic Devices	533
<i>Duan H. Beckett, Ben Hiatt, Ken S. Thomas, Simon J. Cox</i>	
Parallel Linear System Solution and Its Application to Railway Power Network Simulation	537
<i>Muhammet F. Ercan, Yu-fai Fung, Tin-kin Ho, Wai-leung Cheung</i>	
Topic 8: Parallel Computer Architecture and Instruction-Level Parallelism	
Topic Introduction	541
<i>Topic Chairs</i>	
An Overview of the Blue Gene/L System Software Organization	543
<i>George Almási, Ralph Bellofatto, José Brunheroto, Călin Cașcaval, José G. Castaños, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, Alda Sanomiya, Karin Strauss</i>	
Trace Substitution	556
<i>Hans Vandierendonck, Hans Logie, Koen De Bosschere</i>	
Optimizing a Decoupled Front-End Architecture: The Indexed Fetch Target Buffer (iFTB)	566
<i>Juan C. Moure, Dolores I. Rexachs, Emilio Luque</i>	
Clustered Microarchitecture Simultaneous Multithreading	576
<i>Seong-Won Lee, Jean-Luc Gaudiot</i>	
Counteracting Bank Misprediction in Sliced First-Level Caches	586
<i>Enrique F. Torres, P. Ibañez, V. Viñals, J.M. Llabería</i>	
An Enhanced Trace Scheduler for SPARC Processors	597
<i>Spiros Kalogeropoulos</i>	
Compiler-Assisted Thread Level Control Speculation	603
<i>Hideyuki Miura, Luong Dinh Hung, Chitaka Iwama, Daisuke Tashiro, Niko Demus Barli, Shuichi Sakai, Hidehiko Tanaka</i>	

Compression in Data Caches with Compressible Field Isolation for Recursive Data Structures	609
<i>Masamichi Takagi, Kei Hiraki</i>	
Value Compression to Reduce Power in Data Caches	616
<i>Carles Aliagas, Carlos Molina, Montse Garcia, Antonio Gonzalez, Jordi Tubella</i>	

Topic 9: Distributed Algorithms

Topic Introduction	623
<i>Topic Chairs</i>	
Multiresolution Watershed Segmentation on a Beowulf Network	624
<i>Syarrraieni Ishar, Michel Bister</i>	
iRBP – A Fault Tolerant Total Order Broadcast for Large Scale Systems	632
<i>Luiz Angelo Barchet-Estefanel</i>	
Computational Models for Web- and Grid-Based Computation	640
<i>Joaquim Gabarró, Alan Stewart, Maurice Clint, Eamonn Boyle, Isabel Vallejo</i>	
CAS-Based Lock-Free Algorithm for Shared Deques	651
<i>Maged M. Michael</i>	
Energy Efficient Algorithm for Disconnected Write Operations in Mobile Web Environments	661
<i>Jong-Mu Choi, Jin-Seok Choi, Jai-Hoon Kim, Young-Bae Ko</i>	
Distributed Scheduling of Mobile Priority Requests	669
<i>Ahmed Housni, Michel Lacroix, Michel Trehel</i>	
Parallel Distributed Algorithms of the β -Model of the Small World Graphs	675
<i>Mahmoud Rafea, Konstantin Popov, Per Brand, Fredrik Holmgren, Seif Haridi</i>	

Topic 10: Parallel Programming: Models, Methods, and Programming Languages

Topic Introduction	681
<i>Topic Chairs</i>	
Cost Optimality and Predictability of Parallel Programming with Skeletons	682
<i>Holger Bischof, Sergei Gorlatch, Emanuel Kitzelmann</i>	

A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms	694
<i>K. Villaverde, E. Pontelli, H-F. Guo, G. Gupta</i>	
From Complexity Analysis to Performance Analysis	704
<i>Vicente Blanco, Jesús A. González, Coromoto León, Casiano Rodríguez, Germán Rodríguez</i>	
The Implementation of ASSIST, an Environment for Parallel and Distributed Programming	712
<i>Marco Aldinucci, Sonia Campa, Pierpaolo Ciullo, Massimo Coppola, Silvia Magini, Paolo Pesciullesi, Laura Potiti, Roberto Ravazzolo, Massimo Torquati, Marco Vanneschi, Corrado Zoccolo</i>	
The Design of an API for Strict Multithreading in C++	722
<i>Wolfgang Blochinger, Wolfgang Kuchlin</i>	
High-Level Process Control in Eden	732
<i>Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, Nils Weskamp</i>	
Using Skeletons in a Java-Based Grid System	742
<i>Martin Alt, Sergei Gorbachev</i>	
Prototyping Application Models in Concurrent ML	750
<i>David Johnston, Martin Fleury, Andy Downton</i>	
THROOM – Supporting POSIX Multithreaded Binaries on a Cluster	760
<i>Henrik Löf, Zoran Radović, Erik Hagersten</i>	
An Inter-entry Invocation Selection Mechanism for Concurrent Programming Languages	770
<i>Aaron W. Keen, Ronald A. Olsson</i>	
Parallel Juxtaposition for Bulk Synchronous Parallel ML	781
<i>Frédéric Loulergue</i>	
Parallelization with Tree Skeletons	789
<i>Kiminori Matsuzaki, Zhenjiang Hu, Masato Takeichi</i>	
Topic 11: Numerical Algorithms and Scientific Engineering Problems	
Topic Introduction	799
<i>Topic Chairs</i>	
Parallel ScaLAPACK-Style Algorithms for Solving Continuous-Time Sylvester Matrix Equations	800
<i>Robert Granat, Bo Kågström, Peter Poromaa</i>	

RECSY – A High Performance Library for Sylvester-Type Matrix
Equations 810
Isak Jonsson, Bo Kågström

Two Level Parallelism in a Stream-Function Model for Global Ocean
Circulation 820
Martin van Gijzen

Scalable Parallel RK Solvers for ODEs Derived by the Method of
Lines 830
Matthias Korch, Thomas Rauber

Hierarchical Hybrid Grids as Basis for Parallel Numerical
Solution of PDE 840
Frank Hülsemann, Benjamin Bergen, Ulrich Rüde

Overlapping Computation/Communication in the Parallel One-Sided
Jacobi Method 844
El Mostafa Daoudi, Abdelhak Lakhouaja, Halima Outada

**Topic 12: Architectures and Algorithms for Multimedia
Applications**

Topic Introduction 850
Topic Chairs

Distributed Multimedia Streaming over Peer-to-Peer Networks 851
Jin B. Kwon, Heon Y. Yeom

Exploiting Traffic Balancing and Multicast Efficiency in
Distributed Video-on-Demand Architectures 859
*Fernando Cores, Ana Ripoll, Bahjat Qazzaz, Remo Suppi,
Xiaoyuan Yang, Porfidio Hernandez, Emilio Luque*

On Transmission Scheduling in a Server-Less Video-on-Demand
System 870
C.Y. Chan, Jack Y.B. Lee

A Proxy-Based Dynamic Multicasting Policy Using Stream’s Access
Pattern 880
Yong Woon Park, Si Woong Jang

**Topic 13: Theory and Algorithms for Parallel
Computation**

Topic Introduction 884
Topic Chairs

Improving Communication Sensitive Parallel Radix Sort for Unbalanced Data	885
<i>Martin Schmollinger</i>	
Minimizing Global Communication in Parallel List Ranking	894
<i>Jop F. Sibeyn</i>	
Construction of Efficient Communication Sub-structures: Non-approximability Results and Polynomial Sub-cases	903
<i>Christian Laforest</i>	
c -Perfect Hashing Schemes for Binary Trees, with Applications to Parallel Memories	911
<i>Gennaro Cordasco, Alberto Negro, Vittorio Scarano, Arnold L. Rosenberg</i>	
A Model of Pipelined Mutual Exclusion on Cache-Coherent Multiprocessors	917
<i>Masaru Takesue</i>	
Efficient Parallel Multiplication Algorithm for Large Integers	923
<i>Viktor Bunimov, Manfred Schimmler</i>	

Topic 14: Routing and Communication in Interconnection Networks

Topic Introduction	929
<i>Topic Chairs</i>	
Dynamic Streams for Efficient Communications between Migrating Processes in a Cluster	930
<i>Pascal Gallard, Christine Morin</i>	
FOBS: A Lightweight Communication Protocol for Grid Computing	938
<i>Phillip M. Dickens</i>	
Low-Fragmentation Mapping Strategies for Linear Forwarding Tables in InfiniBand TM	947
<i>P. López, J. Flich, A. Robles</i>	
A Robust Mechanism for Congestion Control: INC	958
<i>Elvira Baydal, P. López</i>	
RoCL: A Resource Oriented Communication Library	969
<i>Albano Alves, António Pina, José Exposto, José Rufino</i>	
A QoS Multicast Routing Protocol for Dynamic Group Topology	980
<i>Li Layuan, Li Chunlin</i>	

A Study of Network Capacity under Deflection Routing Schemes	989
<i>Josep Fàbrega, Xavier Muñoz</i>	
Implementation and Performance Evaluation of M-VIA on AceNIC Gigabit Ethernet Card	995
<i>In-Su Yoon, Sang-Hwa Chung, Ben Lee, Hyuk-Chul Kwon</i>	

Topic 15: Mobile and Ubiquitous Computings

Topic Introduction	1001
<i>Topic Chairs</i>	
A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks	1003
<i>I. Chatzigiannakis, T. Dimitriou, M. Mavronicolas, S. Nikolettseas, P. Spirakis</i>	
Network Based Mobile Station Positioning in Metropolitan Area	1017
<i>Karl R.P.H. Leung, Joseph Kee-Yin Ng, Tim K.T. Chan, Kenneth M.K. Chu, Chun Hung Li</i>	
Programming Coordinated Motion Patterns with the TOTA Middleware	1027
<i>Marco Mamei, Franco Zambonelli, Letizia Leonardi</i>	
iClouds – Peer-to-Peer Information Sharing in Mobile Environments	1038
<i>Andreas Heinemann, Jussi Kangasharju, Fernando Lyardet, Max Mühlhäuser</i>	
Support for Personal and Service Mobility in Ubiquitous Computing Environments	1046
<i>K. El-Khatib, N. Hadibi, Gregor v. Bochmann</i>	
Dynamic Layouts for Wireless ATM	1056
<i>Michele Flammini, Giorgio Gambosi, Alessandro Gasparini, Alfredo Navarra</i>	
Modeling Context-Aware Behavior by Interpreted ECA Rules	1064
<i>Wolfgang Beer, Volker Christian, Alois Ferscha, Lars Mehrmann</i>	
A Coordination Model for ad hoc Mobile Systems	1074
<i>Marco Tulio Valente, Fernando Magno Pereira, Roberto da Silva Bigonha, Mariza Andrade da Silva Bigonha</i>	
Making Existing Interactive Applications Context-Aware	1082
<i>Tatsuo Nakajima, Atsushi Hasegawa, Tomoyoshi Akutagawa, Akihiro Ibe, Kouji Yamamoto</i>	

Benefits and Requirements of Using Multi-agent Systems on Smart Devices	1091
<i>Cosmin Carabelea, Olivier Boissier, Fano Ramparany</i>	
Performance Evaluation of Two Congestion Control Mechanisms with On-Demand Distance Vector (AODV) Routing Protocol for Mobile and Wireless Networks	1099
<i>Azzedine Boukerche</i>	
Towards an Approach for Mobile Profile Based Distributed Clustering ...	1109
<i>Christian Seitz, Michael Berger</i>	
Simulating Demand-Driven Server and Service Location in Third Generation Mobile Networks	1118
<i>Geraldo Robson Mateus, Olga Goussevskaia, Antonio A.F. Loureiro</i>	
Designing Mobile Games for a Challenging Experience of the Urban Heritage	1129
<i>Francesco Bellotti, Riccardo Berta, Alessandro De Gloria, Edmondo Ferretti, Massimiliano Margarone</i>	
QoS Provision in IP Based Mobile Networks	1137
<i>Vilmos Simon, Árpád Huszák, Sándor Szabó, Sándor Imre</i>	
Design of a Management System for Wireless Home Area Networking	1141
<i>Tapio Rantanen, Janne Sikiö, Marko Hännikäinen, Timo Vanhatupa, Olavi Karasti, Timo Hämäläinen1</i>	
Short Message Service in a Grid-Enabled Computing Environment	1148
<i>Fenglian Xu, Hakki Eres, Simon Cox</i>	
Service Migration Mechanism Using Mobile Sensor Network	1153
<i>Kyungsoo Lim, Woojin Park, Sinam Woo, Sunshin An</i>	
Topic 16: Distributed Systems and Distributed Multimedia	
Topic Introduction	1159
<i>Topic Chairs</i>	
Nswap: A Network Swapping Module for Linux Clusters	1160
<i>Tia Newhall, Sean Finney, Kuzman Ganchev, Michael Spiegel</i>	
Low Overhead Agent Replication for the Reliable Mobile Agent System ..	1170
<i>Taesoon Park, Ilsoo Byun</i>	
A Transparent Software Distributed Shared Memory	1180
<i>Emil-Dan Kohn, Assaf Schuster</i>	

On the Characterization of Distributed Virtual Environment Systems	1190
<i>Pedro Morillo, Juan M. Orduña, M. Fernández, J. Duato</i>	
A Proxy Placement Algorithm for the Adaptive Multimedia Server	1199
<i>Balázs Goldschmidt, Zoltán László</i>	
A New Distributed JVM for Cluster Computing	1207
<i>Marcelo Lobosco, Anderson Silva, Orlando Loques, Claudio L. de Amorim</i>	
An Extension of BSDI for Multimedia Bitstream Syntax Description.....	1216
<i>Sylvain Devillers</i>	
Fast Construction, Easy Configuration, and Flexible Management of a Cluster System.....	1224
<i>Ha Yoon Song, Han-gyoo Kim, Kee Cheol Lee</i>	
Topic 17: Peer-to-Peer Computing	
Topic Introduction	1229
<i>Topic Chairs</i>	
Hierarchical Peer-to-Peer Systems	1230
<i>L. Garcés-Erice, E.W. Biersack, P.A. Felber, K.W. Ross, G. Urvoy-Keller</i>	
Enabling Peer-to-Peer Interactions for Scientific Applications on the Grid	1240
<i>Vincent Matossian, Manish Parashar</i>	
A Spontaneous Overlay Search Tree	1248
<i>Hung-Chang Hsiao, Chuan-Mao Lin, Chung-Ta King</i>	
Fault Tolerant Peer-to-Peer Dissemination Network	1257
<i>Konstantinos G. Zerfiridis, Helen D. Karatza</i>	
Exploring the Catallactic Coordination Approach for Peer-to-Peer Systems	1265
<i>Oscar Ardaiz, Pau Artigas, Torsten Eymann, Felix Freitag, Roc Messegueur, Leandro Navarro, and Michael Reinicke</i>	
Incentives for Combatting Freeriding on P2P Networks	1273
<i>Sepandar D. Kamvar, Mario T. Schlosser, Hector Garcia-Molina</i>	

Topic 18: Demonstrations of Parallel and Distributed Computing

Topic Introduction	1280
<i>Topic Chairs</i>	
Demonstration of P-GRADE Job-Mode for the Grid	1281
<i>P. Kacsuk, R. Lovas, J. Kovács, F. Szalai, G. Gombás, N. Podhorszki, Á. Horváth, A. Horányi, I. Szeberényi, T. Delaitre, G. Terstyánszky, A. Gourgoulis</i>	
Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster	1287
<i>Jérémie Allard, Bruno Raffin, Florence Zara</i>	
Kerrighed: A Single System Image Cluster Operating System for High Performance Computing	1291
<i>Christine Morin, Renaud Lottiaux Geoffroy Vallée, Pascal Gallard, Gaël Utard, R. Badrinath, Louis Rilling</i>	
ASSIST Demo: A High Level, High Performance, Portable, Structured Parallel Programming Environment at Work	1295
<i>M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, C. Zoccolo</i>	
KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs	1301
<i>Bernd Mohr, Felix Wolf</i>	
Visual System for Developing of Parallel Programs	1305
<i>O.G. Monakhov</i>	
Late Paper	
Peer-to-Peer Communication through the Design and Implementation of Xiangqi	1309
<i>Abdulmotaleb El Saddik, Andre Dufour</i>	
Author Index	1315

The Verifying Compiler: A Grand Challenge for Computing Research

C.A.R. Hoare

Microsoft Research Ltd.,
7 JJ Thomson Ave,
Cambridge CB3 0FB, UK
thoare@microsoft.com

Abstract. I propose a set of criteria which distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers.

The primary purpose of the formulation and promulgation of a grand challenge is to contribute to the advancement of some branch of science or engineering. A grand challenge represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale. The challenge is formulated by the researchers themselves as a focus for the research that they wish to pursue in any case, and which they believe can be pursued more effectively by advance planning and co-ordination. Unlike other common kinds of research initiative, a grand challenge should not be triggered by hope of short-term economic, commercial, medical, military or social benefits; and its initiation should not wait for political promotion or for prior allocation of special funding. The goals of the challenge should be purely scientific goals of the advancement of skill and of knowledge. It should appeal not only to the curiosity of scientists and to the ambition of engineers; ideally it should appeal also to the imagination of the general public; thereby it may enlarge the general understanding and appreciation of science, and attract new entrants to a rewarding career in scientific research.

As an example drawn from Computer Science, I revive an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it. A verifying compiler uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations that are associated with the code of the program, often inferred automatically, and increasingly often supplied by the original programmer. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components. The only limit to its use will be set by an evaluation of the cost and benefits of accurate and complete formalization of the criterion of correctness for the software.

Evolving a Multi-language Object-Oriented Framework: Lessons from .NET

Jim Miller

Microsoft Corporation,
jsmiller@microsoft.com

Abstract. In 2001 Microsoft shipped the first public version of its Common Language Runtime (CLR) and the associated object-oriented .NET Framework. This Framework was designed for use by multiple languages through adherence to a Common Language Specification (CLS). The CLR, the CLS, and the basic level of the .NET Framework are all part of International Standard ISO/IEC 23271. Over 20 programming languages have been implemented on top of the CLR, all providing access to the same .Net Framework, and over 20,000,000 copies have been downloaded since its initial release.

As a commercial software vendor, Microsoft is deeply concerned with evolving this system. Innovation is required to address new needs, new ideas, and new applications. But backwards compatibility is equally important to give existing customers the confidence that they can build on a stable base even as it evolves over time. This is a hard problem in general, it is made harder by the common use of virtual methods and public state, and harder still by a desire to make the programming model simple.

This talk will describe the architectural ramifications of combining ease-of-use with system evolution and modularity. These ramifications extend widely throughout the system infrastructure, ranging from the underlying binding mechanism of the virtual machine, through program language syntax extensions, and into the programming environment.

Databases, Web Services, and Grid Computing – Standards and Directions

Stefan Dessloch

University of Kaiserslautern
Department of Computer Science
Heterogenous Information Systems Group
D-67653 Kaiserslautern, Germany
`dessloch@informatik.uni-kl.de`

Abstract. Over the last two years, web services have emerged as a key technology for distributed object computing, promising significant advantages for overcoming interoperability and heterogeneity problems in large scale, distributed environments. Major vendors have started to incorporate web service technology in their database and middleware products, and companies are starting to exploit the technology in information integration, EAI, and B2B-integration architectures. In the area of Grid computing, which aims at providing a distributed computing architecture and infrastructure for science and engineering, web services have become an important piece of the puzzle by providing so-called Grid Services that help realizing the goal of virtual organizations to coordinate resource sharing and problem solving tasks. The adequate support of data management functionality, or data-oriented services in general within this architectural setting is undoubtedly a key requirement and a number of approaches have been proposed both by research and industry to address the related problems. This talk will give an overview of recent developments in the areas outlined above and discuss important standardization activities as well as trends and directions in industry and research.

Ibis: A Java-Based Grid Programming Environment

Henri E. Bal

Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

`bal@cs.vu.nl`

<http://www.cs.vu.nl/ibis/>

Ibis [3] is an ongoing research project in which we are building a Java-based Grid programming environment for distributed supercomputing applications. Java's high portability allows parallel applications to run on a heterogeneous grid without requiring porting or recompilation. A major problem in using Java for high-performance computing, however, is the inferior performance and limited expressiveness of Java's Remote Method Invocation (RMI). Earlier projects (e.g., Manta [1]) solved the performance problem, but at the cost of using a runtime system written in native code, which gives up Java's high portability. The philosophy behind Ibis is to try to obtain good performance without using any native code, but allow native solutions as special-case optimizations. For example, a Grid application developed with Ibis can use a pure-Java RMI implementation over TCP/IP that will run "everywhere"; if the application runs on, say, a Myrinet cluster, Ibis can load a more efficient RMI implementation for Myrinet that partially uses native code. The pure-Java implementation of Ibis does several optimizations, using bytecode rewriting. For example, it boosts RMI performance by avoiding the high overhead of runtime type inspection that current RMI implementations have. The special-case implementations do more aggressive optimizations, even allowing zero-copy communication in certain cases.

The Ibis programming environment consists of a communication runtime system with a well-defined interface and a range of communication paradigms implemented on top of this interface, including RMI, object replication, group communication, and collective communication, all integrated cleanly into Java. Ibis has also been used to implement Satin [2], which is a Cilk-like wide-area divide-and-conquer system in Java. Experiments have been performed on two Grid test beds, the Dutch DAS-2 system and the (highly heterogeneous) European GridLab test bed. Our current research on Ibis focuses on fault tolerance and on heterogeneous networks.

References

1. J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. on Programming Languages and Systems*, 23(6):747–775, November 2001.
2. R. van Nieuwpoort, T. Kielmann, and H.E. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, Snowbird, Utah, June 2001.
3. Rob van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *ACM JavaGrande ISCOPE 2002 Conference*, pages 18–27, Seattle, WA, November 2002.

Topic 1

Support Tools and Environments

Helmar Burkhart, Rudolf Eigenmann, Tomàs Margalef, and Thomas Ludwig
Topic Chairs

It is always an exciting moment to look at the papers that are submitted to Euro-Par's topic on "Support Tools and Environments". What types of tools will be presented? What shift of focus in research can we see? What can we conclude for the situation of the user who is always looking for sophisticated tools?

This year's topic will mainly focus on performance analysis and well-elaborated monitoring techniques. It seems as if finally our programs are running but need more tuning in order to be efficient.

The research that is conducted in the field of performance analysis is now very advanced. Not only do we find much work with a focus on semi-automatic or even automatic bottleneck detection. Also there is research in the field of comparing multiple experiments with programs and evaluating differences between program runs. This feature does well support the users as code tuning is always an evolutionary process.

Current monitoring techniques used as a basis for different types of tools are more generic than ever. They offer configurability with respect to sensors and actuators which cooperate with the program and to events which are triggered and cooperate with the tools.

Debuggers now deal with race detection, covering the most problematic error situations in parallel programs.

Even with debugging and performance analysis issues being well covered the design and construction of tools remains a challenging task. Users also need tools for e.g., computational steering and load balancing. This is particularly complicated when we consider new architectural structures like the Grid. The distribution of components of tools and programs and the potential failure of connections between them will have to result in more fault tolerant tool architectures. We are convinced that future Euro-Par conferences will reflect this trend.

However, independent of the specific research issues presented this year the goal of the topic is still to bring together tool designers, developers, and users and help them in sharing ideas, concepts, and products in this field.

This year we have received 19 submissions which is considerably more than in the last year. 7 papers were accepted as full papers for the conference (35%) and 2 papers as short presentation. One paper was accepted as demonstration.

We would like to thank all authors who submitted a contribution as well as all the reviewers who spent much time to guarantee a sound selection of papers and thus a high quality of the topic.

One final comment: We have seen a considerable number of submitted papers that more or less omitted comments how their project is related to earlier or ongoing work. We do not consider this as being proper scientific practice and warmly recommend to put more emphasis on related work sections. We have a 25 years history of tools for parallel computers and an active presence: refer to it.

A Hardware Counters Based Tool for System Monitoring

Tiago C. Ferreto^{1*}, Luiz DeRose², and César A.F. De Rose¹

¹ Catholic University of Rio Grande do Sul (PUCRS),
Post-Graduate Program on Computer Science, Porto Alegre, Brazil

{tferreto,derose}@inf.pucrs.br

² IBM T.J. Watson Research Center,
Yorktown Heights, NY, USA
laderose@us.ibm.com

Abstract. In this paper we describe the extensions to the RVision tool to support hardware performance counters monitoring at system level. This monitoring tool is useful for system administrators to detect applications that need tuning. We present a case study using a parallel version of the Swim benchmark from the SPEC suite, running on an Intel Pentium III Linux cluster, where we show a performance improvement of 25%. In addition we present some intrusion measurements showing that our implementation has very low intrusion even with high monitoring frequencies.

1 Introduction

As parallel architectures become more complex it is becoming much more difficult for applications to run at a reasonable fraction of the peak performance of parallel systems. In order to improve program performance, experienced program developers have been using hardware performance counters for application tuning [1]. Likewise, in order to help programmers tune their applications, a variety of utilities and libraries have been created to provide access to the hardware performance counters at user level [2,3,4,5,6,7,8].

Unfortunately, for most users, it is not always clear when their programs need tuning. Thus, system administrators would like to be able to monitor application performance metrics, such as MFlops/sec and cache hit ratios, to identify programs that could be candidates for optimization. However, there are practically no system-monitoring tools available on cache-based systems that provide such information. An approach being used today by system administrators is to apply “*wrappers*” to job submission scripts that activate utilities to collect hardware performance information at the application level, in order to generate summary files at the end of the execution. An example of such an approach is the “*hpmcollect*” interface [9], developed at the Scientific Supercomputing Center at the University of Karlsruhe, which automatically starts an utility [6] to collect hardware performance counters data for all applications submitted for execution, and at the end of the execution, collects and combines the hardware counters output from all parallel tasks, providing a short overview of the total performance and the resource usage of the parallel application.

* Work supported by HP-Brazil

In order to provide a more complete solution for system administrators, we extended the RVision tool for cluster monitoring [10] with a monitoring library to collect hardware performance counters information during program execution, and a monitoring client for presentation in the forms of graphs and tables of the hardware events and derived metrics. In this paper we describe the implementation and the main features of this monitoring system, which include the easy of use, the flexibility in selecting events to be monitored, and its very low intrusion. In addition, we demonstrate the value of this system with an example using the SPEC Swim benchmark [11].

The remainder of this paper is organized as follows. We begin in Section 2 describing the utilization of hardware counters for program optimization and explaining the design and implementation of the modules for Hardware Performance Counters monitoring using RVision. In Section 3 we present intrusion measurements. In Section 4 we present an utilization example of the RVision hardware monitoring module. Finally, we summarize our conclusions and directions for future work in Section 5.

2 Hardware Counters Monitoring

Hardware performance counters are special purpose registers available in most modern microprocessors that keep track of programmable hardware events at every cycle. These events represent hardware activities from all functional units of the processor, allowing the low overhead access to hardware performance information, such as counts of instructions, cache misses, and branch misprediction. These registers were originally designed for hardware debugging, but, although the number of counters and type of events available differs significantly between microprocessors, most of them provide a common subset that allows programmers to use these counters to compute derived metrics to correlate the behavior of the application to one or more of the hardware components. With the availability of kernel level APIs to access the hardware counters at a user level, as well as performance tools and libraries that provide event counts and derived metrics, hardware counters have become an invaluable asset for application performance tuning.

In contrast, system administrators have not been able to exploit effectively the availability of hardware counters, mainly due to the lack of monitoring systems capable of accessing the counters. The access to derived hardware metrics during program execution would be helpful to system administrators to detect applications that need tuning. For example, the value of a particular metric falling constantly below a pre-defined threshold would be an indication that the program is a candidate for optimization.

For our hardware performance counters monitoring approach, we extended RVision, a multi-user monitoring system for GNU/Linux based clusters [10] developed at the Research Center in High Performance Computing (CPAD - PUCRS/HP). It requires the availability of a kernel interface to access the hardware performance counters at a system level, and we consider that programs will have dedicated use of the nodes during execution, which is normally the case on the majority of supercomputing sites.

The RVision monitor was originally developed to acquire information such as processor and memory usage, through kernel system calls. Due to RVision's open architecture, only a new *Monitoring Library* and a new *Monitoring Client*, which are described next, were needed to obtain the hardware counters values. The "Monitoring Library" is re-

sponsible for the capture of selected hardware events, while the “Monitoring Client” is responsible for presenting this new information, which also includes derived performance metrics. We implemented and tested these components on the “*Tropical*” Linux Cluster at CPAD, which has 8 dual-processor (Pentium III-1GHz) nodes, switched via a Fast-Ethernet network. Each node has 256 MBytes of main memory. Each processor has two levels of cache: 32 KBytes of Level 1 (with 4-way set associativity) and 256 KBytes of Level 2 (with 8-way set associativity). In both cases the cache line size is 32 bytes.

2.1 Monitoring Library

Currently, Linux does not provide an interface to access the hardware performance counters. Hence, we patched the Linux kernel using the “perfctr” patch and driver, developed by Mikael Petterson [2]. This patch provides a user and a system level interface to access the performance monitoring counters on Intel X86 processors. Since the number of counters and type of events available differs significantly between microprocessors, the RVision Monitoring Library to capture information provided by the hardware counters is architecture dependent. However, our monitoring library can be easily ported to any other platform that provides a kernel level application programming interface to access the hardware counters at system level.

The selection of events to be monitored were restricted to the hardware counter events available on the Pentium III processor, which provides two performance monitoring counters capable of counting a total of 77 different events (at most two at a time). In addition, the architecture provides a time stamp counter (TSC), which counts the elapsed machine cycles, as well as the CPU frequency. The complete specification of the performance monitoring counters and the description of all of its events are presented in [12].

From the events provided by the Pentium III architecture, we selected the following set of pair of events (in addition to the TSC) to be used for monitoring:

- P6_INST_RETIRED and P6_CPU_CLK_UNHALTED, to count the number of instructions completed and the number of machine cycles used by the program.
- P6_FLOPS and P6_CPU_CLK_UNHALTED, to count the number of floating point instructions and the number of machine cycles used by the program.
- P6_DATA_MEM_REFS and P6_DCU_LINES_IN, to count the number of level 1 accesses and the number of level 1 cache misses, respectively.
- P6_L2_RQSTS and P6_L2_LINES_IN, to count the number of level 2 accesses and the number of level 2 cache misses, respectively.
- P6_INST_DECODED and P6_INST_RETIRED, to count the number of instructions dispatched and the number of instructions decoded.

Depending on the set of counters used, we compute the following derived metrics:

MIPS: the average number of instructions per second (in millions), computed as:

$$P6\_INST\_RETIRED / (1000000 * TSC / CPU \text{ frequency})$$

Utilization Rate: the ratio of CPU time to wall clock time, computed as:

$$P6\_CPU\_CLK\_UNHALTED / TSC$$

IpC: Instructions per Cycle, computed as:

$$P6\_INST\_RETIRED / P6\_CPU\_CLK\_UNHALTED$$

MFlops/sec: Millions of floating point operations per second, computed as:

$$P6\_FLOPS / (1000000 * TSC / CPU \text{ frequency})$$

Level 1 cache hit ratio: Computed as:

$$100 * (1 - (P6\_DCU\_LINES\_IN / P6\_DATA\_MEM\_REFS))$$

Level 2 cache hit ratio: Computed as:

$$100 * (1 - (P6\_L2\_LINES\_IN / P6\_L2\_RQSTS))$$

Percentage of instructions dispatched that completed: Computed as:

$$100 * (P6\_INST\_RETIRED / P6\_INST\_DECODED)$$

2.2 Monitoring Client

The monitoring client is responsible for the presentation of tables and graphics with the performance monitoring counters information and the derived metrics. The communication routines of the monitoring client were implemented in C, and the GUI was implemented in Java, with JNI being used to provide the connection between the languages. A snapshot of the monitoring client is presented in Figure 3. It provides a table presenting the hardware counters values and the derived metric values for each node of the cluster, as well as a histogram containing the average of the derived metric for all processors. The histogram scrolls to the left, presenting the most recent information at the rightmost side.

3 Intrusion Measurement

The main concern when a tool is used to monitor some resource is how much the readings are being affected by the tool. Being a parallel application itself, the monitoring tool, when active, is consuming cluster resources such as CPU and network bandwidth. This intrusion should be minimal to guarantee that the monitored data is accurate and that the behavior of the other application running on the cluster is not considerably affected.

We measured RVision's intrusion by defining a monitoring session capturing hardware counters values for all cluster nodes and requesting this data using online monitoring with a regular time interval. Different applications are executed in the cluster with and without monitoring and we compared the execution times. We varied the monitoring time interval from 2 seconds to 200 milliseconds to simulate a worst-case scenario.

To evaluate RVision under different workloads we used the following programs from the NAS Parallel Benchmarks [13, 14]: Embarassingly Parallel (EP), Integer Sort (IS), LU Decomposition (LU), Conjugate Gradient (CG), and Multigrid (MG). All benchmarks were compiled using class A, defined internally in the NPB, and executed on the 8 nodes of the Tropical Linux Cluster at CPAD [15]. Table 1 presents the intrusion results for all test cases. In each case the test application was executed 10 times for each time interval with monitoring turned on.

All benchmarks used in the measurement generated low intrusivity. The worst-case scenario, using 200 milliseconds as time interval, presented intrusion values of less than 1% for all programs with the exception of the IS benchmark. The Integer Sort benchmark

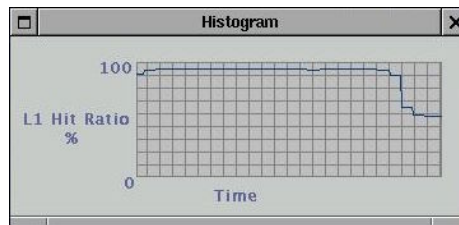
Table 1. Intrusion measurements

Time Interval (sec)	EP	IS	LU	CG	MG
2.0	0.22%	0.12%	0.09%	0.15%	0.03%
1.8	0.22%	0.15%	0.17%	0.33%	0.10%
1.6	0.23%	0.72%	0.10%	0.44%	0.12%
1.4	0.39%	0.85%	0.11%	0.49%	0.12%
1.2	0.50%	0.86%	0.24%	0.53%	0.17%
1.0	0.54%	0.86%	0.34%	0.53%	0.17%
0.8	0.55%	1.21%	0.38%	0.59%	0.21%
0.6	0.56%	1.61%	0.39%	0.63%	0.29%
0.4	0.66%	1.91%	0.54%	0.64%	0.33%
0.2	0.72%	2.13%	0.71%	0.67%	0.51%

has a high network utilization. Hence, it was more affected by the traffic generated by the on-line monitoring.

4 Example of Use

In order to demonstrate the usefulness of the hardware counters monitoring feature of RVision, we ran a parallel version of the SPEC Swim benchmark with problem size defined by $N1=N2=2048$, on the Tropical Linux cluster, and monitored both the MFlops/sec rate and the level 1 cache hit ratio during the execution of the program, using 1 second as monitoring interval. Figure 1 presents the system's average L1 hit ratio when the program started its execution, we observe that the cache hit ratio dropped considerable (most recent information is presented at the rightmost side). Looking at Figure 2, which presents a table with a snapshot with the monitoring values for all processors, we observe that the level 1 hit ratio was in the order of 54% in all processors, indicating that the application needed some sort of program restructuring, due to its poor utilization of the memory subsystem. This poor cache utilization reflects in the MFlops/sec rate, shown in Figure 3, which indicates a sustained performance of around 67 MFlops/sec.

**Fig. 1.** System's average L1 hit ratio running the SPEC Swim benchmark

RVision Monitoring Client						
Node/Proces...	CPU Mhz	tsc	P6_DCUI_LINE...	P6_DATA_ME...	L1 Cache Hit...	
tropical01/0	1.000119 GHz	1010066685	76836746	180031616	57.320415	
tropical01/1	1.000119 GHz	1010066700	77377709	166687741	53.579247	
tropical02/0	1.000069 GHz	1010023365	76847329	180768185	57.488464	
tropical02/1	1.000069 GHz	1010023189	76923796	166765035	53.872955	
tropical03/0	1.000121 GHz	1010025420	77152597	171854148	55.10577	
tropical03/1	1.000121 GHz	1010025058	77873352	180988081	56.97322	
tropical04/0	1.000103 GHz	1010004487	78475250	175237481	55.217773	
tropical04/1	1.000103 GHz	1010004097	78084583	167229630	53.306973	
tropical05/0	1.000086 GHz	1010066663	77630548	170822761	54.554916	
tropical05/1	1.000086 GHz	1010066320	77198987	167094277	53.79914	
tropical06/0	1.000086 GHz	1010044231	77912810	173115181	54.99366	
tropical06/1	1.000086 GHz	1010043923	76793973	173854845	55.82868	
tropical07/0	1.000119 GHz	1010150982	77542917	176194273	55.9901	
tropical07/1	1.000119 GHz	1010149793	77477947	166233787	53.392185	
tropical08/0	1.000103 GHz	1010069971	78425463	170862266	54.100185	
tropical08/1	1.000103 GHz	1010069884	78047726	168014759	53.547096	

Fig. 2. System's L1 hit ratio running the SPEC Swim benchmark

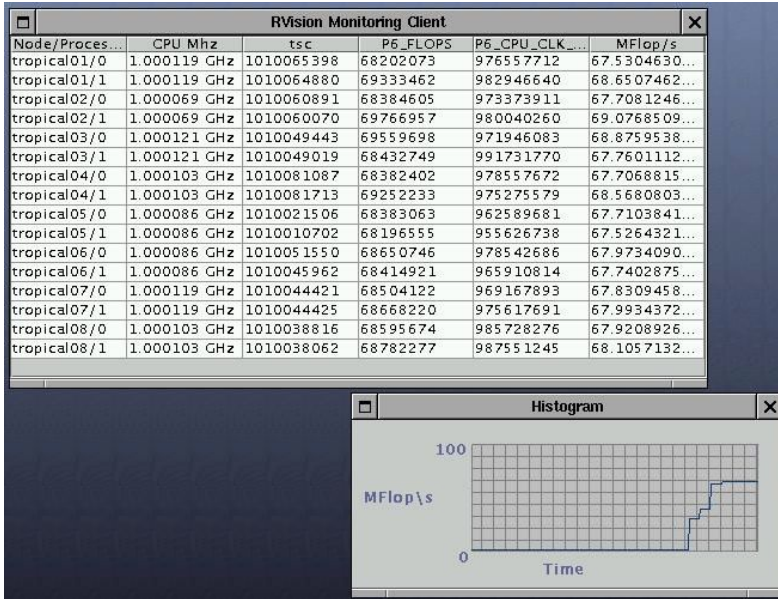


Fig. 3. System's MFlops/sec rate running the SPEC Swim benchmark

Looking at the source code, we observe that the basic data structure of the application is defined in the "Common block" shown in Figure 4, and used in loops such as the one presented in Figure 5. The use of high powers of 2 as array dimensions (2048 in this case) would result in an excessive cache miss ratio for this application, due to the low associativity of the level 1 cache (4-way). This problem occurs because 9 different arrays

are being used in the loop, and since the size of each array is multiple of the cache size, array elements with the same indices will map to the same cache set. Since each set can accommodate only 4 different entries, this loop generates an excessive number of conflict misses. This could be a well-known fact for application tuning specialists and experienced programmers, but not necessarily well understood by scientists with no background in computer architecture. Hence, the availability of monitoring tools that could indicate to system administrators when an application may need tuning is an invaluable asset.

```

PARAMETER (N1=2048, N2=2048)
COMMON U(N1,N2), V(N1,N2), P(N1,N2),
*   UNEW(N1,N2), VNEW(N1,N2), PNEW(N1,N2), UOLD(N1,N2),
*   VOLD(N1,N2), POLD(N1,N2), CU(N1,N2), CV(N1,N2),
*   Z(N1,N2), H(N1,N2), PSI(N1,N2)

```

Fig. 4. Definition of the main data structure in the SPEC Swim benchmark

```

DO 300 J=js,je
DO 300 I=1,M
  UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
  VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
  POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
  U(I,J) = UNEW(I,J)
  V(I,J) = VNEW(I,J)
  P(I,J) = PNEW(I,J)
300 CONTINUE

```

Fig. 5. Utilization of the arrays in one of loops in the SPEC Swim benchmark

In this case, the simple solution would be to “pad” the common block in Figure 4 with declaration of dummy vectors between the array declarations, as shown in Figure 6. The use of dummy vectors of 8 elements each will separate the mapping of the elements of the arrays with the same indices by one cache line, which will reduce the conflicts. When executing the modified program, we observe an increase in the Level 1 cache hit ratio to about 78%, as shown in Figure 7 and a sustained performance in the order of 84 MFlops/sec, as shown in Figure 8, which corresponds to an improvement of 25% in performance.

```

PARAMETER (N1=2048, N2=2048)
COMMON U(N1,N2), D1(8), V(N1,N2), D2(8), P(N1,N2),
*   UNEW(N1,N2), D3(8), VNEW(N1,N2), D4(8), PNEW(N1,N2),
*   D5(8), UOLD(N1,N2), D6(8), VOLD(N1,N2), D7(8),
*   POLD(N1,N2), D8(8), CU(N1,N2), D9(8), CV(N1,N2), D10(8),
*   Z(N1,N2), D11(8), H(N1,N2), D12(8), PSI(N1,N2)

```

Fig. 6. Padded common block for the SPEC Swim benchmark

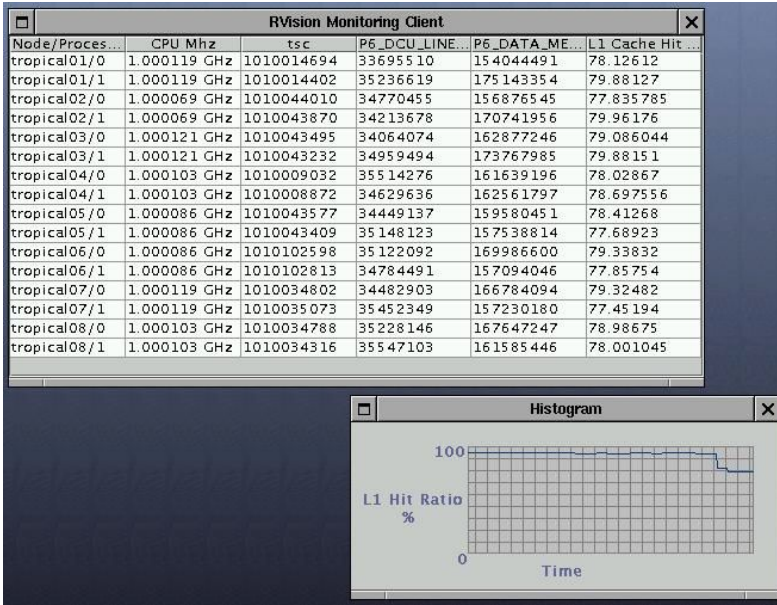


Fig. 7. System's L1 hit ratio running the padded version of the SPEC Swim benchmark

Other examples of performance problems that can be detected with this monitoring approach, which are not presented here due to space limitations are:

Utilization Rate: For a task on a dedicated compute node, this ratio should be close to 1. Lower values would indicate large system activity, which could require some application performance tuning.

IpC: Given that the processor has multiple functional units, a well tuned program should have IpC larger than 1.

Percentage of instructions dispatched that completed: This metric gives an indication of how well the speculation is working for the program. A low percentage could indicate that the program has loops with few iteration counts that could benefit from loop unrolling.

MIPS: This metric can be used for non-floating point intensive codes for monitoring of the sustained performance of the application.

L1 and L2 cache hit ratios: Are also useful to detect problems in the memory subsystem caused by capacity misses. These problems might be solved with loop transformations, such as blocking, to increase data locality.

5 Conclusions and Future Work

In this paper we presented a new approach to system monitoring for cluster architectures based on hardware counters. Our main goal is to provide an efficient tool for system

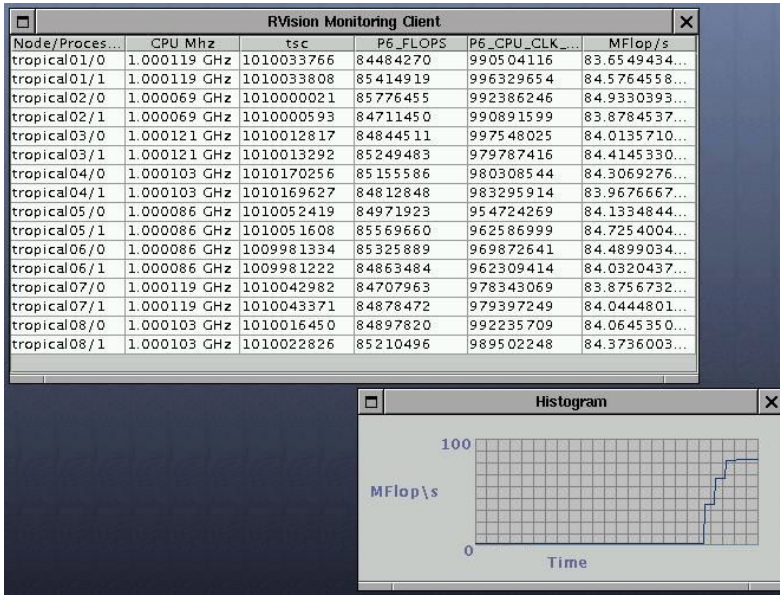


Fig. 8. System's MFlops/sec rate running the padded version of the Swim benchmark

administrators to detect programs that cause bottlenecks in cluster architectures so that throughput can be increased. As expected hardware counters monitoring has a very low intrusion resulting in more precise results. In dedicated systems it is also possible to detect applications that could need tuning, without the instrumentation needed by traditional hardware counters monitoring at application level.

To investigate this new concept we expanded our resource monitor RVision to access hardware counters on Intel Pentium III processors and to calculate derived performance metrics for Linux clusters. With this tool we analyzed the results obtained in a case study where we optimized the execution of a parallel version of the Swim benchmark from the SPEC suite, obtaining a performance increase in the order of 25%, on a Linux cluster with 16 processors. We also presented intrusion results for the expanded version of RVision running the NAS benchmark suite. We observed that in most cases intrusion is less than 1% even with high monitoring frequencies like 200 milliseconds.

We believe that systems oriented hardware monitoring tools are a very interesting alternative for system and application tuning. Our next steps is to support additional processor architectures like the Intel Pentium IV and the IBM Power4, where it will be possible to work with more derived metrics due to the increased number of hardware counters. More information about RVision and the package for download are available at <http://rvision.sourceforge.net>.

References

1. Zagha, M., Larson, B., Turner, S., Itzkowitz, M.: Performance Analysis Using the MIPS R10000 Performance Counters. In: Proceedings of Supercomputing'96. (November 1996)
2. Pettersson, M.: Linux X86 Performance-Monitoring Counters Driver. <http://user.it.uu.se/~mikpe/linux/perfctr/>. Computing Science Department; Uppsala University – Sweden. (2002)
3. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In: Proceedings of Supercomputing'00. (November 2000)
4. Research Centre Juelich GmbH: PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. <http://www.fz-juelich.de/zam/PCL/>. (2002)
5. May, J.M.: MPX: Software for multiplexing hardware performance counters in multithreaded programs. In: Proceedings of 2001 International Parallel and Distributed Processing Symposium. (April 2001)
6. DeRose, L.: The Hardware Performance Monitor Toolkit. In: Proceedings of Euro-Par. (August 2001) 122–131
7. Janssen, C.: The visual Profiler. <http://aros.ca.sandia.gov/~cljanss/perf/vprof/>. Sandia National Laboratories. (2002)
8. Buck, B., Hollingsworth, J.K.: Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In: Proceedings of Supercomputing'01. (November 2001)
9. Geers, N.: Automatic Collection of HPM Data in Parallel Applications. <http://www.uni-karlsruhe.de/~SP/software/tools/hpm/hpmcollect.de.html>. Scientific Supercomputing Center at University of Karlsruhe. (2002)
10. Ferreto, T.C., De Rose, C.A.F., DeRose, L.: RVision: An Open and High Configurable Tool for Cluster Monitoring. In: Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, Berlin, Germany (2002) 75–82
11. Sadourny, R.: The Dynamics of Finite-Difference Model of the Shallow-Water Equations. *Journal of Atmospheric Sciences* **32** (1975)
12. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. (2002)
13. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-929, NASA Ames Research Center (1995)
14. Saphir, W., Wijngaart, R.V.D., Woo, A., Yarrow", M.: New implementations and results for the NAS parallel benchmarks 2. In: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing. (1997)
15. Research Center in High Performance Computing: <http://www.cpad.pucrs.br> (2002)

ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis

Robert Bell, Allen D. Malony, and Sameer Shende

University of Oregon, Eugene, OR 97403 USA
{bertie,malony,sameer}@cs.uoregon.edu

Abstract. This paper presents the design, implementation, and application of *ParaProf*, a portable, extensible, and scalable tool for parallel performance profile analysis. *ParaProf* attempts to offer “best of breed” capabilities to performance analysts – those inherited from a rich history of single processor profilers and those being pioneered in parallel tools research. We present *ParaProf* as a parallel profile analysis framework that can be retargeted and extended as required. *ParaProf*’s design and operation is discussed, and its novel support for large-scale parallel analysis demonstrated with a 512-processor application profile generated using the TAU performance system.

1 Introduction

Perhaps the best known method for observing the performance of software and systems is profiling. Profiling techniques designed over thirty years ago [11], such as *prof* [17] and *gprof* [5] for Unix, are still apparent in the profiling approaches for modern computer platforms (e.g., *vprof* [9] and *cvperf* [20]). While ideological differences exist in profiling instrumentation and data collection, most notably between sample-based versus measurement-based approaches, profiling is the most commonly used tool in the performance analyst’s repertoire. Unfortunately, despite the ubiquitous nature of profiling, profile analysis tools have tended to be system specific, proprietary, and incompatible. Not only does this pose difficulties for cross-platform performance studies, but the lack of reusable profile analysis technology has slowed the development of next-generation tools. With the general availability of hardware counters in modern microprocessors, the complexity of performance profile data will increase, further exacerbating the need for more robust profile analysis tool support. Parallel software and systems introduce more difficult challenges to profile analysis tools. All the concerns for single processor performance analysis are present in parallel profiling, except now the profile data size is amplified by the number of processors involved. Profile analysis scalability is important for large-scale parallel systems where each thread of execution may potentially generate its own profile data set. Parallel execution also introduces new performance properties [1] and problems that require more sophisticated analysis and interpretation, both within a single profiling experiment and across experiments. The identification of parallel inefficiencies and load imbalances requires analysis of all execution threads

in a profile, whereas pinpointing reasons for poor scalability, for example, must combine detailed analysis across profiles generated with different numbers of processors. Lastly, the performance problem solving provided by the parallel profile analysis tool should reflect the system architecture as well as the model of parallelism used in the application.

This paper presents the design, implementation, and application of *ParaProf*, a portable, extensible, and scalable tool for parallel performance profile analysis. ParaProf attempts to offer “best of breed” capabilities to performance analysts – those inherited from a rich history of single processor profilers and those being pioneered in parallel tools research. However, ParaProf should be regarded not as a complete solution, but rather a parallel profile analysis framework that can be retargeted and extended as needed. Thus, in this paper we emphasize, in equal measure, the design of ParaProf and its support for customizability, in addition to its application in the context of the TAU performance system.

In the sections that follow, we first relate ParaProf to selected profiling tools, sequential and parallel, that highlight important features that ParaProf incorporates, or aspires too. Section §3 describes the ParaProf architecture and section §4 goes into more detail of the operation of core components. While ParaProf is being applied in many applications, for this paper, we focus on its novel support for large-scale profile analysis. Section §5 demonstrates ParaProf’s core capabilities on a highly-parallel application developed with the SAMRAI framework [8], one of several large-scale parallel environments where ParaProf is being deployed. In conclusion, we remark on how we see ParaProf evolving and its integration in a parallel performance diagnosis environment.

2 Related Work

Performance profiling characterizes the execution behavior of an application as a set of summary statistics associating performance data and metrics with program structure and semantics.¹ Profiling tools are distinguished by two aspects: what performance information is being analyzed, and how profile results are mapped to the program and presented to the user. Since *prof* [17] and *gprof* [5], execution time has been the standard performance data profiled, and the dominant mapping of execution time has been to program source statements. The inclusion of hardware performance counts in profile data (e.g., SGI’s *ssrun* [20]) has significantly increased the insight on processor and memory system behavior. The performance API (*PAPI* [2]) provides a common interface to hardware performance counters across microprocessors and is in use by most current profiling tools. The types of program mapping to source statements include statement-, loop-, and routine-level mapping. Callgraph profiling, pioneered in *gprof*, has also been extended to callpath profiling [6].

All of the above profiling features can be found in various forms in sequential profile analysis tools (e.g., *cvperf* [20], *DynaProf* [14], and *vprof* [9]). The

¹ We can also speak of profiling the performance of a system, but to simplify the discussion, we will focus on application performance.

HPCView tool [13] best exemplifies the integration of sequential analysis capabilities. Profile data from multiple sources can be input to HPCView, including performance data sets from hardware counters and different program executions. Internally, an extensible profile data format allows different data to be associated with program sites, each identified by a unique name and source mapping information. HPCView can compute derived performance statistics from mathematical expressions that involve performance data variables. The user interface allows navigation, grouping, and sorting of the profile data to assist in results analysis. However, HPCView is not a parallel profile analysis tool, per se. ParaProf can support many of HPCView’s features, as well as provide scalable parallel profile analysis.

Parallel profile analysis tools often target specific parallel programming models. The GuideView [10] tool analyzes performance of multi-threaded OpenMP applications. It has sophisticated analysis functions, including handling of multiple experiments and relating performance metrics to ideal values. VGV [7] extends GuideView to hybrid OpenMP and MPI applications. Unfortunately, neither GuideView nor VGV are open systems (now Intel proprietary), and, thus, are able to accommodate more general parallel profile data. The HPM Toolkit [3] also targets OpenMP and MPI applications, with emphasis on the analysis of hardware performance monitor data and derived statistics, but it only runs on IBM platforms. Aksum [4] handles OpenMP and MPI applications run on Linux clusters and can analyze profiles across multiple experiments. SvPablo [15] processes profiles captured for multiple processes on several platforms, and like HPCView, presents performance results in a source-level view. Expert [19] analyzes more extensive profile information to associate performance properties and problems to different program and execution views. With its general event representation and programmable analysis, ParaProf is able process the profile information for many of the scenarios handled by these tools. Moreover, although none of these tools is specifically focussed on large-scale parallelism, ParaProf’s profile data management and analysis is scalable to thousands of processors.

3 ParaProf Architecture

Software reuse and componentization lies at the heart of much current research in software engineering. Our goal in the ParaProf project is to apply these design principles to performance profile analysis and visualization. Given the commonalities of profile data and semantics, the opportunity is there to develop a framework for analysis and visualization that can be specialized for the parallel profiling problem. To this effect, we have abstracted four key components in the design of ParaProf: the *Data Source System* (DSS), the *Data Management System* (DMS), the *Event System* (ES), and the *Visualization System* (VS). Each component is independent, and provides well-defined interfaces to other components in the system. The result is high extensibility and flexibility, enabling us to tackle the issues of re-use and scalability. The remainder of this section describes each these components.

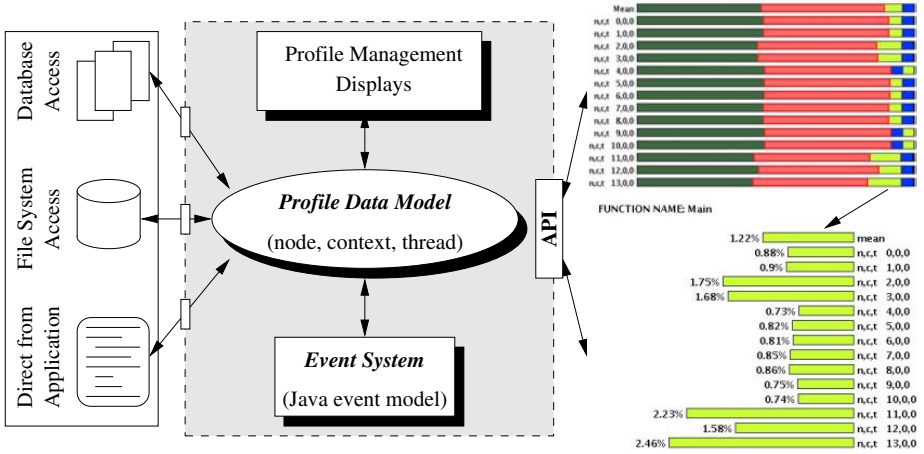


Fig. 1. ParaProf Architecture.

Current performance profilers provide a range of differing data formats. As done in HPCView [13], external translators have typically been used to merge profile data sets. Since much commonality exists in the profile entities being represented, this is a valid approach, but it requires the adoption of a common format. ParaProf’s DSS addresses this issue in a different manner. DSS consists of two parts. One, DSS can be configured with profile input modules to read profiles from different sources. The existing translators provides a good starting point to implement these modules. An input module can also support interfaces for communication with profiles stored in files, managed by performance databases, or streaming continuously across a network. Two, once the profile is input, DSS converts the profile data to a more efficient internal representation.

The DMS provides an abstract representation of performance data to external components. Its supports many advanced capabilities required in a modern performance analysis system, such as derived metrics for relating performance data, cross experiment analysis for analyzing data from disparate experiments, and data reduction for elimination of redundant data, thus allowing large data sources to be tolerated efficiently. The importance of sophisticated data management and its support for exposing data relationships is an increasingly important area of research in performance analysis. The DMS design provides a great degree of flexibility for developing new techniques that can be incorporated to extend its function.

The VS components is responsible for graphical profile displays. It is based on the Java2D platform, enabling us to take advantage of a very portable development environment that continues to increase in performance and reliability. Analysis of performance data requires representations from a very fine granularity, perhaps of a single event on a single node, to displays of the performance characteristics of the entire application. ParaProf’s current set of displays range from purely textual based to fully graphical. Significant effort has been put into

making the displays highly interactive and fast to draw. In addition, it is relatively easy to extend the display types to better show data relations.

Lastly, in the ES, we have provided a well-defined means by which these components can communicate various state changes, and requests to other components in ParaProf. Many of the display types are hyper-linked enabled, allowing selections to be reflected across currently open windows. Support for runtime performance analysis and application steering, coupled with maintaining connectivity with remote data repositories has required us to focus more attention on the ES, and to treat it as a wholly separate component system.

4 Operation

Let us now examine a typical usage scenario. A ParaProf session begins with a top-level profile management window that lets the user decide what performance experiments they want to analyze. An *experiment* is generally defined by a particular application and its set of associated performance profiles coming from experiment *trials*. As shown in Figure 2, the profile manager provides access to different profile sources (e.g., file system, performance database, or online execution) and to different profile experiments. Several profile data sets can be active within ParaProf at the same time. These may be from different experiments, allowing the user to compare performance behavior between applications, or from multiple runs of the same application where each profiled a different performance value. Note that the profile management window is where the user can also specify performance data calculations involving profile data values to derive new performance statistics. We discuss this further below.

Once performance profiles have been selected, ParaProf’s view set then offers means to understand the performance data at a variety of levels. The global profile view is used to see profile data for all application events across all threads² of execution in a single view. Of course, not all of the profile data can be shown, so the user has control over what performance metrics to display through menu options. The global view is interactive in the sense that clicking on various parts of the display provides a means to explore the performance data in more detail through other views. For example, the global view of a 20-processor VTF [18] parallel profile is shown in Figure 3. The two other views show 1) the performance of one event across all threads, gotten from clicking one color segment, and 2) the performance of all events across a single thread, gotten from clicking on the *node/context/thread* identifier. The full profile data for any thread can be shown in an ASCII view at any time.

All of the profile views are scrollable. For large numbers of events or large numbers of threads, scrolling efficiency is important and we have optimized this in our Java implementation. We have also provided support for sizing down the

² We use the term “thread” here in a general sense to denote a thread of execution. ParaProf’s internal profile data is organized based on a parallel execution model of shared-memory computing *nodes* where *contexts* reside, each providing a virtual address space shared by multiple threads of execution.

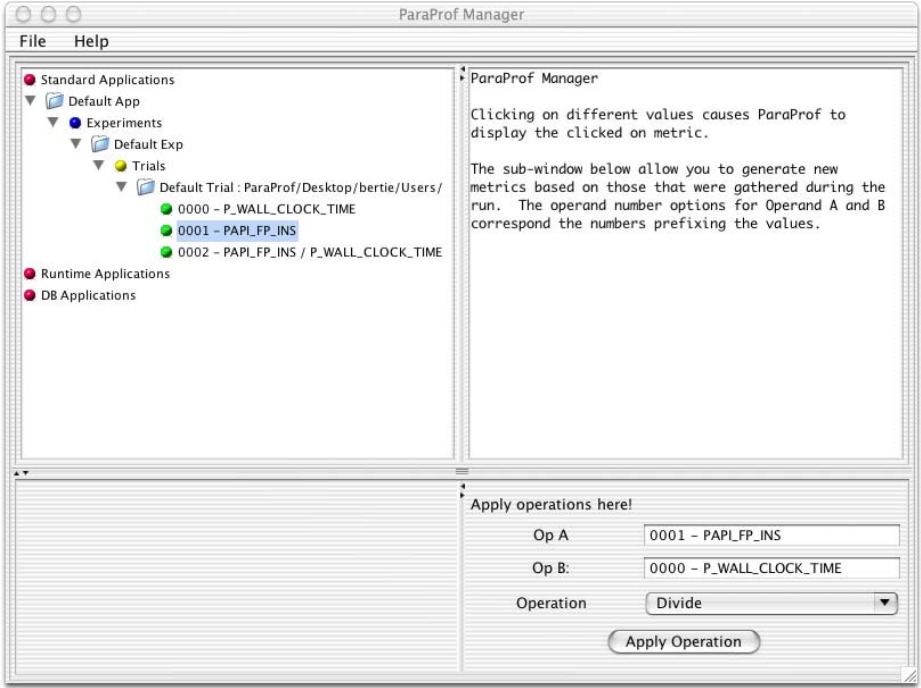


Fig. 2. ParaProf Profile Management Window.

display bars so that larger numbers of threads can be visualized. Nevertheless, for applications with high levels of parallelism, the two-dimensional profile displays can become unwieldy when trying to understand performance behavior. We have recently implemented histogram analysis support to determine the distribution of data values across their value range. A histogram display shows the distribution as a set of value bins (the number of bins is user defined) equally spaced between minimum and maximum. The number of profile values falling in a bin determines the height of the bin bar in the display.

Concurrent graphical and textual data representation at every stage provides for a seamless translation between levels of data granularity. In addition, the DMS controllers enable easy navigation between data sources thus facilitating understanding of data relationships in the AE. To highlight the importance of this last point, let us consider the following important analysis requirement. Most modern CPUs provide support for tracking a variety of performance metrics. The TAU performance system [16], for example, can simultaneously track CPU cycles, floating point instructions, data cache misses, and so on, using the PAPI library. A consistent problem in performance analysis has been how to examine on a large scale other useful performance metrics that can be derived from measured base values. For example, to determine how efficiently the pipelines of modern super-scalar architectures are being used, our performance system

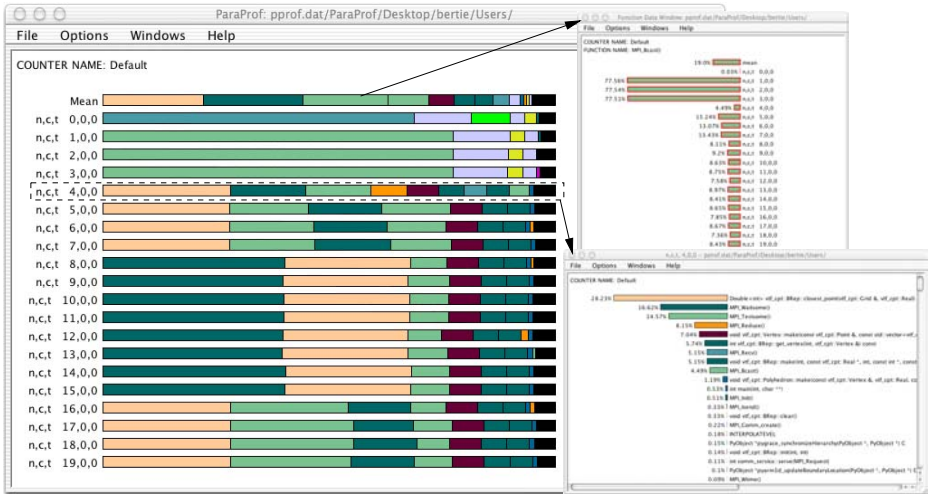


Fig. 3. ParaProf Display of VTF Profile.

might have access to both CPU cycles, and instruction counts, but gives us no correlation between the two. It is cumbersome to identify performance bottlenecks in CPU pipe utilization across thousands of events or threads if one has to make visual comparisons. To solve such problems, ParaProf’s DMS can apply mathematical operations to the performance metrics gathered, thus obtaining more detailed statistics. These operations can be applied to single executions, to executions in different runs, and even across experiments, thus allowing a complex range of derived metrics to be gathered. When examining data across experiments, the DMS can tolerate disparities in data sources (an event might be present in one source, but not another) by examining the commonalities that do exist, and presenting only those. Data sources are treated as operands in the system, and the DMS allows a user to compose operations to produce a variety of derived statistics. Results from this analysis can then be saved for future reference.

To aid in the visual process, the ES passes user activity between windows so that areas of interest highlighted in one window are propagated to all related windows. This greatly reduces the time spent correlating data shown in different displays. Another feature of ParaProf (not shown here) is its ability to recognize and create event groupings. Events can be grouped either at runtime by the performance system (supported in TAU), or post-runtime by ParaProf. Displays can then be instructed to show only events that are members of particular groups. This provides another mechanism for reducing visual complexity, and focusing only on points of interest or concern. ParaProf's DMS demonstrated its ability to simultaneously handle the large quantity of data comfortably. Any data redundancies present in the source data were eliminated as expected, and we showed (via duplication and renaming) that the only practical limits to ParaProf's operation are the memory limitations of the platform. Even these can be

alleviated to a great extent by the use of a central repository of data (such as a database) to which ParaProf can be directed to simply maintain links.

5 ParaProf Application

ParaProf, in its earlier incarnation as *jRacy*, has been applied in many application performance studies over the last several years. The additions we have made for profile management, multi-experiment support, derivative performance analysis, performance database interaction, and large-scale parallel profiles, plus overall improvements in efficiency, have elevated the tool to its new moniker distinction. Here we would like to focus on ParaProf’s support for profile analysis of large-scale parallel applications. This will be an important area for our future work with the DOE laboratories and ASCI platforms.

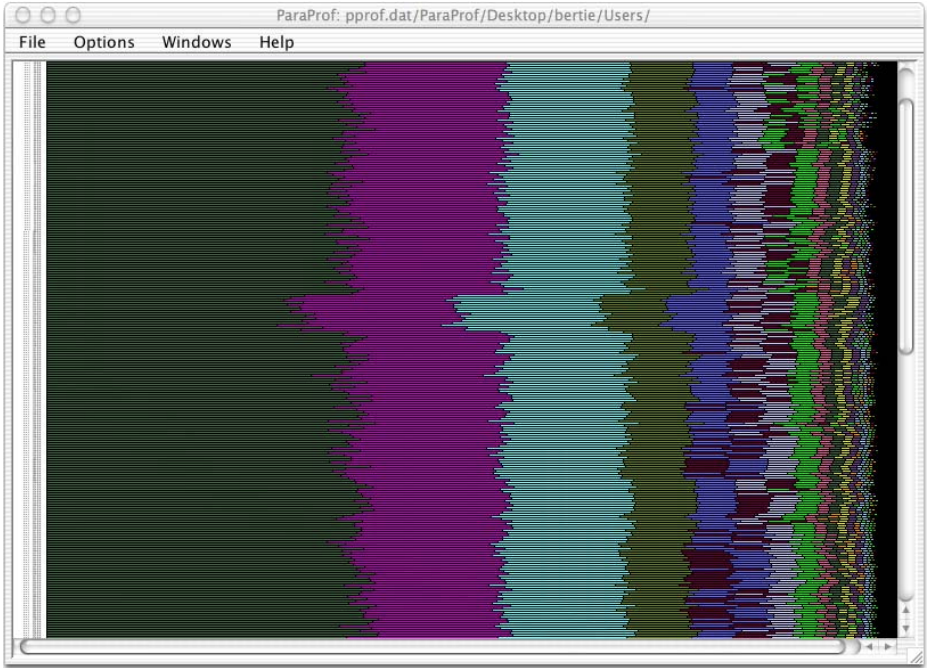


Fig. 4. Scalable SAMRAI Profile Display.

To demonstrate ParaProf’s ability to handle data from large parallel applications in an informative manner, we applied ParaProf to TAU data obtained during the profiling of a SAMRAI [8] application run on 512 processor nodes. SAMRAI is a C++ class library for structured adaptive mesh refinement. Figure 4 shows a view of exclusive wall-clock time for all events. The display is fully interactive, and can be “zoomed” in or out to show local detail. Even so, some

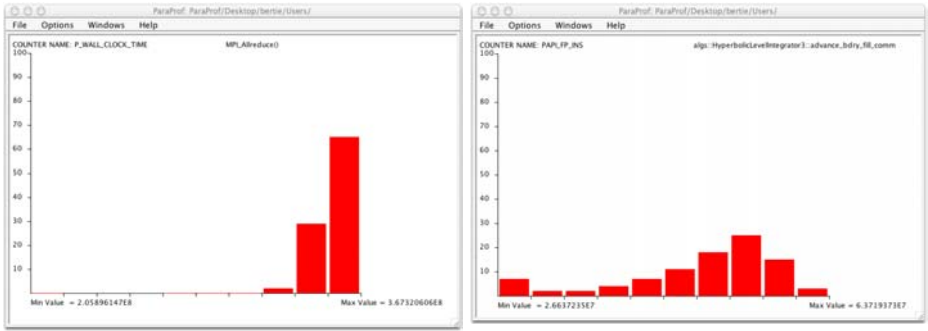


Fig. 5. SAMRAI Histogram Displays.

performance characteristics can still be difficult to comprehend when presented with so much visual data. Figure 5 shows one of ParaProf’s histogramming options that enable global data characteristics to be computed and displayed in a more concise form. In this case, we see binning of exclusive time data across the system for the routine whose total exclusive time is largest, `MPI_Allreduce()`. We have included another histogram showing the floating point instructions of a SAMRAI module; this data was obtained using TAU’s interface to PAPI. Clearly, the histogram view is useful for understanding value distribution.

6 Conclusion

The ParaProf profile analysis framework incorporates important features from a rich heritage of performance profiling tools while addressing new challenges for large-scale parallel performance analysis. Although ParaProf was developed as part of the TAU performance system, our primary goals in ParaProf’s design were flexibility and extensibility. As such, we are positioning ParaProf to accept profile data from a variety of sources, to allow more complete performance analysis. We will also continue to enhance ParaProf’s displays to aid in performance investigation and interpretation. In particular, there are opportunities to apply three-dimensional graphics to visualize profile results for very large processor runs. Currently, we are developing an advanced visualization library based on OpenGL that can be used by ParaProf.

There are two areas where we want to improve ParaProf’s capabilities. First, other parallel profile tools provide linkage back to application source code. The information needed for this is partly encoded in the profile event names, but ParaProf needs to have a standard means to acquire source mapping metadata (e.g., source files, and line and column position) to associate events to the program. We will apply our PDT [12] source analysis technology to this problem, and also hope to leverage the work in HPCView. In addition, source text display and interaction capabilities are required.

The second area is to improve how performance calculations are specified and implemented in ParaProf. Our plan is to develop an easy to use interface to

define analysis formulae, whereby more complex expressions, including reduction operations, can be created. These can then be saved in an analysis library for reuse in future performance profiling studies.

References

1. APART, IST Working Group on Automatic Performance Analysis: Real Tools. See <http://www.fz-juelich.de>.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.
3. L. DeRose, "The Hardware Performance Monitor Toolkit," *Euro-Par 2001*, 2001.
4. T. Fahringer and C. Seragiotto, "Experience with Aksum: A Semi-Automatic Multi-Experiment Performance Analysis Tool for Parallel and Distributed Applications," *Workshop on Performance Analysis and Distributed Computing*, 2002.
5. S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler," *SIGPLAN '82 Symposium on Compiler Construction*, pp. 120–126, June 1982.
6. R. Hall, "Call Path Profiling," *International Conference on Software Engineering*, pp. 296–306, 1992.
7. J. Hoeflinger et al., "An Integrated Performance Visualizer for MPI/OpenMP Programs," *Workshop on OpenMP Applications and Tools (WOMPAT)*, July 2001.
8. R. Hornung and S. Kohn, "Managing Application Complexity in the SAMRAI Object-Oriented Framework, *Concurrency and Computation: Practice and Experience*, special issue on Software Architectures for Scientific Applications, 2001.
9. C. Janssen, "The Visual Profiler." <http://aros.ca.sandia.gov/cljanss/perf/vprof/>.
10. KAI Software, a division of Intel Americas, "GuideView Performance Analyzer," 2001. <http://www.kai.com/parallel/kapro/guideview>.
11. D. Knuth, "An Empirical Study of FORTRAN Programs," *Software – Practice and Experience*, **1**:105–133, 1971.
12. K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, "Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," *Proc. Supercomputing 2000*, November, 2000.
13. J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A Tool for Top-down Analysis of Node Performance," *The Journal of Supercomputing*, **23**:81–104, 2002.
14. P. Mucci, "Dynaprof." <http://www.cs.utk.edu/mucci/dynaprof>
15. D. Reed, L. DeRose, and Y. Zhang, "SvPablo: A Multi-Language Performance Analysis System," *10th International Conference on Performance Tools*, pp. 352–355, September 1998.
16. TAU (Tuning and Analysis Utilities). <http://www.acl.lanl.gov/tau>.
17. Unix Programmer's Manual, "**prof** command," Section 1, Bell Laboratories, Murray Hill, NJ, January 1979.
18. VTF, Virtual Test Shock Facility, Center for Simulation of Dynamic Response of Materials. <http://www.cacr.caltech.edu/ASAP>.
19. F. Wolf and B. Mohr, "Automatic Performance Analysis of SMP Cluster Applications," Technical Report IB 2001-05, Research Centre Jülich, 2001.
20. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Supercomputing '96*, November 1996.

On Utilizing Experiment Data Repository for Performance Analysis of Parallel Applications^{*}

Hong-Linh Truong and Thomas Fahringer

Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
{truong,tf}@par.univie.ac.at

Abstract. Performance data usually must be archived for various performance analysis and optimization tasks such as multi-experiment analysis, performance comparison, automated performance diagnosis. However, little effort has been done to employ data repositories to organize and store performance data. This lack of systematic organization of data has hindered several aspects of performance analysis tools such as performance comparison, performance data sharing and tools integration. In this paper we describe our approach to exploit a relational-based experiment data repository in SCALEA which is a performance instrumentation, measurement, analysis and visualization tool for parallel programs. We present the design and use of SCALEA's experiment data repository which is employed to store information about performance experiments including application, source code, machine information and performance data. Performance results are associated with experiments, source code and machine information. SCALEA is able to offer search and filter capabilities, to support multi-experiment analysis as well as to provide well-defined interfaces for accessing the data repository and leveraging the performance data sharing and tools integration.

1 Introduction

Collecting and archiving performance data are important tasks required by various performance analysis and optimization processes such as multi-experiment analysis, performance comparison and automated performance diagnosis. However, little effort has been done to employ data repositories to organize and store performance data. This lack of systematic organization of data has hindered several aspects of performance analysis tools. For example, users commonly create their own performance collections, extract performance data and use external tools to compare performance outcome of several experiments manually. Moreover, different performance tools employ different performance data formats and they lack well-defined interfaces for accessing the data. As a result, the collaboration among performance tools and high-level tools is hampered.

^{*} This research is partially supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.

Utilizing a data repository and providing well-known interfaces to access data in the repository can help to overcome the abovementioned limitations. We can structure the data associated with performance experiments thus performance results can always be associated with their source codes and machine description on which the experiment has been taken. Based on that, any other performance tool can store its performance data for a given application to the same repository thus providing a large potential to enable more sophisticated performance analysis. And then, any other tools or system software can easily access the performance data through a well-defined interface. To do so, we have investigated and exploited a relational-based experiment data repository in SCALEA [12,11] which is a performance instrumentation, measurement, analysis and visualization tool for parallel programs.

In this paper, we present the design and use of SCALEA's experiment data repository which is employed to store performance data and information about performance experiment which alleviates the association of performance information with experiments and source code. SCALEA's experiment data repository has been implemented with relational database, SQL and accessed through interfaces based on JDBC. We demonstrate significant achievements gained when exploiting this data repository such as the capabilities to support search and multi-experiment analysis, to facilitate and leverage performance data sharing and collaboration among different tools. We also discuss other directions on utilizing performance data repository for performance analysis.

The rest of this paper is organized as follows: Section 2 details the SCALEA's experiment data repository. We then illustrate achievements gained from the use of the experiment data repository in Section 3. We discuss other directions to utilize the experiment data repository in Section 4. The related work is presented in Section 5 followed by the conclusion and future work in Section 6.

2 Experiment Data Repository

2.1 Experiment-Related Data

Figure 1 shows the structure of the data stored in SCALEA's experiment data repository. An *experiment* refers to a sequential or parallel execution of a program on a given target architecture. Every experiment is described by *experiment-related data*, which includes information about the application code, the part of a machine on which the code has been executed, and performance information. An application (program) may have a number of code versions, each of them consists of a set of source files and is associated with one or several experiments. Every source file has one or several static code regions (ranging from entire program units to single statements), uniquely specified by their positions – start/end line and column – where the region begins and ends in the source file.

Experiments are associated with virtual machines on which they have been taken. The virtual machine is a collection of physical machines to execute the experiment; it is described as a set of *computational nodes* (e.g. single-processor

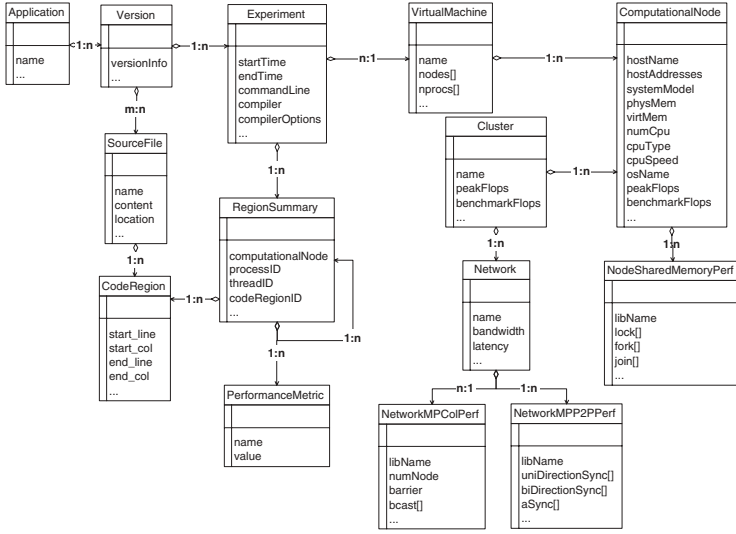


Fig. 1. SCALEA's Experiment-Related Data Model

systems, SMP nodes) of clusters. A *Cluster* is a group of computational nodes (physical machines) connected by specific networks. Computational nodes in the same cluster have the same physical configuration. Note that this structure is still suitable for a network of workstations as workstations can be classified into groups of machines having the same configuration. Specific information of physical machines such as memory capacity, peak FLOPS is measured and stored in the data repository. In addition, for each computational node, performance characteristics of shared memory operations (e.g. lock, barrier, fork/join thread) are benchmarked and stored in *NodeSharedMemoryPerf*. Similarly, for each network of a cluster, performance characteristics of message passing model are also benchmarked and stored in *NetworkMPColPef* and *NetworkMPP2PPerf* for collective and point-to-point operations, respectively.

A *region summary* refers to the performance information collected for a given code region on a specific *processing unit* (consisting of computational node, process, thread). The region summaries are associated with performance metrics that comprise performance overheads, timing information, and hardware parameters. A region summary has a link to a parent region summary; this link reflects the calling relationship between the two regions recorded in the dynamic code region call graph [12].

2.2 Implementation Overview

Figure 2 depicts components that interact with the experiment data repository. The *post-processing* is used to store source programs and the instrumentation description file [12] generated by SCALEA instrumentation system [11] into the

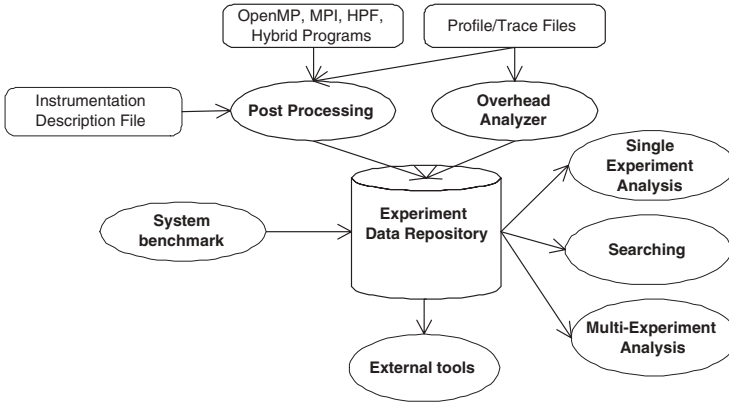


Fig. 2. Components interacting with SCALEA’s Experiment Data Repository

data repository. In addition, it filters raw performance data collected for each experiment and stores the filtered data into the repository.

The *overhead analyzer* performs the overhead analysis according to the overhead classification [11] based on filtered data in the repository (or trace/profile files). The resulting overhead is then stored into the data repository.

The *system benchmark* is used to collect system information (memory, hard-disk, etc), to determine specific information (e.g. overhead of probes, time to access a lock), and to perform benchmarks for every target machine of interest. By using available MPI and OpenMP benchmarks, reference values of system performance are obtained for both message passing (e.g. blocking send/receive, collective) and shared memory operations (e.g. lock, barrier) with various networks (e.g. Fast-Ethernet, Myrinet), libraries (e.g. MPICH, PGI OpenMP) on different architectures (e.g. Linux x86, Sun Sparc). The obtained results aim to assist the correlation analysis between application- and system-specific metrics.

Based on data availability in the repository, various analyses can be conducted such as single- and multi-experiment analysis, searching. Moreover, data can be exported into XML format which further facilitates accessing performance information by other tools (e.g. compilers or runtime systems) and applications.

The experiment data repository is implemented with PostgreSQL [8] which is a relational database system supported on many platforms. All components interacting with the data repository are written in Java and the connection realizing between these components with the database is powered by JDBC.

2.3 Experiment-Related Data APIs

Each table shown in Fig. 1 is associated with a Java class. In addition, we define two classes *ProcessingUnit* and *ExperimentData*. The first class is used to describe the *processing unit* where the code region is executed; a process unit consists of information about *computational node*, *process*, *thread*. The latter

implements interfaces used to access experiment data. Bellow, we just highlight some classes with few selected data members and methods:

```
public class PerformanceMetric {
    public String metricName;
    public Object metricValue;
    ...
}
public class ProcessingUnit {
    ...
    public ProcessingUnit(String node,int process, int thread) {...}
}
public class RegionSummary {
    ...
    public PerformanceMetric[] getMetrics(){...}
    public PerformanceMetric getMetric(String metricName){...}
}
public class ExperimentData {
    DatabaseConnection connection;
    ...
    public ProcessingUnit[] getProcessingUnits(Experiment e){...}
    public RegionSummary[] getRegionSummaries(CodeRegion cr, Experiment e)
        {...}
    public RegionSummary getRegionSummary(CodeRegion cr, ProcessingUnit
        pu, , Experiment e) {...}
    ...
}
```

Based on the well-defined APIs, external tools can easily access data in the repository, using the data for their own purpose. For example, a tool can compute $\frac{\text{identified overhead}}{\text{total execution time}}$ ratio for code region 1 in computational node *gsr1.vcpc.univie.ac.at*, process 1, thread 0 of experiment 1 based on identified overhead (denoted by *oall_ident*) and total execution time (denoted by *wtime*) as follows:

```
CodeRegion cr = new CodeRegion("region1");
Experiment e = new Experiment("experiment1");
ProcessingUnit pu = new ProcessingUnit("gsr1.vcpc.ac.at",1,0);
ExperimentData ed = new ExperimentData(new DatabaseConnection(...));
RegionSummary rs = ed.getRegionSummary (cr,pu,e);
PerformanceMetric overhead=rs.getMetric("'oall_ident'");
PerformanceMetric wtime =rs.getMetric("'wtime'");
double overheadRatio=((Double)overhead.metricValue).doubleValue()/
    ((Double)wtime.metricValue).doubleValue();
```

3 Achievements of Using Experiment Data Repository

3.1 Search and Filter Capabilities

Most existing performance tools lack basic search and filter capabilities. Commonly, the tools allow the user to browse code regions and associated performance metrics through various views (e.g. process time-lines with zooming and

scrolling, histograms of state durations and message data [14,3]) of performance data. Those views are crucial but mostly require all data to be loaded into the memory, eventually making the tools in-scalable. Moreover, the user has difficulty to find out the occurrence of events with interesting criteria of performance metrics, e.g. code regions with overhead of data movement [11] larger than 50% of total execution time. Search of performance data will allow the user to quickly identify interesting code regions. Filtering performance data being visualized helps to increase the scalability of performance tools. Utilizing a data repository allows to power the archive, to facilitate search and filter with great flexibility and robustness based on SQL language and to minimize the implementation's cost.

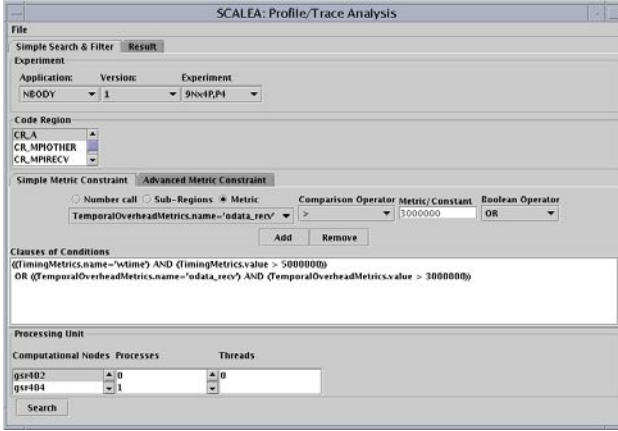


Fig. 3. Interface for Search and Filter in SCALEA



Fig. 4. Specify complex performance conditions

Figure 3 presents the interface for search and filter in SCALEA. The user can select any experiment for searching code regions under selection criteria. For each experiment, the user can choose *code region types* (e.g. send/receive, OpenMP loop), specify *metric constraints* based on performance metrics (timing, hardware parameter, overhead) and opt the *processing unit* (computational nodes, processes, threads) on which the code regions are executed. Metric constraints can be made in a simple way by forming clauses of selection conditions based on available performance metrics (see Fig. 3). They can also be constructed by selecting quantitative characteristics (Figure 4). For instance, the user may define characteristics for L2 cache miss ratio by expressing L2 cache miss ratio as $r_{L2 \text{ cache miss ratio}} = \frac{L2 \text{ cache misses}}{L2 \text{ cache accesses}}$ and then discretizing the L2 cache miss ratio as follows: *good* if $r_{L2 \text{ cache miss ratio}} \leq 0.3$, *average* if $0.3 < r_{L2 \text{ cache miss ratio}} < 0.7$, and *poor* if $r_{L2 \text{ cache miss ratio}} \geq 0.7$. These quantitative characteristics can be stored into the experiment data repository for later use. SCALEA will transfer user-specified conditions into SQL language, perform the search and

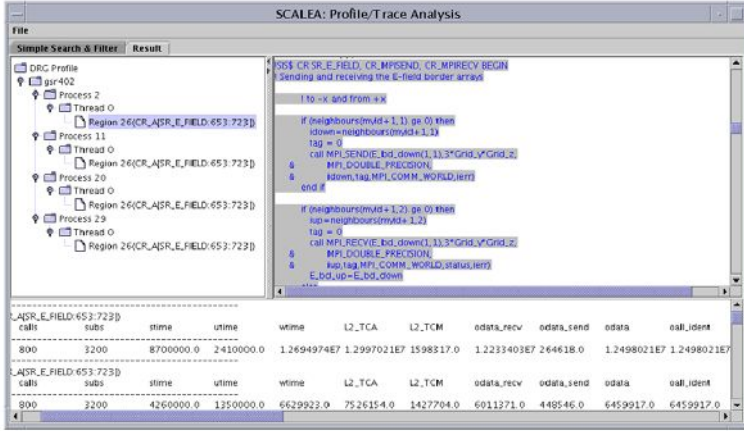


Fig. 5. Results of Performance Search

return the result. For example, the searching result based on the conditions in Fig. 3 is shown in Fig. 5. In the top-left window, the code region summaries met the search conditions are visualized. By clicking into a code region summary, the source code and corresponding performance metrics of the code region summary will be shown in the top-right and bottom window, respectively.

3.2 Multi-experiment Analysis

Most existing performance tools [5,3,13] investigate the performance for individual experiments one at a time. To archive experiment data in the repository, SCALEA goes beyond this limitation by supporting multi-experiment analysis. The user can select several experiments, code regions and performance metrics of interest of which associated data are stored in the data repository (see Figure 6). The outcome of every selected code regions and metrics is then analyzed and visualized for all experiments. SCALEA's multi-experiment analysis supports:

- **performance comparison for different sets of experiments:** The user can analyze the overall execution of the application across different sets of experiments; experiments in a set are grouped based on their properties (e.g. problem sizes, communication libraries, platforms).
- **overhead analysis for multi-experiment:** Various sources of performance overheads across experiments can be examined.
- **study parallel speedup and parallel efficiency at both program and code region level:** Commonly, those metrics are applied only at the level of the entire program. SCALEA, however, supports to examine the scalability at both program and code region level.

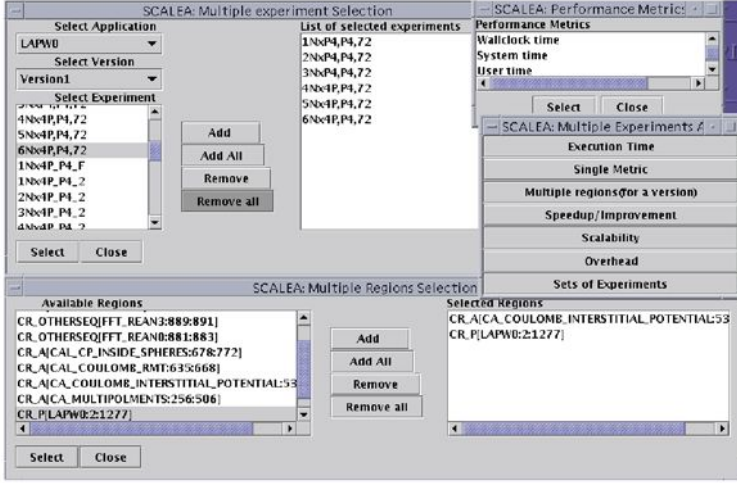


Fig. 6. Interface for Multi-Experiment Analysis

3.3 Data Sharing and Tools Integration

One of the key reasons for utilizing the data repository is the need to support data sharing and tools integration. Via well-defined interfaces, other tools can retrieve data from the experiment data repository and use the data for their own purpose. For example, AKSUM [2] which is a high-level semi-automatic performance bottleneck analysis has employed SCALEA to instrument user's programs, to measure and analyze performance overheads of code regions of the programs. AKSUM then accesses performance data (overheads, timing, hardware parameters) of code regions on the data repository, computes performance properties [1] and conducts the high-level bottleneck analysis based on these properties. In another case, the PerformanceProphet [7] which supports performance modeling and prediction on cluster architectures has used data stored in the experiment data repository to build the cost functions for the application model represented in UML forms. The model is then simulated in order to predict the execution behaviour of parallel programs.

Performance data can also be exported into XML format so that it can easily be transferred and processed by other tools. For example, performance data of *code region 1* can be saved in XML format as follows:

```
<coderegion id="1">
  <metric name="wtime" value="1.09039995E8" />
  <metric name="odata_send" value="2986000.0" />
  <metric name="odata_rcv" value="5.6923546E7" />
</coderegion>
```

where each performance metric of the region is represented as a tuple (*name,value*). Note that *wtime* (stands for wall clock time), *odata_send* (overhead of send operations), *odata_rcv* (overhead of rcv operations) are unique

performance metric names described in a *performance metric catalog* which holds information of performance metrics (e.g. unique metric name, data type, well-defined meaning) supported by SCALEA.

4 Further Directions

In this section, we discuss further directions on utilizing the experiment data repository for performance analysis:

- **A query language for performance data** can be designed to support ad hoc and interactive searching and filtering for occurrence of events with criteria of performance metrics and/or performance problems in order to facilitate flexible and efficient discovery of interesting performance information. Such a language can be implemented on top of search facilities provided by database systems and be based on performance property language [1].
- **Automatic scalable analysis techniques** such as decision trees, rule associations, clusters, classifications should be exploited to discover the knowledge of performance information on the repository. These techniques are particularly useful for executing very complex queries on non-main-memory data. However, currently these techniques are rarely used in performance analysis due to lack of systematic organization of performance data.
- **Providing standardization APIs for acquiring and exploiting performance data** is one of the keys to bring the simplicity, efficiency and success to the collaboration among tools. The well-defined APIs should be independent on the internal data representation and organization of each tool but based on an agreement of well-defined semantics.

5 Related Work

Significant works on performance analysis have been done by Paradyn [10], TAU [5], VAMPIR [3], EXPERT [13], etc. SCALEA differs from these approaches by storing experiment-related data to a data repository, and by supporting also multi-experiment performance analysis.

In [4], information about each experiment is stored in a Program Event and techniques for comparison between experiments are done automatically. A prototype of Program Event has been implemented, however, the lack of capability to export and share performance data has hindered external tools from using and exploiting data in Program Events.

Prophesy [9] provides a repository to store performance data for performing the automatic generation of performance models. Data measured and analyzed by SCALEA can be used by Prophesy for modeling systems.

USRA Tool family [6] collects and combines information of parallel programs from various sources at the level of subroutines and loops. Information is stored in flat files which can further be saved in a format understood by spreadsheet

programs. SCALEA's repository provides a better infrastructure for storing, querying and exporting performance data with a relational database system.

APART proposes the performance-related data specification [1] which stimulates our experiment-related data model. Besides performance-related data, we also provide system-related data.

6 Conclusions and Future Work

The main contributions of this paper are centered on the design and achievements of the experiment data repository in SCALEA which is a performance analysis tool for OpenMP/MPI and mixed parallel programs. We have described a novel design of SCALEA's experiment data repository holding all relevant experiment information and demonstrated several achievements gained from the employment of the data repository. The data repository has increasingly supported the automation of the performance analysis and optimization process.

However, employing the data repository introduces extra overheads in comparison with other non-employing-data-repository tools; the extra overheads occur in filtering and storing raw data to and retrieving data from the database. In the current implementation, we observed the bottleneck in accessing the data repository with large data volume. We are going to enhance our access methods and database structure to solve this problem. In addition, we intend to work on the issues discussed in Section 4.

References

1. T. Fahringer, M. Gerndt, Bernd Mohr, Felix Wolf, G. Riley, and J. Träff. Knowledge Specification for Automatic Performance Analysis, Revised Version. APART Technical Report, Workpackage 2, Identification and Formalization of Knowledge, Technical Report <http://www.kfa-juelich.de/apart/result.html>, August 2001.
2. T. Fahringer and C. Seragiotto. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *International Conference On High Performance Computing (HiPC 2002)*, Bangalore, India, December 2002. Springer Verlag.
3. Pallas GmbH. VAMPIR: Visualization and Analysis of MPI Programs. <http://www.pallas.com/e/products/vampir/index.htm>.
4. Karen L. Karavanic and Barton P. Miller. Experiment Management Support for Performance Tuning. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE.
5. Allen Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. In *In G. Kotsis and P. Kacsuk (Eds.), Third International Austrian/Hungarian Workshop on Distributed and Parallel Systems (DAP-SYS 2000)*, pages 37–46. Kluwer Academic Publishers, Sept. 2000.
6. Insung Park, Michael Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *International Journal of Parallel Programming*, 26(5):541–??, 1998.

7. S. Pllana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.
8. PostgreSQL 7.1.2. <http://www.postgresql.org/docs/>.
9. V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and Ivan R. Judson. Prophesy: An Infrastructure for Analyzing and Modeling the Performance of Parallel and Distributed Applications. In *Proc. of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC's 2000)*, Pittsburgh, August 2000. IEEE Computer Society Press.
10. Paradyn Parallel Performance Tools. <http://www.cs.wisc.edu/paradyn/>.
11. Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International EuroPar Conference (EuroPar 2002)*, Lecture Notes in Computer Science, Paderborn, Germany, August 2002. Springer-Verlag.
12. Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Proceeding of the 9th IEEE/ACM High-Performance Networking and Computing Conference (SC'2001)*, Denver, USA, November 2001.
13. Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP-11)*, pages 13–22. IEEE Computer Society Press, February 2003.
14. Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.

Flexible Performance Debugging of Parallel and Distributed Applications

Jacques Chassin de Kergommeaux¹, Cyril Guilloud², and B. de Oliveira Stein³

¹ Laboratoire LSR-IMAG, B.P. 72, 38402 St. Martin d'Hères Cedex, France
Jacques.Chassin-de-Kergommeaux@imag.fr, <http://drakkar.imag.fr/~chassin>

² Laboratoire ID-IMAG, ENSIMAG Antenne de Montbonnot
ZIRST, 51 avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France

Cyril.Guilloud@imag.fr, <http://www-id.imag.fr/~guilloud>

³ Departamento de Eletrônica e Computação,
Universidade Federal de Santa Maria – RS, Brazil.
benhur@inf.ufsm.br

Abstract. The Pajé approach to help performance debugging of parallel and distributed applications is to provide behavioral visualizations of their program executions to programmers. This article describes how Pajé was designed to be usable for a variety of programming models. Trace recording and trace manipulation facilities can be generated from a description of the events to be traced. The visualization tool includes a trace-driven simulator reconstructing the behavior of the observed executions of the applications being debugged. The Pajé simulator is generic and can be specialized by the types of the objects to be visualized, the representation of these objects and their relations.

1 Introduction

Performance debugging requires that programmers of parallel and distributed applications identify the bottlenecks, overheads or out of balance computations resulting in inefficient use of parallel resources by their programs. Tools supporting the identification of such problems should help programmers understand the runtime behavior of their applications. Of particular interest are behavioral visualization tools which ease the understanding of the interactions of processes and/or threads during parallel and distributed executions. Behavioral visualization tools reconstruct an execution of a debugged application from execution traces. Such tools need to adapt easily to the variety and continuous evolution of programming models and visualization techniques.

This article describes the Pajé tracing and visualization tools, aimed at easing performance debugging of parallel and distributed applications. These tools adapt to a variety of programming models. Traces are recorded by tracing functions, inserted in the observed applications and automatically generated by “Tumit¹” from a description of the events to be recorded. The Pajé visualization

¹ Inuit word meaning animal track, track and trace being the same word in French.

tool is generic, allowing users to describe what they would like to see and how this should be visualized.

The next section discusses the notion of behavioral observation of executions of parallel and distributed applications. The following gives a brief overview of the user-guided generation of trace monitoring and manipulation functions. The genericity of the Pajé visualization tool is then briefly recalled and exemplified. The two last sections survey rapidly related work and conclude the article.

2 Behavioral Observation of Parallel and Distributed Executions

2.1 Monitoring Techniques

Most monitoring tools are either clock driven or event driven [4] (see figure 1). Clock driven monitoring amounts to have the state of the observed system registered at periodical time intervals. This technique may be useless to identify the origins of some overheads of parallel programs: bottlenecks, communication or idling times. Event driven monitoring is triggered by the advent of observable events. In the *timing* approach, the time spent in various parts of the observed program is measured. In the *counting* approach, the number of occurrences of the observed events is recorded in global performance indices. *Tracing* is done by recording each of the observed events in a performance trace. Each record is composed of several parameters describing the event (see §3.2).

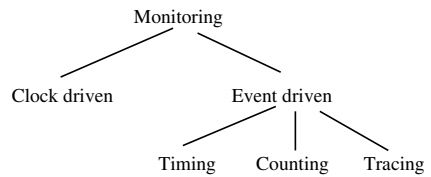


Fig. 1. Classification of monitoring tools

2.2 Tracing and Behavioral Visualization

Event driven monitoring is intensively used for performance debugging of parallel and distributed applications. Counting and timing are used to compute global performance indices whose evolution can sometimes be related to locations (procedures) [7] or execution phases [1] of the observed applications so that programmers can identify where and when the time goes. However, to reconstruct the behavior of the observed execution, more information is required and implies the use of execution traces. Traces can be used to reconstruct the dynamic behavior of an observed application so that the activity of processes, threads and processing elements and their communications can be checked along the execution time axis [12] (see figure 6).

2.3 Versatile Visualization Tool

Behavioral visualization tools include a trace-driven simulator to reconstruct the behavior of traced executions. Such simulators are usually dedicated to a given

programming model. However, there exist various programming models that are used to program parallel or distributed applications [2,9,15,13]. In addition, these applications are often programmed using layered middleware systems; explaining performance problems may imply to observe their execution at several layers simultaneously, to relate a performance problem observed at one layer — MPI program for example — to the behavior of a lower abstraction layer — transport layer for example. Therefore it is necessary that behavioral visualization tools can be adapted to various programming models and able to reconstruct executions at several abstraction layers simultaneously.

2.4 The Pajé Approach

Pajé is a trace-driven behavioral visualization tool combining interactivity, scalability, extensibility and genericity properties [3,5]. Interactivity allows users to move back and forth in time, to obtain more details on displayed objects or relate a visualization to a source program. Scalability is mainly provided by zooming functionalities in a fully interactive way so that application developers can zoom in from data aggregated along the space (groups of threads, of nodes) or time dimension to a detailed visualization of where the problem appears to be. Extensibility is the possibility of adding easily new functionalities to the tool; it is provided in Pajé by a careful modular design. Genericity is the faculty allowing users to adapt the Pajé visualization to various programming models by describing in the input data what they wish to visualize and how this should be done (see §4). Pajé could already be used in a variety of domains such as performance debugging of parallel [3] and distributed applications and monitoring of clusters [5]. The latest developments concern the production of execution traces which should be simple for all programming models that can be visualized with Pajé. The Tunit tool, described in the next section, allows tracing functions to be generated automatically as well as trace processing functions. The aim is to provide at least the same flexibility for monitoring as is provided by Pajé for visualization.

3 Tunit

The aim of Tunit is to help middleware or application developers to easily generate trace recording and trace manipulation functions. “Classical” tracing problems such as global dating or intrusion limitation and compensation [4] are not dealt with in this section.

3.1 Software Tracing Instrumentation Techniques

Instrumentation is the insertion of code to detect and record the application events. It can be done at several possible stages: in the source code of the traced application or at compile-time or by instrumenting the middleware used by the

application or by direct instrumentation of compiled object code [4]. Compile-time and object code instrumentation contradict the objective of being easy to adapt: the former requires to have access to the source code of the compiler and modify it while the latter is dependent on the operating system and the target platform. Instrumentation of communication libraries or runtime systems is not difficult for the developers of these middleware systems. However, if tracing functions are not included in the middleware, they have to be realized by the implementors of visualization tools or by their users; these people need to have access to the source codes of the middleware to be instrumented and also to update the instrumentation each time a new version of this middleware is released. For these reasons, it was decided not to realize another tracer for Pajé but instead to assist users in the development of ad hoc tracing facilities. From a description of the events to be traced, Tunit generates a set of tracing functions. Calls to these functions are inserted in the applications or middleware systems to be observed. Tunit also provides a set of data manipulation facilities to transform local into global dates or convert the monitored data from one format into another one (see §3.2). To guarantee the maximum flexibility, format conversion functions are not directly provided but instead generated from a set of rewriting rules provided by the users (see §3.3).

3.2 Data Format

A trace is composed of event records. Each record contains at least the following information: type of the event, (physical) date of the event and process identification of the process having performed the event. Some records contain additional parameters of the traced event such as receiver (sender) identification and message length in case of message emission (reception). Traces are usually recorded in a compact binary format to avoid memory buffer saturation and frequent disk accesses likely to alter the behavior of the observed application. They can be later processed — for example to correct the offsets and drifts of local clocks with respect to a global one [10] —, sorted and converted in the input data format of a visualization tool. The visualization tool input data format is a textual self defined data format (see [6] and §4). In Pajé, trace format conversion is considered inevitable and therefore assisted by Tunit. Trace format conversion might involve more than coding the same data in a different way. Suppose for example that a blocking receive is recorded — by a “semantic-aware” tracer — as a single event whose format includes two dates bounding the period when the emitting thread is blocked. When converting into the Pajé visualization input format, this event has to be transformed in two different visualization events, corresponding to the blocking and unblocking of the thread; the reason is that Pajé does not have notions of programming model semantics and has no notion of blocking receive.

3.3 Tunit Instrumentation Technique

Tunit performs direct code instrumentation: the instructions generating the event records must be inserted in the programs before compilation (by the user of Pajé or by a preprocessor). Traced events are user-defined which provides complete flexibility in numbers and types of the recorded parameters. Tracing user-defined events requires specific functions to record these events and process the event records. In Tunit, these functions are automatically generated from events definitions. To illustrate the Tunit approach, we present examples extracted from the tracing and visualization of a small program using the Taktuk [11] communication library. Taktuk uses binomial tree topologies to launch parallel programs in logarithmic time on large-sized clusters of computers and to establish efficient communication networks. It can be seen as a communication library using two levels of parallelism: nodes and threads.

```

LEVEL (COM)                                     // Definition of the COM level
EVENT (PACK_MSG, int node, int thread)          // packing message to send
EVENT (POST, int node, int thread, int dst, int stamp) // send
EVENT (RECV, int node, int thread, int src, int stamp) // receive
EVENT (END_UNPACK, int node, int thread)        // end of receive

```

Fig. 2. Definition of events

Events definition. In this short example (see figure 2), four events are defined inside one level called COM: events are grouped in levels to ease the filtering of monitored data or allow users to mix traces recorded at different abstraction levels. For each event we define its name and for each user parameter, its name and type. The identification of the process (or thread) emitting the event has to be provided explicitly by the user since this identification is not standard, process or nodes in some cases, nodes and local thread identification in other cases. Stamps are used to match POST and RECV events.

```

(COM, PACK_MSG) -> ("PajePushState @ S t-@-@ TakPack", date, node, thread)
(COM, POST)      -> ("PajeStartLink @ COM TP TakCom T-@-@ key-@-@-@",
                    date, node, thread, thread, dst, stamp)
                  -> ("PajePopState @ S t-@-@", date, node, thread)
(COM, RECV)      -> ("PajeEndLink @ COM TP TakCom T-@-@ key-@-@-@",
                    date, node, thread, src, thread, stamp)
                  -> ("PajePushState @ S t-@-@ TakRecv", date, node, thread)
(COM, END_UNPACK) -> ("PajePopState @ S t-@-@", date, node, thread)

```

Fig. 3. Example Tunit conversion rules

```

// Definition of the hierarchical tree represented graphically in figure 6
// Keywords of the Pajé input data format are written in bold font.
PajeDefineContainerType P 0 "Prog"           // 0: no parent in type hierarchy
PajeDefineContainerType N P "Node"          // There are nodes in a prog
PajeDefineContainerType T N "Thread"        // There are threads in a node
PajeDefineStateType S T "State"             // A thread has a State
PajeDefineLinkType COM P T T "Comm"        // Com. from Thread to Thread

// Value definition of previously defined entities
PajeDefineEntityValue TakWork S "ThreadWorking"
PajeDefineEntityValue TakPack S "PackingTakCom"
PajeDefineEntityValue TakRecv S "ReceivingTakCom"
PajeDefineEntityValue TakCom COM "TaktukCom"

```

Fig. 4. Definition of a hierarchy in a Pajé trace

Generation of tracing functions. From each event definition, Tunit generates a tracing function which can then be called from the user source code. The execution of a tracing function includes three phases: binding to a buffer and allocate another one if necessary; building an event record containing the parameters appearing in the events' definitions; if necessary, flushing the buffer to disk.

Generation of trace manipulation and conversion functions. Trace files need to be processed in order to be usable. Tunit generates functions allowing event records to be read (written) from (into) trace files or to be displayed under textual form. These functions are used in the implementation of more complex functionalities such as correcting the local time-stamps of the events to obtain globally coherent dates [10], sorting or filtering event trace files. As explained in §3.2, format conversion is often required. Tunit uses re-writing rules definitions to generate trace conversion functions. Figure 3 shows an example where these rules are used to transform monitoring data into the Pajé visualization tool format and produce the example of §4. The output format is described as a character string where escape characters (@ sign) indicate substitutions by the recorded values of parameters. The corresponding parameters are written after the string. A rule with two arrows means that two events will be produced in the converted trace.

4 Genericity of the Pajé Visualization Tool

The Pajé visualization tool can be specialized for a given programming model. The visualization to be constructed from the traces can be described by the user, provided that the types of the objects appearing in the visualization are hierarchically related and that this type hierarchy can be described as a tree

(see figures 4 and 6). The Pajé self-defined input data format [5] is based on two generic types: containers and entities. While entities can be considered as elementary visual objects, containers are complex objects, each including a (potentially high) number of entities. An entity type is further specified by one of the following generic types: event, state, link or variable. Each entity also has a value, which allows its identification among other entities of the same type. In the example of figure 4, three hierarchically related containers are defined, named **Prog**, **Node** and **Thread**. The **Thread** container is composed of entities of type **S** or **ThreadState** of the generic type “state” and will therefore be represented as rectangles showing the successive states of the thread. These states may have three possible values, **ThreadWorking** or **PackingTakCom** or **ReceivingTakCom**. Another entity type named **Comm** of type “link”, connecting threads, is defined and will be represented as an arrow with a source and destination.

Monitoring data, recorded during the execution of a small Taktuk program, using trace functions generated by Tunit from the event definitions of §3.3, was converted, using the rewrite rules presented in §3.3, in the Pajé data format defined above. A small excerpt of this Pajé file is given in figure 5: it exemplifies the creation of two threads and the sending of a message from one thread to the other. The identification of each thread is obtained by concatenation of a global node identification with a thread identification local to the node.

```

PajeCreateContainer 0.00 TP P 0 "Test Prog"           // Monitored program
PajeCreateContainer 0.00 N1 N TP "Node 1"             // Node 1 created
PajeCreateContainer 0.00 N2 N TP "Node 2"             // Node 2 created
PajeCreateContainer 0.12 T-1-1 T N1 "Thread-1-1"      // Creation of thread-1-1
PajeSetState 0.12 S T-1-1 TakWork                     // T-1-1 initial state
PajeCreateContainer 0.15 T-2-1 T N1 "Thread-2-1"     // Creation of thread-2-1
PajeSetState 0.15 S T-2-1 TakWork                     // T-2-1 initial state
PajePushState 0.25 S T-1-1 TakPack                   // T-1-1 packs message
PajeStartLink 0.42 COM TP TakCom T-1-1 key121         // T-1-1 sends message
PajePopState 0.42 S T-1-1                            // T-1-1 back to work
PajeEndLink 0.55 COM TP TakCom T-2-1 key121          // T-2-1 receives msg
PajePushState 0.55 S T-2-1 TakRecv                   // T-2-1 changes state
PajePopState 0.75 S T-2-1                            // T-2-1 back to work

```

Fig. 5. Excerpt of converted trace file

5 Related Work

VampirTrace and Vampir [12] allow users to obtain easily behavioral visualizations of parallel programs executions. However these tools are bound to a programming model such as MPI and cannot be easily adapted. Tau [14] is a very flexible and portable performance monitoring tool which records information using mainly the counting and timing techniques (see §2.1). The tracing facility which should be provided through the definition of “user events” was not yet

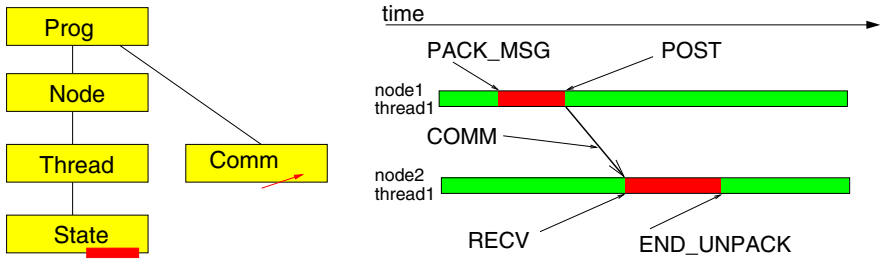


Fig. 6. Simple Pajé type hierarchy and visualization of Taktuk communication events

available in Tau at the time Tunit was designed. In the Paraver visualization and analysis tool [8], users can assign semantics — within a programming model paradigm — to the types of event records to indicate how trace file should be visualized.

6 Conclusion and Future Work

The Pajé approach for performance debugging is based on event tracing and behavioral visualizations of the observed executions. One of the objectives of Pajé is to face the multiplicity of parallel and distributed programming models. From a description of events to be traced, Tunit generates event recording functions and trace manipulation programs. The Pajé visualization tool can be adapted to various programming models. Future work includes experimenting with multi-level tracing and visualizations — mixing hardware counters, thread management, communications and high level parallel programming model — or adapting global clock implementation techniques to scale to hundreds of processing elements or more.

Acknowledgements. Jacques Briat suggested several of the proposed techniques. This work is partly supported by INRIA Rhône-Alpes via the LIPS project (joint project of INRIA and BULL).

References

1. B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of SuperComputing Applications and High Performance Computing*, 14(4):317–329, Winter 2000.
2. R. Chandra et al. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.
3. J. Chassin de Kergommeaux, B. de Oliveira Stein, and P. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, aug 2000.

4. J. Chassin de Kergommeaux, E. Maillet, and J.-M. Vincent. Monitoring parallel programs for performance tuning in cluster environments. In J. Cunha et al., editors, *Parallel Program Development for Cluster Computing*, volume 5, chapter 6, pages 131–150. Nova Science, 2001.
5. J. C. de Kergommeaux and B. de Oliveira Stein. Flexible performance visualization of parallel and distributed applications. *Future Generation Computer Systems*, 19(5):735–748, 2003.
6. B. de Oliveira Stein, J. Chassin de Kergommeaux, and G. Mounié. Pajé trace file format. Technical report, ID-IMAG, Grenoble, France, 2002. <http://www-id.imag.fr/Logiciels/paje/publications>.
7. L. DeRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *Procs of the International Conference on Parallel Processing (ICPP'99)*, pages 311–318, Fukushima, Japan, September 1999.
8. European Center for Parallelism of Barcelona, <http://www.cepba.upc.es/paraver>. *Paraver: Parallel Program Visualization and Analysis Tool*, 2001.
9. M. P. I. Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.
10. É. Maillet and C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 28:84–93, 1995.
11. C. Martin and O. Richard. Parallel launcher for cluster of PC. In *Proceedings of ParCo 2001*. Imperial College Press, London, 2001.
12. W. Nagel et al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
13. A. Pope. *The CORBA reference guide*. Addison Wesley, 1998.
14. S. Shende, A. D. Malony, et al. Portable profiling and tracing for parallel scientific applications using C++. In *Proc. of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, Aug. 1998.
15. Sun Microsystems Inc., <http://java.sun.com/docs/books/tutorial/>. *The Java(tm) Tutorial*, 2003.

EventSpace – Exposing and Observing Communication Behavior of Parallel Cluster Applications^{*}

Lars Ailo Bongo, Otto J. Anshus, and John Markus Bjørndalen

Department of Computer Science,
University of Tromsø
{larsab, otto, johnm}@cs.uit.no

Abstract. This paper describes the motivation, design and performance of EventSpace, a configurable data collecting, management and observation system used for monitoring low-level synchronization and communication behavior of parallel applications on clusters and multi-clusters. *Event collectors* detect events, create *virtual events* by recording timestamped data about the events, and then store the virtual events to a *virtual event space*. *Event scopes* provide different *views* of the application, by combining and pre-processing the extracted virtual events. Online monitors are implemented as consumers using one or more event scopes. Event collectors, event scopes, and the virtual event space can be configured and mapped to the available resources to improve monitoring performance or reduce perturbation. Experiments demonstrate that a wind-tunnel application instrumented with event collectors, has insignificant slowdown due to data collection, and that monitors can re-configure event scopes to trade-off between monitoring performance and perturbation.

1 Introduction

As the complexity and problem size of parallel applications and the number of nodes in clusters increase, communication performance becomes increasingly important. Of eight scalable scientific applications investigated in [12], most would benefit from improvements to MPI's collective operations, and half would benefit from improvements in point-to-point message overhead and reduced latency.

The performance of collective operations has been shown to improve by a factor of 1.98 by using better mappings of computation and data to the clusters [2]. Point-to-point communication performance can also be improved by tuning configurations. In order to tune the performance of collective and point-to-point communication, fine-grained information about the applications communication events is needed to compute how the application behaves.

^{*} This research was supported in part by the Norwegian Science Foundation project “NOTUR”, sub-project “Emerging Technologies – Cluster”

In this paper we describe EventSpace, an approach and a system for online monitoring the communication behavior of parallel applications. It is the first step toward a system that can dynamically reconfigure an applications communication structure and behavior.

For low-level performance analysis [3,11] and prediction [14,5], large amounts of data may be needed. For some purposes the data must be consumed at a high rate [9]. When the data is used to improve the performance of an application at run-time, low perturbation is important [8,10]. To meet the needs of different monitors the system should be flexible, and extensible. Also the sample rate, latency, perturbation, and resource usage should be configurable [8,5,11]. Finally, the complexity of specifying the properties of such a system must be handled.

The approach is to have a *virtual event space*, that contains traces of an applications communication (including communication used for synchronization). *Event scopes* are used by consumers to extract and combine virtual events from the virtual event space, providing different *views* of an applications behavior. The output from event scopes can be used in applications and tools for adaptive applications, resource performance predictors, and run-time performance analysis and visualization tools. When triggered by communication events, *event collectors* create a virtual event, and store it in the virtual event space. A virtual event comprises timestamped data about the event.

The EventSpace system is designed to scale with regards to the number of nodes monitored, the amount of data collected, the data observing rate, and the introduced perturbation. Complexity is handled by separating instrumentation, configuration, data collection, data storage, and data observing.

The prototype implementation of the EventSpace system is based on the PATHS [1] system. PATHS allows configuring and mapping of an applications *communication paths* to the available resources. PATHS use the concept of *wrappers* to add code along the communication paths, allowing for various kinds of processing of the data along the paths. PATHS use the PastSet [13] distributed shared memory system. In PastSet *tuples* are read from and written to named *elements*.

In EventSpace, an application is instrumented when the configurable communication paths are specified. Event collectors are implemented as PATHS wrappers integrated in the communication system. They are triggered by PastSet operations invoked through the wrapper. The virtual event space is implemented by using PastSet. There is one *trace element* per event collector.

Event scopes are implemented using scalable hierarchical *gather trees*, used to gather data from a set of trace elements. PATHS wrappers to filter, sort, convert, or reduce data can be added to the tree as needed. To trade-off between performance and perturbation the tree can be mapped to the available resources, and wrapper properties can be set.

This paper proceeds as follows. In section 2 we discuss related work. The architecture and implementation of EventSpace are described in sections 3 and 4. Monitors using EventSpace are described in section 5. The performance of

EventSpace is evaluated in section 6. In section 7 we draw conclusions and outline future work.

2 Related Work

There are several performance analysis tools for parallel programs [6]. Generally such tools provide coarse grained analysis with focus on processor utilization [11]. EventSpace supplements these tools by providing detailed information about the internal behavior of the communication system.

NetLogger [11] provides detailed end-to-end application and system level monitoring of high performance distributed systems. Analysis is based on life-lines describing the temporal trace of an object through the distributed system. EventSpace provides data for similar paths inside the communication system. However, data is also provided for joined and forked paths forming trees, used to implement collective operations and barriers.

There are several network performance monitoring tools [5,7]. While these often monitor low level network data, EventSpace is used by monitors monitoring paths that are used to implement point-to-point and collective communication operations. Such a path may in addition to a TCP connection, have code to process the data, synchronization code, and buffering.

JAMM [10] is a monitoring sensor management system for Grid environments. Focus is on automating the execution of monitoring sensors and the collection of data. In JAMM sensors generate events that can be collected, filtered and summarized by consumers using event gateways. In EventSpace, events are generated by sensors integrated in the communication system, and consumers collect, filter, and preprocess data using event scopes. Event scopes are implemented by extending the PATHS system, allowing the data flow to be configured at a fine granularity.

The Network Weather Service [14] is a system service producing short-term forecast for a set of network and computational resources. In EventSpace monitoring is per application, hence data is only collected for the resources used by the application simplifying the resource usage control.

EventSpace, and many other monitoring systems [5,8,11,14] separate producers and consumers, and allow intermediates to filter, broadcast, cache, and forward data. Most monitoring systems also have a degree of configurability [5, 8,11,14]. In EventSpace focus is on allowing all parts of the monitoring to be configured and tuned at a fine granularity.

3 EventSpace Architecture

The architecture of the EventSpace system is given in figure 1. An application is instrumented by inserting *event collectors* into its communication paths. Each event collector record data about communication events, creates a virtual event based on the data, and stores it in a virtual event space. Different *views* of the

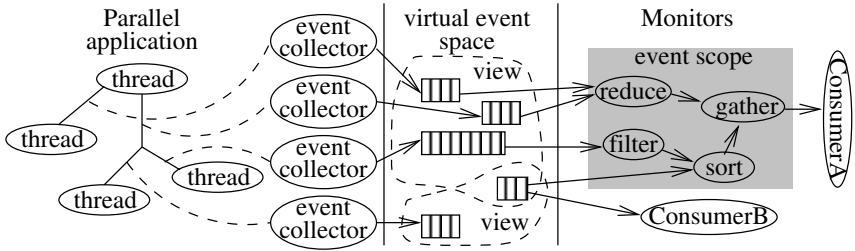


Fig. 1. EventSpace overview.

communication behavior can be provided by extracting and combining virtual events provided by different event collectors. Consumers use an *event scope* to do this.

An event collector record operation type, operation parameters, and start and completion times¹ of all operations sent through it. Typically, several event collectors are placed on a path to collect data at multiple points.

EventSpace is designed to let event collectors create and store events, with low overhead introduced to the monitored communication operations. Shared resources used to extract and combine virtual events are not used until the data is actually needed by consumers. We call this *lazy event processing*. By using lazy processing we can, without heavy performance penalties, collect more data than may actually be needed. This is important because we do not know the actual needs of the consumers, and we expect the number of writes to be much larger than the number of reads [9].

EventSpace is designed to be extensible and flexible. The event collectors and event scopes can be configured and tuned to trade off between introduced perturbation and data gathering performance. It is also possible to extend EventSpace by adding other event collectors, and event scopes.

The communication paths used by event collectors and event scopes can also be instrumented and monitored. Consequently, a consumer can monitor its own, another consumer's, or an event collectors performance and overhead.

4 EventSpace Implementation

The implementation of EventSpace is built on top of PATHS and PastSet. Presently, the monitored applications must also use PATHS and PastSet.

PastSet is a structured distributed shared memory system in the tradition of Linda [4]. A PastSet system comprises a number of user-level *tuple servers* hosting PastSet *elements*. An element is a sequence of tuples of the same type. Tuples can be read from and written to the element using blocking operations.

PATHS supports mapping of threads to processes, processes to hosts, specifying and setting up physical communication paths to individual PastSet elements,

¹ Using the Pentium timestamp counter.

and insertion of code in the communication paths. This code is executed every time the path is used.

A path is specified by listing the stages from a thread to a PastSet element. At each stage the wrapper type and parameters used to initialize an actual instance of the wrapper are specified. A wrapper is typically used to run code before and after forwarding the PastSet operation to the next stage in the path.

Paths can be joined or forked forming a tree structure. This supports implementation of collective operations, barriers and EventSpace gather trees.

The threads, elements, and all communication paths used by an application are specified in a *pathmap*. It also contains information about which paths are instrumented, and the properties of event collectors and trace elements. The path specifications are generated by path generate functions. As input to these functions three maps are used: (1) An application map describing which threads access which elements. (2) A cluster map, describing the topology and the nodes on each cluster. (3) An application to cluster map, that describes the mapping of threads and elements to the nodes.

Threads can use elements in an access and location transparent manner, allowing the communication to be mapped onto arbitrary cluster configurations simply by reconfiguring the pathmap. Presently, run-time reconfiguration of the pathmap is not implemented.

4.1 Event Collectors

When triggered, an event collector creates a virtual event in the form of a *trace tuple*, and writes it to a *trace element*. A virtual event space is implemented by a number of trace elements in PastSet. Each trace element can have a different size, lifetime, and be stored in servers locally or remotely from where the communication event took place.

The trace tuple is written to a trace element using a blocking PastSet operation. As a result it is important to keep the introduced overhead low. If the trace element is located in the same process as the event collector (a local server), the write only involves a memory copy and some synchronization code. To keep the introduced overhead low, trace tuples are usually stored locally.

Tuples can be removed either by explicit calls, or automatically discarded when the number of tuples is above a specified threshold (specified on a per element basis). Presently, for persistent storage some kind of *archive consumers* are needed.

The recorded data is stored in a 36 byte tuple. Since write performance is important, tuples are stored in binary format, using native byte ordering. For heterogeneous environments, the tuple content can be parsed to a common format when it is observed.

By using PATHS to specify the path from an event collector wrapper to a trace element, we can specify the location of the trace element, its size, and its properties.

Presently, the amount of tracing cannot dynamically be adjusted as in [8,11]. However, when a consumer decides to start monitoring a part of the system, a backlog of collected events can be examined.

4.2 Event Scopes

An event scope is used to gather and combine virtual events providing a specific *view* of an applications communication behavior. It can also do some pre-processing on the virtual events. An event scope is implemented using a configurable *gather tree*. The tree is built using PATHS wrappers. Even if a tree can involve many trace elements with wrappers added at arbitrary places, the complexity of building and configuring it is reduced due to the hierarchical nature of views, resulting in a regular structure of the tree. For example, the Heartbeat view in figure 2a, comprises two node views, each comprising two thread views. The tree can be built hierarchically by creating similar sub-trees for each sub-view. Any number of event scopes can be dynamically built by different consumers using the applications pathmap as input.

The desired performance and perturbation of a gather tree are achieved by mapping the tree to available resources and setting properties of the wrappers. Data can be reduced or filtered close to the source, to avoid sending all data over a shared resource such as Ethernet, or a slow Internet link. Also some data preprocessing can be done on the compute clusters, thereby reducing the load on the node where the consumer is running. Since data is gathered on-demand, shared resources such as CPU, memory and networks are only used when needed.

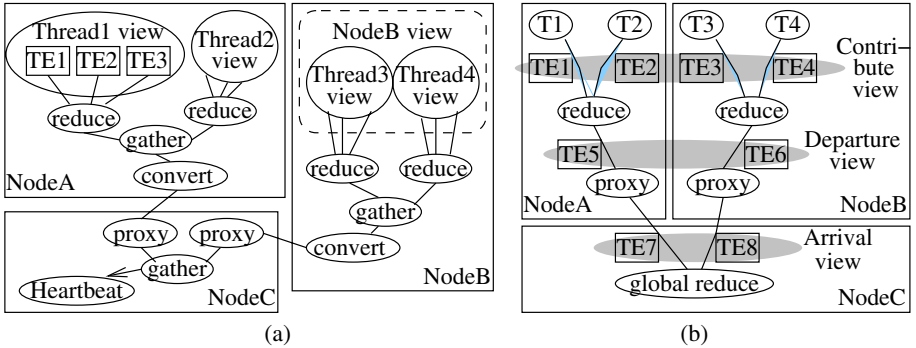


Fig. 2. (a) Heartbeat gather tree. (b) Data collected for a reduce operation tree.

5 Monitors – Consumers of Virtual Events

In this section we describe several monitors that use event scopes to extract and process virtual events.

5.1 Heartbeat Monitor

The Heartbeat monitor has the task of establishing whether a thread has reached a stable state where no further progress can take place or not. The monitor uses an event scope as shown in figure 2a, to find the last virtual event for each thread of the application. The event scope extract the latest virtual event for every communication path used by a thread, and then selects the last of these events (“reduce”). The output (“gather”) from the event scope is one single timestamp for each thread representing the last event happening in each thread.

After a set time the monitor use the event scope to reduce and gather new timestamps. These are compared with the previous values. If no change is detected the thread has had no progress.

5.2 Get Event Monitor

GetEventMon just consumes virtual events without further processing of the data. In [3] the data collected by GetEventMon is used for performance analysis and visualization of the behavior of an application.

GetEventMon uses an event scope with a gather tree similar to Heartbeat’s, except that all reduce wrappers are replaced with gather wrappers.

5.3 Collective Operation Monitor

ColOpMon monitors the performance of MPI type collective operation, implemented and instrumented using PATHS. As these operations use broadcast, gather, scatter, and reduce trees, we collect data about the activity in the trees for later extraction from the virtual event space, and analysis by ColOpMon.

ColOpMon is a multi-threaded distributed application, with its own pathmap. It has threads monitoring the performance of each internal tree node, and one thread monitoring the contribution times of threads participating in the collective operation.

In figure 2b the different views for (a part of) a reduce operation tree are shown, where four threads, T1 - T4 on nodes A and B, do a partial reduce on each node before the global reduce on node C. The path is instrumented using event collectors that store events in trace elements TE1 - TE8. An internal node monitor thread gathers, does clock synchronization, and analyzes events from the departure and arrival views. A contribution time monitor gathers and analyzes events from the contribute view.

6 Experiments

To demonstrate the feasibility of the approach and the performance of the EventSpace prototype, we monitor a wind-tunnel application.

The hardware platform comprise two clusters:

4W: Eight four-way Pentium Pro 166 MHz, 128 MB RAM.

8W: Four eight-way Pentium Pro 200 MHz, 2GB RAM.

The clusters use TCP/IP over a 100 Mbps Ethernet for intra and inter-cluster communication. Communication to and from the 4W cluster goes through a two-way Pentium II 300 MHz with 256 MB RAM, while the 8W nodes are directly accessible.

The wind-tunnel is a Lattice Gas Automaton doing particle simulation. We use eight $2N \times N$ matrices. Each matrix is split into 140 slices, which are assigned to 140 threads. Each thread uses 12 PastSet elements to exchange the border rows of its slices with threads computing on neighboring slices. We use three different problem sizes: *large*, *medium* and *small*.

Large is the largest problem size that does not trigger paging on the 4W nodes, while *medium* and *small* are 1/2 and 1/8 of the size of *large*².

For the large problem size, border rows are exchanged approximately every 300 ms (and thus virtual events are produced at about 3.3 Hz). For *medium* and *small*, rows are exchanged every 70 ms and every 5 ms respectively. The discard threshold is set to 200 tuples for all 2240 trace elements. Less than 0.5 MB of memory on a 4W node is used for trace tuples.

6.1 Event Collecting Overhead

The overhead introduced to the communication path by a single event collector wrapper is measured by adding event collector wrappers before and after it. The average overhead, calculated using recorded timestamps in these wrappers, is 1.1 μ s on a 1.8 GHz Pentium 4 and 6.1 μ s on a 200 MHz 8W node. This is comparable to overheads reported for similar systems [8,11].

For *large* and *medium*, the slowdown due to data collection is insignificant. For *small*, the slowdown is 1.03.

Further experiments are needed to determine if the insignificant and small slowdowns are due to unused cluster resources, and the effect of the overhead introduced by event collectors on other applications.

6.2 Event Scope Perturbation and Performance

In this section we document how reconfiguring an event scope can be used to trade-off between the rate at which virtual events can be observed for a given view, and the perturbation of monitored application. We measure how many times a monitor can pull, using an event scope, one tuple from all trace elements in a view during a run of the wind-tunnel. All monitors are run on a computer outside the cluster³, if not otherwise stated.

GetEventMon consumes events from a single trace element, a thread view with 14 elements, and a node view with 266 elements (19 thread views) with no slowdown for *large*, and 1.03 for *small*. Events are consumed at 2500 Hz, 1500 Hz, and 200 Hz respectively.

² As the size of the matrices, N , increase the computation increase by $O(N^2)$, and the communication by N . For *large*, N is 7680.

³ A 1.8 GHz Pentium 4 with 2 GB RAM.

When consuming events from a multi-cluster view with 1948 trace elements (140 thread views), the wind-tunnel has no slowdown for *large*. However when using the small problem size, the event scope introduces a slowdown of 1.36. Using a different gather tree shape increases the observe rate from 58 Hz to 106 Hz, and the slowdown to 1.50. Reconfiguring the event scope to use less threads, reduces the slowdown to 1.07, and the observe rate to 25 Hz.

When consuming from another multi-cluster view for *large*, with only two trace elements per thread, there is no difference in sample rates and slowdown when consuming events sequentially and concurrently. However, when more processing are added to the communication paths the concurrent version is faster, due to better overlap of communication and computation.

The event scopes used by the collective operation monitor, ColOpMon, results in a slowdown of 1.17. We discovered that an event scope actually perturbs the wind-tunnel more than the computation intensive internal node monitoring threads running on the clusters. By reconfiguring the event scope to use less resources the slowdown is reduced to 1.08 (the observe rates are also decreased by about 50%).

When running four monitors concurrently, the slowdown is about the same as the largest slowdown caused by a single monitor (ColOpMon). For the consumers running on the computer outside the cluster, observe rates are reduced by 10-50%. For the ColOpMon internal monitor threads running on the clusters, observe rates are unchanged.

7 Conclusions and Future Work

The contributions of this work are two-fold: (i) we describe the architecture and design of a tunable, and configurable framework for low-level communication monitoring, and (ii) we demonstrate its feasibility and performance.

The approach is to have event collectors integrated in the communication system, that when triggered by communication events, create a virtual event that contains timestamped information about the event. The virtual events are then stored in a virtual event space from where they can be extracted by consumers using different event scopes.

Low-level communication monitoring is implemented by adding event collection code to communication paths. The data is stored in a structured shared memory. The data is extracted, combined and pre-processed using configurable communication paths. The architecture and its implementation, allows consumers, producers, and data management issues to be clearly separated. This makes handling the complexity simpler.

We have described several monitors using EventSpace, and given initial performance results. The results show that a wind-tunnel application instrumented with event collectors, has insignificant slowdown due to data collection. We have also documented how event scopes can be reconfigured to trade-off between monitoring performance and perturbation.

Currently, we are using other types of applications and monitors to evaluate the performance and configurability of the system. We are also using EventSpace to analyze and predict the performance of MPI type collective operations, with the purpose of dynamically adapting these for better performance.

References

1. BJØRNDALEN, J. M., ANSHUS, O., LARSEN, T., AND VINTER, B. Paths - integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed application. *Norsk Informatikk Konferanse* (2001).
2. BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Configurable collective communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (2002).
3. BONGO, L. A., ANSHUS, O., AND BJØRNDALEN, J. M. Using a virtual event space to understand parallel application communication behavior, 2003. Technical Report 2003-44, Department of Computer Science, University of Tromsø.
4. CARRIERO, N., AND GELERTER, D. Linda in context. *Commun. ACM* 32, 4 (April 1989).
5. DINDA, P., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND SUTHERLAND, D. The architecture of the Remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing* (2001).
6. MOORE, S., D.CRONK, LONDON, K., AND J.DONGARRA. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Y. Cotronis and J. Dongarra, Eds., Springer Verlag.
7. <http://www.caida.org/tools/taxonomy/>.
8. RIBLER, R. L., VETTER, J. S., SIMITCI, H., AND REED, D. A. Autopilot: Adaptive control of distributed applications. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing* (1998).
9. TIERNEY, B., AYDT, R., GUNTER, D., SMITH, W., TAYLOR, V., WOLSKI, R., AND SWANY, M. A grid monitoring architecture. *Tech. Rep. GWD-PERF-16-2, Global Grid Forum, January 2002*. (2002).
10. TIERNEY, B., CROWLEY, B., GUNTER, D., HOLDING, M., LEE, J., AND THOMPSON, M. A monitoring sensor management system for Grid environments. In *Proc. 9th IEEE Symp. On High Performance Distributed Computing* (2000).
11. TIERNEY, B., JOHNSTON, W. E., CROWLEY, B., HOO, G., BROOKS, C., AND GUNTER, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing* (1998).
12. VETTER, J. S., AND YOO, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002).
13. VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, University of Tromsø, 1999.
14. WOLSKI, R., SPRING, N. T., AND HAYES, J. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 15, 5-6 (1999).

A Race Detection Mechanism Embedded in a Conceptual Model for the Debugging of Message-Passing Distributed Programs*

Ana Paula Cláudio and João Duarte Cunha

¹Faculdade de Ciências da Universidade de Lisboa
Departamento de Informática, Campo Grande, Edifício C5, Piso 1, 1700 LISBOA Portugal
apc@di.fc.ul.pt

²Laboratório Nacional de Engenharia Civil
Av. do Brasil, 101, 1799 LISBOA CODEX, Portugal
jdc@lnec.pt

Abstract. An object-oriented conceptual model for the debugging of message-passing distributed programs incorporates several debugging facilities: generation of a space-time diagram showing the progression of the execution being studied, detection of race conditions, detection of particular kinds of predicates and representation of causality cones. The focus of this paper is the race detection mechanism. The proposed mechanism comprises two steps: detection of pairs of receive events in the same process potentially involved in race conditions and verification of the legitimacy of the potential race condition. The mechanism relies on the analysis of two arguments, *process id* and *message tag*, in receive events and consume events, which are considered as distinct types of occurrences in the conceptual model.

1 Introduction

This paper describes a race detection mechanism embedded in a conceptual model for message-passing distributed programs. This model was conceived according to the object-oriented methodology and incorporates several debugging facilities.

Due to race conditions the internal work of a distributed program may be nondeterministic, that is, two successive executions of the program with the same input behave differently. Therefore, a bug observed in one execution may vanish in a later one and a mechanism for detecting these conditions is of utmost importance.

However, it should be stressed that the programmer may intentionally introduce race conditions in order to obtain improved performances. So, only the programmer is able to distinguish between intentional and non-intentional races.

The conceptual model is the core of MPVisualizer, a debugging tool for message-passing programs that is capable of operating either in reexecution mode or in *post-mortem* mode. Besides the race detection mechanism, the conceptual model incorporates the following debugging facilities which are available in the tool:

* This work has been partially funded by FCT(Portugal) under Project POSI/39351/SRI/2001 Mago2.

- Generation of a space-time diagram showing the progression of the execution being studied.
- Detection of local predicates as well as disjunction and conjunction of local predicates. Local predicates are defined by the user and, as explained in [1], a specific class must be rewritten in order to prepare the tool for the detection of each local predicate. Once the local predicates are defined no further programming is needed for the detection of disjunctive and conjunctive predicates.
- Representation of causality cones. The tool has the capability of adding to the space-time diagram a representation of the causality cone [2] or causal history [3] of an event previously selected by the user. The causality cone of an event contains all the events in the program, which can possibly have influenced the event. Conjunctive and disjunctive predicates are evaluated in the frontier of a causality cone. The frontier of a causality cone includes the last event belonging to the cone in each process.

Detection of disjunctive and conjunctive predicates as well as the representation of causality cones did not exist in the previous version of MPVisualizer [1]. Also, the race detection mechanism has undergone a major improvement in the current version.

In the next section a brief description of MPVisualizer is given. The race detection mechanism is thoroughly explained in section three. Section four is dedicated to the conclusions.

2 MPVisualizer

As mentioned above, the core of MPVisualizer is the conceptual model. Conceived according to the object-oriented methodology, this model comprises two groups of classes: kernel classes and graphical classes. Classes in the first group are independent from the message-passing software. Classes in the second group are subclasses of the classes in the first group and encapsulate all the programming details that depend on the graphical software.

MPVisualizer builds, either in reexecution or in *post-mortem* mode, a concrete model of the execution being studied. This model is an instance of the conceptual model and is called the functional model.

MPVisualizer has three components: the visualization engine, the reexecution mechanism and the graphical interface.

The visualization engine contains the functional model, the entity responsible for its management, called *manager*, and a set of processes with special tasks, the *collectors* and the *central process*. *Collectors*, one per machine, receive data from local processes in the execution being studied and forward them to the *central process*, which is in charge of sending to the manager all the information, needed to build the functional model. The code executed by *collectors* depends on the message-passing software used by the program being studied; everything else in the visualization engine is independent of it.

The reexecution mechanism is based on *Instant Replay* [4], a well known mechanism for shared memory communication that we have adapted for message-passing communication.

The reexecution mechanism includes two phases: trace and replay. In the trace phase, minimal information is stored to minimize the inevitable intrusion caused by

tracing activities. Although minimal, the stored information is sufficient to ensure that, during the replay phase, each process will consume the same messages, in the same order.

The monitoring code has been inserted in the standard libraries of the message-passing software. Program code is kept untouched but linking is different in the trace and replay phases. In the trace phase, the program code must be linked with one modified library, the trace library; for the replay phase, it must be linked with a second modified library, the replay library. The routines in this last library force the re-execution to respect the causal order of the communication events recorded during the original execution. They also force the processes to send blocks of information to the local *collector* process.

The graphical interface is composed of a menu bar and a drawing area where the space-time diagram is shown. Space-time diagrams are built in a bidimensional orthogonal referential: one of the axis corresponds to the continuous variable time and the other axis to the set of all the processes in the execution which is a discrete variable. Each process is represented by a line-segment parallel to the time axis and growing accordingly to the evolution of the process. Every kind of communication event has its own graphical symbol drawn over the line-segment of the corresponding process in order to signal its occurrence. An oblique line-segment connects each consume event to the corresponding send event. The diagram is consistent with the *happened-before* relation defined by Lamport [5].

By appropriate manipulation of the graphical symbols in the diagram, the user can profit from all the facilities offered by MPVisualizer.

3 Race Detection Mechanism

The race detection mechanism comprises two steps: detection of pairs of receive events in the same process potentially involved in race conditions and verification of the legitimacy of the potential race condition.

The original mechanism [1] included also the same two steps. However, the first step has been improved in order to enable the identification of a broader spectrum of race conditions.

The mechanism, embedded in the conceptual model, performs the detection of race conditions while the construction of the functional model is taking place.

The description of the mechanism is divided in three subsections: architecture, implementation and mechanism flaws.

3.1 Architecture

According to Netzer and Miller [6]: “Two messages race if either *could have* been accepted first by some receive, due to variations in message latencies or process scheduling”. Therefore, the entities involved in a race condition are: a couple of messages sent to the same process and at least one receive event that consumes the message arrived in the first place. Sometimes, there is a second receive event involved, which may or may not consume the message arrived in the second place.

The proposed mechanism relies on the analysis of two arguments, *process id* and *message tag*, in receive and consume events. In the conceptual model, receive and consume events are considered as distinct types of occurrences:

- A receive event occurs when a process intends to consume a message; it can be either blocking or non-blocking. A *process id* and a *message tag* are specified; both values are integers and one or both can have the value “-1” which means “any”. We assume that this value is strictly reserved for this meaning.
- A consume event occurs when a process consumes a message. An event of this kind is always preceded by a receive event. It contains the tag of the message and its sender id. Both values are different from “-1”.

A consume event is always preceded by a receive event but, it is possible that a receive event is not followed by a consume event. Therefore, for any pair of receive events, four distinct cases can be identified:

- RCRC: both receive events, either blocking or non-blocking, are followed by consume events.
- RCR-: the first receive event, either blocking or non-blocking, is followed by a consume event. The same does not happen with the second receive event which is not immediately followed by a consume event. If this receive event is a blocking one the execution of the process remains suspended.
- R-RC: the first receive event is not followed by a consume event while the second one is. This case is not possible when the first receive is blocking.
- R-R-: none of the two receive events is followed by a consume event. This case is not possible when the first receive is blocking. If the first receive event is non-blocking and the second receive event is blocking, the execution of the process is suspended.

Potential race conditions are considered as non-legitimate, *i.e.* spurious, when one of the following situations holds:

- Both messages have the same origin. Assuming that communication channels are FIFO, two messages sent by the same process to an equal destination can not be racing with one another.
- Consumption of the first message “happens before” the sending of the second one, in the sense introduced by Lamport [5]. These situations are called *false race condition* [6].

The race detection mechanism compares the pair of values (*process id*, *message tag*) of a receive event with the equivalent pair of a later receive event in the same process. This comparison deals with $16=2^4$ different combinations of values. Table 1 details all the sixteen different situations. For referencing purposes, each situation is identified in the first column by an integer ranging from 1 to 16.

The second and third columns correspond to the arguments *process id* and *message tag* of the first receive event; the fourth and fifth columns correspond to the same arguments of the second receive event.

The conceptual model assumes that every message carries its origin and its tag, thereafter called *message.pid* and *message.tag*, respectively. Every consume event contains the pair of values that corresponds to the consumed message. These values are useful in both detection steps; however they are essential for the second one, when the legitimacy of potential races is verified.

Table 1. Sixteen combinations of pairs (*process id*, *message tag*) in two receive events.

n°	recv1		recv2		Comments
1	pid1	tag1	pid2	tag2	There is no race condition
2	pid1	-1	pid2	tag2	There is no race condition
3	-1	tag1	pid2	tag2	There is a potential race condition if tag1=tag2
4	-1	-1	pid2	tag2	There is a potential race condition
5	pid1	tag1	pid2	-1	There is no race condition
6	pid1	-1	pid2	-1	There is no race condition
7	-1	tag1	pid2	-1	There is a potential race condition if both messages are consumed and both have a tag equal to tag1 or if the message consumed in the first place has a <i>process id</i> equal to pid2
8	-1	-1	pid2	-1	There is a potential race condition
9	pid1	tag1	-1	tag2	There is no race condition
10	pid1	-1	-1	tag2	There is no race condition
11	-1	tag1	-1	tag2	There is a potential race condition if tag1=tag2
12	-1	-1	-1	tag2	There is a potential race condition
13	pid1	tag1	-1	-1	There is no race condition
14	pid1	-1	-1	-1	There is no race condition
15	-1	tag1	-1	-1	There is a potential race condition if both messages have been consumed and if they have equal tags
16	-1	-1	-1	-1	There is a potential race condition

In the set of situations listed on table 1 four subsets can be identified:

- Subset 1: situations 1, 2, 5, 6, 9, 10, 13 and 14. In these situations there are no race conditions because the message consumed in the first place has to be the one sent by the process specified in the corresponding receive event parameter.
- Subset 2: situations 4, 8, 12 and 16. These situations correspond to potential race conditions. Since the first receive event does not specify either the *process id* of the origin or the tag of the expected message, the message consumed first is the one that arrived first. The second message may or may not be consumed by the destination process.
- Subset 3: situations 3 and 11. In these situations there are potential race conditions if message tags are the same in both receive events.
- Subset 4: situations 7 and 15. In these situations there are potential race conditions if both messages have been consumed and both verify *message.tag* = *tag1*. Additionally, in situation number 7, a potential race condition also occurs if the message consumed in the first place verifies *message.pid* = *pid2*.

3.2 Implementation

The following piece of pseudo-code explains the implementation of the first step in the mechanism. In the code, *potential_race* denotes a boolean variable, *recv1.tag* and *cons1.pid* correspond, respectively, to the arguments *process id* and *message tag* of the first receive event, $\exists \text{cons1}$ denotes a boolean expression that is true when the consumption of the first message has taken place; expressions *recv2.tag*, *cons2.pid* and $\exists \text{cons2}$ have similar meanings.

```

1  potential_race = false
2  if (recv1.pid = -1)
    { the condition is true in situations 3, 4, 7, 8, 11, 12, 15 and 16 }
3  then if (recv1.tag = -1)
    { the condition is true in situations 4, 8, 12 and 16 }
4      then potential_race = true
5      else { recv1.tag ≠ -1 }
6      if (recv1.tag = recv2.tag) { the condition is true in 3 and 11 }
7      then potential_race = true
8      else { recv1.tag ≠ recv2.tag }
9      if (recv2.tag = -1) { the condition is true in 7 and 15 }
10     then if (∃cons1)
11         then if (cons1.pid = recv2.pid) { false in 15 }
12         then potential_race = true
13         else { cons1.pid ≠ recv2.pid }
14         if (∃cons2)
15             then if (cons1.tag = cons2.tag)
16                 then potential_race = true

```

Condition $\text{pid1} \neq -1$ is sufficient to guarantee that we are not dealing with a pair of receive events involved in a race condition. This condition excludes situations 1, 2, 5, 6, 9, 10, 13 and 14.

The condition on line 3 is evaluated for the remaining eight situations. It is true in situations 4, 8, 12 and 16, whose first receive events do not specify either the process id, or the tag of the expected message. These situations are signaled as potential race conditions, whether or not any of the corresponding consumptions have occurred.

The condition on line 6 is evaluated for situations 3, 7, 11 and 15. It is false for situations 7 and 15 and it can be true or false in situations 3 and 11. So, conditions 3 and 11 that satisfy the condition on line 6 are classified as potential race conditions.

Condition $\text{recv2.tag} = -1$, on line 9, is false for situations 3 and 11. This fact excludes these situations from the subsequent tests. The condition is true for situations 7 and 15.

Condition $\exists \text{cons1}$, on line 10, is true for situations 7 and 15 corresponding to cases RCR- or RCRC. Within this group, situations satisfying the condition $\text{cons1.pid} = \text{recv2.pid}$ on line 11 are classified as potential race conditions. This condition is false for situation 15 and true for situations 7 similar to the one illustrated in figure 1, or when two messages with the same origin have been consumed. Both cases are marked as potential race conditions, but the last one is discarded during the verification process since it corresponds to a spurious race.

Condition $\exists \text{cons2}$, on line 14, is true for all RCRC cases in situation 15 and for situation 7 when condition $\text{cons1.pid} = \text{recv2.pid}$ does not hold. In all these situations, there is a potential race condition when the consumed messages have the same tag, that is, when condition $\text{cons1.tag} = \text{cons2.tag}$, on line 15, holds.

A brief explanation of how this code has been incorporated in MPVvisualizer follows. The mechanism is embedded in specific methods of class *Process*, the kernel class modeling any individual process in the execution. In situations 4, 8, 12 and 16, the detection of potential race conditions takes place when the first receive event is notified; in situations 3, 11 and some situations of type 7, the detection takes place

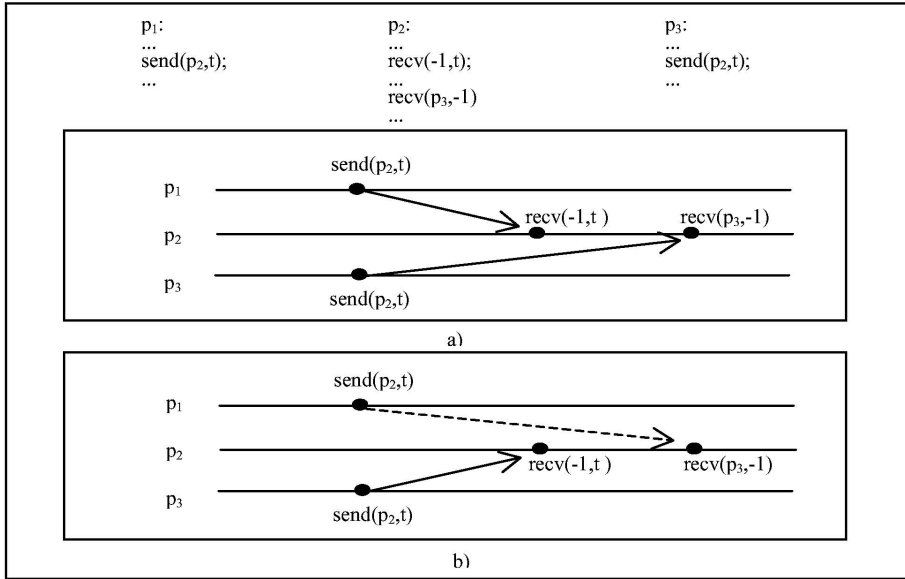


Fig. 1. Situation 7 of table 1. Dashed arrows correspond to messages that were not consumed.

when the second receive event is notified; in situations 15 as well as a different set of situations of type 7, the detection occurs when the second consume is notified. Therefore, detection code is embedded in methods `b_recv`, `nb_recv` and `m_consumed`, which are invoked respectively when a blocking receive, a non-blocking receive and a consume event occurs.

Whenever any of these methods detects a potential race condition, it invokes a private method of *Process* called `notify_potential_race`, which then verifies if the condition is a legitimate race.

Two receive events in the same process are classified as *adjacent* when no receive event occurred between them. In the current version of MPVisualizer, for efficiency reasons, detection of adjacent race conditions is performed automatically while detection of non-adjacent race conditions is performed only on demand, when the user selects the graphical symbol of a `recv(-1, -1)` event.

3.3 Mechanism Flaws

The mechanism is embedded in a conceptual model and therefore limited by the amount of information contained in the model. Since the model does not contain information about messages that arrived at a process but were not consumed, legitimization is not always feasible and it is possible to identify race conditions that are not detectable by the mechanism.

The mechanism is not able to legitimate potential races, namely when the second consumption does not occur, either because the receiver process did not execute the second receive event that could lead to the consumption of the late message, or because it did execute the second receive but the late message could not be consumed since its characteristics did not match the arguments of that receive event.

However, even when legitimization is not feasible, the tool signals the occurrence of a potential race condition.

Potential race conditions not detectable by the tool are those that involve messages that were not actually consumed. For instance in a situation of type 7, similar to the one illustrated in figure 1a) but without message consumptions because both receive events are non blocking and both messages arrived too late to be consumed.

4 Conclusions

A race detection mechanism is a valuable debugging facility. Nevertheless, it is relatively unusual to have this facility available in debugging tools. Out of 12 debugging tools compared with MPVVisualizer, only 4 claim to detect race conditions.

The tools compared with MPVVisualizer were: Atemp (the debugging tool of MAD) [7]; DDBG (a debugging tool previously used by GRADE) [8]; DETOP (the debugging tool of TOOL-SET) [9]; DIWIDE (the debugging tool of WINPAR and P-GRADE) [10]; p2d2 [11]; Panorama [12]; the debugging tool of Parascopes [13]; PDBG (the debugging tool of DAMS) [14]; POET [15]; TotalView [16]; Xmdb [17] and XPVM [18].

Among these tools and environments only MAD, POET, XMDB, and the debugging tool of Parascopes are able to detect race conditions.

The debugging tool of Parascopes detects data races during execution and the compiler in the same environment is able to signal potential races.

MAD is able to detect race conditions based on the analysis of trace files produced by the trace/replay mechanism.

In POET the detection is performed by a component in charge of the trace/replay mechanism.

Both MAD and POET allow the user to test a different order of communication events and to conclude if the reordering of communication events causes different results. We consider that this is a useful feature to include in a future version of MPVVisualizer, combined with the possibility of incremental reexecution.

Users of Xmdb can, *a priori*, list the race conditions classified as harmless; these are not signaled by the tool. MPVVisualizer opens a pop-up window every time a race condition involving two adjacent receptions is detected. Depending on the program being studied, an overwhelming volume of information can be displayed. Therefore, future versions, will include filtering mechanisms to avoid this kind of effect.

The race detection mechanism described is based on an exhaustive study of all possible combinations of the arguments *process id* and *message tag* in two receive events in the same process. The mechanism is not perfect since it is possible to identify race conditions that are not detectable. However, this flaw is due to the fact that the mechanism is embedded in a conceptual model and therefore limited by the amount of information contained in the model. Without a thorough reformulation of the conceptual model itself this has to be considered as a feature.

References

1. Cláudio, A.P., Cunha, J.D. Carmo, M.B.: *Monitoring and Debugging Message Passing Applications with MPVisualizer*. Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society, pages 376-382, Rodhes, January 2000.
2. M. Raynal. *About Logical Clocks for Distributed Systems*. ACM Operating System Review, 26(1): 41-48, 1992.
3. C. Fidge. *Partial Orders for Parallel Debugging*. Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices: 24(1), pages 183-194, 1989.
4. T. Leblanc, J. Mellor-Crummey. *Debugging Parallel Programs with Instant Replay*. IEEE Transactions on Computers, C-36(4), pages 471-482, April 1987.
5. L. Lamport. *Time, Clocks and the Ordering of Events in a Distributed System*. Communications of the ACM, 21(7): 558-565, July 1978.
6. R. Netzer, B. Miller. *Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs*. Proceedings of Supercomputing'92, Minneapolis, USA, pages 502-511, November 1992.
7. D. Kranzlmüller, R. Hügl, J. Volkert. *MAD - A Top Down Approach to Parallel Program Debugging*. Proceedings of HPCN'99, LNCS 1593, Springer-Verlag, pages 1207-1210, April 1999.
8. J.C. Cunha, J. Lourenço, T. Antão. *An Experiment in Tool Integration: The DDBG Parallel and Distributed Debugger*. Euromicro Journal of Systems Architecture, 45(11):897-907, Elsevier Science Press, 1999.
9. R. Wismuller et al.. *The Tool-Set Project: Towards an Integrated Tool Environments for Parallel Programming*. Proceedings of the 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, Koblenz, Germany, September 1997.
10. <http://www.lpds.sztaki.hu>.
11. <http://science.nas.nasa.gov/Groups/Tools/Projects/P2D2>.
12. J. May, F. Berman. *Designing a Parallel Debugger for Portability*. Proceedings of the International Parallel Processing Symposium, 1994.
13. K. Cooper et al.. *The Parascope Parallel Programming Environment*. Proceedings of the IEEE: 81(2), February 1993.
14. J.C. Cunha, J. Lourenço, J. Vieira, B. Moscão, D. Pereira. *A Framework to Support Parallel and Distributed Debugging*. Proceedings of HPCN'98, LNCS 1401, Springer-Verlag, pages 707-717, April 1998.
15. <http://styx.uwaterloo.ca/poet/>.
16. <http://www.pallas.de/pages/totalv.htm>.
17. <http://www.lanl.gov/orgs/cic/cic8/para-dist-team/mdb/mdb.html>.
18. <http://www.netlib.org/utk/icl/xpvm/xpvm.html>.

DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications^{*}

Hua Liu and Manish Parashar

The Applied Software Systems Laboratory

Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA
{marialiu,parashar}@caip.rutgers.edu

Abstract. This paper presents the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for enabling rule based autonomic adaptation and control of distributed scientific applications. DIOS++ provides: (1) abstractions for enhancing existing application objects with sensors and actuators for runtime interrogation and control, (2) a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and (3) a distributed rule engine that enables the runtime definition, deployment and execution of rules for autonomic application management. The framework is currently being used to enable autonomic monitoring and control of a wide range of scientific applications including oil reservoir, compressible turbulence and numerical relativity simulations.

1 Introduction

High-performance parallel/distributed simulations are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities. These simulations and the phenomena that they model are large, inherently complex and highly dynamic. Furthermore, the computational Grid, which is emerging as the dominant paradigm for distributed computing, is similarly heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. As a result, applications must be capable of dynamically managing, adapting and optimizing their behaviors to match the dynamics of the physics they are modeling and the state of their execution environment, so that they continue to meet their requirements and constraints. Autonomic computing [3] draws on the mechanisms used by biological systems to deal with such complexity, dynamics and uncertainty to develop applications that are self-defining, self-healing, self-configuring, self-optimizing, self-protecting, contextually aware and open. This paper presents the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for supporting the autonomic adaptation and control of distributed scientific applications. While a number of existing systems support interactive monitoring

^{*} Support for this work was provided by the NSF via grants numbers ACI 9984357(CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

and steering capabilities, few existing systems (e.g. CATALINA [4] and Autopilot [2]) support automated management and control. DIOS++ enables rules and polices to be dynamically composed and securely injected into the application at runtime so as to enable it to autonomically adapt and optimize its behavior. Rules specify conditions to be monitored and operations that should be executed when certain conditions are detected. Rather than continuously monitoring the simulation, experts can define and deploy appropriate rules that are automatically evaluated at runtime. These capabilities require support for automated and controlled runtime monitoring, interaction, management and adaptation of application objects.

DIOS++ provides: (1) abstractions for enhancing existing application objects with sensors and actuators for runtime interrogation and control, (2) a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these objects at runtime, and (3) a distributed rule engine mechanism that enables the runtime definition, deployment and execution of rules for autonomic application management. Access to an object's sensors and actuators is governed by access control polices. Rules can be dynamically composed using sensors and actuators exported by application objects. These rules are automatically decomposed, deployed onto the appropriate processors using the control network, and evaluated by the distributed rule-engine.

DIOS++ builds on the DIOS [1], a distributed object substrate for interactively monitoring and steering parallel scientific simulations, and is part of the Discover ¹ computational collaboratory. Discover enables geographically distributed clients to collaboratively access, monitor and control Grid applications using pervasive portals. It is currently being used to enable interactive monitoring, steering and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

2 DIOS++ Architecture

DIOS++ is composed of 3 key components: (1) autonomic objects that extend computational objects with sensors (to monitor the state of an object), actuators (to modify the state of an object), access policies (to control accesses to sensors and actuators) and rule agents (to enable rule-based autonomic self-management), (2) mechanisms for dynamically and securely composing, deploying, modifying and deleting rules, and (3) a hierarchical control network that is dynamically configured to enable runtime accesses to and management of the autonomic objects and their sensors, actuators, access policies and rules.

2.1 Autonomic Objects

In addition to its functional interfaces, an autonomic object exports three aspects: (1) a *control aspect*, which defines sensors and actuators to allow the object's state to be externally monitored and controlled, (2) an *access aspect*, which controls accesses to these sensors/actuators and describes users' access privileges based on their capabilities,

¹ <http://www.discoverportal.org>

and (3) a *rule aspect*, which contains rules that can autonomically monitor, adapt and control the object. These aspects are described in the following subsections. A sample object that generates a list of random integers (*RandomList*) is used as a running example. The number of integers and their range are allowed to be set at run time.

Control aspect. The control aspect specifies the sensors and actuators exported by an object. Sensors provide interfaces for viewing the current state of an object, while actuators provide interfaces for processing commands to modify the object's state. For example, a *RandomList* object would provide sensors to query the current length of the list or the maximum value in the list, and an actuator for deleting the current list. Note that sensors and actuators must be co-located in memory with the computational objects and must have access to their internal state, since computational objects may be distributed across multiple processors and can be dynamically created, deleted, migrated and redistributed.

DIOS++ provides programming abstraction to enable application developers to define and deploy sensors and actuators. This is achieved by deriving computational objects from virtual base object provided by DIOS++. The derived objects can then selectively overload the base object methods to specify their sensors and actuators interfaces. This process requires minimal modification to the original computational objects and has been successfully used by DIOS++ to support interactive steering.

Access aspect. The access aspect addresses security and application integrity. It controls the accesses to an object's sensors and actuators and restricts them to authorized users. The role-based access control model is used, where users are mapped to roles and each role is granted specific access privileges defined by access policies.

The DIOS++ access aspect defines three roles: owner, member, and guest. Each user is assigned a role based on her/his credentials. Owner can modify access policies, define access privileges for members and guests, and enable or disable external accesses. Policies define which roles can access a sensor or actuator and in what way. Owners can also enable or disable a sensor or actuator. Access policies can be defined statically during object creation using the DIOS++ API, or can be injected dynamically by the owner at runtime via secure Discover portals.

Rule aspect. The DIOS++ framework uses user-defined rules to enable autonomic management and control of applications. The rule aspect contains rules that define actions that will be executed when specified conditions are satisfied. The conditions and actions are defined in terms of the control aspect (e.g. sensors and actuators). A rule consists of 3 parts: (1) Condition part, defined by the keyword "IF" and composed of conditions which are conjoined by logical relationships (AND, OR, NOT etc.), (2) Then action part, defined by the keyword "THEN" and composed of operations that are executed when the corresponding condition is true, (3) Else action part, defined by the keyword "ELSE" and composed of operations that are executed when condition is not fulfilled.

For example, consider the *RandomList* object with 2 sensors: (1) *getLength()* to get the current length of the list and (2) *getMaxValue()* to get the maximal value in the list, and 1 actuator *append(length, max, min)* that creates a list of size *length* with random integers between *max* and *min*, and appends it to the current list.

```

IF RandomList.getLength()<10 AND RandomList.getMaxValue()<=50
THEN RandomList.append(10, 100, 0)

```

Note that rules are separated from the application logic. This provides flexibility and allows users to dynamically create, delete and change rules without modifying the application. Users use these rules to monitor and control their applications at run time. Rules can be added, deleted, changed on the fly without stopping and restarting the application. Rules are handled by rule agents and the rule engine, which are part of the control network (described in the following subsection) and are responsible for storing, evaluating and executing rules.

2.2 Control Network

The DIOS++ control network (see Figure 1) is a hierarchical structure consisting of a rule engine and gateway, autonomic objects (composed of computational objects and rule agents), and computational nodes. It is automatically configured at run time using the underlying messaging environment (e.g. MPI) and the available processors.

The lowest level of the control network hierarchy consists of computational nodes. Each node maintains a local autonomic object registry containing references to all autonomic objects currently active and registered. At the next level of hierarchy, the Gateway represents a management proxy for the entire application. It combines the registries exported by the nodes and manages a registry of the interaction interfaces (sensors and actuators) for all the objects in the application. It also maintains a list of access policies related to each exported interface and coordinates the dynamic injection of rules. The Gateway interacts with external interaction servers or brokers such as those provided by Discover.

Co-located with Gateway, the rule engine accepts and maintains the rules for the application. It decomposes these rules and distributes them to corresponding rule agents, coordinates the execution of rule agents, keeps track of rule execution and reports them to the user. Each rule agent executes its rules using an execution script, and reports the rule execution status to the rule engine. The execution script is also defined by the rule engine and specifies the rule execution sequence and the rule agent's behavior. The specification and execution of scripts and the coordination between the rule engine and rule agents are illustrated in the following subsections.

In DIOS++, although rule execution is coordinated by the rule engine, rules are evaluated and executed in parallel. This central-control and distributed-execution mechanism has the following advantages: (1) Rule execution which can be compute-intensive is done in parallel by rule agents. This reduces the rule execution time as compared to

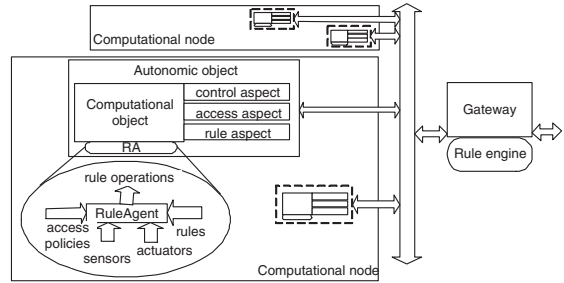


Fig. 1. DIOS++ architecture.

a sequential rule execution. (2) Rule agents are created dynamically and delegated to autonomic objects. This solution requires less system resources than static rule agents as the agents are created only when need. It also leads to more efficient rule execution. (3) Rule agent's behavior is based on script, which allows it to adapt to the execution environment and the rules that it needs to execute. Rule agent scripts can be calibrated at runtime by the rule engine to make rule agents more adaptive.

The operation of the control network is explained below using an example. Consider a simple application that generates a list of integers and then sorts them. This application contains two objects: (1) *RandomList* that provides a list of random integers, and (2) *SortSelector* that provides several sorting algorithms (bubble sort, quick sort, etc.) to sort integers.

Initialization. During initialization, the application uses the DIOS++ APIs to create and register its objects, and to export its aspects, interfaces and access policies to the local computational node. Each node exports these specifications for all its objects to the Gateway. The Gateway then updates its registry. Since the rule engine is co-located with Gateway, it has access to the Gateway's registry. The Gateway interacts with the external access environment (Discover servers in our prototype) and coordinates accesses to the application's sensor/actuators, policies and rules.

Interaction and rule operation. At runtime the Gateway may receive incoming interaction or rule requests from users. The Gateway first checks the user's privileges based on her/his role, and refuses any invalid access. It then transfers valid interaction requests to corresponding objects and transfers valid rule requests to the rule engine. Finally, the responses to the user's requests or from rules executions are combined, collated and forwarded to the user. Once again we use the example to describe this process.

Rule definition: Suppose *RandomList* exports 2 sensors: *getLength()*, and *getList()*. *SortSelector* exports no sensors, and 2 actuators: *sequentialSort()* and *quickSort()*. The owner can access all these interfaces. Members can only access *getLength()* and *getList()* in *RandomList*, and *sequentialSort()* in *SortSelector*. Guests can only access *getLength()* in *RandomList*.

Using DIOS++, users can view, add, delete, modify and temporarily disable rules at runtime using a graphical rule interface integrated with the Discover portal. An application's sensors, actuators and rules are exported to the Discover server and can be securely accessed by authorized users (based on access control policies) via the portal. Authorized users can compose rules using the sensors and actuators. Note that rules may be defined for individual objects or for the entire application and span multiple objects. Users specify a priority for each rule, which is then used to resolve rule conflicts.

Rule deployment: Consider the following rules defined by a user. Let Rule1 have a higher priority than Rule2:

```
Rule1: IF RandomList.getLength()<100 THEN RandomList.getList()
      ELSE RandomList.getLength()
Rule2: IF RandomList.getLength()<50 THEN SortSelector.sequentialSort()
      ELSE SortSelector.quickSort()
```

Rule1 is an object rule, which means that the rule only applies to one object. Rule2 is an application rule, which means that the rule can affect several objects. When the Gateway receives the two rules, it will first check the user's privileges. If the rules are defined by member users, Rule2 will be rejected by Gateway since member users do not have the privilege to access *quickSort()* interface in *SortSelector*.

The Gateway transfers valid rules to the rule engine. The rule engine dynamically creates rule agents for the objects if they do not already exist. It then composes a script for each agent, which defines the rule agent's lifetime and rule execution sequence based on rule priorities. For example, the script for the rule agent for *RandomList* may specify that this agent will terminate itself when it has no rules, and that Rule1 is executed first. Note that this script is extensible.

In the case of an object rule, the rule engine just injects the object rule into its corresponding rule agent. In the case of an application rule, the rule engine will first decompose the rule into triggers and then inject triggers into corresponding agents. So Rule2 is decomposed into 3 triggers: (1) *SortSelector.sequentialSort()*, (2) *SortSelector.quickSort()*, and (3) *RandomList.getLength() < 50*. These triggers are injected into corresponding agents as shown in Figure 2.

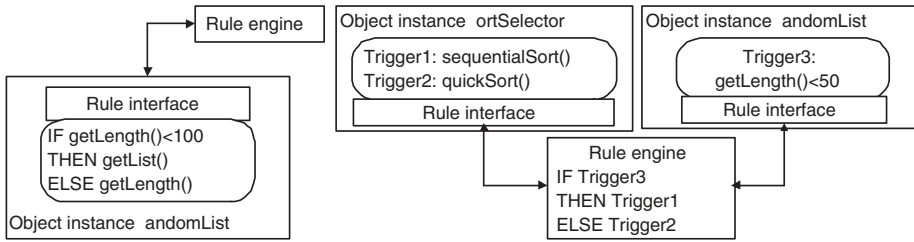


Fig. 2. Left: rule engine deploys an object rule to its corresponding rule agent. Right: rule engine decomposes an application rule into triggers and injects triggers to corresponding rule agents.

Rule execution and conflicts: During interaction periods in the application's execution, the rule engine fires all the rule agents at the same time and rule agents work in parallel. Rule agents execute object rules and return the results to the rule engine. The rule engine then reports them to the user through the Gateway. Rule agents also execute triggers, which are part of application rules, and report corresponding results to the rule engine. The rule engine collects the required trigger results, evaluates condition combinations, and then issues corresponding actions if the conditions are fulfilled. Application rule results are also reported to the user through the Gateway.

While typical rule execution is straightforward (actions are issued when their required conditions are fulfilled), the application dynamics and user interactions make things unpredictable. As a result, rule conflicts must be detected at runtime. In DIOS++, rule conflicts are detected at runtime and are handled by simply disabling the conflicting rules with lower priorities. This is done by locking the required sensors/actuators. For example, suppose that a user defines two rules for the object instance *RandomList*. Rule1 requires setting the minimal integer value to 5 when the list length is less than 100 and

larger than 50, and Rule2 requires the minimal value to be 6 when the list length is larger than 30 and less than 70. Rule1 has higher priority than Rule2. The two rules conflict with each other, for example, when the list length is 60.

```
Rule1: IF RandomList.getLength()>50 AND RandomList.getLength()<100
      THEN RandomList.setMinInt() = 5
Rule2: IF RandomList.getLength()>30 AND RandomList.getLength()<70
      THEN RandomList.setMinInt() = 6
```

The script asks rule agent to fire Rule1 first. After Rule1 is executed, the interface of *setMinInt()* is locked during the period when the length is less than 100 and larger than 50. When Rule2 is issued, it cannot be executed as the required interface is locked. The interface will be unlocked when the length value is not within the range of 50 to 100.

The rule agent at an object will continue existing or destroy itself according to the lifetime specified in its script. Rule engine can dynamically modify the scripts to change the behavior of rule agents.

3 Experimental Evaluation

DIOS++ has been implemented as a C++ library. This section summarizes an experimental evaluation of the DIOS library using the IPARS reservoir simulator framework on a 32 node beowulf cluster. IPARS is a Fortran-based framework for developing parallel/distributed reservoir simulators. Using DIOS++/Discover, engineers can interactively feed in parameters such as water/gas injection rates and well bottom hole pressure, and observe the water/oil ratio or the oil production rate. The evaluation consists of 3 experiments:

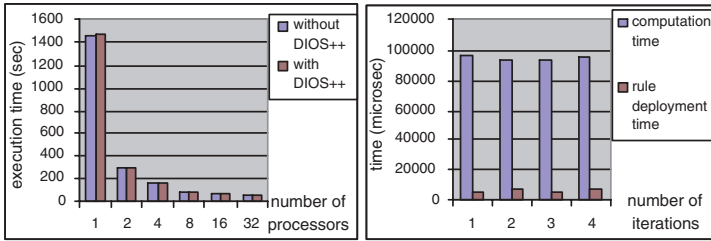


Fig. 3. Left: Runtime overheads introduced in the DIOS++ minimal rule execution mode. Right: Comparison of computation and rule deployment time.

Experiment 1 (Figure 3, left): This experiment measures the runtime overhead introduced to the application in DIOS++ minimal rule execution mode. In this experiment, the application automatically updates the Discover server and its connected clients with current state of autonomic objects and rules. Explicit interaction and rule execution are disabled during the experiment. The application's run time with and without DIOS++

are plotted in the left figure in Figure 3. It can be seen that the runtime overheads due to DIOS++ are very small and are within the error of measurements.

Experiment 2 (Figure 3, right): This experiment compares computation time and rule deployment time for successive iterations. In this experiment, we deployed two object rules and two application rules in 4 successive iterations (object rules are deployed in the first and third iterations; application rules are deployed in the second and fourth iterations). The experiment shows that object rules need less deployment time than application rules. This is true since rule engine only has to inject object rules to corresponding rule agents, while it has to decompose application rules to triggers, and inject triggers to corresponding rule agents.

Experiment 3 (Figure 4): This experiment compares computation time, object rule execution time and application rule execution time for successive application iterations. The experiment shows that an application rule requires a larger execution time than an object rule, since rule engine has to collect results from all the triggers, check whether the conditions are fulfilled and invoke corresponding actions.

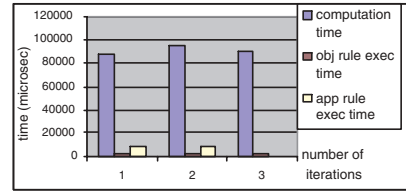


Fig. 4. Comparison of computation, object rule execution and application rule execution times.

4 Summary and Conclusions

This paper presented the design, prototype implementation and experimental evaluation of DIOS++, an infrastructure for supporting the autonomic adaptation and control of distributed scientific applications. DIOS++ enables rules and policies to be dynamically composed and securely injected into application at runtime so as to enable it to autonomously adapt and optimize its behavior. The framework is currently being used, along with Discover, to enable autonomic monitoring and control of a wide range of scientific applications, including oil reservoir, compressible turbulence and numerical relativity simulations.

References

1. R. Muralidhar and M. Parashar, "A Distributed Object Infrastructure for Interaction and Steering", *Concurrency and Computation: Practice and Experience*, John Wiley and Sons, 2003 (to appear)
2. R. Ribler, J. Vetter, H. Simitci, and D. Reed, "Autopilot: Adaptive Control of Distributed Applications", *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998
3. P. Horn, "Autonomic Computing: IBM's perspective on the State of Information Technology", IBM Corp., October 2001. <http://researchweb.watson.ibm.com/autonomic/>
4. S. Hariri, C.S. Raghavendra, Y. Kim, M. Djunaedi, R. P. Nellipudi, A. Rajagopalan, P. Vadlamani, Y. Zhang, "CATALINA: A Smart Application Control and Management", *the Active Middleware Services Conference* 2000

DeWiz – A Modular Tool Architecture for Parallel Program Analysis

Dieter Kranzlmüller, Michael Scarpa, and Jens Volkert

GUP, Joh. Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria/Europe
kranzlmueeller@gup.jku.at
<http://www.gup.uni-linz.ac.at/~dk>

Abstract. Tool support is an important factor for efficient development of parallel programs. Due to different goals, target systems, and levels of abstraction, many specialized tools and environments have been developed. A contrary approach in the area of parallel program analysis is offered by DeWiz, which focuses on unified analysis functionality based on the event graph model. The desired analysis tasks are formulated as a set of graph filtering and transformation operations, which are implemented as independent analysis modules. The concrete analysis strategy is composed by placing these modules on arbitrary networked machines, arranging and interconnecting them. The resulting DeWiz system processes the data stream to extract useful information for program analysis.

1 Introduction

Program analysis includes all activities related to observing a program's behavior during execution, and extracting information and insight for corrective actions. Corresponding functionality is offered by dedicated software tools, which support the user during the diverse activities of program analysis. The complexity of these tasks is affected by a series of factors, which often represent substantial problems for software tool developers.

A serious challenge is introduced by the characteristics of parallel and distributed programs, where the program's behavior is constituted by multiple concurrently executing and communicating processes. Different means of process interaction, e.g. shared memory or message passing, and substantially distinct hardware architectures call for dedicated solutions. Different goals regarding the intended improvements, e.g. performance tuning or correctness debugging, and different levels of abstraction, from source code to machine level, require dedicated functionality. Additional difficulties are given by the typical scale of such programs, e.g. execution times of days, months, or even longer, and the possibly large number of participating processes.

In contrast to many existing and specialized tools, e.g. AIMS [16], Paje [3], Paradyn [11], Paragraph [4], PROVE [5], and VAMPIR [2], the approach of DeWiz tries to offer a unified solution for parallel program analysis. The idea of DeWiz stems from the fact that, despite the many differences, most tools rely

of graph-based analysis methods or utilize space-time diagrams for visualization of program behavior. Consequently, a unified directed graph may be used to capture different program properties and to represent the basis for subsequent analysis activities.

This paper describes the software architecture of DeWiz and discusses its benefits for parallel program analysis. Section 2 provides an overview of related work and the basics of the universal event graph model. Section 3 presents the tool architecture of DeWiz and the concrete activities of constructing a DeWiz system for a particular analysis task. Some examples of DeWiz are presented in Section 4, before conclusions and an outlook on future work summarize the paper.

2 The Universal Event Graph Model

The approach of DeWiz stems from our work on the Monitoring And Debugging environment MAD [6]. MAD is a collection of software tools for debugging message-passing programs based on the MPI standard [13]. At the core of MAD are the monitoring tool NOPE and the visualization tool ATEMPT [7].

Similar to MAD, the basic representation utilized during all analysis activities of DeWiz is the abstract event graph, which can be defined as follows [7]:

Definition 1: Event graph [7]

An event graph is a directed graph $G = (E, \rightarrow)$, where E is the non-empty set of events e of G , while $\rightarrow$ is a relation connecting events, such that $x \rightarrow y$ means that there is an edge from event x to event y in G with the "tail" at event x and the "head" at event y .

The events $e \in E$ of an event graph are the events e_p^i observed during program execution, with i denoting the sequential order of the events relative to object p that is responsible for the event. The vertices of such a graph represent a subset of the state data at a concrete point of time during the program's execution, while the edges represent the transition from one state to another.

Table 1. Event graph data corresponding to target system and event type

Target system	Event	Event Data
parallel/distributed message-passing program	send/receive	source, destination, message type, parameters, ...
multi-threaded shared memory	read/write memory	memory address, size, contents, ...
database/transaction	db_sum	table, location, access time, ...
file I/O, disk access	write	filename, device, buffer size, ...
web server	get	client-IP, URL, ...

Using the event graph for program analysis activities requires to map events onto actual operations of the target program. The universal properties of the event graph allow its application for different kinds of programs and different levels of granularity and abstraction. Some examples are given in Table 1 (compare with [9]).

The relation connecting the events of an event graph is the happened-before relation, which is the transitive, irreflexive closure of the union of the relations $\xrightarrow{S}$ and $\xrightarrow{C}$ as follows:

Definition 2: Happened-before relation [10]

The happened-before relation $\rightarrow$ is defined as $\rightarrow = (\xrightarrow{S} \cup \xrightarrow{C})^+$, where $\xrightarrow{S}$ is the sequential order of events relative to a particular responsible object, while $\xrightarrow{C}$ is the concurrent order relation connecting events on arbitrary responsible objects.

The sequential order $e_p^i \xrightarrow{S} e_p^{i+1}$ defines, that the i^{th} event e_p^i on any (sequential) object p occurred before the $i + 1^{th}$ event e_p^{i+1} on the same object. The concurrent order $e_p^i \xrightarrow{C} e_q^j$ defines, that the i^{th} event e_p^i on any object p occurred directly before the j^{th} event e_q^j on any object q , if e_p^i and e_q^j are somehow connected by their operation.

With the event graph, $e_p^i \rightarrow e_q^j$ means, that e_p^i preceded e_q^j (and e_q^j occurred after e_p^i). For program analysis, a very important view is that $e_p^i \rightarrow e_q^j$ describes the possibility for event e_p^i to causally affect event e_q^j , or in other words, event e_q^j may be causally affected by event e_p^i , if the state of object q at event e_q^j depends on the operation carried out at event e_p^i . Consequently, a program's behavior can be described by its set of program state transitions in the event graph, and this representation can be used for program analysis activities.

3 DeWiz – Tool Architecture

The event graph model as defined above is the basic fundament of DeWiz. The tool itself consists of three main components, the modules, the protocol, and a framework.

The *DeWiz Modules* represent the work units of DeWiz. Each module processes the event graph stream in one way or another. Depending on the particular task of a module, we distinguish between event graph generation, automatic analysis, and data access modules

Event graph generation modules are used at the head of the analysis pipeline to generate the event graph stream for a given program execution. The data is produced either on-line while the monitoring tool is running, or post-mortem by reading corresponding tracefiles. The automatic analysis modules perform the actual operations on the event graph to detect the interesting information, while data access modules are used at the tail of the pipeline to display the results to the user. In most cases, some kind of visualization tool will be used, which draws the resulting event graph as a space-time diagram.

The *DeWiz Protocol* is utilized between modules in order to transport the event graph stream. The protocol defines the particular communication interface (at present, only TCP/IP is supported), and the structure of the transported data.

The *DeWiz Framework* offers the required functionality to implement DeWiz modules for a particular programming language. At present, the complete functionality of DeWiz is supported in Java, while smaller fragments of the analysis pipeline are already available in C. The framework also hides the DeWiz protocol in order to ease the development of DeWiz modules.

Each module of the framework must implement the following functionality: (1) Open event graph stream interface; (2) Filter relevant events for processing; (3) Process event graph stream as desired; (4) Close event graph stream interface.

The functions to open and close interfaces are used to establish and destroy interconnections between modules. The interfaces transparently implement the DeWiz protocol, while filtering and processing of events is performed within the main loop of each protocol. With the DeWiz framework, new modules can be implemented by programming the filtering rules and the intended processing functionality. The remaining functionality of the modules is automatically provided by the framework.

If the modules for a concrete analysis task are available, the user may start to construct a corresponding *DeWiz System*. The modules are placed and initialized on arbitrary networked computing nodes. A dedicated module, the *DeWiz Sentinel* is used to control a particular DeWiz system and to coordinate interconnection between modules. Upon initialization, each module connects to the Sentinel and offers its services. With a controller interface, the registered modules may be arbitrarily interconnected by identifying corresponding input and output interfaces.

An example of the DeWiz controller interface is shown in Figure 1. The smaller window in front shows the module table, including all registered modules (by id and name), their available interfaces and status, the implemented features (send, receive, or none), and the id's of corresponding consumer or producer modules. The larger background window of Figure 1 provides the same information in form of a module diagram.

4 Example Event Graph Analysis Activities

For using DeWiz within a particular programming environment, dedicated event graph generation modules have been implemented. A first module represents a post-mortem trace reader for our monitoring tool NOPE [8]. Additionally, an on-line interface for the OMIS Compliant Monitor OCM [14] has been implemented recently. This allows to analyze arbitrary event graph streams on the fly as they are generated by the OCM tool. To test DeWiz for shared-memory programs, event graph generation functionality has been included in the OpenMP Pragma and Region Instrumentor OPARI [12]. With this module, DeWiz can also be applied to analyze set and unset operations in shared memory programs

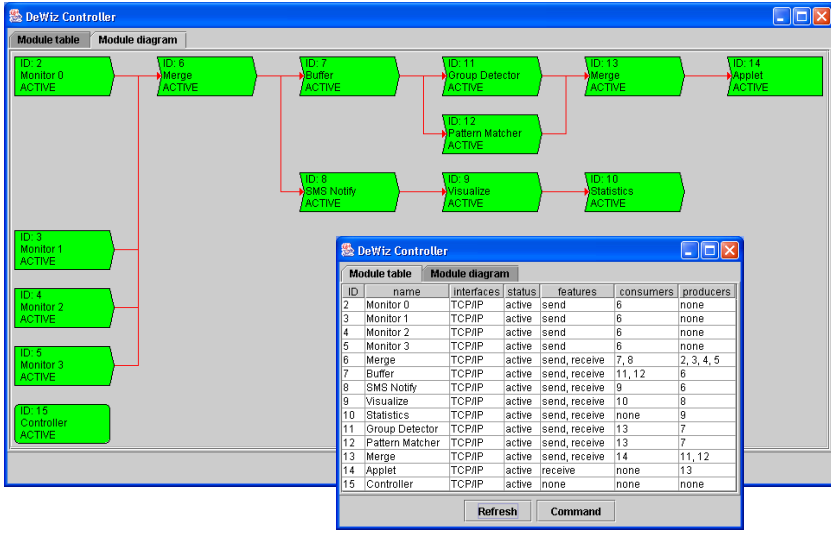


Fig. 1. DeWiz controller showing tool architecture during runtime.

using OpenMP, thus emphasizing the universal applicability of the event graph approach.

In terms of data access modules, DeWiz provides an interface to the analysis tool ATEMPT, a Java applet to display the event graph stream in arbitrary web browsers, and a SMS notifier for critical failures during program execution.

The analysis functionality already implemented in DeWiz is described with the following two examples: Extraction of communication failures and pattern matching and loop detection.

Communication failures can be detected by pairwise analyzing of communication partners. A set of analysis activities for message passing programs is described in [7]. An example is the detection of different message length at send and receive operations. Other examples include isolated send or receive operations, where the communication partner is absent.

A more complex analysis activity compared to the extraction of communication failures is pattern matching and loop detection. The goal of the corresponding DeWiz modules is to identify repeated process interaction patterns in the event graph.

5 Conclusions and Future Work

The DeWiz tool architecture described in this paper offers a universal approach to parallel program analysis. While some analysis tasks are limited to fixed targets, many of them are useful in a broader context. With DeWiz it is possible to reuse a large set of analysis modules for different activities.

Another benefit due to the modular architecture, is scalability. Since DeWiz modules can be placed on arbitrary networked resources, even large amounts of

analysis data can be processed, assuming that the computing power is somewhere available. In this context, DeWiz is also a good candidate for grid computing: On the one hand, large-scale grid applications can be analyzed, on the other hand, DeWiz may utilize grids to perform the analysis activities.

The future work in this project is focused on how to formulate event-based analysis techniques within DeWiz modules. At present, the user can rely on a well-defined set of formal graph-based methods, which must be transformed into a set of Java classes. For the future we plan to include some kind of formal language interpreter for this task. Related work in this area, e.g. the Event Analysis and Recognition Language (EARL) [15], or the Event Based Behavioral Abstraction technique EBBA [1], have already hinted at the feasibility of this approach.

Acknowledgments. We are most grateful for the support by our colleagues, Rene Kobler, Johannes Hoelzl, and Bernhard Aichinger.

References

1. P. Bates. *Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior*. ACM Transactions on Computer Systems, Vol. 13, No. 1, pp. 1–31 (February 1995).
2. H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach*. Proc. ICCS 2001, Intl.Conf. on Computational Science, Springer-Verlag, LNCS, Vol. 2074, San Francisco, CA, USA (May 2001).
3. J. Chassin de Kergommeaux, B. Stein. *Paje: An Extensible Environment for Visualizing Multi-threaded Program Executions*. Proc. Euro-Par 2000, Springer-Verlag, LNCS, Vol. 1900, Munich, Germany, pp. 133–144 (2000).
4. M.T. Heath, J.A. Etheridge. *Visualizing the Performance of Parallel Programs*. IEEE Software, Vol. 13, No. 6, pp. 77–83 (November 1996).
5. P. Kacsuk. *Performance Visualization in the GRADE Parallel Programming Environment*. Proc. HPC Asia 2000, 4th Intl. Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, Peking, China, pp. 446–450 (2000).
6. D. Kranzlmüller, S. Grabner, and J. Volkert. *Debugging with the MAD Environment*. Parallel Computing, Vol. 23, Nos. 1–2, pp. 199–217 (1997).
7. D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Joh. Kepler University Linz (September 2000). <http://www.gup.uni-linz.ac.at/~dk/thesis>.
8. D. Kranzlmüller, J. Volkert. *Debugging Point-To-Point Communication in MPI and PVM*. Proc. EuroPVM/MPI 98, Intl. Conference, Liverpool, GB, pp. 265–272 (Sept. 1998).
9. H. Krawczyk and B. Wiszniewski. *Analysis and Testing of Distributed Software Applications*. Research Studies Press Ltd., Baldock, Hertfordshire, UK (1998).
10. L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM, Vol. 21, No. 7, pp. 558–565, (July 1978).
11. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall. *The Paradyn Parallel Performance Measurement Tool*. IEEE Computer, Vol. 28, No. 11, pp. 37–46 (November 1995).

12. B. Mohr, A.D. Malony, S. Shende, and F. Wolf. *Design and Prototype of a Performance Tool Interface for OpenMP*. Proc. LACSI Symposium 2001, Los Alamos Computer Science Institute, Santa Fe, New Mexico, USA (October 2001).
13. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard – Version 1.1*. <http://www.mcs.anl.gov/mpi/> (June 1995).
14. R. Wismüller, J. Trinitis, and T. Ludwig. *OCM – a Monitoring System for Interoperable Tools*. Proc. SPDT 98, 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, ACM Press, Welches, Oregon, USA, pp. 1–9 (August 1998).
15. F. Wolf and B. Mohr. *EARL – A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*. Technical Report FZJ-ZAM-IB-9803, <http://www.kfa-juelich.de/zam/docs/printable/ib/ib-98/ib-9803.ps>, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Germany (1998).
16. J.C. Yan, H.H. Jin, and M.A. Schmidt. *Performance Data Gathering and Representation from Fixed-Size Statistical Data*. Technical Report NAS-98-003, <http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-003.pdf>. NAS Systems Division, NASA Ames Research Center, Moffet Field, CA, USA (1998).

Why Not Use a Pattern-Based Parallel Programming System?

John Anvik, Jonathan Schaeffer, Duane Szafron, and Kai Tan

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8

{janvik, jonathan, duane, cavalier}@cs.ualberta.ca

Abstract. Parallel programming environments provide a way for users to reap the benefits of parallelism, while reducing the effort required to create parallel applications. The CO₂P₃S parallel programming system is one such tool, using a pattern-based approach to express concurrency. This paper demonstrates that the CO₂P₃S system contains a rich set of parallel patterns for implementing a wide variety of applications running on shared-memory or distributed-memory hardware. Code metrics and performance results are presented to show the usability of the CO₂P₃S system and its ability to reduce programming effort, while producing programs with reasonable performance.

1 Introduction

In sequential programming, a powerful paradigm has emerged to simplify the design and implementation of programs. *Design patterns* encapsulate the knowledge of solutions for a class of problems [4]. To solve a problem using a design pattern, an appropriate pattern is chosen and adapted to the particular problem. By referring to a problem by the particular strategy that may be used to solve it, a deeper understanding of the solution to the problem is immediately conveyed and certain design decisions are implicitly made. Just as there are sequential design patterns, there exist *parallel design patterns* which capture the synchronization and communication structure for a particular parallel solution. The notion of these commonly-occurring parallel structures has been well-known for decades in such forms as skeletons [3,5], or templates [7]. Examples of common parallel design patterns are the fork/join model, pipelines, meshes, and work piles.

Although there have been many attempts to build pattern-based high-level parallel programming tools, few have gained acceptance by even a small user community. The idea of having a tool that can take a selected parallel structure and automatically generate correct structural code is quite appealing. Typically, the user would only fill in application-dependent sequential routines to complete the application. Unfortunately these tools have not made their way into practice for a number of reasons:

1. Performance. Generic patterns produce generic code that is inefficient and suffers from loss of performance.

2. Utility. The set of patterns in a given tool is limited, and if the application does not match the provided patterns, then the tool is effectively useless.
3. Extensibility. High-level tools contain a fixed set of patterns and the tool cannot be extended to include more.

The CO₂P₃S parallel programming system uses design patterns to ease the effort required to write parallel programs. The system addresses the limitations of previous high-level parallel programming tools in the following ways:

1. Performance. CO₂P₃S uses *adaptive generative parallel design patterns* [6], augmented design patterns which are parameterized so that they can be readily adapted for an application, and used to generate a parallel framework tailored for the application. In this manner the performance degradation of generic frameworks is eliminated.
2. Utility. CO₂P₃S provides a rich set of parallel design patterns, including support for both shared-memory and distributed-memory environments.
3. Extensibility. MetaCO₂P₃S is a tool used for rapidly creating and editing CO₂P₃S patterns [2]. CO₂P₃S currently supports 17 parallel and sequential design patterns, with more patterns under development.

This paper focuses on the utility aspect of CO₂P₃S. The intent is to show that the use of a high-level pattern-based parallel programming tool is not only possible, but more importantly, it is practical. The CO₂P₃S system can be used to quickly generate code for a diverse set of applications with widely different parallel structures. This can be done with minimum effort, where effort is measured by the number of additional lines of code written by the CO₂P₃S user. The Cowichan Problems [8] are used to demonstrate this utility by showing the breadth of applications which can be written using the tool. Furthermore, it is shown that a shared-memory application can be recompiled to run in a distributed-memory environment with only trivial changes to the user code. We know of no other parallel programming tool that can support the variety of applications that CO₂P₃S can, is extensible, supports both shared-memory and distributed-memory architectures, and achieves good performance commensurate with user effort expended.

2 The CO₂P₃S Parallel Programming System

The CO₂P₃S¹ parallel programming system is a tool for implementing parallel programs in Java [6]. CO₂P₃S generates parallel programs through the use of *pattern templates*. A pattern template is an intermediary form between a pattern and a framework, and represents a parameterized family of design solutions. Members of the solution family are selected based upon the values of the parameters for the particular pattern template. This is where CO₂P₃S differs from other pattern-based parallel programming tools. Instead of generating an application

¹ Correct Object-Oriented Pattern-based Parallel Programming System, ‘cops’.

framework which has been generalized to the point of being inefficient, CO₂P₃S produces a framework which accounts for application-specific details through parameterization of its patterns. A framework generated by CO₂P₃S provides the communication and synchronization for the parallel application, and the user provides the application-specific sequential code. These code portions are added through the use of sequential *hook methods*. This abstraction maintains the correctness of the parallel application since the user cannot change the code which implements the parallelism at the pattern level. However, due to the layered model of CO₂P₃S [6], the user has access to lower abstraction layers when necessary in order to tune the application.

Extensibility of a programming system supports increased utility. CO₂P₃S improves its utility by allowing new pattern templates to be added to the system using the MetaCO₂P₃S tool [2]. Pattern templates added through MetaCO₂P₃S are indistinguishable in form and function from those already contained in CO₂P₃S. This allows CO₂P₃S to adapt to the needs of the user; if CO₂P₃S lacks the necessary pattern for a problem then MetaCO₂P₃S supports its rapid addition to CO₂P₃S.

3 Using CO₂P₃S to Implement the Cowichan Problems

The number of test suites which address the *utility* or *usability* of a system are few. For parallel programming systems, we know of only one non-trivial set, the Cowichan Problems [8,9], a suite of seven problems specifically designed to test the breadth and ease of use of a parallel programming tool.² When the Cowichan Problems were analyzed, it became evident that CO₂P₃S lacked the necessary patterns to implement four of these problems. For any other high-level programming system, the experiment would have been over. However, using MetaCO₂P₃S, we were able to extend CO₂P₃S to fit our requirements through the addition of two new patterns: the Wavefront pattern and the Search-Tree pattern [1]. For each of these patterns, simple parameterization made them general enough to handle a wide class of applications.

Table 1 provides a summary of which CO₂P₃S pattern was used to solve each of the Cowichan Problems. Note that while CO₂P₃S supports using multiple patterns in an application, this capability was not needed here.

4 Evaluating CO₂P₃S

The results of using CO₂P₃S to implement solutions to the Cowichan Problems are presented here. The results take on two forms: code metrics to show the effort required by a user to take a sequential program and convert it into a

² One modification was made to the original problem set. The single-agent search problem (Active Chart Parsing) takes only a few seconds of CPU time on a modern processor. Therefore, a different single-agent search (IDA*), which runs longer and is representative of this class of problems, was used.

Table 1. Patterns used to solve the Cowichan Problems.

Algorithm	Application	Pattern
IDA* search	Fifteen Puzzle	Search-Tree
Alpha-Beta search	Kece	Search-Tree
LU-Decomposition	Skyline Matrix Solver	Wavefront
Dynamic Programming	Matrix Product Chain	Wavefront
Polygon Intersection	Map Overlay	Pipeline
Image Thinning	Graphics	Mesh
Gauss-Seidel/Jacobi	Reaction/Diffusion	Mesh

Table 2. Code metrics for the shared-memory implementations.

Application	Sequential	Parallel	Generated	Reused	New
Fifteen Puzzle	125	308	123	122	47
Kece	375	539	135	362	42
Skyline Matrix Solver	196	390	224	144	22
Matrix Product Chain	68	296	223	60	13
Map Overlay	85	455	235	60	160
Image Thinning	221	529	350	170	9
Reaction/Diffusion	263	434	205	177	52

parallel program, and performance results. Together, these results show that with minimal user effort, reasonable speedups can be achieved. The speedups are not necessarily the best, since the applications could be further tuned to improve performance using the CO₂P₃S layered model [6]. The results are presented in two tables. Table 2 shows the code metrics from the various implementations and contains the sizes of the sequential and parallel programs, how much of the parallel code was generated by CO₂P₃S, how much code was reused from the sequential application, and how much new sequential code the user was required to write. Table 3 provides performance results for a shared-memory computer.

Table 2 shows that a sequential program can be adapted to a shared-memory parallel program with little additional effort on user’s part. The time required to move from a sequential implementation to a parallel implementation took in the range of a few hours to a few days (if a new pattern had to be created) in each case. The additional code that the user was required to write was typically changes to the sequential driver program to use the parallel framework, and/or changes necessary due to the use of the sequential hook methods. The extreme case of this is for the Map Overlay problem where there was a fundamental change in paradigm between the two implementations. To use the Pipeline pattern, the user is required to create classes for various stages of the pipeline. Each of these classes is required to contain a specific hook method for performing the computation of that stage, and for transforming the current object to the object representing the next stage. As this was not necessary in the sequential application, the user had to write more code to use the Pipeline pattern. For all

Table 3. Speedups for the shared-memory implementations.

Application	2	4	8	16
Fifteen Puzzle	1.74	3.56	6.70	10.60
Kece	1.93	3.42	4.83	5.80
Skyline Matrix Solver	1.93	3.89	7.84	14.86
Matrix Product Chain	1.81	3.64	7.80	13.37
Map Overlay	1.56	3.11	4.67	-
Image Thinning	1.88	3.53	6.39	10.43
Reaction/Diffusion	1.75	3.13	4.92	6.50

Table 4. Code metrics for the distributed-memory implementations.

Application	Sequential	Parallel	Generated	Reused	New
Skyline Matrix Solver	196	1929	1760	144	25
Matrix Product Chain	68	1534	1458	60	16
Image Thinning	221	2138	1968	170	12
Reaction/Diffusion	263	1476	1304	177	55

the other patterns, the user only had to fill in the hook methods for a generated class.

The performance results presented in Table 3 are for a shared-memory architecture. The machine used to run the applications was an SGI Origin 2000 with 46 MIPS R100 195 MHz processors and 11.75 GB of memory. A native threaded Java implementation from SGI (Java 1.3.1) was used with optimizations and JIT turned on, and the virtual machine was started with 1 GB of heap space.

Table 3 shows that the use of the patterns can produce programs that have reasonable scalability. Again, these figures are not the best that can be achieved; all of these programs could be furthered tuned to improve the performance. While most of the programs show reasonable scalability, the two that do not, Kece and Map Overlay, are the result of application-specific factors and not a consequence of the use of the specific pattern. In the case of Kece, the number of siblings processed in parallel during the depth-first search was found to never exceed 20, and was usually less than 10, resulting in processors being starved for work. For the Map Overlay, the problem was only run using up to 8 processors, as the application ran for 5 seconds using 8 processors for the largest dataset size that the JVM could support.

Table 4 shows the code metrics for using CO₂P₃S to generate distributed-memory code. As the distributed implementations of the Pipeline and Search-Tree patterns have not been done yet, only a subset of the problems are shown. A key point is that although CO₂P₃S generates very different frameworks for the shared and distributed-memory environments, the code that the user provides is almost identical. There are only two small one-line differences required for handling distributed exceptions.

5 Conclusions

While parallel programs are known to improve the performance of computationally-intensive applications, they are also known to be challenging to write. Parallel programming tools, such as CO₂P₃S, provide a way to alleviate this difficulty. The CO₂P₃S system is a relatively new addition to a collection of such tools and before it can gain wide user acceptance there needs to be a confidence that the tool can provide the assistance necessary. To this end, the utility of the CO₂P₃S system was tested by implementing the Cowichan Problem Set. This required the addition of two new patterns to CO₂P₃S, highlighting the extensibility of CO₂P₃S—an important contribution to a system’s utility. Parallel computing must eventually move away from MPI and OpenMP. High-level abstractions have been researched for years. The most serious obstacles—performance, utility, and extensibility—are all addressed by CO₂P₃S.

CO₂P₃S is available for download at

<http://www.cs.ualberta.ca/~systems/cops/index.html>. A longer version of this paper is available at

<ftp://ftp.cs.ualberta.ca/pub/TechReports/2003/TR03-13>.

Acknowledgments. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta’s Informatics Circle of Research Excellence (iCORE).

References

1. J. Anvik. Asserting the utility of COPS using the Cowichan Problems. Master’s thesis, Department of Computing Science, University of Alberta, 2002.
2. S. Bromling. Meta-programming with parallel design patterns. Master’s thesis, Department of Computing Science, University of Alberta, 2002.
3. M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computations*. MIT Press, 1988.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
5. D. Goswami, A. Singh, and B. Priess. Architectural skeletons: The re-usable building-blocks for parallel applications. In *Parallel and Distributed Processing Techniques and Applications (PDPTA ’99)*, pages 1250–1256, 1999.
6. S. MacDonald. *From Patterns to Frameworks to Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, 2001.
7. J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.
8. G. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1994.
9. G. Wilson and H. Bal. An empirical assessment of the usability of Orca using the Cowichan problems. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

Topic 2

Performance Evaluation and Prediction

Jeff Hollingsworth, Allen D. Malony, Jesús Labarta, and Thomas Fahringer
Topic Chairs

Performance is the reason for parallel computing. Despite years of work, many applications achieve only a few percentage of theoretical peak. Performance measurement and analysis tools exist to identify the problems with current programs and systems. Performance Prediction is intended to identify issues in new code or systems before they are fully available. These two topics are closely related since most prediction requires data to be gathered from measured runs of program (to identify application signatures or to understand the performance characteristics of current machines).

Several issues continue to confront tool builders. First, how can tools be scaled up to run on very large systems. There are a growing number of parallel and distributed systems with over 1,000 processors and machines with 10,000 to 100,000 processors are being considered. Many tools were designed to work with a few dozen processors. Second, memory system bottlenecks are often the largest source of the gap between theoretical peak and achieved performance. How can tools help programmers to understand where this loss of performance is coming from? Third, how can we achieve high quality predictions of the performance of large-scale applications. Many existing simulation tools are several orders of magnitude slower than running the application. How can we improve the running time of these tools without sacrificing too much accuracy?

Out of 16 submitted papers to this topic, 6 have been accepted as regular papers (37%), and two as short papers. Thanks to all of the contributing authors as well as all reviewers for their work. A special thanks to the program committee Thomas Fahringer, Jesus Labarta, and Allen Malony.

Symbolic Performance Prediction of Speculative Parallel Programs

Hasyim Gautama and Arjan J.C. van Gemund

Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
{h.gautama, a.j.c.vangemund}@its.tudelft.nl

Abstract. Speculative parallelism refers to searching in parallel for a solution, such as finding a pattern in a data base, where finding the first solution terminates the whole parallel process. Different performance prediction methods are required as compared to traditional parallelism. In this paper we introduce an analytical approach to predict the execution time distribution of data-dependent parallel programs that feature N -ary and binary speculative parallel compositions. The method is based on the use of statistical moments which allows program execution time distribution to be approximated at $O(1)$ solution complexity. Measurement results for synthetic distributions indicate an accuracy that lies in the percent range while for empirical distributions on internet search engines the prediction accuracy is acceptable, provided sufficient workload unimodality.

1 Introduction

Analytically predicting the execution time distribution of dependent parallel programs is an extremely challenging problem. Even for fixed problem size the variety of typical input data sets may cause a considerable execution time variance. Especially for time-critical applications, merely predicting the mean execution time does not suffice and knowledge on the execution time distribution is essential. In particular, the execution time distribution of a *parallel* composition of tasks with stochastic workloads is computationally demanding.

Parallel composition can be distinguished in *and*-parallel and *or*-parallel compositions. Of the two forms, *and*-parallel composition is most common in parallel computing, and typically results from *task* or *data* parallelism, where each task essentially involves different computation (workload), or the same computation on different data, respectively. Consider an *and*-parallel composition of N tasks having an execution time X_i . As in *and*-parallelism the slowest computation (largest workload) determines overall execution time Y , it follows that

$$Y = \max_{i=1}^N X_i. \quad (1)$$

Or-parallel composition, in contrast, results from *speculative* parallelism where each task is initiated without being sure that its execution should complete. Instead of *and*-parallel composition, an *or*-parallel composition terminates

when *one* of the parallel tasks has been completed. Examples of or-parallel composition include searching in parallel for a solution, such as finding patterns in a data base, where the first solution found terminates the whole parallel process. As the fastest task completion now determines the performance, the overall execution time Y is given by

$$Y = \min_{i=1}^N X_i. \quad (2)$$

Other areas where computing the minimum of a set of stochastic numbers is of interest include reliability analysis (e.g., system breakdown is determined by earliest component failure), and road traffic analysis (e.g., speed of a vehicle chain on a single lane is determined by the slowest).

In this paper we consider the solution of Eq. (2) which has received less attention compared to Eq. (1), despite its obvious utility. While many authors have studied Eq. (1) as part of their prediction techniques [2,3,5,6,7,8,9,10,13,14,15,16], some authors also include Eq. (2) in their analysis [6,9,14], but restrict the analysis either to specific distributions rarely found in (parallel) programs, or assume distributions to be symmetric [6].

When X_i are deterministic Eq. (2) provides an exact and low-cost prediction tool. When, as in data-dependent programs, X_i is stochastic, interpreting Eq. (2), e.g., in terms of mean values in the sense of

$$E[Y] = \min_{i=1}^N E[X_i] \quad (3)$$

produces a severe error, which for symmetric distributions X_i is linear in the variance of X_i and logarithmic in N [6]. The same applies to Eq. (1).

Solving Eq. (2) presents a major problem that is well-known in the field of order statistics. There are a number of approaches to express an execution time distribution, the choice of which largely determines the trade-off between accuracy and cost involved in solving Y . An exact, closed-form, solution for the distribution of Y is expressed by using the cumulative density function (cdf) [18]. Unfortunately, many execution time distributions do not have a closed-form cdf function, such as the normal distribution and most of the distributions obtained from measurements. Alternative approaches restrict the type of distribution to those for which Eq. (2) can be solved, such as the class of exponential distributions [14] or Erlang and Hyperexponential distributions [9]. While proven a powerful tool for, e.g., reliability modeling, such a distribution workload rarely occurs in practice.

Another, more general way of expressing distributions is based on approximating an execution time distribution in terms of a limited number of statistical moments. In [1] an analytic performance prediction technique based on the first four statistical moments is presented for sequential and conditional task compositions. The raw moment of X , denoted $E[X^r]$, is defined by

$$E[X^r] = \int_{-\infty}^{\infty} x^r dF_X(x). \quad (4)$$

Although the approach is straightforward in the sequential domain, for parallel composition, however, there is in general no analytic, closed-form solution for $E[Y^r]$ in terms of $E[X^r]$ due to the following integration problem. From Eqs. (2) and (4) we obtain

$$E[Y^r] = \int_{-\infty}^{\infty} x^r d(1 - (1 - F_X(x))^N). \quad (5)$$

The right-hand side integration is fundamentally impossible to solve analytically in such a way that $E[Y^r]$ can be expressed directly as function of $E[X^r]$. This problem becomes even more complicated when different workloads are involved.

Recently, a similar integration problem, arising from the *and*-parallel case, has been solved, by introducing generalized lambda distributions as an approximation of the execution time distribution [2,3]. Based on a similar approach, in this paper we solve the integration problem in Eq. (5) for the *or*-parallel composition, for both N -ary or-parallel composition (identical workload distributions), and binary or-parallel composition (different workload distributions). Our contribution has the specific advantage that the approximation of $E[Y^r]$, now readily expressed in terms of the first four input moments $E[X^r]$, has high accuracy and $O(1)$ solution complexity. Furthermore, this approach extends the use of statistical moments, shown to be successful in the sequential domain [1] and the *and*-parallel domain [2,3] into the *or*-parallel domain. To the best of our knowledge such an approach towards solving $E[Y^r]$ in terms of $E[X^r]$ has not been described elsewhere.

The remainder of the paper is organized as follows. Section 2 presents the rationale behind our choice of generalized lambda distributions and describes how our closed-form solution for $E[Y^r]$ is derived. The accuracy of our method is described in Section 3 using standard distributions that are characteristic of workload distributions found in practice. In Section 5 we state our conclusion.

2 Analysis

2.1 Rationale

As mentioned in Section 1, our performance prediction method is based on the use of a limited number of statistical moments (typically the first four moments, i.e., mean, variance, skewness and kurtosis) aimed to pair good accuracy with minimum solution complexity. Previously, the method has been applied to compute the execution time distribution of sequential programs based on the distribution parameters of the constituent program parts such as loops, branches, and basic blocks [1]. The method provides a very good prediction of the execution time of sequential programs, while the analysis is straightforward. In contrast, the analysis of determining the distribution of a *parallel* composition in terms of input distributions is problematic as described in Section 1. Directly applying the moment method in the analysis is fundamentally impossible due to integration problems. In order to extend the application of our moment method to parallel

composition we introduce the use of generalized lambda distributions (GLD) for two reasons. First and most importantly, due to the specific formulation of the GLD we can solve our integration problem in Eq. (5) so that it becomes possible to obtain the moments of Y . Second, unlike other approximating distributions such as, e.g., Pearson distributions, the GLD comprises of only four parameters which makes it directly compatible to our four-moment method used in the sequential domain.

2.2 Generalized Lambda Distributions

The generalized lambda distribution, $\text{GLD}(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$, is a four-parameter distribution defined by the percentile function R as function of the cdf, $0 \leq F \leq 1$, according to [12]

$$X = R_X(F) = \lambda_1 + \frac{F^{\lambda_3} - (1 - F)^{\lambda_4}}{\lambda_2} \quad (6)$$

where λ_1 is a location parameter, λ_2 is a scale parameter and λ_3 and λ_4 are shape parameters. While the cdf does not exist in simple closed form, the probability density function (pdf) is given by

$$f(x) = \frac{dF}{dR_X(F)} = \frac{1}{R'_X(F)} = \frac{\lambda_2}{\lambda_3 F^{\lambda_3-1} + \lambda_4 (1 - F)^{\lambda_4-1}}. \quad (7)$$

This four-parameter distribution includes a wide range of curve shapes. Specifically, the distribution can provide good approximations to well-known densities as found in [12]. Provided that the lambda values are given, obtaining the statistical moments proceeds as follows. Without loss of generality let $\lambda_1 = 0$. Then from Eqs. (4) and (6) the r th raw moment of X is given by

$$\mathbb{E}[X^r] = \frac{1}{\lambda_2^r} \sum_{i=0}^r \binom{r}{i} (-1)^i B(\lambda_3(r-i) + 1, \lambda_4 i + 1) \quad (8)$$

where B denotes the beta function as defined by

$$B(a, b) = \int_0^1 t^{a-1} (1-t)^{b-1} dt. \quad (9)$$

Since the beta function has been provided in the current standard mathematical library, the computation cost of Eq. (8) for r up to four is negligible.

To obtain the four lambda values from the moments we can refer to a table provided in [12]. If more precise lambda values are required, an alternative to looking up the table is to apply a method based on function minimization such as the Nelder-Mead simplex procedure [11]. This procedure requires a constant number of iteration steps. Even for a 10^{-10} precision for the lambda values, our experiments show that no more than 100 iteration steps are required which involves evaluating Eq. (8) (i.e., in total less than 10^5 floating point operations). Thus providing the first four moments, the cost of obtaining the lambda values from the table is also negligible.

2.3 Or-Parallel Composition

In this section we present how the GLD is applied in the analysis of speculative, or-parallel composition. Our results are stated in terms of the following two theorems for N -ary and binary compositions, respectively.

Theorem 1. The n th order statistic

Let $Y_1 \leq Y_2 \leq \dots \leq Y_N$ be random variables obtained by permuting N independent and identical distributed (iid) variates of continuous random variables X , i.e., $X_1, X_2, \dots, X_N$, in increasing order. Let $E[X^r]$, $r = 1, 2, 3, 4$ exists while X can be expressed in terms of $GLD(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ where λ_j are functions of $E[X^r]$. Without loss of generality let $\lambda_1 = 0$. Then for $n = 1, 2, \dots, N$ the r th raw moment of Y_n is given by

$$E[Y_n^r] = \frac{n}{\lambda_2^r} \binom{N}{n} \sum_{i=0}^r \binom{r}{i} (-1)^i B(\lambda_3(r-i) + n, \lambda_4 i + N - n + 1) \quad (10)$$

where B denotes the beta function.

□

For the proof we refer to [4].

Theorem 1 enables us to express $E[Y^r]$ in terms of $E[X^r]$, using the GLD by simply looking up a table [12] and subsequently using Eq. (10). Note, that our analysis technique is parametric in the problem size N (the number of tasks involved in the parallel section). Moreover, note that the solution complexity is entirely *independent* of N (i.e., $O(1)$). If more precise lambda values are required, an alternative to looking up the table is to apply a method based on function minimization such as the Nelder-Mead simplex procedure [11]. This procedure requires a constant number of iteration steps. Even for a 10^{-10} precision for the lambda values, our experiments show that no more than 100 iteration steps are required which involves evaluating Eq. (8) (i.e., in total less than 10^5 floating point operations).

As in speculative parallel composition Y is determined by the smallest X_i , the specific instance $n = 1$ of Theorem 1 applies (i.e., the smallest order statistic). Our result is stated in the following corollary.

Corollary 1. N -ary or-parallel composition

Under the same assumption as in Theorem 1 let random variable Y be defined as

$$Y = \min(X_1, X_2, \dots, X_N). \quad (11)$$

Then the r th raw moment of Y is given by

$$E[Y^r] = \frac{N}{\lambda_2^r} \sum_{i=0}^r \binom{r}{i} (-1)^i B(\lambda_3(r-i) + 1, \lambda_4 i + N). \quad (12)$$

The proof can be straightforward obtained from Eq. (10) by substituting $n = 1$. Similar to sequential compositions the solution complexity of Eq. (12) is entirely *independent* of N (i.e., $O(1)$) while its computation cost is negligible small (cf. Eq. (8)).

While Theorem 1 and associated corollary restrict the workload X_i to be iid, for *different* workload in binary or-parallel composition we present the following theorem.

Theorem 2. Binary or-parallel composition

Let random variable Y be defined as

$$Y = \min(X_1, X_2) \quad (13)$$

where X_i are independent random variables for which $E[X_i^r]$ for $r = 1, 2, 3, 4$ exist. Let X_i can be expressed in terms of GLD($\lambda_1, \lambda_2, \lambda_3, \lambda_4$) where $\lambda_{i,j}$ are functions of $E[X_i^r]$. Then the raw moments of Y are given by

$$E[Y^r] = \int_0^1 (R_{X_1}(F))^r (1 - F_{X_2}(R_{X_1}(F))) + (R_{X_2}(F))^r (1 - F_{X_1}(R_{X_2}(F))) dF. \quad (14)$$

□

Again, for the proof we refer to [4].

A straightforward implementation of Eq. (14) is as follows

$$E[Y^r] = \frac{1}{K} \sum_{k=1}^K ((R_{X_1}(\frac{k}{K}))^r (1 - F_{X_2}(R_{X_1}(\frac{k}{K}))) + (R_{X_2}(\frac{k}{K}))^r (1 - F_{X_1}(R_{X_2}(\frac{k}{K})))) \quad (15)$$

where K is large. In practice, $K = 10^4$ is sufficiently large.

Based on the GLD an alternative approach to compute $F_{X_i}(R_{X_j}(k/K))$ is using Newton's method according to the following.

$$F_{X_{i,l+1}}(R_{X_j}(k/K)) = F_{X_{j,l}} - R_{X_j}(F_{X_{j,l}}) / R'_{X_j}(F_{X_{j,l}}) \quad (16)$$

where $F_{X_{j,0}} = 1/K$. The use of Newton's method is appropriate since we need the GLD only to represent the execution distribution in the integration (Eq. (14)). Furthermore Newton's method converges quadratically and converges to a solution within a few iteration due to specific property of cdf. The evaluation of Eq. (15) requires 60 ms on a 1 GHz Pentium III processor.

3 Synthetic Distributions

This section describes the quality of our prediction approach when applied to some of the frequently-used standard distributions. The estimation quality is defined by the relative error ε_r evaluated from the predicted moments $E[Y_p^r]$ and the measured moments $E[Y_m^r]$, according to

$$\varepsilon_r = \frac{|E[Y_m^r] - E[Y_p^r]|}{E[Y_m^r]} \quad (17)$$

where $E[Y_p^r]$ may be Eq. (12) or (15) for N -ary or binary parallel composition, respectively. In all experiments in this paper we use ε_r in Eq. (17) to determine

the estimation error of our method using 6,000 data samples. For the standard distributions $E[Y_m^r]$ is computed using Eq. (5) since the cdf's are known.

For the uniform distribution, $E[Y^r]$ is determined exactly since the GLD and Newton's method express this distribution exactly.

The exponential distribution (used by many authors) is typical for workloads occurring when, e.g., searching for a key in a data base. The results for $E[X] = 1$ ε_r [%] for N -ary parallel composition is shown by Figure 1 (left) while for binary composition where $E[X_1] = 1$, and $E[X_2] = 1/\theta$ ε_r [%] is shown by Figure 1 (right). The figure shows that ε_r is in the percent range and insensitive to θ value variation.

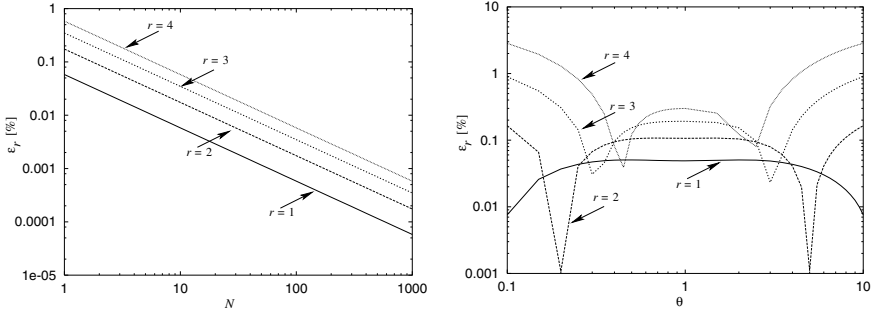


Fig. 1. ε_r [%] for N -ary and binary exponential distribution

Next, we consider the normal distribution, which is also appropriate to model workloads as found in many programs. For N -ary composition we let $E[X] = 0$ and $\text{Var}[X] = 1$ while N varies to 10,000. ε_r is in the percent range as shown by Figure 2. For binary composition we let $E[X_1] = 1$ and $\text{Var}[X_1] = 1$, while $E[X_2] = \mu$ and $\text{Var}[X_2] = \sigma^2$. We vary μ and σ as shown in Figure 3. In Figure 3 (left), ε_r is in the percent range and increases logarithmically as function of σ . Numerical instabilities in the computation of $E[Y_m^r]$ prohibit larger values for σ . In Figure 3 (right), ε_r decreases logarithmically as function of μ , and $\varepsilon_r < 1\%$.

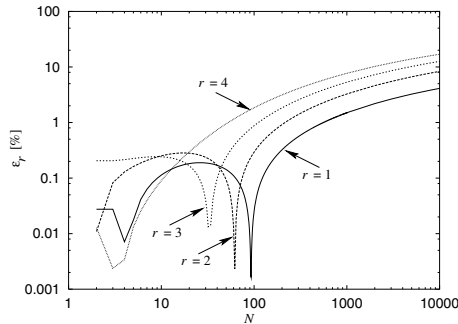


Fig. 2. ε_r [%] for N -ary normal distribution

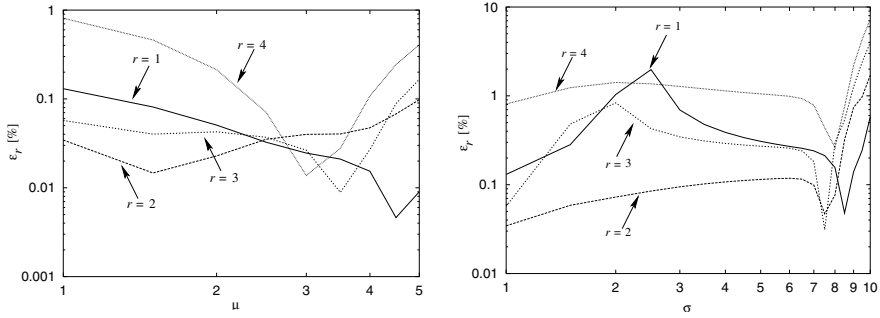


Fig. 3. ε_r [%] for binary normal distribution

4 Empirical Distributions

While the previous distributions already provide an indication of the performance of our technique, in this section we present the results for empirical distributions as measured from a distributed search application (using search engines) on the internet. For a large number of queries, each query is sent to each site in parallel, the response time being determined by the fastest response.

The search engines are located at URL addresses in the USA (dot com), Europe (dot nl), and Latin America (dot cl). As query we search 2,000 large cities around the world while as response we obtain html text, which includes the search time (in seconds) excluding internet communication delay. The workload distribution is shown in Figure 4 (left). The dot cl domain has a left-skewed distribution while the dot com and dot nl domains have bimodal distributions.

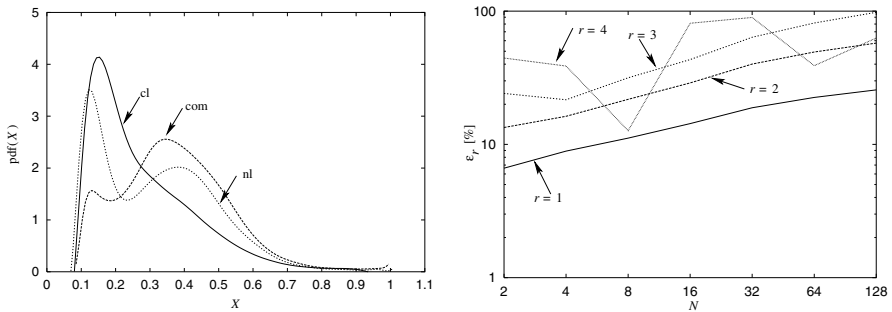


Fig. 4. Empirical distributions (left) and ε_r [%] for .cl (right)

In an N -ary or-parallel search, we represent all sites on one particular domain in terms of one workload distribution X and apply Eq. (12). To resemble identical distributions for X we repeat the search on one domain up to 128 times in unconsecutive manner to avoid probable caching mechanism in the search engine.

The prediction error is shown in Figures 4 (right) and 5 for the dot cl, dot com and dot nl domains, respectively. For all three cases ε_r increases for larger N . The average error for the dot cl domain is appreciable since the distribution is sharply left-skewed where the distribution mass is concentrated near the minimum value. In such case the fitting for lambda values is very sensitive. The error for the dot com and dot nl domains are much worse and increase rapidly for larger N , which is caused by the fact that the distribution of X is no longer unimodal, a requirement of our technique.

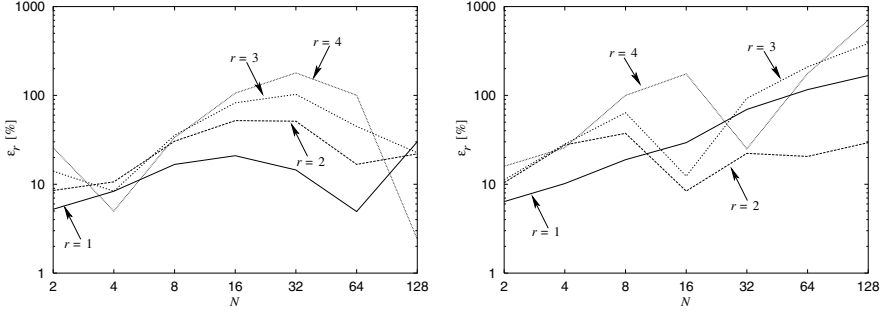


Fig. 5. ε_r [%] for N -ary or-parallel (.com: left and .nl: right)

In a binary or-parallel search we combine each possible combination of two domains and apply Eq. (15). The prediction error for the mean is in the percent range as given in Table 1 while for the higher moments the prediction error is moderate, since again, the workloads are no longer unimodal.

Table 1. ε_r [%] for empirical distributions

X_1	X_2	ε_1 [%]	ε_2 [%]	ε_3 [%]	ε_4 [%]
.cl	.com	4.0	8.5	21	73
.com	.nl	1.1	9.3	21	45
.nl	.cl	1.8	4.7	19	37

5 Conclusion

In this paper we have presented an analytical approach to predict the execution time distribution of N -ary and binary speculative parallel composition at a mere $O(1)$ solution complexity. Measurement results for uniform, exponential and normal distributions indicate an accuracy that lies in the percent range while for real workload distributions on internet search engines the prediction error is in the 10 percent range, provided workload unimodality.

References

1. Gautama, H. and van Gemund, A.J.C.: "Static performance prediction of data-dependent programs," in *ACM WOSP 2000*, Sept. 2000, pp. 216–226. Ottawa, Canada.
2. Gautama, H. and van Gemund, A.J.C.: "Low-cost performance prediction of data-dependent data parallel programs," in *Proc. of MASCOTS 2001*, IEEE Computer Society Press, Aug. 2001, pp. 173–182. Cincinnati, Ohio.
3. Gautama, H. and van Gemund, A.J.C.: "Performance prediction of data-dependent task parallel programs," in *Proc. of EuroPar 2001*, Aug. 2001, pp. 106–116. Manchester, United Kingdom.
4. Gautama, H.: "A statistical approach to performance prediction of speculative parallel programs," Tech. Rep. PDS-2003-007, Delft University of Technology, Delft, The Netherlands, May 2003.
5. Gelenbe, E., Montagne, E., Suros, R. and Woodside, C.M.: "Performance of block-structured parallel programs," in *Parallel Algorithms and Architectures* (Cosnard, M. et al., (eds.)), Amsterdam: North-Holland, 1986, pp. 127–138.
6. Gumbel, E.J.: "Statistical theory of extreme values (main results)," in *Contributions to Order Statistics* (Sarhan, A.E. and Greenberg, B.G. eds.), New York: John Wiley & Sons, 1962, pp. 56–93.
7. Kruskal, C.P. and Weiss, A.: "Allocating independent subtasks on parallel processors," *IEEE TSE*, vol. 11, Oct. 1985, pp. 1001–1016.
8. Lester, B.P., "A system for the speedup of parallel programs," in *Proceedings of the 1986 Intl. Conference on Parallel Processing*, IEEE, Aug. 1986, pp. 145–152. Maharishi Intl. U, Fairfield, Iowa.
9. Liang, D.-R. and Tripathi, S.K.: "On performance prediction of parallel computations with precedent constraints," *IEEE TPDS*, vol. 11, no. 5, 2000, pp. 491–508.
10. Madala, S. and Sinclair, J.B.: "Performance of synchronous parallel algorithms with regular structures," *IEEE TPDS*, vol. 2, Jan. 1991, pp. 105–116.
11. Olsson, D.M. and Nelson, L.S.: "Nelder-Mead simplex procedure for function minimization," *Technometrics*, vol. 17, 1975, pp. 45–51.
12. Ramberg, J.S., Tadikamalla, P.R., Dudewicz, E.J. and Mykytka, F.M.: "A probability distribution and its uses in fitting data," *Technometrics*, vol. 21, May 1979, pp. 201–214.
13. Robinson, J.T.: "Some analysis techniques for asynchronous multiprocessor algorithms," *IEEE TSE*, vol. 5, Jan. 1979, pp. 24–31.
14. Sahner, R.A. and Trivedi, K.S.: "Performance and reliability analysis using directed acyclic graphs," *IEEE TSE*, vol. 13, Oct. 1987, pp. 1105–1114.
15. Schopf, J.M. and Berman, F.: "Performance prediction in production environments," in *Proceedings of IPPS/SPDP-98*, Los Alamitos, IEEE Computer Society, Mar. 30–Apr. 3 1998, pp. 647–653.
16. Sötz, F.: "A method for performance prediction of parallel programs," in *Proc. CONPAR 90- VAPP IV (LNCS 457)* (Burkhart, H., (ed.)), Springer-Verlag, 1990, pp. 98–107.
17. Stuart, A. and Ord, J.K.: *Kendall's Advanced Theory of Statistics*, vol. 1. New York: Halsted Press, 6 ed., 1994.
18. Trivedi, K.S.: *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

A Reconfigurable Monitoring System for Large-Scale Network Computing^{*}

Rajesh Subramanyan¹, José Miguel-Alonso², and José A.B Fortes³

¹ School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, USA

² Department of Computer Architecture and Technology, The University of the Basque Country UPV/EHU, San Sebastian, Spain

³ Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA

Abstract. The dynamic nature of large-size Network Computing Systems (NCSs) and the varying monitoring demands from the end-users pose serious challenges for monitoring systems (MSs). A statically configured MS initially adjusted to perform optimally may end performing poorly. A reconfiguration mechanism for a distributed MS is proposed. It enables the MS to react to changes in the available resources, operating conditions, and monitoring requirements, while maintaining high performance and low monitoring overheads. A localized decision process involving two adjacent intermediate-level managers (ILMs) and values of a local node performance parameter called *temperature* together determine transformations (merge, split, migrate) for each ILM. The reconfiguration mechanisms are derived reusing SNMP primitives. Interactions between MS and NCS are studied by defining a queuing model, and by evaluating different configuration schemes using simulation. Results for the static and reconfigurable schemes indicate that reconfiguration improves performance in terms of lower processing delays at the ILMs.

1 Introduction

A Network Computing System (NCS) is a large collection of heterogeneous resources spread across networks that provides application services to many users. The Grid [1], is an example of such a system. We assume that in NCS, nodes leave and join continually, and are separated by sizeable network distances with unpredictable delays. The performance of the MS is critical to the accuracy and timeliness of monitoring information. A centralized MS cannot scale for large sizes. In [2], we describe a scalable, hierarchical, distributed MS called SIMONE.

Most distributed MS are static, that is, configured manually at start-up time based on the prevalent conditions. The dynamic nature of the NCS environment has been largely overlooked (some exceptions are [3], [4] and [5]). This dynamic nature arises due to (i) changing operating conditions of the NCS, e.g. network

^{*} This work has been done with the support of the Ministerio de Ciencia y Tecnología, Spain, under contract MCYT TIC2001-0591-C02-02.

load, processor load, varying number of nodes in the NCS pool (alters monitor's target size and available resources), and (ii) changing monitoring requirements, e.g. increase parameter update frequency. The application, user, or the system may determine these requirements.

In a dynamic NCS, a distributed configuration initially designed to be optimal may perform poorly over extended periods of time. Monitoring overheads on the NCS may increase due to the skewed usage of resources by the MS.

In this paper, we propose a reconfigurable MS as a solution to the above performance problem. Reconfiguration is an adaptation technique whereby the components of the MS, or the (logical) connections among them, change automatically, adapting themselves to the changing environment with the aim of fulfilling monitoring requirements and reducing overheads. Additional reasons for reconfiguration could include administrative demands, node failures, freeing nodes that are needed for running critical applications.

The remainder of the paper is organized as follows. Section 2 provides some architectural background. Section 3 models the NCS, MS components, and applications running on the NCS, and describes metrics for assessing MS performance. The reconfiguration system and simulation results are discussed in Sections 4 and 5. Related work and conclusions are discussed in Sections 6 and 7 respectively.

2 Towards a Reconfigurable Architecture

In a SNMP-based MS, a manager (client) requests and obtains monitoring information from several agents (server). Agents listen and respond to manager requests (Figure 1). A single centralized manager becomes a bottleneck for agents exceeding 100 to 200 [2]. In the distributed solution [2], several intermediate-level managers (ILMs), in one or more levels, are delegated managerial tasks by the top-level manager (TLM). TLM retains the overall MS control. An ILM is a dual entity: an agent to its manager (a TLM or an upper-level ILM) and a manager to its agents (Figure 1(b)). Its duties are defined as downloadable scripts, providing flexibility, while communication continues to be strictly SNMP. The distributed SIMONE reduces monitoring latency by an order of 2-10 [2]. The configuration is determined at start-up time and remains static.

A reconfiguration mechanism that can be incorporated in SIMONE is proposed. Depending on the temperature (node performance metric) value, a merge, split or a migrate transformation may be performed on the ILM, resulting in the reassignment of managerial tasks or responsibilities (measured in terms of number of managed nodes). The solution must be simple and efficient (imposing low overheads), scalable (we anticipate its use in large-scale NCS), capable of integration with existing monitoring methods and fault-tolerant.

3 Model of the MS in an NCS Environment

A queuing model has been developed to assess the impact of introducing a MS in an NCS and study MS-NCS interactions. Details are described in [6]. Below,

is a summary of the most salient aspects of the model (see Figure 2). Nodes and jobs are classified based on their monitoring roles.

1. There are three classes of nodes: leaves (computing nodes), ILMs, and TLM.
2. A node can have an input queue to receive jobs, a CPU queue to process them, and an output queue to send jobs upward in the monitoring tree.
3. A computing node (leaf, ILM) processes NCS jobs (computing jobs), modeling the actual computing tasks assigned by an external scheduler (NCS scheduling is outside MS control). Also, it processes MS-Poll jobs, modeling the monitoring tasks, and generates MS-Report jobs that are sent to their managers.
4. A manager node (ILM, TLM) processes MS-Report jobs. All nodes run agents (MS-agent jobs).
5. The distribution of the inter-arrival time of MS-Poll and MS-agent jobs are obtained from manager polling and agent update periods (MS operational parameters) respectively. Measurements from an actual MS (SIMONE) were used to determine the CPU usage required to process MS-Poll and MS-Report jobs.
6. Analysis of data from an actual Network Computing System (PUNCH [7], developed and operational at Purdue University) allowed us to get good approximations of realistic distributions for inter-arrival time and duration of NCS jobs.

3.1 Performance Metrics

In this paper, latency is measured for a node and not end-to-end. Node latency is the time between a MS-poll job entering a node until it leaves the node (or the last queue). We introduce **temperature** of a node as a local metric, which assesses the performance of an ILM node and the impact of monitoring tasks on NCS applications in the reconfigurable system. It is expressed as $T = aI + bL$, where, “I” is CPU intrusion and “L” is (node) latency, and “a” and “b” are weight factors. Intrusion affects latency as a heavily used node takes longer to process jobs. Since load is subsumed in latency value, we currently assign a weight 1 to

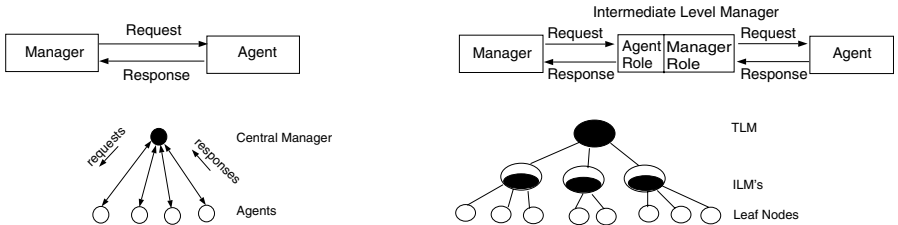


Fig. 1. Model of the client/server paradigm in the centralized (left) and distributed (right) MS

latency and 0 to intrusion. Weights, thresholds and temperature expressions are set and can be dynamically changed by the administrator based on user needs. We define two local metrics and a global one.

- Average node latency: mean delay experienced by poll jobs in an ILM. Each ILM has an average node latency.
- Standard deviation of node latency: measures variations in latency experienced by poll jobs in the ILM node over a period of time.
- Global node latency: mean value of average node latencies of all the ILM nodes. A single value for the whole MS.

4 Reconfiguration

Mechanisms to reduce the overall system temperature fall into two categories,

1. Reducing the activity of the monitoring system [5], e.g. increasing AUP and/or MPP, to reduce the number of parameters measured.
2. Modifying architectural characteristics of the MS to improve its performance without changing MS operational parameters.

Both the approaches are complementary and can be combined. Currently, we assume that the MS operational parameters are already optimally adjusted to satisfy application requirements and further changes compromise monitoring requirements. In this paper we focus only on the second approach.

MS performance can be improved beyond decentralization [2] if the ILM configuration is allowed to change. The arrangement of (logical) connections from leaves to ILMs can be dynamically modified to improve efficiency by moving managerial MS tasks (in terms of number of managed nodes) from heavily loaded ILMs to less loaded ILMs. We call this reconfiguration. Thus, the responsibility of managing node(s) is transferred between ILMs.

The transformations to be applied on ILMs are decided based on current values of temperature which are periodically computed. Keeping the large size

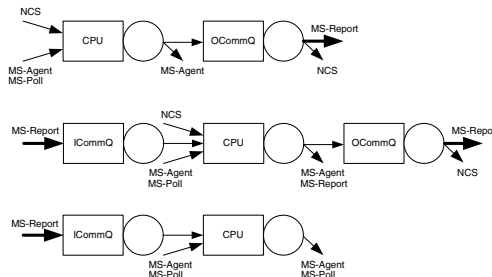


Fig. 2. Models of the components of the monitored NCS: leaf (top), ILM (middle), TLM (bottom). TLM does not process NCS jobs.

of NCS in mind, a localized decision process involving two adjacent levels of nodes is used (see Figure 3). Node A collects temperature and support node lists (SL) for level $I + 1$, computes, and then updates the new support list at level $I + 1$ thus changing their managerial loads. If Node A detects failure(s) at level $I + 1$ (lack of response within timeout), it reassigns the support list of the affected ILM(s) saved from the previous poll, among the remainder level $I + 1$ ILMs. ILMs receive a trigger to initiate its monitoring activities [2]; the same is used to trigger transformations. The transformations are now described as,

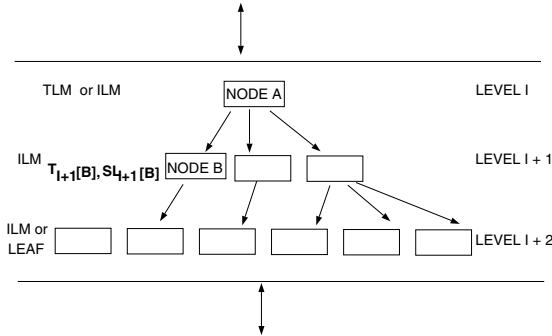


Fig. 3. Segment of the distributed MS showing level I , $I + 1$ and $I + 2$. Each ILM stores temperature, T , and a list of its supported nodes. Transformations for ILMs at level $I + 1$ are decided by level I nodes.

- Migrate – a node stops running an ILM and its managerial tasks are assigned to another, less “hot” node.
- Split – a node does not stop running an ILM but decides to pass part of its managerial tasks to another node.
- Merge– the complement of split. A “cold” ILM assumes the managerial tasks of another cold ILM, which then becomes a regular node.

Migrate and split operations try to reduce the temperature of an ILM. The utility of the merge operation when two nodes are operating within allowable ranges may be debated. The reasons are to free a lightly loaded node that can be fully dedicated to process NCS jobs, experiments show that due to additional communication and synchronization costs, too many ILMs are counterproductive to performance [2], and for symmetry, just as new resources are added when required (split), resources should be freed when not required.

Agents and managers need to be modified to implement these mechanisms. Adhering to SNMP, our design uses an extended MIB to store information needed for reconfiguration. Mechanisms reusing simple SNMP communication primitives are devised for triggers, computation and communication.

The overhead cost, from decision process and transformation enactment, needs to be weighed in before enacting reconfiguration. These are measured in

terms of complexity of the algorithm (computational costs and memory), storage costs, communication costs and delays. Decisions and node list computations are performed between two adjacent levels locally. If ILM at level i supports X ILMs, the overheads for ILM level i are: computational complexity of decisions is $O(X)$, storage is $(X + C)$ MIB variables, and communication: $2X$ SNMP requests to send and receive node lists.

5 Simulation, Results, and Interpretation

The Ptolemy [8] environment was used to design simulation setup for centralized, static distributed and reconfigurable schemes. Additional components including statistical recorders and processor sharing servers were designed by us. Simulations were conducted for single level 2-64 ILMs, 64-2048 leaf nodes for NCS-P¹ NCS-H (higher load and arrival rate than NCS-P) jobs using job distribution characteristics reported in [6] and data from previous experiments [2]. The simulation setup was validated using experimental data for central and distributed schemes. Both, the experimental and simulation values showed similar trends [6], despite the comparisons being approximate (experiments measured end-to-end delay and used different NCS loads).

Time units were defined as simulated time units (stu), where each stu is a clock tick in the simulation and set to be equal to 100 ms of real time. Simulations were run for identical number of stu's. Statistical components record timing data by observing various stages of the job flow. The following measurements were performed.

- Global and ILM latencies for different configurations of static and reconfigurable schemes, using NCS-P and NCS-H jobs.
- Impact of thresholds and trigger periods on MS performance.
- MS performance in hot zone (high temperature region) for different schemes.
- Improvements by including past values in temperature computation.

5.1 Global and Average Latency, Deviation in Latency

Simulations were conducted using NCS-P jobs for different configurations (64-2048 leaves, single level 2-64 ILMs) for static and reconfigurable schemes. For small (> 32) fanin's, both schemes have low global latencies, with static being slightly lower. Saturation occurs at large fanin's, making comparisons irrelevant. Reconfiguration delays saturation and reduces global latencies as jobs submitted to lighter nodes cause smaller increases in latency. Two ILMs have different average latencies despite reallocation, as a fraction of jobs (MS) alone are rescheduled. Results indicate that the reconfigurable MS performs better than the static; has 10-60% lower global latencies for intermediate fanin's, has smaller and flatter ILM latencies, and smaller latency deviations (sample results are shown in Figure 4).

¹ P refers to PUNCH, the NCS whose traces were used to model the distribution of jobs

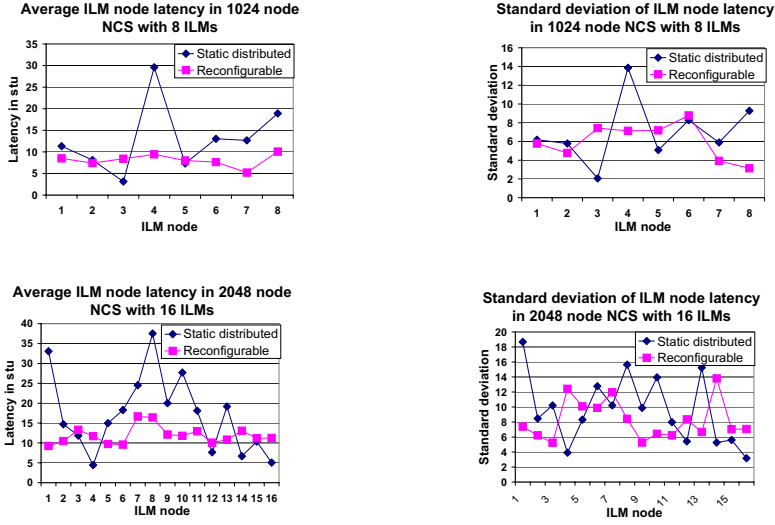


Fig. 4. Comparing average and standard deviation of ILM latencies for static distributed and reconfigurable schemes for 1024 nodes/8 ILMs and 2048 nodes/16 ILMs using NCS-H jobs. All units are in stu.

5.2 Effect of Varying Thresholds and Trigger Periods

Threshold values are expressed as [merge, split, migrate], e.g. [0.1-25-500]. Here temperature below 0.1 triggers merge, 0.1 and 25 causes no change, between 25 and 500, a split, and above 500, a migrate. Results of varying one threshold value at a time (Figure 5), increasing one threshold region at a time (Table 1), and varying the trigger period (Figure 5) using 1024 ILMs with 8 ILMs and NCS-H jobs are shown. The optimal values appear in the middle, a common pattern observed in all cases. At the end regions, either several spikes occur with too few transformations, or frequent transformations lead to poor decisions (an ILM loaded heavily for short intervals need not undergo transformations).

At low split threshold, global latency may even exceed that of static distribution due to very frequent transformations. Large migrate thresholds yield low latencies suggesting sparing use of migrate transformation. Merge frees up ILM nodes, but increases load on remaining ILMs. Low merge thresholds are recommended particularly with few ILMs. Larger region for split (more split transformations) improves performance, provided the split thresholds are not selected too low. Trigger rate improves performance by prompt reaction to spikes, however, very high rates are counterproductive (Figure 5).

Currently, we base threshold selections on user requirements rather than achieving optimality. Results indicate strong effect of threshold and trigger values on MS performance, suggesting automation.

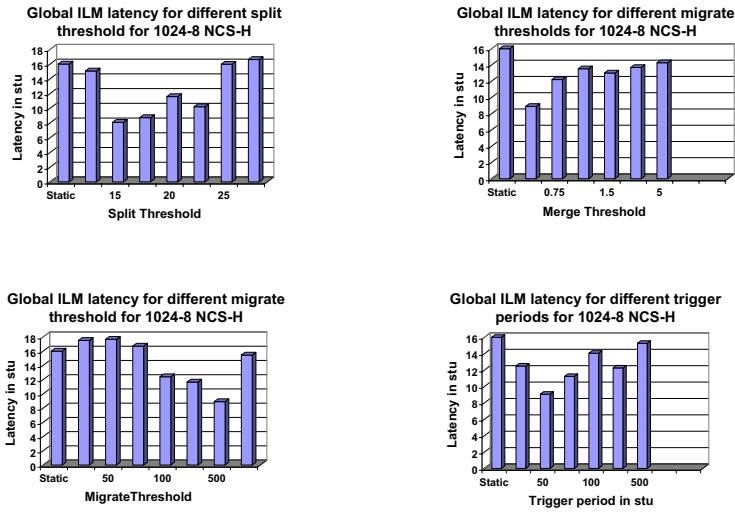


Fig. 5. Effect of varying threshold or trigger for 1024 node NCS/8 ILMs reconfigurable scheme, using NCS-H jobs. The figures above are (i) vary split threshold, merge threshold = 0.1, migrate = 500, (ii) vary migrate, merge = 0.1, split = 17.5, (iii) vary merge, split = 17.5, migrate = 500, (iv) vary trigger period. Global latencies are given in units of stu.

5.3 Operating in Hot Zone

“Hot zone” is a region of high temperature for ILM operation. ILMs with temperature exceeding 1.2 times average global latency are considered to operate in the hot zone. Figure 6(i) indicates that reconfiguration significantly reduces the percentage of time spent by ILMs in the hot zone. Medium thresholds yield best results. While ILMs in static scheme spent on an average 37% of total time in hot zone, the corresponding values were 23%, 10% and 24% for reconfigurable schemes with low, medium and high thresholds under identical conditions.

ILMs in static scheme spent on an average 19 trigger intervals per visit to the hot zone, while the corresponding values for the reconfigurable scheme with low, medium and high thresholds were 4, 3, and 7 (Figure 6(ii)). The average length

Table 1. Effect of thresholds for 1024-8 reconfigurable configuration, using NCS-H jobs. Global latencies are given in units of stu.

Threshold Set	Transformation with increased threshold range	Global latency (stu)
—	Static distribution	15.89
0.1-20-500	Normal	11.6
0.1-10-500	Split	14.27
0.1-20-50	Migrate	17.7
1-20-500	Merge	13.3

of time spent in the hot zone for each visit reflects how prompt and effective the transformations are. High thresholds increases the time spent in the hot zone. In the static scheme, ILMs leave hot zone only due to a fall in NCS activity.

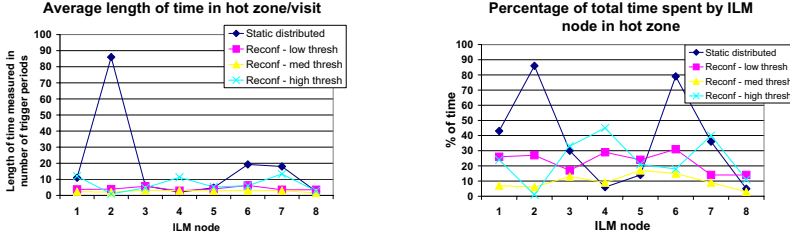


Fig. 6. (i) Percentage of total time spent by ILM in hot zone. Threshold values are low = [0.1-10-500], medium = [0.1-17.5-500], high = [0.1-25-500]. (ii) Average duration (counted in number of trigger cycles) spent by ILM node in the hot zone/visit.

5.4 Temperature Computation with History

Two sets of simulations were carried out under identical conditions, except one uses current temperature (T_c), while the other uses a “weighted” temperature (T_w). Higher weights are accorded to recent polls, and selected as follows: $T_w = 0.57T_c + 0.28T_{c_1} + 0.15T_{c_2}$, where T_c is the temperature during the current poll, T_{c_1} the temperature during the previous poll, and so on. Weighted temperature yields better results by helping avoid transformations for short temporary spikes. Using current temperature, global latency values were 14.25, 8.09 and 15.8 for low, medium and high threshold. Corresponding figures with weighted temperature were 13.07, 10.27 and 12.47 respectively.

6 Related Work

Tools designed for network computing environment are: NWS [9] which provides dynamic resource performance forecasts; Globus’ Gloperf [10] tool which works like NWS, and periodically schedules end-to-end tests to retrieve latency and bandwidth information; Netlogger [11], a trace-based system used mainly for diagnosis, and Remos [12] which combines different monitoring techniques such as SNMP and end-to-end tests focusing on providing information for network-aware applications. Performance issues when working in large networks are tackled by using a hierarchy of groups in NWS and Gloperf, by stopping and starting logging process in Netlogger, and by implementing a hierarchical monitoring architecture using a data collector in Remos. These solutions are statically configured and use SNMP as an enhancement, thus differing from our solution. The Grid Performance Working Group of the Grid Forum has ongoing work towards defining a monitoring service architecture for the Grid [1]. SIMONE will be made compliant with these specifications to allow interaction with other Grid systems.

While reconfiguration, migration and other concepts have been widely studied in other areas [13], most monitoring systems have static configurations. Mobile code paradigms for network management have suggested taking advantage of migration. Liotta [13] uses a dynamic hierarchical manager model based on delegation using mobile agents, for scalable monitoring. Liotta indicates the complexities and drawbacks of using mobile code. Our solution strictly uses SNMP mechanisms and is applicable to any SNMP-based MS. Lutfiyya's [3] adaptive model, studies the reconfiguration of monitoring elements in CMIP [14] in order to optimally minimize monitoring overheads in small systems. Our algorithm chooses a simple approximate solution, which may be better suited for large systems. Hollingsworth [5] proposes an adaptive cost system for Paradyn for maintaining perturbation [15] levels in applications by delaying dynamic instrumentation until execution. This allows the flexibility of dynamic insertion and alteration. Performance is gained by reducing instrumentation. Our solution assumes reducing polling frequency would compromise monitoring accuracy requirements. The Quorum [16] program by DARPA has several ongoing initiatives on resource management, scheduling etc.

7 Conclusions

A reconfigurable mechanism is proposed that reassigns monitoring responsibilities (managed nodes) among ILMs in response to changing operating conditions, thus maintaining MS performance. Temperature, threshold, transformations, trigger and other mechanisms are designed fully within the SNMP context and can be integrated with existing SIMONE. Design features allow ILM robustness, local transformation decisions aid scalability, and MIB extensions and SNMP reuse makes the solution simple and interoperable. A queuing model was developed to describe MS-NCS interactions, and distribution of inputs were derived. These were used to design and perform simulations for 64-2048 nodes and 2-64 single level ILMs configurations.

Simulation results indicate 10-60% smaller global latencies and less variation in average ILM latencies in the reconfigurable scheme. Too few or too many transformations reduces performance as observed while varying thresholds or trigger period. The performance is sensitive to threshold selection, suggesting automation. Infrequent merge and migrate is desired combined with medium occurrence of split transformation. Under identical conditions, static ILMs spent 37% of the time in hot zone compared to 10-24% for reconfigurable schemes. Correspondingly, the average stay/visit in hot zone was 19 for static ILMs as compared to 3-7 in reconfigurable schemes, expressed in units of trigger interval.

The above results indicate that reconfigurable scheme improves performance and significant gains may be achieved with proper selection of thresholds. Adding fork and join transformations, automating threshold selection, and developing backup TLM are future enhancements proposed.

References

1. I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations.", in *Intl. J. Supercomputer Applications*, 2001.
2. R. Subramanyan, J. Miguel-Alonso and J.A.B Fortes, "A Scalable SNMP-based Distributed Monitoring System for Heterogeneous Network Computing", in *Supercomputing*, Nov. 2000.
3. Hasina Abdu, Hanan Lutfiyya and Michael A. Bauer, "A Testbed for Optimizing the Monitoring of Distributed Systems", in *Proceeding of PDCS '98*, 1998.
4. A. Liotta, G. Pavlou, G. Knight, "A Self-adaptable Agent System for Efficient Information Gathering.", in *MATA*, 2001.
5. J.K. Hollingsworth and B.P. Miller, "An Adaptive Cost System for Parallel Program Instrumentation", in *Euro-Par '96*, Aug. 1996.
6. R. Subramanyan, *Scalable SNMP-Based Monitoring Systems for Network Computing*, PhD thesis, Purdue University, Aug. 2002.
7. N.H. Kapadia and J.A.B. Fortes, "PUNCH: An Architecture for Web-Enabled Wide-Area Network-Computing", *Cluster Computing*, Sept. 1999.
8. *Ptolemy 0.7 User's Manual*, UC Berkeley, <http://ptolemy.eecs.berkeley.edu>.
9. R. Wolski, "Dynamically Forecasting Network Performance using the Network Weather Service", *Cluster Computing*, 1998.
10. C.A. Lee, J. Stepanek, R. Wolski, C. Kesselman and I. Foster, "A Network Performance Tool for Grid Environments", in *HPDC 98*, 1998.
11. B. Tierney, W. Johnston and B. Crowley, "The Netlogger Methodology for High Performance Distributed Systems Performance Analysis", in *HPDC 98*, 1998.
12. Nancy Miller and Peter Steenkiste, "Collecting Network Status Information for Network-Aware Applications", in *Proceeding of Infocom 2000*, 2000.
13. Antonio Liotta, Graham Knight and George Pavlou, "On the Performance and Scalability of Decentralized Monitoring using Mobile Agents", in *DSOM*, 1999.
14. U. Black, *Network Management Standards*, McGraw-Hill, 1995.
15. A.D. Malony, D.A. Reed and H.A.G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis", in *IEEE Transactions on Parallel and Distributed Systems*, July 1992.
16. DARPA, *Quorum Project*, <http://www.darpa.mil/ito/research/quorum/index.html>.
17. G. Goldszmidt, *Distributed Management by Delegation*, PhD thesis, Columbia University, Dec. 1995.
18. A. Waheed, D.T. Rover, M.W. Mutka, H. Smith, and A. Bakic, "Modeling, Evaluation, and Adaptive Control of an Instrumentation System", in *Proc. Real-Time Technology and Applications Symposium (RTAS '97)*, June 1997.
19. M. Siegl, *Design and Realization of a Mid-Level Management System*, PhD thesis, TUW-HDNM, Vienna University of Technology, Nov. 1996.
20. A. Pras, *Network Management Architectures*, PhD thesis, Centre for Telematics and Information Technology, University of Twente, April 1995.

Obtaining Hardware Performance Metrics for the BlueGene/L Supercomputer

Pedro Mindlin, José R. Brunheroto, Luiz DeRose, and José E. Moreira

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
{pamindli, brunhe, laderose, jmoreira}@us.ibm.com

Abstract. Hardware performance monitoring is the basis of modern performance analysis tools for application optimization. We are interested in providing such performance analysis tools for the new BlueGene/L supercomputer as early as possible, so that applications can be tuned for that machine. We are faced with two challenges in achieving that goal. First, the machine is still going through its final design and assembly stages and, therefore, it is not yet available to system and application programmers. Second, and most important, key hardware performance metrics, such as instruction counters and Level 1 cache behavior counters, are missing from the BlueGene/L architecture. Our solution to those problems has been to implement a set of nonarchitected performance counters in an instruction-set simulator of BlueGene/L, and to provide a mechanism for executing code to retrieve the value of those counters. Using that mechanism, we have ported a version of the libHPM performance analysis library. We validate our implementation by comparing our results for BlueGene/L to analytical models and to results from a real machine.

1 Introduction

The BlueGene/L supercomputer [1] is a highly scalable parallel machine being developed at the IBM T. J. Watson Research Center in collaboration with Lawrence Livermore National Laboratory. Each of the 65,536 compute nodes and 1,024 I/O nodes of BlueGene/L is implemented using system-on-a-chip technology. The node chip contains two independent and symmetric PowerPC processors, each with a 2-element vector floating-point unit. A multi-level cache hierarchy, including a shared 4 MB level-2 cache, is part of the chip. Figure 1 illustrates the internals for the BlueGene/L node chip.

Although it is based on the mature PowerPC architecture, there are enough innovations in the BlueGene/L node chip to warrant new work in optimizing code for this machine. Even if we restrict ourselves to single-node performance, understanding the behavior of the memory hierarchy and floating-point operations is necessary to optimize applications in general.

The use of analysis tools based on hardware performance counters [2,5,6,7,10,12,13,14,15] is a well established methodology for understanding and optimizing the performance of applications. These tools typically rely on software instrumentation to obtain the value of various hardware performance counters through kernel services. They also rely on kernel services that *virtualize* those counters. That is, they provide a different set

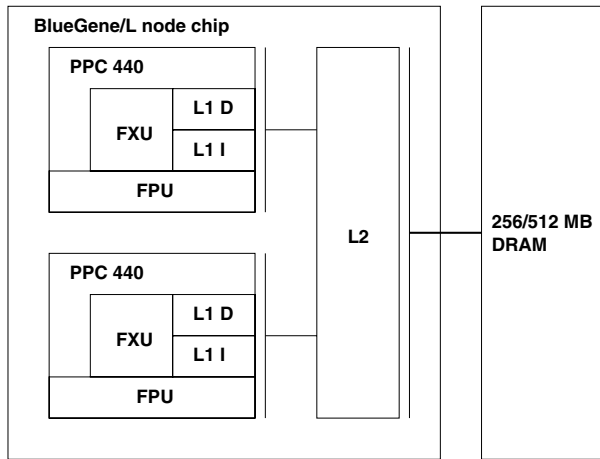


Fig. 1. Internal organization of the BlueGene/L node chip. Each node consists of two PowerPC model 440 CPU cores (PPC 440), which include a fixed-point unit (FXU) and 32KB level-1 data and instruction caches (L1 D and L1 I). Associated with each core is a 2-element vector floating-point unit (FPU). A 4MB shared level-2 unified cache (L2) is also included in the chip. 256 or 512 MB of DRAM (external to the chip) complete a BlueGene/L node.

of counters for each process, even when the hardware only implements a single set of physical counters. The virtualization also handles overflowing of counters by creating virtual 64-bit counters from the physical (which are normally 32-bit wide) counters.

We plan to provide a version of the libHPM performance analysis library, from the Hardware Performance Monitor (HPM) toolkit [10], on BlueGene/L. We believe this will be useful to applications developers in that machine, since many of them are already familiar with libHPM in other platforms. However, a deficiency of the performance data that can be obtained from the BlueGene/L machine is the limited scope of its available performance counters. The PowerPC 440 core used in BlueGene/L is primarily targeted at embedded applications. As is the case with most processors for that market, it does not support any performance counters. Although the designers of the BlueGene/L node chip have included a significant amount of performance counters outside the CPU cores (e.g., L2 cache behavior, floating-point operations, communication with other nodes), there are critical data not available. In particular, counters for instruction execution and L1 cache behavior are not present in BlueGene/L.

Hence, our solution to providing complete hardware performance data for BlueGene/L applications relies on our architecturally accurate system simulator, BGLsim. This simulator works by executing each machine instruction of BlueGene/L code. We have extended the simulator with a set of non-architected performance counters, that are updated as each instruction executes. Since they are implemented in the internals of the simulator, they are not restricted by the architecture of the real machine. Therefore, we can count anything we want without disturbing the execution of any code. Per process and user/supervisor counters can be implemented directly in the simulator, without any

counter virtualization code in the kernel. Finally, we provide a lightweight architected interface for user- or supervisor-level code to retrieve the counter information. Using this architected interface, we have ported libHPM to BlueGene/L. As a result, we have a well-known tool for obtaining performance data implemented with a minimum of perturbation to the user code. We have validated our implementation through analytical models and by comparing results from our simulator with results from a real machine.

The rest of this paper is organized as follows. Section 2 presents the BGLsim architectural simulator for BlueGene/L. Section 3 discusses the specifics of implementing all layers needed to support a hardware performance monitoring tool on BGLsim, from the implementation of the performance counters to the modifications needed to port libHPM. Section 4 discusses the experimental results from using our performance tool chain. Finally, Section 5 presents our conclusions.

2 The BGLsim Simulator

BGLsim [8] is an architecturally accurate instruction-set simulator for the BlueGene/L machine. BGLsim exposes all architected features of the hardware, including processors, floating-point units, caches, memory, interconnection, and other supporting devices. This approach allows an user to run complete and unmodified code, from simple self-contained executables to full Linux images. The simulator supports interaction mechanisms for inspecting detailed machine state, providing more monitoring capabilities beyond what is possible with real hardware. BGLsim was developed primarily to support development of system software and application code in advance of hardware availability. It can simulate multi-node BlueGene/L machines, but in this paper we restrict our discussion to the simulation of a single BlueGene/L node system.

BGLsim can simulate up to 2 million BlueGene/L instructions per second when running on a 2 GHz Pentium 4 machine with 512 MB of RAM. We have found the performance of BGLsim adequate for the development of system software and for experimentation with applications. In particular, BGLsim was used for the porting of the Linux operating system to the BlueGene/L machine.

We routinely use Linux on BGLsim to run user applications and test programs. This is useful to evaluate the applications themselves, to test compilers, and to validate hardware and software designs choices. At this time, we have successfully run database applications (eDB2), codes from the Splash-2, Spec2000, NAS Parallel Benchmarks, and ASCI Purple Benchmarks suites, various linear algebra kernels, and the Linpack benchmark.

While running an application, BGLsim can collect various execution data for that application. Section 3 describes a particular form of that data, represented by performance counters. BGLsim can also generate instruction traces that can be fed to trace-driven analysis tools, such as SIGMA [11].

BGLsim supports a *call-through* mechanism that allows executing code to communicate directly with the simulator. This call-through mechanism is implemented through a reserved PowerPC 440 instruction, which we call the call-through instruction. When BGLsim encounters this instruction in the executing code, it invokes its internal call-through function. Parameters to that function are passed in pre-defined registers. De-

pending on those parameters, specific actions are performed. The call-through instruction can be issued from kernel or user mode. Examples of call-through services provided by BGLsim include:

- *File transfer*: transfers a file from host machine to simulated machine.
- *Task switch*: informs the simulator that the operating system (Linux) has switched to a different process.
- *Task exit*: informs the simulator that a given process has terminated.
- *Trace on/off*: controls the acquisition of trace information.
- *Counter services*: controls the acquisition of data from performance counters.

As we will see in Section 3, performance monitoring on BGLsim relies heavily on call-through services. In particular, we have modified Linux to use call-through services to inform BGLsim about which process is currently executing. BGLsim then uses this information to control monitoring (e.g., performance counters) associated with a specific process.

The PowerPC 440 processors supports two execution modes: *supervisor* (or *kernel*) and *user*. A program that executes in supervisor mode is allowed to issue privileged instructions, as opposed to a program running in user mode. BGLsim is aware of the execution mode for each instruction, and this information is also used to control monitoring. By combining information about execution mode with the information about processes discussed above, BGLsim can segregate monitoring data by user process and, within a process, by execution mode.

3 Accessing Hardware Performance Counters on BGLsim

The BlueGene/L node chip provides the user with a limited set of hardware performance counters, implemented as configurable 32- or 64-bit registers. These counters can be configured to count a large number of hardware events but there are significant drawbacks. The number of distinct events that can be simultaneously counted is limited by chip design constraints, therefore restraining the user's ability to make thorough performance analysis with a single application run. Moreover, the set of events to be simultaneously counted cannot be freely chosen, and some important events, such as level-1 cache hits and instructions executed, cannot be counted at all.

As an architecturally accurate simulator, BGLsim should only implement the hardware performance counters actually available in the chip. However, we found that BGLsim allowed for much more: the creation of a set of non architected counters, which could be configured to count any set of events at the same time.

The BGLsim non architected performance counters were organized as two sets of 64 counters. The counters are 64-bit wide. Pairs of counters, one in each set, count the same event, but in separate modes: one set for user mode and the other set for supervisor mode. Furthermore, we have implemented an array of 128 of these double sets of counters, making it possible to assign each double set to a different Linux process id (PID), thus acquiring data from up to 128 distinct Linux concurrent processes.

To provide applications with performance data, we had first to enable access to the BGLsim non architected performance counter infra-structure to executing code. This

was accomplished through the call-through mechanism described in Section 2. Through the counter services of the call-through mechanism, executing code can reset and read the values of the non architected performance counters. We used the task switch and task exit services (used by Linux) to switch the (double) set of counters being used for counting events. We also used the knowledge of execution mode to separate counting between the user and supervisor sets of counters. That way, each event that happens (*e.g.*, instruction execution, floating-point operation, load, store, cache hit or miss), is associated with a specific process and a specific mode.

To make the performance counters easily accessible to an application, we defined the *BGLCounters* application programming interface. This API defines the names of the existing counters in BGLsim to the application, and creates wrapper functions that, in turn, call the corresponding call-through services with the appropriate parameters. The BGLCounters API provides six basic functions: initialize the counters, select which counters will be active (usually all), start the counters, stop the counters, reset the counters, and read the current values of the counters.

Finally, we ported libHPM to run on BGLsim. This port was done by extending the library to use the BGLCounters API, adding support for new hardware counters and derived metrics that are related to the BlueGene/L architecture, such as the 2-element vector floating-point unit, and by exploiting the possibility of counting both at user mode and at supervisor mode during the same execution of the program. Figure 2 displays a snapshot of the libHPM output from the execution of an instrumented CG code, from the NAS Parallel Benchmarks [4]. It exhibits the feature of providing the counts for both user and kernel activities, which, in contrast with other hardware counters based tools, is a unique feature of this version of libHPM.

Counter	User	Kernel
BGL_INST (Instructions completed)	308099072	9715300
BGL_LOAD (Loads completed)	31607898	1055210
BGL_STORE (Stores completed)	1207204	593399
BGL_D1_HIT (L1 data hits (1/s))	85123481	1633792
BGL_D1_L_MISS (L1 data load misses)	17041434	166502
BGL_D1_S_MISS (L1 data store misses)	301437	51876
BGL_I1_MISS (L1 instruction misses)	10406	6930
BGL_D2_MISS (L2 data load misses)	149	150
BGL_DTLB_MISS (Data TLB misses)	107001	201
BGL_ITLB_MISS (Instruction TLB misses)	2283	0
BGL_FP_LOAD (Floating point loads)	66920747	198
BGL_FP_STORE (Floating point stores)	2730585	198
BGL_FP_DIV (Floating point divides)	780	0
BGL_FMA (Floating point multiply-add)	32662140	0
BGL_FLOPS (Floating point operations)	33755085	0

Fig. 2. libHPM output for the Instrumented CG code running on BGLsim.

4 Experimental Results

In order to validate BGLsim's hardware performance counters mechanism, we performed two sets of experiments. The first validation was analytical, using the ROSE [9] set of micro-benchmarks, which allows us to estimate the counts for specific metrics related with the memory hierarchy, such as loads, stores, cache hits or misses, and TLB misses. In the second set of experiments we used the serial version of the NAS Parallel Benchmarks and generated code for BlueGene/L and for an IBM Power3 machine, using the same compiler (GNU gcc and g77 version 3.2.2). We ran these programs on BGLsim and on a real Power3 system, collecting the hardware metrics with libHPM. With this approach, we were able to validate metrics that should match in both architectures, such as instructions, loads, stores, and floating point operations.

4.1 Analytical Comparison

The Blue Gene memory hierarchy parameters used in this analysis are: level-1 cache: 32 Kbytes with line size of 32 bytes; level-2 cache: 4 Mbytes with line size of 128 bytes; Linux page size: 4 Kbytes; TLB: 64 entries, fully associative.

We used seven functions from the ROSE micro-benchmark set: (1) *sequential stores*, (2) *sequential loads*, (3) *sequential loads and stores*, (4) *random loads*, (5) *multiple random loads*, (6) *matrix transposition*, and (7) *matrix multiplication*. The first five functions use a vector D with $n = 2^{20}$ double precision elements (8 Mbytes). The functions random loads and multiple random loads also use a vector, I , of 1024 integers (4 Kbytes), containing random indices in the range $(1..n)$. We also executed these two functions (and sequential loads) using a vector D_2 with twice the size of D . Finally, we used matrices of 64×64 double precision elements (32 Kbytes each) in the matrix functions. The transposition function only uses one matrix, while the multiplication uses three matrices. We called a function to evict all matrix elements between the two calls (transpose and multiply). Those functions perform the following operations (where c is a constant, and s is a double precision variable):

Sequential stores: $D[k] = c$, for $k = 1..n$.

Stores 1M double precision words (8 Mbytes) sequentially; we expect approximately 2K page faults, generating 2K TLB misses. Each page has 128 Level 1 and 32 Level 2 lines. Hence, this function should generate 256K store misses at Level 1 and 64K store misses at Level 2.

Sequential loads: $s = s + c * D[k]$, for $k = 1..n$.

Loads 1M double precision words. Again we expect approximately 2K page faults, 2K TLB misses, 256K Level 1 load misses, and 64K Level 2 misses.

Sequential loads and stores: $D[k] = D[k] + c * D[k - 1]$, for $k = 2..n$.

Loads 2M double precision words and stores 1M double precision words. Since it is loading adjacent entries and always storing one of the loaded entries, we should expect only 1 miss for every 8 loads, with a total of 256K load misses in Level 1 and no store misses. In Level 2 we expect one fourth of the misses in Level 1 (64K).

Random loads: $s = s + D[I[k]]$, $k = 1..1024$.

Access 1K random elements of the vector D . We expect approximately 1K TLB

misses and 1K load misses at level 1. D has twice the Level 2 cache size and was just traversed. Hence, accesses to its second half should hit in Level 2 cache. Since we are dealing with random elements, we would expect that approximately half of these accesses would cause misses. We also expect to have an additional 2 TLB misses, 128 Level 1 misses, and 32 Level 2 misses, due to the sequential access to I . When running this function using D_2 , we expect the same number of TLB and Level 1 misses, but approximately $3/4$ of the accesses should miss in Level 2, since D_2 is four times the size of the Level 2 cache.

Multiple random loads: $s = s + D[I[k] + 16 * j], j = 0..31, k = 1..1024$

For each entry of I it accesses 32 consecutive entries from D , with a stride of 16, which corresponds to four Level 1 cache lines and one Level 2 cache line. we expect again one Level 1 miss and $1/2$ Level 2 miss for each load, with a total of 16K Level 1 misses and 8K Level 2 misses, but only 2K TLB misses overall¹. A similar behavior from the one described for the function Random loads is expected for the Index vector I and the runs using D_2 .

Matrix transposition: This function executes the operations shown in Figure 3a, where N is 64. It performs $2 * (64^2 - 64) = 8064$ loads and the same number of stores. Since the matrix fits exactly in the Level 1 cache, we expect approximately 1 load miss for every 8 loads, and no store misses. Also, since the matrix has 32 Kbytes, we expect 9 TLB misses.

Matrix multiplication: Executes the operations shown in Figure 3b, assuming the matrix B is transposed. It loads matrices A and B 64 times, and matrix C once. Hence, it performs $129 * 64^2$ loads and 64^2 stores. We would expect 8 TLB misses for each matrix, with an extra TLB miss to compensate for not starting the arrays at a page boundary. In each iteration the outer loop traverses B completely. Since B has exactly the size of the level-1 cache, and one row of A and C are also loaded per iteration, there would be no reuse of B in subsequent iterations of the outer loop. B should generate $(64^2)/4$ Level 1 misses in each iteration of the outer loop. The rows of A and C are reused during the iterations of the two inner loops. This function should generate approximately $(64 + 1 + 1) * 64^2/4$ level-1 misses (66K). Since all three matrices fit in Level 2, we expect to have approximately $3 * 64^2/16$ Level 2 misses (768).

<pre> for (i = 0; i < N; i++) for (j = 0; j < N; j++) if (i != j) { aux = B[i][j]; B[i][j] = B[j][i]; B[j][i] = aux; } </pre>	<pre> for (i = 0; i < N; i++) for (j = 0; j < N; j++) for (k = 0; k < N; k++) C[i][j] += A[i][k]*B[j][k]; </pre>
(a)	(b)

Fig. 3. Pseudo-codes for matrix transposition (a) and matrix multiplication (b)

¹ Unless the first access to D hits the beginning of a page, a page boundary will be crossed during the span of 32 accesses, causing the second TLB miss for each entry of I .

Table 1 summarizes the expected values for each function and presents the error (difference) between observed and expected counts. In all cases the measured values are close to the expected numbers, with the largest differences being less than 2%. Also, the expected numbers for functions random loads and multiple random loads should be viewed as upper bounds, because they deal with random entries. The larger the vector D gets, the closer the measured value should be from these bounds.

Table 1. Expected counts and error (difference) between expected and observed counts for the micro-benchmark functions.

functions	Stores				Loads				Total		TLB	
	FP		L1 Misses		FP		FX		L1 Misses		L2 Misses	
	exp.	error	exp	error	exp.	error	exp	error	exp.	error	exp	error
seq stores	1M	0	256K	+1	0	+1	0	0	0	+1	64K	+8
seq loads	0	0	0	0	1M	+1	0	0	256K	+2	64K	+4
seq ld & st	1M	-1	0	0	2M	-1	0	0	256K	+2	64K	+8
random ld D	0	0	0	0	1K	0	1K	0	1152	-53	544	+8
random ld D_2	0	0	0	0	1K	0	1K	0	1152	-53	800	+3
mult rnd ld D	0	0	0	0	32K	0	1K	0	32896	-244	16416	+242
mult rnd ld D_2	0	0	0	0	32K	0	1K	0	32896	-367	24608	+119
transpose	8064	0	0	0	8064	0	0	0	1K	+298	252	+14
matrix mult	4K	0	0	0	516K	0	0	0	66K	+1153	768	+3

4.2 Validation Using a Real Machine

Table 2 shows the counts for the 8 codes in the NAS Parallel Benchmarks. (We used the serial version of these benchmarks, which run on a single processor.) The programs were compiled with the GNU gcc (IS) and g77 (all the others) compilers, with -O3 optimization flag. We ran the Class S problem sizes for the serial versions of the benchmarks on both an IBM Power3 system and on BGLsim. Due to the architectural differences in the memory hierarchy of the BlueGene/L and the Power3 systems, we restricted this comparison to metrics that should match in both architectures (floating-point operations, load, stores, and instructions).

We observe that for some benchmarks (e.g., CG and IS) the agreement between Power3 and BGLsim is excellent, while for others (e.g., EP and LU) the difference is substantial. We are investigating the source of those differences. It should be noted that, even though we are using the same version of the GNU compilers in both cases, those compilers are targeting different machines. It is possible the compilers are performing different optimizations for the different machines. As future work, we will pursue executing the exactly same object code on both targets, to factor out compiler effects.

We also compared the wall clock time for executing the 8 benchmarks on the Power3 machine and on BGLsim, running on a 600 MHz Pentium III processor. The wall clock slowdown varied from 500 (for IS) to 3000 (BT).

5 Conclusions

Performance analysis tools, such as the HPM toolkit, are becoming an important component in the methodology application programmers use to optimize their code. Central

Table 2. Counts from the NAS benchmark, when running on a Power3 and on BGLsim. Note that counts are in units of one thousand (K).

Code	System	Counters								
		Instructions	FX loads	FP loads	Total loads	FX stores	FP stores	Total stores	FP ops	FMA's
BT	Power3	603,174K			215,921K			92,015K	228,617K	13,671K
	BGLsim	530,599K	335K	157,081K	157,417K	9,993K	80,923K	90,916K	210,607K	11,475K
	difference	12.03%			27.10%			1.19%	7.88%	16.06%
CG	Power3	306,865K			98,616K			3,970K	33,762K	32,663K
	BGLsim	308,099K	31,607K	66,920K	98,528K	1,207K	2,730K	3,937K	33,755K	32,662K
	difference	0.4%			0.09%			0.83%	0.02%	0.00%
EP	Power3	5,893,189K			1,248,195K			363,302K	2,501,740K	391,379K
	BGLsim	6,267,337K	403,626K	743,087K	1,146,714K	617,816K	338,413K	956,230K	1,431,830K	350,281K
	difference	6.35%			8.13%			50.28%	42.77%	10.50%
FT	Power3	551,146K			109,204K			97,260K	160,806K	37,226K
	BGLsim	524,995K	7,585K	95,145K	102,730K	2,917K	93,361K	96,278K	147,363K	37,225K
	difference	4.74%			5.93%			1.01%	8.36%	0.00%
IS	Power3	8,767K			2,008K			1,352K		
	BGLsim	8,770K	2,007K		2,007K	1,351K		1,352K		
	difference	0.03%			0.01%			0.00%		
LU	Power3	291,671K			94,758K			27,351K	108,157K	18,560K
	BGLsim	233,270K	4,432K	66,677K	71,109K	8,860K	27,798K	30,659K	77,100K	15,470K
	difference	20.02%			24.96%			12.09%	28.79%	16.65%
MG	Power3	20,827K			5,072K			2,581K	5,517K	367K
	BGLsim	16,956K	271K	3,860K	4,131K	1,257K	1,131K	2,388K	3,516K	367K
	difference	18.59%			18.54%			7.47%	36.27%	0.00%
SP	Power3	276,419K			88,658K			28,278K	78,299K	13,884K
	BGLsim	263,325K	726K	71,592K	72,318K	2,274K	26,893K	29,168K	75,812K	13,884K
	difference	4.74%			18.43%			3.15%	3.18%	0.00%

to those performance tools is hardware performance monitoring. We are interested in providing such performance analysis tools for the new BlueGene/L supercomputer as early as possible, so that applications can be tuned for that machine.

In this paper, we have shown that an architecturally accurate instruction-set simulator, BGLsim, can be extended to provide a set of performance counters that are not present in the real machine. Those nonarchitected counters can be made available to running code, and integrated with a performance analysis library (libHPM), providing detailed performance information on a per process and per mode (user/supervisor) basis.

We validated our implementation through two sets of experiments. We first compared our results to expected values from analytical modes. We also compared results from our implementation of libHPM in BlueGene/L to results from libHPM in a real Power3 machine. Because of differences in architectural details between the BlueGene/L PowerPC 440 processor and the Power3 processor (caches size and associativity, TLB size and associativity), we restricted our comparison to values that should match in both architectures (numbers of floating-point operations, load, stores, instructions). Through both sets of experiments, we have shown coherent results from libHPM in BGLsim in some codes from the NAS Parallel Benchmarks.

Our implementation of libHPM on BGLsim has already been successfully used to investigate the performance of an MPI library for BlueGene/L [3]. We want to make it even more useful by extending it to measure and display multi-chip performance information. Given the limitations of performance counters in the real BlueGene/L hardware, we expect that libHPM in BGLsim will be useful even after hardware is available. The simultaneous access to a large number of performance counters, and consequently to

a large number of derived metrics, allows the user of BGLsim to obtain a wide range of performance data from a single application run. Most hardware implementations of performance counters limit the use to just a few simultaneous counters, forcing the user to repeat application runs with different configurations.

The BGLsim approach can be used as a tool for architectural exploration. We can verify the sensitivity of performance metrics of applications to various architectural parameters (e.g., cache size), and guide architectural decisions for future machines.

References

1. N. Adiga et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
2. D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
3. G. Almasi, C. Archer, J. G. Castaños, M. Gupta, X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. MPI on BlueGene/L: Designing an efficient general purpose messaging solution for a large cellular system. Technical report, IBM Research, 2003.
4. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-929, NASA Ames Research Center, December 1995.
5. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of Supercomputing'00*, November 2000.
6. B. Buck and J. K. Hollingsworth. Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In *Proceedings of Supercomputing'02*, November 2000.
7. J. Caubet, J. G. J. Labarta, L. DeRose, and J. Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools – WOMPAT 2001*, pages 53–67, July 2001.
8. L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Caçaval, J. G. C. nos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full circle: Simulating linux clusters on linux clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, June 2003.
9. L. DeRose. The Reveal Only Specific Events (ROSE) Micro-benchmark Set for Verification of Hardware Activity. Technical report, IBM Research, 2003.
10. L. DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
11. L. DeRose, K. Ekanadham, and J. K. Hollingsworth. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
12. C. Janssen. *The visual Profiler*. <http://aros.ca.sandia.gov/~cljanss/perf/vprofl/>. Sandia National Laboratories, 2002.
13. J. M. May. MPX: Software for Multiplexing Hardware Performance Counters in Multi-threaded Programs. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium*, April 2001.
14. M. Pettersson. *Linux X86 Performance-Monitoring Counters Driver*. <http://user.it.uu.se/~mikpe/linux/perfctr/>. Uppsala University – Sweden, 2002.
15. Research Centre Juelich GmbH. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*. <http://www.fz-juelich.de/zam/PCL/>.

Presentation and Analysis of Grid Performance Data^{*}

Norbert Podhorszki and Peter Kacsuk

MTA SZTAKI, Budapest, H-1528 P.O.Box 63, Hungary
{pnorbert,kacsuk}@sztaki.hu

Abstract. In this paper, data presentation and analysis tools, which use R-GMA, the information and monitoring system of the EU DataGrid project, are presented. The information browser is used to quickly browse among large amount of data. Pulse, a flexibly configurable tool is used to analyse and visualise performance data of grid resources and services. PROVE, a performance visualisation tool is used to analyse the performance of parallel grid applications.

1 Introduction

A fundamental requirement in the case of any computer system is that its behaviour must be observed and possibly controlled. Therefore, information must be acquired about its operation but it is a crucial issue what sort of data should be gathered and what is the way they can be obtained. For instance, to check its performance, archived data from the efficiently working system and new data about its current behaviour are needed as well as the right methods to compare them. The bigger and the more complex a system is the more data should be collected. A grid is a large system of computational resources and thus, huge amount of information is generated all the time.

Yet, there is a debate on what type of information and what data should be collected for grids at all. What status information should be recorded, what frequency of data collection is needed, what performance metrics can be defined and what performance data should be collected are questions yet to be discussed. However, for observing that high amount of information arriving from the many many sensors, the right tools should be used for the right purpose.

In this paper, three different tools are presented. All of them are developed in the EU DataGrid project [1] and as such, they get data from the monitoring and information system of this project. First, the information and monitoring system used in this project and its information browser, the R-GMA Browser are shortly introduced. Pulse, a new flexible tool for analysing and visualising information about grid resources and services is presented in the third section. It provides a framework to represent data in a general form and it consists of analysis and visualisation components. Finally, PROVE, a trace visualisation tool is presented that is used to analyse the behaviour and performance of parallel applications executed in the grid.

^{*} The work described in this paper has been supported by the following grants: EU DataGrid IST-2000-25182, Hungarian DemoGrid OMFB-01549/2001 and OTKA T042459.

2 R-GMA, a Relational Grid Monitoring Architecture

R-GMA, [2]) is being developed as a Grid Information and Monitoring System for both the grid itself and for use by applications. It is based on the GMA concept [3] from Global Grid Forum and on the relational data model. It offers a global view of the information as if each Virtual Organisation had one large relational database.

It provides a number of different Producer types with different characteristics; for example some of them support streaming of information. It also provides combined Consumer/Producers, which are able to combine information and republish it. At the heart of the system is the mediator, which for any query is able to find and connect to the best Producers to do the job.

R-GMA provides a way of using the relational data model in a Grid environment [4] but it is not a general distributed RDBMS. All the producers of information are quite independent. It is relational in the sense that Producers announce what they have to publish via an SQL CREATE TABLE statement and publish with an SQL INSERT and that Consumers use an SQL SELECT to collect the information they need. R-GMA is built using servlet technology and is being migrated rapidly to web services and specifically to fit into an OGSA (Open Grid Services Architecture, [5]) framework.

2.1 Information Browsing

The information in R-GMA can be browsed by web pages and servlets provided with the R-GMA package. The R-GMA Browser provides the list of available tables within R-GMA and detailed information about each table. A query can be composed for a selected table and R-GMA Browser returns a table of the result. It is a Java servlet that generates web pages about the rarely changing content of R-GMA, i.e., list of tables and definitions of tables. For dynamic information, the servlet provides functions. If the user is interested in the current information available within R-GMA for a selected table, a servlet function creates a Consumer that reads data from R-GMA and the servlet function generates a dynamic web page containing the table as a result.

The user can select a table, can compose a query for the table in the SQL language and also can choose among the available producers that gives information according to the selected table. If a general query is to be issued, the Mediator of R-GMA can be used to find and connect all producers and deliver all available information. The R-GMA Browser can be used to discover available information in R-GMA, sites in the grid, services and resources at the sites and status information on selected services or resources. A working R-GMA Browser can be accessed from the web page [6].

3 Pulse: Online Presentation of Multiple Streams of Data

The static nature of web technology, even using servlets, disables the user to stream data from R-GMA to the browser. The other shortcoming is that the user is not able to perform analysis or create new ways of presentations. For establishing streaming connection and performing specific analysis, another tool is available. Pulse is an efficient tool to compose an analysing chain for performance data about services or resources.

Basically, it provides a framework to represent data in a general form and it provides data analysis/presentation components to work on that data.

Pulse is able to subscribe to different data sources (not only to R-GMA producers, but to anything for which an appropriate input component is implemented), drive the incoming data through analysis/filtering components and present the result in a tabular or in a graphical representation. The computations and screen updates can be driven in two different ways. In the case of a source that is streaming data, the incoming data rate determines the updates in Pulse. In the case of a source that has to be polled for information, Pulse can poll it by a given interval.

In Fig. 1 a configuration for displaying load average and memory usage information as histograms can be seen. The top components are sensors that read the status information of the machine (under Linux). Each sensor is connected to a buffered sensor (same name with an asterisk) that stores the values. The output of the buffered sensors are displayed by histogram plotters that are put together into one window frame. The window appearing in Pulse can be seen in Fig. 2.

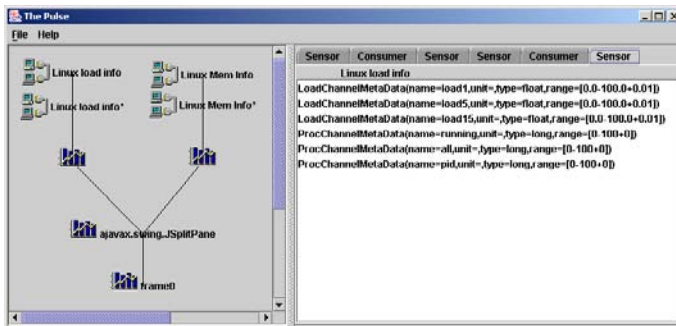


Fig. 1. Configuration of components in Pulse

3.1 Data Representation in Pulse

Pulse uses the concept of data channels. One channel has a data type so a row of data with the same type (e.g., integers) is delivered through it. There is one piece of data at a time in one channel thus, the row can be considered as data produced in time. For example, a service reporting the number of connected clients regularly generates a row of integers in time and can be considered as one channel. If a source of information provides more complex data than the basic data types (integers, floats, strings etc.), several channels has to be defined and the data has to be split into those channels. For example, a query to a relational database returns relational records. A record consists of columns with basic data type so the record can be split into channels, one column to one channel. Components in one hierarchy are connected with channels and data is flowing from the bottom to the top of the hierarchy through the channels.

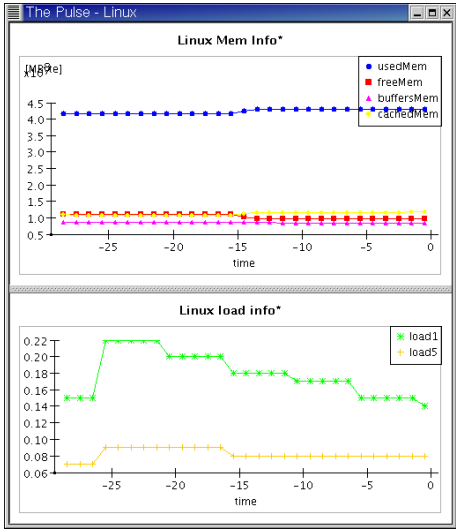


Fig. 2. Load average and memory usage histograms

3.2 Input Components

One of the basic components is the *Sensor*, the input component in Pulse. The task of such a component is to give an interface to any kind of data source. Pulse needs different channels (each typed) for different elements of source data. The sensor has to translate data coming from outside into the channel based representation of data.

An example for such sensors in Pulse is the *LinuxLoadInfo* (see Fig. 1 top left icon), which simply reads the content of the file `\proc\loadavg` under Linux and creates three data channels for the load average values: 1, 5 and 15 minutes load averages, plus three other data channels giving the current number of the running processes, all processes and the value of the largest process id (see the list of channels on the right side of the figure).

The next component in the hierarchy should be a *buffered sensor* which provides buffering of data between a sensor and a processing component. Thus, the processing components need not to care about the sensor’s capability of buffering. E.g. a histogram plotter needs data back for a given time interval. An implemented instance of this component for general use is the *SamplerBuffer*. However, the buffered sensor is just an interface and some sensors can also implement the interface and behave as a buffered sensor. Thus, there is no need to connect them to a *SamplerBuffer* but directly to other processing components. The *RGMAConsumer* sensor that makes the connection between R-GMA and Pulse is, for example, a buffered sensor.

3.3 Analysis and Presentation Components

A presentation component of data is needed to show anything in Pulse. The most basic presentation component is a tabular view of data. One row in the table shows one data set

read from a sensor. The columns represent the different channels defined by the sensor. It can be considered as a relational record. The rows contain data read from the sensor regularly.

Another presentation component is able to draw histograms of data values using the Java Analysis Studio's plotters [7], see Fig. 2. Data arriving on one channel are considered as the values for one histogram line. Channels containing numbered values should be selected to use a histogram plotter.

The regularity of update is determined by the sensor to which the presentation component is connected. The processing elements subscribe to the connected sensor and they receive a notification when the sensor's status is changed, i.e. new data arrived or some failure happened.

Presentation components are the other end of the chain of the components. They produce an output on the screen to the user. Analysis components in-between the chain can be used to modify, filter or analyse data going through the chain. The simplest (and implicit component) one is a channel selector for the histogram plotter. It simply keeps the channels selected for drawing and drops the others. It is implicit as it is created automatically below the histogram plotter in the hierarchy.

Another basic analysis components can merge data coming from different sensors (provided they have the same type), filter one channel or compute new value from several channels. Binding such components together in a hierarchy, a dataflow graph can be created that the input sensors pour data into, and one or more presentation modules provide output to the user.

It must be mentioned here that in most cases a performance parameter taken on one machine can be compared to the same parameter on another machine. There is an ongoing research in our laboratory about defining grid metrics and procedures for transforming local information into globally interpretable grid information.

3.4 Configuration of Pulse

The graph of the components should be described in a configuration file using XML. The simplest component configuration consist of one data producer component, one sampler buffer component, one tabular viewer of data and one window frame. The use of independent components allows a flexible configuration of Pulse for actual data processing and analysis needs.

New components can be implemented as Java classes and they can be referred by their names in the configuration file. A new component should implement the interfaces of Pulse according to its role. Input components should implement the Sensor API while presentation components should implement the data consumer API. Analysis components should implement both of them.

4 Application Performance Monitoring

GRM [8] is a monitoring tool for performance monitoring of parallel applications running in the grid. PROVE is a visualisation tool for GRM traces. GRM has been developed for the P-GRADE [9] graphical parallel programming environment and it has been used

for on-line monitoring of P-GRADE applications in cluster environment. GRM collects trace data from all machines where the application is running any time when requested and the trace is visualised by the PROVE performance visualisation tool of P-GRADE.

These tools are the basis in the development of a grid application monitor that supports on-line monitoring and visualisation of parallel applications in the grid. The current version of GRM uses R-GMA to deliver generated trace data to the visualisation tool. The DataGrid software enables the users to execute an MPI application (as a parallel program) on one site of the grid but multi-site execution is not supported. GRM and R-GMA can be used for monitoring of MPI applications running on one site. Thus, the problem of the different timestamps of different sites is not an issue here.

PROVE is an on-line trace visualisation tool to present traces of parallel programs (see Fig. 3) collected by GRM. Its main purpose is to show a time-space diagram from the trace but it also generates several statistics from the trace that help to discover the performance problems, e.g. Gantt chart, communication statistics among the processes/hosts and detailed run-time statistics for the different blocks in the application process.

To enable monitoring of a grid application, the user should first instrument the application with trace generation functions. There is no automatic tool for instrumentation therefore, the source code of the application should be instrumented by hand. GRM provides an instrumentation API and library for tracing which are available for C/C++ and Fortran. The basic instrumentation functions are for the start and exit, send and receive, multicast, block begin and end events (giving the user the option to decide what a block means in her application code, inserting the begin and end statements by hand). However, more general tracing is possible by *user defined events*. For this purpose, first a format string of a new user event should be defined, similarly to C printf format strings. Then the predefined event format can be used for trace event generation, always passing the arguments only.

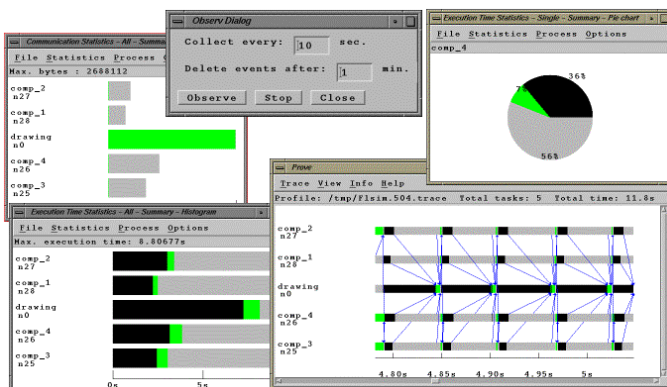


Fig. 3. Visualisation of trace and statistics in PROVE

The instrumentation functions automatically connect to R-GMA at the start of the processes and trace events are published to R-GMA. Thus, instrumented application

processes act as Producers of R-GMA. GRM's main monitor acts as a Consumer of R-GMA, looking for trace data and receiving it from R-GMA. The delivery of data from the machines of the running processes to the collection host is the task of R-GMA. R-GMA is using several servlets and buffers to deliver the trace data to the consumers. There is a local buffer in the application process itself that can be used to temporarily store data if a large amount of trace is generated fast. The processes are connected to ProducerServlets that are further connected to ConsumerServlets. Both kind of servlets create distinguished buffers for each Producer/Consumer that connect to them. GRM's main monitor receives the whole trace data in one single stream at the end.

The distinction between the traces of different applications is made by a unique id for each application. This id works as a key in the relational database schema and one instance of GRM is looking for one application with a given id/key.

After the application is submitted and GRM's main monitor is started, the main monitor connects R-GMA immediately and subscribes for traces with the id of the application. When R-GMA gives a positive response GRM starts continuously reading trace from R-GMA and the execution behaviour can be followed by the user on the displays of PROVE.

5 Related Work

R-GMA is yet deployed within the DataGrid project only. Other grid projects are mostly based on MDS [10], the LDAP based information system of Globus but, e.g., the GridLab project is developing their own MDS information system and a new monitoring system [11].

To present the collected information, several different tools are used in different projects. Java Analysis Studio [7] has been developed to analyse and visualise large amount of scientific data stored in various formats for High Energy Physics but its features allow it to be used for more general use. Its plotting features are separated into a stand-alone package and this package is used in Pulse to create histograms. Nagios [12], a distributed host and service monitor running under Linux provides graphical plots to present monitoring information in a web browser.

Similarly to the case of GRM, the OMIS [13] on-line monitoring interface, developed for clusters and supercomputers is the basis for a grid application monitoring system within the CrossGrid project.

Netlogger is used for monitoring distributed applications in the grid rather than for parallel programs. Its time-space visualisation display concept is orthogonal to PROVE. In the vertical axis different types of events are defined while in PROVE the processes of the parallel programs are presented. Netlogger can be used for finding performance/behaviour problems in a communicating group of distributed applications/services while PROVE for a parallel program that is heavily communicating within itself. Netlogger, PROVE and other tools like Network Weather Service and Autopilot has been compared in the beginning of the DataGrid project in detail, see [14].

6 Conclusion

R-GMA is a relational Grid Monitoring Architecture delivering the information generated by the components in the grid. The large amount of monitoring information can be presented with different techniques. From simple web browser based queries to on-line monitoring of parallel applications, there are different tools available to get information from R-GMA and present it in an appropriate form. The simplest tool, the R-GMA Browser is used to search among the available information and look at the static or slowly changing behaviour of the grid infrastructure. Pulse can be used to build an analysis chain to process dynamic information about the services and the infrastructure on the fly and to create graphical plots to show the result of the analysis. GRM/PROVE is a parallel application monitoring toolset that is now connected to the grid. It can be used for on-line monitoring and performance analysis of parallel applications running in the grid environment.

References

1. EU DataGrid Project Home Page: <http://www.eu-datagrid.org>
2. S. Fisher et al. R-GMA: A Relational Grid Information and Monitoring System. 2nd Cracow Grid Workshop, Cracow, Poland, 2002
3. B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. GGF Informational Document, GFD-I.7, GGF, 2001, URL: <http://www.gridforum.org/Documents/GFD/GFD-I.7.pdf>
4. Steve Fisher. Relational Model for Information and Monitoring. GGF Technical Report GWDPerf-7-1, 2001. URL: <http://www.didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-7-1.pdf>
5. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, and P. Vanderbilt. Grid service specification. GGF Draft Document, 2002, URL: <http://www.gridforum.org/meetings/ggf6/ggf6-wg-papers/draft-ggf-ogsi-gridservice-04.2002-10-04.pdf>
6. R-GMA Browser. URL: <http://hepunix.rl.ac.uk/edg/wp3/>
7. Java Analysis Studio. URL: <http://www.sldnt.slac.stanford.edu/jas/>
8. Z. Balaton, P. Kacsuk, N. Podhorszki, F. Vajda. From Cluster Monitoring to Grid Monitoring based on GRM. Proc. of EuroPar'2001, Manchester, pp. 874–881
9. P. Kacsuk. Visual Parallel Programming on SGI Machines. Proc. of the SGI Users' Conference, Cracow, Poland, pp. 37–56, 2000.
10. K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. Grid Information Services for Distributed Resource Sharing. Proc. of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.
11. G. Gombás and Z. Balaton. A Flexible Multi-level Grid Monitoring Architecture. 1st European Across Grids Conference, Universidad de Santiago de Compostela, Spain, Feb. 2003
12. Nagios network monitoring tool. URL: <http://www.nagios.org>
13. T. Ludwig and R. Wismüller. OMIS 2.0 – A Universal Interface for Monitoring Systems. In M. Bubak, J. Dongarra, and J. Wasniewski, eds., Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 4th European PVM/MPI Users' Group Meeting, LNCS vol. 1332, pp. 267–276, Cracow, Poland, 1997. Springer Verlag.
14. Information and Monitoring: Current Technology. DataGrid Deliverable DataGrid-03-D3.1. URL: <https://edms.cern.ch/document/332476/2-0>

Distributed Application Monitoring for Clustered SMP Architectures^{*}

Karl F rlinger and Michael Gerndt

Institut f r Informatik,
Lehrstuhl f r Rechnertechnik und Rechnerorganisation
Technische Universit t M nchen
{fuerling, gerndt}@in.tum.de

Abstract. Performance analysis for terascale computing requires a combination of new concepts including distribution, on-line processing and automation. As a foundation for tools realizing these concepts, we present a distributed monitoring approach for clustered SMP architectures that tries to minimize the perturbation of the target application while retaining flexibility with respect to filtering and processing of performance data. We achieve this goal by dividing the monitor in a passive monitoring library linked to the application and an active component called runtime information producer (RIP) that provides performance data (metric- and event based) for individual nodes. Instead of adding an additional layer in the monitoring system that integrates performance data from the individual RIPs we include a directory service as a third component in our approach. Querying this directory service, tools discover which RIPs provide the data they need.

1 Introduction

Performance analysis of applications in terascale computing requires a combination of new concepts to cope with the difficulties that arise with thousands of processors and gigabytes of performance data. The classic approach of collecting the performance data in a central file and running a post-mortem analysis tool (e.g., Vampir [12]) is hardly feasible in this context.

The new concepts include the *distribution* of the performance analysis system and *on-line processing*, together enabling the analysis of performance data close to its origin in the target application. *Automation* seeks to alleviate the user of the burden of locating performance problems in the interaction of thousands of processors hidden behind massive amounts of data.

Before we present our monitoring approach in detail in Section 4, we first give a short overview of existing monitoring solutions in Section 2. Then we briefly describe our design of a performance tool realizing the aforementioned concepts for which the presented monitoring approach forms the basis in Section 3.

^{*} Part of this work is funded by the Competence Network for High-Performance Computing in Bavaria KONWIHR (<http://konwihr.in.tum.de>) and by the European Commission via the APART working group (<http://www.fz-juelich.de/apart>).

2 Related Work

As an overview of the available monitoring solutions we present three classes of approaches and describe one representative for each.

The *classic* approach to performance analysis relies on a monitoring library that is linked to the application writing event records to a tracefile that is analyzed after the application has terminated. A good representative here is Vampir-trace, the monitoring library that comes with Vampir [12], a powerful visualization tool that is available for many platforms. Originally limited to pure message passing applications, work is now in progress to support hybrid OpenMP/MPI programming in the library as well as in the visualizer. The current version does not support performance counters. For performance reasons, the trace data is held in local memory and dumped to a file at the end of the target application or when the buffer is full. The library supports flexible filtering based on the event type through a configuration file, but this cannot be changed at runtime. Additionally, the analysis is a pure post-mortem approach suffering of the massive amounts of performance data generated by thousands of processors.

An interesting approach to minimize the monitoring overhead and to limit the amount of data generated is the *dynamic instrumentation* approach of DPCL [4] which is based on Dyninst [5]. Executable instrumentation code patches ('probes') can be inserted to and removed from a target application at runtime by calling functions of the API of the DPCL C++ class library. DPCL translates these calls to requests that are sent to DPCL daemons that attach themselves to the target application processes and install or remove the probes. The probes within the target application send data to the DPCL daemon which forwards the data to the analysis tool, triggering the appropriate callback routine. The advantage of the dynamic instrumentation approach is that the monitoring overhead can be limited to its absolute minimum since probes can be removed from the target application as soon as the desired information has been retrieved.

In the context of *Grid* computing the existing monitoring approaches have been found to be unsuitable and several projects for grid monitoring were initiated. As an example, OCM-G [1], the grid-enabled OMIS [11] compliant monitor, is an autonomous and distributed grid application monitoring system currently being developed in the CrossGrid [3] project. The OCM-G features transient as well as permanent components. The transient component is called the local monitor and is embedded in the address space of the application. The persistent component consists of one service manager per grid site. OCM-G supports selective (activated or deactivated) monitoring to minimize overhead and perturbation and to limit the amount of monitored data to the really relevant parts. It also supports higher-level performance properties and application defined metrics and it allows the manipulation of the executable (e.g., stopping a thread) besides pure monitoring). Furthermore it includes infra-structural data like the status of the network connections, and as it is designed as a grid-wide permanent service, it includes support for monitoring of several applications by several tools and several users simultaneously.

3 The Peridot Project

Each of the monitoring solutions presented in Section 2 was considered unsuitable for the Peridot project, where we plan to implement a distributed automated on-line performance analysis system, primarily for the Hitachi SR8000 system installed at the Leibnitz-Rechenzentrum in Munich and similar clustered SMP architectures. Here we provide an overview of the target architecture as then briefly introduce the design of our performance analysis system.

3.1 The Hitachi SR8000

The Hitachi SR8000 at the Leibnitz-Rechenzentrum (LRZ) in Munich was the first teraflop computer in Europe. Initially installed in the first quarter of 2000, after a substantial upgrade in the beginning of 2002 the machine now delivers 2.0 TFlops peak performance [8].

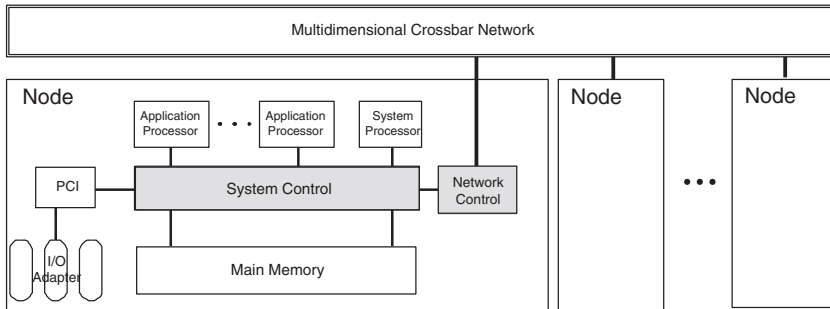


Fig. 1. The Hitachi SR8000 is a clustered SMP system with eight application processors plus one system processor per node connected to a multidimensional crossbar network.

The architecture of the Hitachi is schematically depicted in Fig. 1. It is a clustered SMP design, each node consists of eight application processors plus one system processor sharing a common memory through a crossbar switch. The processor is based on the RS6000, substantially redesigned it includes a large register set and pre-load and pre-fetch operations to enable a continuous data stream from main memory into cache or the floating point registers, respectively. Each node has a storage controller that supports efficient access to memory on remote nodes (remote direct memory access, RDMA) and takes care of cache coherency.

The SR8000 can be used for pure message passing (inter- and intra-node) as well as for hybrid shared memory/message passing programming. OpenMP and COMPAS (COoperative MicroProcessors in single Address Space) are supported for shared memory programming within a single node.

The processors have support for eight fixed performance counters: instruction and data TLB miss count, instruction and data cache miss count, number

of memory access and floating point instructions, total number of executed instructions and number of cycles.

3.2 Automated Distributed Online Analysis

This section outlines the general architecture of the analysis system currently under development within the Perdiot project, for details consult [6].

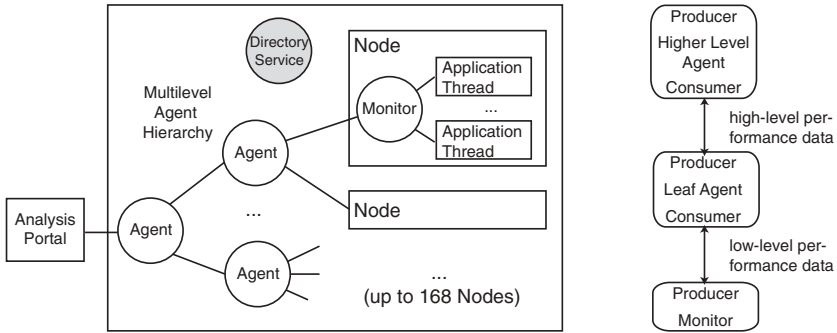


Fig. 2. Our performance analysis system for the Hitachi SR8000 consists of a set of analysis agents arranged in a hierarchy (left). The interactions between the agents at various levels and the monitor can be regarded as producer-consumer relations (right).

Our distributed performance analysis system is composed of a set of analysis agents (left part of Fig. 2), that cooperate in the detection of performance properties and problems. The agents are logically arranged into a hierarchy and each agent is responsible for the detection of performance problems related to its level in the hierarchy. Specifically, the leaf agents (the lowest level of the hierarchy) are responsible for the collection and analysis of performance data from one or more nodes, which they request from monitors.

The detection of performance problems is based on an automatic evaluation of performance *properties* specified in the APART specification language [9,10]. Higher level agents combine properties detected by lower level agents and they assign subtasks for the evaluation for a global property to the appropriate lower level agents. The degree of autonomy of the agents at the various levels is not yet determined in our design, first versions of our system will probably follow a more conservative global steering approach.

The interaction among the agents and between leaf agent and monitor can be regarded as a producer-consumer relation (right part of Fig. 2). On the lowest level, the monitors generate the performance data and leaf agents request and receive this data. On higher levels, agents act as consumers as well as producers of refined (higher-grade) performance data in the form of (partially evaluated) performance properties. The monitors and the agents register their producer

and consumer parts in a central directory service together with the type of performance data they are able to produce or consume.

4 Distributed Monitoring

As mentioned in Section 3.2, the monitor acts as a producer of performance data in our approach. This implies an *active* implementation where the monitor is waiting to serve requests from agents. Since we want to avoid the overhead and perturbation of the target application resulting from a monitoring library spawning its own thread or process, we split the monitor in two components coupled by a ring (circular) buffer.

The first (passive) component is the *monitoring library* linked to the application and the second (active) component is the *runtime information producer (RIP)*. This separation keeps the monitoring overhead and the perturbation of the target application small while flexibility with respect to filtering and pre-processing of monitoring data can be retained.

4.1 Monitoring Library

The monitoring library is completely passive, i.e., it executes only through calls of the instrumented application.

Instrumentation. On the Hitachi we can integrate three instrumentation approaches. OpenMP regions are instrumented by OPARI [2] (a source-to-source instrumenter), while MPI calls are captured using the usual MPI wrapper technique. Additionally, functions and procedures are instrumented by Hitachi's C/C++ and Fortran compilers using the `-Xfuncmonitor` option. This instructs the compiler to call a procedure of our monitoring library on each function entry and exit. A string specifying the location of the function (source file name plus function name) as well as the line number of the beginning of the function in the source code are passed as arguments [7].

For each *event* (i.e., call to one of the procedures of the monitoring library) an event packet is assembled and stored in a ring buffer. The event packet consists of a header that specifies the size of the packet, the type of the event and a sequence number. The body of the packet contains a wall-clock time stamp and the current values of the Hitachi's performance counters. Additionally, event-type specific data is stored in the body. For OpenMP regions this includes the name of the OpenMP construct affected and its location in the source code (file name and line numbers denoting beginning and end of the construct).

Ring Buffer. The ring buffer connects the two components of our monitoring approach. Data written by the monitoring library is read by runtime information producers (RIPs). A separate ring buffer is allocated by the monitoring library for each OpenMP thread, avoiding the overhead associated with locking a single

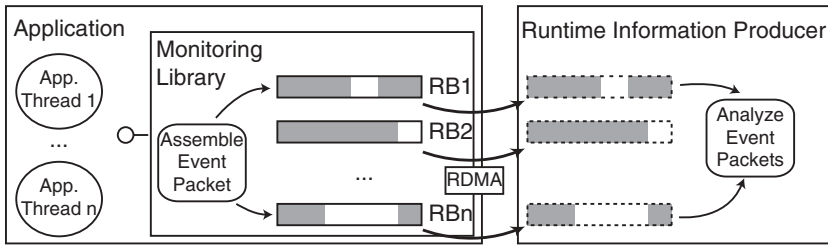


Fig. 3. The monitoring library writes event packets to ring buffers (one per application thread). In the RDMA setting on the Hitachi SR8000, the runtime information producer works on a copy transferred into its own address space.

buffer per process for several threads. As the buffer is organized as a ring (circular) buffer and the packets are of varying length (due to different lengths for file and function names), a new event packet may overwrite one or more older packets.

In order to process the event packet, a RIP must acquire access to ring buffers embedded in the monitoring library. This can be organized in two ways. The first approach is to assign one RIP per application node which is responsible for all ring buffers of that node. A RIP can then simply map these buffers into its own virtual address spaces, provided they are allocated in shared memory segments (for example using System V `shmget()` and `shmat()`). Although this approach is feasible for any SMP machine, it can lead to artificial load imbalance since one processor per node must execute the RIP in addition to its application load.

To circumvent this problem, it would be convenient to take advantage of the system processor on the Hitachi. However, this special processor is used internally by the operating system and special (root) privileges are required to execute programs there. Currently we are working together with the Leibnitz-Rechenzentrum to investigate the requirements concerning the utilization of the system processor for our analysis system, in order not to disturb normal operation of the operating system.

The second approach is to use the remote direct memory access (RDMA) facility of the Hitachi, allowing the RIP to execute on any processor of the machine. The RIP transfers a *copy* of the ring buffers of a node into its own address space and works on this copy when analyzing events (Fig. 3). As this does not require intervention of the processors of the remote node (holding the ring buffer and executing the target application), this approach is very efficient and does not lead to artificial load imbalance. However, it requires one or more nodes being set aside for the performance analysis system.

In both approaches, the buffers must be locked by the monitoring library as well as the RIPs for write or read access. Fortunately the Hitachi supports efficient locks across nodes. Note that the original ring buffer is never emptied in RDMA case, since the RIP always works on a copy of the original buffer.

4.2 Runtime Information Producer

A runtime information producer (RIP) forms the active part of our monitoring approach. Its task is to provide the analysis agents of our system with the required performance data. The functionality and the data are accessed through a monitoring request interface (MRI) implemented by the RIP. Current efforts in the APART working group to standardize the MRI will enable other tools to use the functionality of our monitors as well.

Node-level Monitoring. A runtime information producer (RIP) is responsible for reading and processing event packets from ring buffers of its assigned application nodes. On startup, it queries the directory service that is part of our performance analysis system for the information required to access the memory holding the buffer. In the shared memory case, this is the key and the size of the shared memory segment, in the RDMA case, the coordinates of the affected node are needed additionally. Subsequently, in regular intervals (one second, say) the RIP acquires access to the buffers and processes them.

If a ring buffer is a copy transferred using RDMA, it might contain packets that were already present in a previous copy. Checking the sequence numbers, the RIP recognizes new packets, from which it then creates a representation of the monitored performance data as a collection of C++ objects which lend themselves to efficient and straightforward post-processing.

High-level Monitoring. Analyzing certain application behavior, notably for MPI applications, requires the collection of data from several nodes. For example, to analyze message transit time we need to monitor matching `MPI_Sends` and `MPI_Recv`s. Hence, we need to integrate data from several nodes, generally not covered by the same RIP.

In our approach we deliberately do not provide this inter-node data at the monitoring level. Instead we focus on efficient monitoring of single nodes at minimal cost and RIPs can be queried not only for aggregated data, but also for single events. Hence, a tool which requires inter-node event data registers with the RIPs responsible for the respective nodes and is then able to retrieve the desired information.

We feel that this ‘flat’ monitoring is advantageous to a hierarchical approach (e.g., OCM-G) since the latter leads to considerable complexity in the monitor to distribute the performance requests and to integrate the results. Note that it is still possible to provide similar distributed monitoring functionality at the tools level. For example an ‘adapter’ can be implemented that provides monitoring functionality for the whole machine, handling the distribution and integration of requests and responses, respectively. The directory service would be consulted to discover the location of the RIPs responsible for the individual nodes assigned to our target application.

5 Conclusion

We have presented our approach for monitoring clustered SMP architectures with the goal of minimizing overhead while enabling flexible on-line analysis. This was achieved by separating the required active component from the monitoring library into a distinct component, called runtime information producer (RIP). A ring buffer allocated in a shared memory segment couples the monitoring library and the RIP. To efficiently access the ring buffer we can take advantage of service processor and the RDMA facility of the Hitachi SR8000, our primary target machine.

The third component of our monitoring approach is the directory service used by the RIP to retrieve the required information to access the ring buffers. Additionally, RIPs publish the type of performance data they provide in the directory service. Consumers, such as the agents of our distributed analysis system can then locate and query the RIPs to access the desired performance data.

References

1. Bartosz Balis, Marian Bubak, Wlodzimierz Funika, Tomasz Szepieniec, and Roland Wism ller. Monitoring of Interactive Grid Applications. To appear in Proceedings of Dagstuhl Seminar 02341 on Performance Analysis and Distributed Computing. Kluwer Academic Publishers. 2003.
2. Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting. In EWOMP'01 Third European Workshop on OpenMP, Sept. 2001.
3. CrossGrid Project: <http://www.eu-crossgrid.org>
4. Dynamic Probe Class Library. <http://oss.software.ibm.com/dpctl/>
5. Dyninst. An Application Program Interface (API) for Runtime Code Generation. <http://www.dyninst.org>
6. Michael Gerndt and Karl F rlinger. Towards Automatic Performance Analysis for Large Scale Systems. At the 10th International Workshop on Compilers for Parallel Computers (CPC 2003). Amsterdam, The Netherlands. January 2003.
7. The Hitachi Performance Monitor Function (Hitachi System Documentation).
8. The Top 500 Supercomputer Sites. <http://www.top500.org>
9. T. Fahringer, M. Gerndt, G. Riley, and J.L. Tr ff. Formalizing OpenMP Performance Properties with the APART Specification Language (ASL), International Workshop on OpenMP: Experiences and Implementation, Lecture Notes in Computer Science, Springer Verlag, Tokyo, Japan, pp. 428–439, October 2000.
10. T. Fahringer, M. Gerndt, G. Riley, and J.L. Tr ff. Knowledge Specification for Automatic Performance Analysis. APART Technical Report. <http://www.fz-juelich.de/apart>. 2001.
11. T. Ludwig, R. Wism ller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen Vol 9, LRR-TUM Research Report Series, (1997). <http://www.bode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz>
12. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. Supercomputer, 12(1):69–80, January 1996. <http://www.pallas.com/e/products/vampir/index.htm>

An Emulation System for Predicting Master/Slave Program Performance^{*}

Yasuharu Mizutani, Fumihiko Ino, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{mizutani, ino, hagihara}@ist.osaka-u.ac.jp

Abstract. This paper describes the design and implementation of a performance prediction system, named Master/Slave Emulator (MSE) for message passing programs. The aim of MSE is to assist the performance study of master/slave (M/S) programs on clusters of PCs. MSE provides accurate predictions, so that enables us to analyze the performance of M/S programs.

1 Introduction

Master/slave (M/S) paradigm is an adaptive programming paradigm for computing on heterogeneous systems such as clusters and grids. This paradigm provides a dynamic load-balancing mechanism, so that allows us to develop high performance programs on such heterogeneous systems. However, M/S program performance can decrease when the master is responsible for excessive slaves. Therefore, performance prediction is useful to investigate the optimal number of slaves as well as to study the performance of load-balancing strategies based on the M/S paradigm.

One approach for predicting the performance of M/S programs is to use a theoretical analysis specific to the M/S paradigm. Although this approach is helpful in determining the number of slaves, it requires strict assumptions like that all tasks assigned by the master must be of same-size [1] and that workloads must be a representative distribution such as an exponential distribution.

Another approach is to use a realistic parallel computational model such as the LogP [2] family of models [3,4,5]. LogGP [3] has been validated with a wavefront application within 7% error [6]. On the other hand, to improve the accuracy of the model, LogGPS [4] and LoGPC [5] capture synchronous messages and network contention, respectively. Thus, many works have validated the accuracy of the LogP family and demonstrated accurate performance predictions with parallel programs. However, few works except CLUE [7] have validated these models with M/S programs. CLUE predicts the performance of an M/S program by executing the programs and modeling communication like LogP. Although CLUE shows accurate predictions on a 5-node SMP cluster, it leaves unclear whether its prediction is accurate enough to investigate the optimal number of slaves as well as to analyze the behavior of sophisticated M/S programs.

^{*} This work was partly supported by the Foundation for C&C Promotion, Network Development Laboratories, NEC, JSPS Research for the Future Program JSPS-RFTF99I00903, and JSPS Grant-in-Aid for Young Scientists (B)(15700030), for Scientific Research (C)(2)(14580374).

The goal of this work is to develop a performance prediction system for the performance study of M/S programs on clusters. Our target programs include not only M/S programs where workloads can be dynamically determined but also M/S-based load-balancing strategies where the master can be dynamically generated, migrated, and structured in a hierarchical manner. To achieve this goal, we have developed an emulation system, named Master/Slave Emulator (MSE) for Message Passing Interface (MPI) programs, by incorporating following three approaches: (1) a dynamic behavior emulation by direct execution; (2) a low-overhead prediction by a realistic parallel computational model; and (3) a saturation point modeling by adapting the used model to the M/S paradigm.

2 Design Aspects for Predicting Master/Slave Program Performance

2.1 Dynamic Behavior Emulation for Performance Study

To provide an accurate prediction for M/S-based load-balancing strategies, we have to capture the dynamic behavior of programs influenced by the strategies.

To capture this, we currently execute the target program without omitting any portion of program code. Although a few performance prediction systems omit some code by compiler analysis [8], we avoid this omission since sophisticated M/S programs contain a code for load balancing, and distinguishing the code automatically from given programs brings hard issues. For example, such programs perform task queueing as well as exchange workload information in addition to the original calculation and communication. Therefore, we have decided to use a run-time emulation approach to capture the dynamic behavior of the target programs.

2.2 Low-Overhead Prediction for Accurate Performance Prediction

In M/S programs, the master's performance can determine their overall performance. Therefore, to realize an accurate prediction of M/S program performance, we have to predict the master's behavior in precise. One approach for this is to simulate network in detail, however, detailed simulation requires an unpredictable large amount of computational effort, which perturbs the master's behavior. To avoid this perturbation, we have decided to use a low-overhead approach, or simulation by a realistic model such as LogGPS. LogGPS is an extension of LogGP and abstracts the communication of messages through the use of seven parameters: L , o , g , G , P , s , and S [4].

Note here that LogGPS captures no network contention. Although other contention-based models such as LoGPC can be used, we have selected contention-less LogGPS since it has shown to be accurate on our cluster, as presented later in Section 3. Furthermore, LogGPS has two advantages to LoGPC when network contention has little effect on the overall performance. First, LogGPS is simpler than LoGPC, so that causes less perturbation on the master. Second, the LogGPS parameters depend only on the target hardware while the LoGPC parameters depend on both the target hardware and software. For example, LoGPC requires application-specific parameters such as the message injection rate and the fraction of messages determined for every pair of processors. Deriving

Table 1. RTT_P : Round trip time for 1-byte message on Fast Ethernet with P processors.

MPI implementation	RTT_P (μs)		
	$P = 2$	$P = 16$	$P = 64$
MPICH	144.0	152.1	185.8
LAM	125.8	147.2	194.8
MPICH-SCore	95.4	97.6	99.2

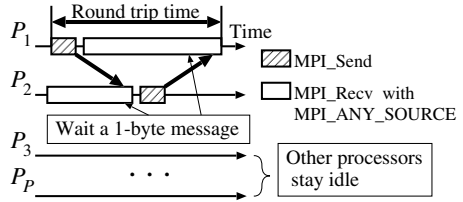


Fig. 1. Measurement of round trip time

these parameter values is complicated for some sophisticated programs [9] since such programs dynamically generate and migrate the master.

2.3 Saturation Point Modeling for Scalability Analysis

From our preliminary experiments, we found that the LogP family fail to produce the performance curve as shown later in Fig. 2. This failure is a critical problem especially when we analyze the scalability of M/S program performance. In the following, we explain why this failure happens and how we addressed this problem.

Table 1 shows the round trip time (RTT) measured on a Fast Ethernet network by using three MPI implementations. Note here that $P - 2$ of P processors stay idle while the remaining two processors exchange a 1-byte message as shown in Fig. 1. In Table 1, although two common processors transmit messages for all P , RTT increases with P . This increase can strongly effect on the performance of M/S programs since the performance can be determined by the RTT between the master and the slaves. Therefore, performance prediction for P processors requires the parameter values derived from the same P .

However, smaller P is desirable for the scalability analysis. To estimate the RTT on a large P from a small P , we have adapted LogGPS by representing the overhead, $o = o' + Ok$, as a linear function of P : $o = o'_a + o'_b P + Ok$, where k is the message length; o' and O are LogGPS constants [4]; and o'_a and o'_b are additional constants derived from two values of o' measured on a pair of a small P (described in Section 3).

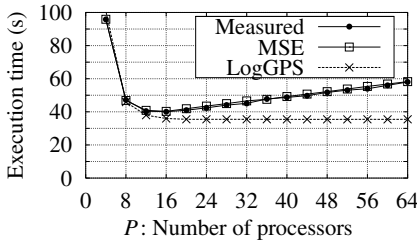
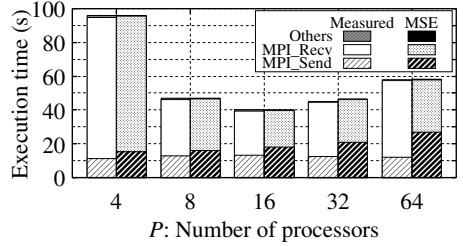
Our linear representation for the overhead is appropriate from the following reason. The increase of RTT has no relation to network contention since the message length and the communication pattern are fixed for all P during the RTT measurement. It is due to the increase of the overhead required for retrieving arrival messages. For example, MPICH's MPI_Recv calls a select system call and Linux/FreeBSD's select retrieves the arrival-state sockets by the linear search. Since the number of sockets is equal to $P - 1$, o increases with P , and thereby RTT increases with P .

3 Experimentation

To validate our MSE on its prediction accuracy, we applied it to an M/S program: a parallel mandelbrot set explorer (MASE) for fractal visualization which solves a set of independent tasks whose workloads are dynamically determined. A task of MASE corresponds to test whether a point on the complex plane is included in the mandelbrot

Table 2. LogGPS parameter values for MPICH on Fast Ethernet, derived from 8 processors. P and k represent the number of processors and the length for messages, respectively.

	L (μ s)	G (μ s)	o for MPI_Send (μ s)	o for MPI_Recv (μ s)
MSE	50.0	0.0268	$12.1 + 0.182P + 0.0708k$	$12.1 + 0.182P + 0.0722k$
LogGPS			13.0 $+ 0.0706k$	13.0 $+ 0.0723k$

**Fig. 2.** Measured and predicted execution time for single master implementation (SI).**Fig. 3.** Breakdown analysis of master's execution time (SI).

set. We implemented them in three variations: single (SI), multiple (ML), and dynamic (DY) master implementations (described later in Section 3.2).

We used a 64-node cluster with Pentium III 1GHz processors for experiments. Each node in the cluster connects to Myrinet and Fast Ethernet switches, yielding full-duplex bandwidth of 2 Gb/s and 100 Mb/s, respectively.

In the experiments, we emulated MPICH programs on Fast Ethernet by executing MPICH-SCore [10] programs on Myrinet with the same P processors. Note here that MSE currently requires a high-speed network for emulation and this network must be faster than the target network. This is due to the lack of a virtual time mechanism [7], however, this is no critical problem to validate the prediction accuracy of MSE.

Table 2 shows the LogGPS parameter values for MPICH on Ethernet, derived from 8 processors. Here, we only show the values for asynchronous messages since MASE transmitted no synchronous message. Besides, we disregarded g since g is encapsulated in o on current machines [4,5]. To derive o'_a and o'_b , we first derived two LogGPS constants, o' on $P = 2$ and $P = 8$, in accordance with [4], then solved a pair of equations in o'_a and o'_b variables. As shown in Table 2, the coefficient of P has significant influence on o . For example, for $k = 1$, o doubles when P increases from 2 to 66 processors.

3.1 Validating Prediction Accuracy

To validate the saturation point modeling of MSE, we predicted the performance with the minimum task granularity, where the master became a bottleneck. Fig. 2 shows the measured and the predicted execution time for the SI implementation. Our MSE shows an accurate prediction within 3% error. It also shows the performance saturation point in precise, so that the optimal P obtained by MSE agrees with the measured results, $P = 16$. However, LogGPS fails to show the performance curve, so that drops its accuracy as P increases. This failure can be explained as follows. Since MASE performs 1,048,576 round trip communications between the master and the slaves, the master takes at least

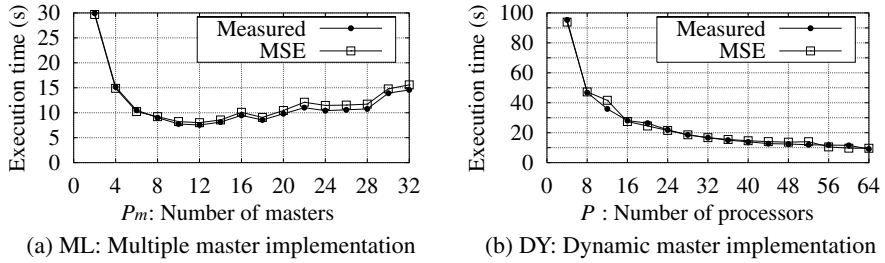


Fig. 4. Measured and predicted execution time for MASE.

$1,048,576 \cdot 20 \mu s$. Substituting $P = 8$ and $P = 64$ to o (see Table 2) and subtracting them gives $1,048,576 \cdot 2 \cdot 0.182 \cdot (64 - 8) / 10^6 = 21.4$ s, or the contribution of our linear overhead when $P = 64$. This value is close to 22.5 s, or the difference between the measured and the LogGPS time when $P = 64$. Thus, our linear overhead representation is necessary for the scalability analysis for M/S programs since it can significantly effect on the performance on large P .

Fig. 3 shows the breakdown of the master's execution time. We can see that MSE estimates the total time in precise for all P , however, with the increase of P , it estimates longer time for MPI_Send and shorter time for MPI_Recv. This discrepancy is due to our simplified model, which assumes that both the send and the receipt overheads require the same o' . That is, to simplify our model, we have made the assumption, but the actual send overhead contains no cost proportional to P . Our assumption works fine for M/S programs where assigned tasks and their results are unredirected. However, for M/S programs where the master migrates before receiving the results of assigned tasks, the total number of the master's MPI_Send differs to that of its MPI_Recv, so that we have to avoid this simplification. One solution for this is to distinguish o'_a and o'_b between the send and the receipt overheads.

3.2 Validating Dynamic Behavior Emulation

We applied our MSE to two sophisticated variations of MASE. One is the ML implementation, in which each of P_m masters has P/P_m responsible slaves, where $2 \leq P_m \leq P/2$. The other is the DY implementation, in which the number of masters can change during program execution. In DY, every master can split its responsible tasks and slaves in two groups and assign one group to a responsible slave. The assigned slave then becomes the master of the half slave and works as a secondary master until it completes the assigned tasks. The splitting and merging of the master are triggered by the statistics of the slaves' wait time from the request of a task until its assignment.

Fig. 4(a) shows the results for ML on $P = 64$. In Fig. 4(a), although the measured time appears as an irregular graph, our MSE presents an accurate prediction within 10% error. This good accuracy comes from our emulation approach since it realizes almost the same workload distributions as the original. Actually, the measured time increases when $P = 16$ due to the splitting of tasks with imbalanced workloads.

Fig. 4(b) shows the results for DY. Although DY behaves in a more complex manner, MSE shows a similar performance to the measured results. From the above study, we

can conclude that ML gives the best performance among the three implementations if we can detect the optimal number of P_m . Thus, MSE successfully shows the irregular performance of M/S programs, which is not easy for static and post-mortem approaches.

4 Conclusion

We have presented the design and implementation of a performance prediction system, named MSE, which aims at assisting the performance study of MPI programs based on the M/S paradigm. Our MSE employs a best effort emulation method and a realistic parallel computational model, the LogGPS model. It also adapts LogGPS to the M/S paradigm by capturing the overhead required for retrieving messages arrived from slaves. MSE has demonstrated an accurate prediction within 10% error for a 64-node cluster. It also has shown that our linear overhead representation is necessary for M/S programs, especially on a large number of processors.

References

1. Beaumont, O., Legrand, A., Robert, Y.: The master-slave paradigm with heterogeneous processors. In: Proc. 3rd IEEE Int'l Conf. on Cluster Computing. (2001) 419–426
2. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proc. 4th ACM SIGPLAN Symp. on Principles Practice of Parallel Programming. (1993) 1–12
3. Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.: LogGP: Incorporating long messages into the LogP model for parallel computation. J. of Parallel and Distributed Computing **44** (1997) 71–79
4. Ino, F., Fujimoto, N., Hagihara, K.: LogGPS: A parallel computational model for synchronization analysis. In: Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. (2001) 133–142
5. Moritz, C.A., Frank, M.I.: LoGPC: Modeling network contention in message-passing programs. IEEE Trans. on Parallel and Distributed Systems **12** (2001) 404–415
6. Adve, V.S., Bagrodia, R., Browne, J.C., Deelman, E., Dube, A., Houstis, E.N., Rice, J.R., Sakellariou, R., Sundaram-Stukel, D.J., Teller, P.J., Vernon, M.K.: POEMS: End-to-end performance design of large parallel adaptive computational systems. IEEE Trans. on Software Engineering **26** (2000) 1027–1048
7. Kvasnicka, D.F., Hlavacs, H., Ueberhuber, C.W.: Simulating parallel program performance with CLUE. In: Proc. 2001 Int'l Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'01). (2001) 140–149
8. Adve, V.S., Bagrodia, R., Deelman, E., Sakellariou, R.: Compiler-optimized simulation of large-scale applications on high performance architectures. J. of Parallel and Distributed Computing **62** (2002) 393–426
9. Czarnul, P., Tomko, K., Krawczyk, H.: Dynamic partitioning of the divide-and-conquer scheme with migration in PVM environment. In: Proc. 8th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'01). (2001) 174–182
10. O'Carroll, F., Tezuka, H., Hori, A., Ishikawa, Y.: The design and implementation of zero copy MPI using commodity hardware with a high performance network. In: Proc. 12th ACM Int'l Conf. on Supercomputing, <http://www.pcccluster.org/> (1998) 243–250

POETRIES: Performance Oriented Environment for Transparent Resource-Management, Implementing End-User Parallel/Distributed Applications¹

Eduardo Cesar, J.G. Mesa, Joan Sorribes, and Emilio Luque

Computer Science Department. Universitat Autònoma de Barcelona
08193 Bellaterra, Spain

{Eduardo.Cesar, Joan.Sorribes, Emilio.Luque}@uab.es

Abstract. Parallel/Distributed application development is a very difficult task for non-expert programmers, and support tools are therefore needed for all phases of the development cycle of these kinds of application. This study specifically presents a programming environment oriented towards dynamic performance tuning based on frameworks with an associated performance model. The underlying idea is that programmers only use frameworks to concentrate on solving the computational problem at hand, and the dynamic performance tuning tool uses the framework-associated performance model to improve the performance of the applications at run-time.

1 Introduction

Parallel/distributed programming constitutes a highly promising approach to the improvement of the performance of many applications. However, in comparison to sequential programming, several new problems have emerged in all phases of the development cycle of these kinds of applications. One of the best ways to solve these problems would be to develop tools that support the design, coding, and analysis and/or tuning of parallel/distributed applications.

In the particular case of performance analysis and/or tuning, it is important to note that the best way to analyze and tune parallel/distributed applications depends on some of their behavioral characteristics. If the application to be tuned behaves in a regular way, then a static analysis would be sufficient. However, if the application changes its behavior from execution to execution, or even in a single execution, then dynamic monitoring and tuning techniques should be used instead.

The key issue in dynamic monitoring and tuning is that decisions must be taken efficiently while minimizing intrusion on the application. In this sense, we have designed a dynamic tuning environment, shown in figure 1, where decisions are taken on the basis of the high level structure of the application [1][2].

The main idea is that it is possible to define a performance model associated to the most common application frameworks. This model includes a set of *performance functions* to not only detect performance drawbacks, but also detect which application

¹ This work was supported by MCyT-Spain under contract TIC 2001-2592 and partially supported by the *Generalitat de Catalunya – Grup de Recerca Consolidat 2001* SGR-00218

parameters have to be measured (*measure points*) to evaluate these functions and which actions should be taken to overcome detected performance drawbacks (*tuning points*).

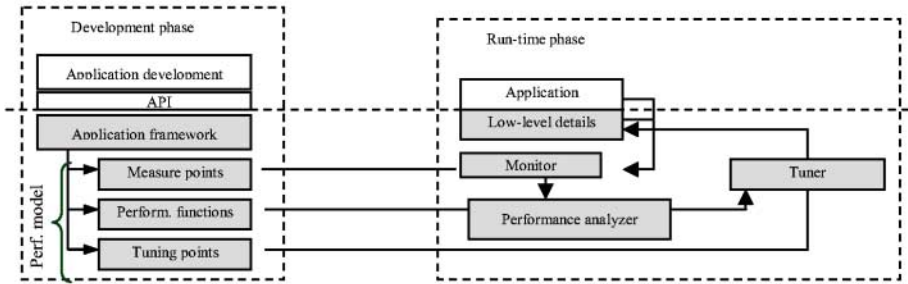


Fig. 1. Structure of the Dynamic Tuning Environment.

These definitions make up the static part of the environment (Development phase). After that, there is a dynamic tuning tool (Run-time phase) which, at application execution time, uses the performance model to monitor the appropriate parameters for evaluation of the performance functions (performance analyzer) and which, using the evaluation results as a basis, takes the required actions to improve the application performance (tuner).

To implement the *development phase* of the described tuning environment we have created POETRIES, an environment that integrates a framework-based parallel/distributed programming environment with the performance model needed to perform the dynamic analysis and tuning of these kinds of applications.

In section 2, we present an overview of the related studies in this field. In section 3, we describe the general methodology used to develop the environment. In section 4, we illustrate the use of this methodology through an analytical description of the performance model for homogeneous Master-Worker applications, as well as the experimental results that have been obtained to validate it. Finally, in section 5, we present the conclusions of this study.

2 Related Studies

It is possible to classify related studies into two big sets; firstly, those studies that address the problem of parallel/distributed application development in its different phases, and secondly, those studies related to performance monitoring, analysis and tuning, which in turn could be broadly divided into those that propose a static (predictive or trace-based) analysis, and those that propose a dynamic one.

In the first set, some studies, such as [3] and CO₂P₃S[4], are based on design patterns which could be applied to those problems suitable for being solved in parallel, while other studies, such as Skil[5], Eden[6] and “Haskell with Ports”[7] take advantage of the high abstraction degree and transformation possibilities of functional languages. Finally, [8] suggest the use of a known modeling language, such as UML, extended to model the most important parallel/distributed constructs.

In the set of studies related to performance monitoring, analysis and tuning we have found tools that make a trace-based analysis, such as Kappa-Pi[9] and EXPERT [10], along with other studies based on static prediction, such as P³T+[11], which predicts performance mainly based on information gathered at compilation time.

In the same set, but with a dynamic approach, we find Paradyn[12] and its Performance Consultant, which dynamically searches performance bottlenecks.

It is worth noting that of the programming tools only [6] and [10] mention the possibility of taking advantage of the knowledge of the application structure for performance reasons, and not one of the performance analysis tools uses information about the application structure to perform its analysis. The main contribution of our study is precisely to demonstrate that this information could be very useful for dynamically analyzing and tuning application performance.

3 Environment Design Methodology

The main goal of POETRIES, performance oriented programming environment, is to offer programmers a set of tools to allow them to develop parallel applications while concentrating solely on the computational problem to be solved. This goal poses two different, although related, problems. Firstly, tools and structures (frameworks) to simplify application development have to be available, and secondly, a methodology must be designed that enables dynamic and automatic performance tuning.

In general, any programming tool based on the application structure that fulfills the requirements of flexibility, usability and independence of the communication library could be used in our environment, because the performance model is not related to any specific implementation of the structure, but rather to the structure itself. However, in the initial phase of our study, we decided to develop our own set of frameworks in order to gain flexibility and knowledge about their implementation.

As we outlined in the introduction, in order to be efficient the dynamic tuning tool needs to know the performance model associated to the structure of the application being tuned. It includes the parameters of the application (*measure points*) that have to be monitored at runtime in order to evaluate the *performance functions* and decide what actions should be taken to overcome performance drawbacks (*tuning points*).

Our main goal is to minimize the execution time of the application, so the performance functions should describe the general behavior of the application as a function of the computing and communication times of the application processes and the structure of the framework used to implement it. Consequently, the general methodology used to define a performance model, useful for the dynamic tuning environment, consists of finding for each framework:

- The parameters (*measure points*) which must be monitored at runtime to allow for the evaluation of the performance functions
- A set of *performance functions* that reflect the behavior of the application in terms of execution time allowing the dynamic detection of performance drawbacks and the impact of dynamically introducing changes to the application.
- The actions (*tuning points*) that should be taken dynamically by the tuning tool to improve the performance of the application.

It is worth noting that the conditions that fire an action must be persistent enough to be sure of the benefits of any improvements that could be obtained from that action. In other words transient problems are not worth the effort of solving.

4 Performance Model for Homogeneous Applications

To illustrate our methodology, we show its applicability in the simple case of Master-Worker homogeneous applications, where homogeneous means that we assume that all tasks are approximately of the same size and require the same processing time.

The most important step to achieve this *performance model* is to define the *performance functions* associated to the framework, because from this definition we will get the parameters that must be measured (*measure points*) to evaluate them, and we also will infer the actions that should be taken (*tuning points*) to improve performance. The analytical definition of the performance functions is shown in section 4.1 and their experimental validation appears in section 4.2.

4.1 Homogeneous Master-Worker Performance Analysis

In our analysis we will use the following terminology:

- tl, λ . Network parameters (time overhead per message and inverse bandwidth)
- V, v_m, v_i . Size of task sent to worker i in bytes (v_i). Size of results sent back to master in bytes (v_m), and total data volume ($V = \sum_i (v_i + v_m)$).
- n = current number of workers in the application.
- Tt, Tc, tc_i . Time that each worker spends to process a task (tc_i), total computing time ($Tc = \sum_i tc_i$), and total execution time (Tt).
- $Nopt$ = number of workers needed to obtain the minimum Tt (best performance).

Now, we can describe the analysis performed to construct the performance functions of these kinds of applications:

- At the beginning, the master sends one task to each worker, the time spent for this operation is $n*tl$ (network overhead) + $\lambda*v_i$ (last task communication time).
- Next, we just have to add the processing time of one worker, tc_i (Tc/n).
- In order to evaluate what happens to results sent back to the master we only need to count the communication time for the last message, which is $tl + \lambda*v_m$.
- The total iteration time is formed by adding these quantities together, and gives: $Tt = n*tl + \lambda*v_i + Tc/n + tl + \lambda*v_m$, as $\lambda*v_m + \lambda*v_i = \lambda*V/n$, we obtain:

$$Tt = (tl*n^2 + \lambda*V + 2*Tc)/n + tl \quad (1)$$

- If we calculate $dTt/dn = 0$ for expression (1) we obtain an expression to calculate the number of workers needed to minimize Tt , which is:

$$Nopt = \text{sqr}(\lambda*V + Tc) / tl \quad (2)$$

It can be seen that the *measure points* needed for the performance model associated to this framework are:

- tl and λ , which could be calculated at the beginning of the execution.

- Tasks & results size (v_i, v_m), which could be captured in communication functions.
- The time workers spend on each task to calculate the total computing time (T_c).

The *performance functions* that have to be evaluated are:

- Expression (1), to predict the application performance for any number of workers.
- Expression (2), because it indicates what has to be done to obtain the best application performance under the conditions that hold for a certain time period.

Finally, the main *tuning point* to improve performance is:

- Adding or deactivating workers when the performance functions indicate that a better performance could be obtained.

4.2 Experimental Validation

In order to validate the analytical results described above we have written a synthetic parametrical master-worker application generator. This application uses MPI and all experiments were executed on a homogeneous Linux cluster of workstations.

We generated two applications, the first was aimed at getting the best performance out of a few workers (6) and the second made intensive computations aimed at getting the best performance out of many workers (80). In both cases, the synthetic application was executed 100 times for 1 to 10 workers and the mean execution time was recorded. Using the *performance model* associated to this framework, the *measure points* were monitored and the *performance functions* were evaluated.

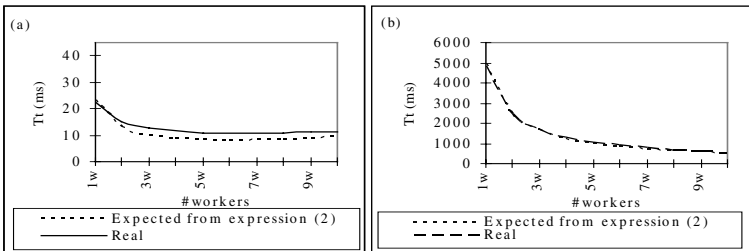


Fig. 2. Best performance obtained with (a) 6 workers (b) 80 workers.

The results of this experiment are shown in figure 3 (a) for the first application and 3 (b) for the second one, where real and expected (expression (1)), execution times are outlined for each case. As it can be seen, the application's behavior is closely captured by expression (2), a fact that validates it. But also expression (3), which indicates the optimal number of workers, is also validated.

5 Conclusions and Future Studies

In this study, we have shown the design of POETRIES to be an environment that integrates a framework-based parallel/distributed programming environment with the performance model needed to carry out the dynamic analysis and tuning of these

kinds of applications. The main underlying idea is to knowing the structure of an application in advance is a good way to make appropriate global decisions regarding the dynamic improvement of its performance.

To show how to build this environment, we have presented the requirements that frameworks must fulfill in order to be useful in such an environment and, of course, to programmers, and the methodology for the definition of framework performance models.

To illustrate the usefulness of relating application structure to its performance model for dynamic analysis and tuning we have presented the case of Master-Worker homogeneous applications. The results obtained demonstrate that it is possible to define analytical models that can be used by tuning tools to improve performance.

Of course much more work still remains to be done, with the building of more general performance models that cover non-homogeneous applications being the most important challenge. However, we believe this is an extremely promising approach for broadening and encouraging the use of parallel/distributed applications.

References

1. A. Morajko, E. Cesar, T. Margalef, J. Sorribes, E. Luque, "Dynamic Performance Tuning Environment", Proc. Euro-Par 2001, pp. 36–45, Manchester, UK, August 2001.
2. E. Cesar, A. Morajko, T. Margalef, J. Sorribes, A. Espinosa, E. Luque, "Dynamic Performance Tuning Supported by Program Specification", Scientific Programming 10, pp 35–44, IOS Press, (2002).
3. B. L. Massingill, T. G. Mattson, B. A. Sanders, "A Pattern Language for Parallel Application Progrmas", Proc. Of the 6th International Euro-Par Conference (Euro-Par 2000).
4. S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, K. Tan, "From Patterns to Frameworks to Parallel Programs", Parallel Computing, Vol. 28, n. 12, pp 1663–1683, 2002.
5. T. Richert, "Skil: Programming with Algorithmic Skeletons – A Practical Point of View", Proceedings of the 12th Int. Workshop on Impl. of Funct. Lang., pp. 15-30, Germany, 2002.
6. U. Klusik, R. Loogen, S. Priebe, F. Rubio, "Implementation Skeletons in Eden: Low-Effort Parallel Programming", Lecture Notes in Computer Science, Vol. 2011, pp. 71–88, 2001.
7. F. Huch, U. Norbistrath, "Distributed Programming in Haskell with Ports", Proceedings of the 12th International Workshop on Implementation of Functional Languages, 2002.
8. S. Pillana, T. Fahringer, "On Customizing the UML for Modelling Performance-Oriented Applications", UML 2002 "Model Engineering, Concepts and Tools", Springer-Verlag, Dresden, Germany, September 2002.
9. A. Espinosa, T. Margalef, E. Luque, "Integrating Automatic Techniques in a Performance Analysis Session", LNCS, Vol. 1900 (Euro-Par 2000), pp. 173–177, Springer-Verlag, 2000.
10. F. Wolf, B. Morh, "Automatic Performance Analysis of SPM Cluster Applications", Technical Report IB-2001-05, 2001.
11. T. Fahringer, A. Pogaj, "P³T+: A Performance Estimator for Distributed and Parallel Programs", Scientific Programming, IOS Press, Vol. 8, no. 2, The Netherlands, 2000.
12. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, T. Newhall, "The Paradyn Parallel Performance Measurement Tool", IEEE Computer 28, 11, pp. 37–46, November 1995.

Topic 3

Scheduling and Load Balancing

Yves Robert, Henri Casanova, Arjan van Gemund, and Dieter Kranzlmüller
Topic Chairs

Scheduling and load balancing techniques are crucial issues when executing applications in parallel and distributed systems. The quest for high performance computing (HPC) requires substantial support from efficient scheduling and load balancing techniques. Even the most powerful HPC environments require proper support of these techniques in order to fulfill the users expectations in terms of performance and efficiency.

For this reason, research on optimization of scheduling decisions and improvements of load distribution are addressed in many past and on-going projects. Such techniques can be provided either at the application level or at the system level. Both scenarios are of interest for this workshop and are covered by the contributed papers. Both, theoretical and practical aspects, are presented and discussed by the authors.

In addition, this year puts special emphasis on the characteristics of scheduling and load balancing algorithms for current and new parallel and distributed systems, such as clusters, computational grids and global computing environments. These massively parallel and often inhomogeneous systems represent major challenges for todays scheduling and load balancing algorithms. The variety of systems and the possible distribution over long distance resources requires dedicated solutions for different resource constellations as well as adaptations to changes of the execution environment.

The selection of papers has been initiated with 25 contributions. Based on the reviews, the workshop consists of 1 distinguished paper, 3 regular papers, and 9 short papers. Each paper received between 3 and 4 reviews, and the decisions have been taken unanimously within the topic committee. A broad spectrum of techniques is covered by these contributions, including job allocation on clusters and replication on computational grids. The number of short papers indicates the wide variety of results in different areas of this research domain.

Finally, we would sincerely like to thank all of the contributing authors for their work, as well as the reviewers for their support during the selection process.

Static Load-Balancing Techniques for Iterative Computations on Heterogeneous Clusters

Hélène Renard, Yves Robert, and Frédéric Vivien

LIP, UMR CNRS-INRIA-UCBL 5668

ENS Lyon, France

{Helene.Renard | Yves.Robert | Frederic.Vivien}@ens-lyon.fr

Abstract. This paper is devoted to static load balancing techniques for mapping iterative algorithms onto heterogeneous clusters. The application data is partitioned over the processors. At each iteration, independent calculations are carried out in parallel, and some communications take place. The question is to determine how to slice the application data into chunks, and to assign these chunks to the processors, so that the total execution time is minimized. We establish a complexity result that assesses the difficulty of this problem, and we design practical heuristics that provide efficient distribution schemes.

1 Introduction

In this paper, we investigate static load balancing techniques for iterative algorithms that operate on a large collection of application data. The application data will be partitioned over the processors. At each iteration, some independent calculations will be carried out in parallel, and then some communications will take place. This scheme is very general, and encompasses a broad spectrum of scientific computations, from mesh based solvers to signal processing, and image processing algorithms. The target architecture is a fully heterogeneous cluster, composed of different-speed processors that communicate through links of different capacities. The question is to determine the best partitioning of the application data. The difficulty comes from the fact that both the computation and communication capabilities of each resource must be taken into account.

An abstract view of the problem is the following: the iterative algorithm repeatedly operates on a large rectangular matrix of data samples. This data matrix is split into vertical slices that are allocated to the computing resources (processors). At each step of the algorithm, the slices are updated locally, and then boundary information is exchanged between consecutive slices. This (virtual) geometrical constraint advocates that processors be organized as a virtual ring. Then each processor will only communicate twice, once with its (virtual) predecessor in the ring, and once with its successor. There is no reason *a priori* to restrict to a uni-dimensional partitioning of the data, and to map it onto a uni-dimensional ring of processors: more general data partitionings, such as two-dimensional, recursive, or even arbitrary slicings into rectangles, could be

considered. But uni-dimensional partitionings are very natural for most applications, and, as will be shown in this paper, the problem to find the optimal one is already very difficult.

We assume that the target computing platform can be modeled as a complete graph:

- Each vertex in the graph models a computing resource P_i , and is weighted by the relative cycle-time of the resource. Of course the absolute value of the time-unit is application-dependent, what matters is the relative speed of one processor versus the other.
- Each edge models a communication link, and is weighted by the relative capacity of the link. Assuming a complete graph means that there is a *virtual* communication link between any processor pair P_i and P_j . Note that this link does not necessarily need to be a direct physical link. There may be a path of physical communication links from P_i to P_j : if the slowest link in the path has maximum capacity $c_{i,j}$, then the weight of the edge will be $c_{i,j}$.

We suppose that the communication capacity $c_{i,j}$ is granted between P_i and P_j (so if some communication links happen to be physically shared, we assume that a fraction of the total capacity, corresponding to the inverse of $c_{i,j}$, is available for messages from P_i to P_j). This assumption of a fixed capacity link between any processor pair makes good sense for interconnection networks based upon high-speed switches like Myrinet [2].

Given these hypotheses, the optimization problem that we want to solve is the following: how to slice the matrix data into chunks, and assign these chunks to the processors, so that the total execution time for a given sweep step, namely a computation followed by two neighbor communications, is minimized? We have to perform resource selection, because there is no reason *a priori* that all available processors will be involved in the optimal solution (for example some fast computing processor may be left idle because its communication links with the other processors are too slow). Once some resources have been selected, they must be arranged along the best possible ring, which looks like a difficult combinatorial problem. Finally, once a ring has been set up, there remains to load-balance the workloads of the participating resources.

The rest of the paper is organized as follows. In Section 2, we formally state the previous optimization problem, which we denote as SLICERING. If the network is homogeneous (all links have same capacity), then SLICERING can be solved easily, as shown in Section 3. But in the general case, SLICERING turns out to be a difficult problem: we show in Section 4 that the decision problem associated to SLICERING is NP-complete, as could be expected from its combinatorial nature. We derive in Section 5 a formulation of the SLICERING problem in terms of an integer linear program, thereby providing a (costly) way to determine the optimal solution. In Section 6, we move to the design of polynomial-time heuristics, and we report some experimental data. We survey related work in Section 7, and we provide a brief comparison of static versus dynamic strategies. Finally, we state some concluding remarks in Section 8.

2 Framework

In this section, we formally state the optimization problem to be solved. As already said, the target computing platform is modeled as a complete graph $G = (P, E)$. Each node P_i in the graph, $1 \leq i \leq |P| = p$, models a computing resource, and is weighted by its relative cycle-time w_i : P_i requires $S.w_i$ time-units to process a task of size S . Edges are labeled with communication costs: the time needed to transfer a message of size L from P_i to P_j is $L.c_{i,j}$, where $c_{i,j}$ is the capacity of the link, i.e. the inverse of its bandwidth. The motivation to use a simple linear-cost model, rather than an affine-cost model involving start-ups, both for the communications and the computations, is the following: only large-scale applications are likely to be deployed on heterogeneous platforms. Each step of the algorithm will be both computation- and communication-intensive, so that start-up overheads can indeed be neglected. Anyway, most of the results presented here extend to an affine cost modeling, $\tau_i + S.w_i$ for computations and $\beta_{i,j} + L.c_{i,j}$ for communications.

Let W be the total size of the work to be performed at each step of the algorithm. Processor P_i will accomplish a share $\alpha_i.W$ of this total work, where $\alpha_i \geq 0$ for $1 \leq i \leq p$ and $\sum_{i=1}^p \alpha_i = 1$. Note that we allow $\alpha_j = 0$ for some index j , meaning that processor P_j do not participate in the computation. Indeed, there is no reason *a priori* for all resources to be involved, especially when the total work is not so large: the extra communications incurred by adding more processors may slow down the whole process, despite the increased cumulated speed.

We will arrange the participating processors along a ring (yet to be determined). After updating its data slice, each active processor P_i sends some boundary data to its neighbors: let $\text{pred}(i)$ and $\text{succ}(i)$ denote the predecessor and the successor of P_i in the virtual ring. Then P_i requires $H.c_{i,\text{succ}(i)}$ time-units to send a message of size H to its successor, plus $H.c_{i,\text{pred}(i)}$ to receive a message of same size from its predecessor. In most situations, we will have symmetric costs ($c_{i,j} = c_{j,i}$) but we do not make this assumption here. To illustrate the relationship between W and H , we can view the original data matrix as a rectangle composed of W columns of height H , so that one single column is exchanged between any pair of consecutive processors in the ring (but clearly, the parameter H can represent any fixed volume of communication).

The total cost of a single step in the sweep algorithm is the maximum, over all participating processors, of the time spent computing and communicating:

$$T_{\text{step}} = \max_{1 \leq i \leq p} \mathbb{I}\{i\} [\alpha_i.W.w_i + H.(c_{i,\text{succ}(i)} + c_{i,\text{pred}(i)})]$$

where $\mathbb{I}\{i\}[x] = x$ if P_i is involved in the computation, and 0 otherwise. In summary, the goal is to determine the best way to select q processors out of the p available, and to arrange them along a ring so that the total execution time per step is minimized. We formally state this optimization problem as follows:

Definition 1 ($\text{SliceRing}(p, w_i, c_{i,j}, W, H)$). *Given p processors of cycle-times w_i and $p(p-1)$ communication links of capacity $c_{i,j}$, given the total workload W and the communication volume H at each step, determine*

$$T_{\text{step}} = \min_{1 \leq q \leq p} \min_{\sigma \in \mathcal{S}_{q,p}} \max_{1 \leq i \leq q} \left(\sum_{i=1}^q \alpha_{\sigma(i)} = 1 \right. \\ \left. (\alpha_{\sigma(i)} \cdot W \cdot w_{\sigma(i)} + H \cdot (c_{\sigma(i), \sigma(i-1 \bmod q)} + c_{\sigma(i), \sigma(i+1 \bmod q)})) \right) \quad (1)$$

Here $\mathcal{S}_{q,p}$ denotes the set of one-to-one functions $\sigma : [1..q] \rightarrow [1..p]$ which index the q selected processors, for all candidate values of q between 1 and p .

From Equation 1, we see that the optimal solution will involve all processors as soon as the ratio $\frac{W}{H}$ is large enough: in that case, the impact of the communications becomes smaller in front of the cost of the computations, and these computations should be distributed to all resources. But even in that case, we still have to decide how to arrange the processors along a ring. Extracting the “best” ring out of the interconnection graph seems to be a difficult combinatorial problem. Before assessing this result (see Section 4), we deal with the much easier situation when the network is homogeneous (see Section 3).

To conclude this section, we point out that this framework is more general than iterative algorithms: in fact, our approach applies to any problem where independent computations are distributed over heterogeneous resources. The only hypothesis is that the communication volume is the same between adjacent processors, regardless of their relative workload.

3 Homogeneous Networks

Solving the optimization problem, i.e. minimizing expression (1), is easy when all communication times are equal. This corresponds to a homogeneous network where each processor pair can communicate at the same speed, for instance through a bus or an Ethernet backbone.

Let us assume that $c_{i,j} = c$ for all i and j , where c is a constant. There are only two cases to consider: (i) only the fastest processor is active; (ii) all processors are involved. Indeed, as soon as a single communication occurs, we can have several ones for the same cost, and the best is to divide the computing load among all resources. In the former case (i), we derive that $T_{\text{step}} = W \cdot w_{\min}$, where $w_{\min}$ is the smallest cycle-time. In the latter case (ii), the load is most balanced when the execution time is the same for all processors: otherwise, removing a small portion of the load of the processor with largest execution time, and giving it to a processor finishing earlier, would decrease the maximum computation time. This leads to $\alpha_i \cdot w_i = \text{Constant}$ for all i , with $\sum_{i=1}^p \alpha_i = 1$. We derive that $T_{\text{step}} = W \cdot w_{\text{cumul}} + 2H \cdot c$, where $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^p \frac{1}{w_i}}$. We summarize these results:

Proposition 1. *The optimal solution to SLICERING(p, w_i, c, W, H) is*

$$T_{step} = \min \{W.w_{\min}, W.w_{cumul} + 2H.c\}$$

where $w_{\min} = \min_{1 \leq i \leq p} w_i$ and $w_{cumul} = \frac{1}{\sum_{i=1}^p \frac{1}{w_i}}$.

If the platform is given, there is a threshold, which is application-dependent, to decide whether only the fastest computing resource, as opposed to all the resources, should be involved. Given H , the fastest processor will do all the job for small values of W , namely $W \leq H \cdot \frac{2c}{w_{\min} - w_{cumul}}$. Otherwise, for larger values of W , all processors should be involved.

4 Complexity

The decision problem associated to the SLICERING optimization problem is:

Definition 2 (SliceRingDec($p, w_i, c_{i,j}, W, H, K$)). *Given p processors of cycle-times w_i and $p(p-1)$ communication links of capacity $c_{i,j}$, given the total workload W and the communication volume H at each step, and given a time bound K , is it possible to find an integer $q \leq p$, a one-to one mapping $\sigma : [1..q] \rightarrow [1..p]$, and nonnegative rational numbers α_i with $\sum_{i=1}^q \alpha_{\sigma(i)} = 1$, such that*

$$T_{step} = \max_{1 \leq i \leq q} \{ \alpha_{\sigma(i)} \cdot W.w_{\sigma(i)} + H \cdot (c_{\sigma(i), \sigma(i-1 \bmod q)} + c_{\sigma(i), \sigma(i+1 \bmod q)}) \} \leq K?$$

The following result states the intrinsic difficulty of the problem (see [9] for the proof):

Theorem 1. SLICERINGDEC($p, w_i, c_{i,j}, W, H, K$) is NP-complete.

5 ILP Formulation

When the network is heterogeneous, we face a complex situation: how to determine the number of processors that should take part to the computation already is a difficult question.

In this section, we express the solution to the SLICERING optimization problem, in terms of an Integer Linear Programming (ILP) problem. Of course the complexity of this approach may be exponential in the worst case, but it will provide useful hints to design low-cost heuristics. We start with the case where all processors are involved in the optimal solution. We extend the approach to the general case later on.

5.1 When All Processors Are Involved

Assume first that all processors are involved in an optimal solution. All the p processors require the same amount of time to compute and communicate: otherwise, we would slightly decrease the computing load of the last processor to

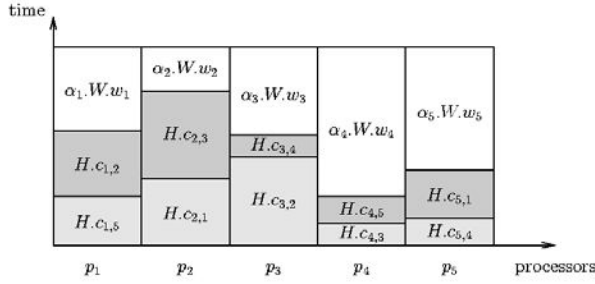


Fig. 1. Summary of computation and communication times with p processors.

complete its assignment (computations followed by communications) and assign extra work to another one. Hence (see Figure 1 for an illustration) we have

$$T_{\text{step}} = \alpha_i.W.w_i + H.(c_{i,i-1} + c_{i,i+1}) \quad (2)$$

for all i (indices in the communication costs are taken modulo p). As $\sum_{i=1}^p \alpha_i = 1$, we derive that $\sum_{i=1}^p \frac{T_{\text{step}} - H.(c_{i,i-1} + c_{i,i+1})}{W.w_i} = 1$. Defining $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^p \frac{1}{w_i}}$ as before, we have:

$$\frac{T_{\text{step}}}{W.w_{\text{cumul}}} = 1 + \frac{H}{W} \sum_{i=1}^p \frac{c_{i,i-1} + c_{i,i+1}}{w_i}. \quad (3)$$

Therefore, T_{step} will be minimal when $\sum_{i=1}^p \frac{c_{i,i-1} + c_{i,i+1}}{w_i}$ is minimal. This will be achieved for the ring that corresponds to the shortest Hamiltonian cycle in the graph $G = (P, E)$, where each edge $e_{i,j}$ is given the weight $d_{i,j} = \frac{c_{i,j} + c_{j,i}}{w_i}$. Once we have this path, we derive T_{step} from Equation 3, and then we determine the load α_i of each processor using Equation 2.

To summarize, we have the following result:

Proposition 2. *When all processors are involved, finding the optimal solution is equivalent to solving the Traveling Salesman Problem (TSP) in the weighted graph (P, E, d) , $d_{i,j} = \frac{c_{i,j} + c_{j,i}}{w_i}$.*

Of course we are not expecting any polynomial-time solution from this result, because the decision problem associated to the TSP is NP-complete [6] (even worse, because the distance d does not satisfy the triangle inequality, there is no polynomial-time approximation [1]), but this equivalence gives us two lines of action:

- For platforms of reasonable size, the optimal solution can be computed using an integer linear program that returns the optimal solution to the TSP
- For very large platforms, we can use well-established heuristics which approximate the solution to the TSP in polynomial time, such as the Lin-Kernighan heuristic [8,7].

The following result (see [9] for a proof) solves the problem whenever all processors are involved:

Proposition 3. *When all processors are involved, finding the optimal solution is equivalent to solving the following integer linear program:*

TSP integer linear programming formulation

MINIMIZE $\sum_{i=1}^p \sum_{j=1}^p d_{i,j} \cdot x_{i,j}$,

SUBJECT TO

$$\begin{cases} (1) \sum_{j=1}^p x_{i,j} = 1 & 1 \leq i \leq p \\ (2) \sum_{i=1}^p x_{i,j} = 1 & 1 \leq j \leq p \\ (3) x_{i,j} \in \{0, 1\} & 1 \leq i, j \leq p \\ (4) u_i - u_j + p \cdot x_{i,j} \leq p - 1 & 2 \leq i, j \leq p, i \neq j \\ (5) u_i \text{ integer}, u_i \geq 0 & 2 \leq i \leq p \end{cases}$$

5.2 General Case

How to extend the ILP formulation to the general case? For each possible value of q , $1 \leq q \leq p$, we will set up an ILP problem giving the optimal solution with exactly q participating resources. Taking the smallest solution among the p values returned by these ILP problems will lead to the optimal solution.

For a fixed value of q , $1 \leq q \leq p$, we use a technique similar to that of Section 5.1, but we need additional variables. Here is the ILP (see [9] for the proof that the solution is a ring of q participating resources):

q-ring integer linear programming formulation

MINIMIZE T ,

SUBJECT TO

$$\begin{cases} (1) \alpha_i \cdot w_i + \frac{H}{W} \sum_{j=1}^p (x_{i,j} c_{i,j} + x_{j,i} c_{j,i}) \leq T + \left(1 - \sum_{i=1}^p x_{j,i}\right) \cdot K & 1 \leq i \leq p \\ (2) \sum_{i=1}^p x_{i,j} = \sum_{i=1}^p x_{j,i} & 1 \leq j \leq p \\ (3) \sum_{i=1}^p x_{i,j} \leq 1 & 1 \leq j \leq p \\ (4) \sum_{i=1}^p \sum_{j=1}^p x_{i,j} = q & \\ (5) x_{i,j} \in \{0, 1\} & 1 \leq i, j \leq p \\ (6) \sum_{i=1}^p y_i = 1 & \\ (7) -p \cdot y_i - p \cdot y_j + u_i - u_j + q \cdot x_{i,j} \leq q - 1 & 1 \leq i, j \leq p, i \neq j \\ (8) y_i \in \{0, 1\} & 1 \leq i \leq p \\ (9) u_i \text{ integer}, u_i \geq 0 & 1 \leq i \leq p \end{cases}$$

where $K = \max_{1 \leq j \leq p} w_{i,j} + 2 \frac{H}{W} \cdot \max_{1 \leq j \leq p} c_{i,j}$. In any optimal solution of this linear program, $T = \frac{T_{\text{step}}}{W}$ and, because of the term $(1 - \sum_{i=1}^p x_{j,i}) \cdot K$, the constraints (1), for $1 \leq i \leq p$, induce constraints on T only for resources participating in the solution ring.

Proposition 4. *The SLICERING optimization problem can be solved by computing the solution of p integer linear programs, where p is the total number of resources.*

6 Heuristics and Experiments

After the previous theoretically-oriented results, we adopt a more practical approach in this section. We aim at deriving polynomial-time heuristics for solving the SLICERING optimization problem.

Having expressed the problem in terms of a collection of integer linear programs enables us to try to compute the optimal solution with softwares like PIP [4,3] (at least for reasonable sizes of the target computing platforms). We compare this optimal solution with that returned by two polynomial-time heuristics, one that approximates the TSP problem (but only returns a solution where all processors are involved), and a greedy heuristic that iteratively grows the solution ring.

6.1 TSP-Based Heuristic

The situation where all processors are involved in the optimal solution is very important in practice. Indeed, only very large applications are likely to be deployed on distributed heterogeneous platforms. And when W is large enough, we know from Equation 1 that all processors will be involved.

From Section 5.1 we know that the optimal solution, when all processors are involved, corresponds to the shortest Hamiltonian cycle in the graph (P, E, d) , with $d_{i,j} = \frac{c_{i,j} + c_{j,i}}{w_i}$. We use the well-known Lin-Kernighan heuristic [8,7] to approximate this shortest path. By construction, the TSP-based heuristic always returns a solution where all processors are involved. Of course, if the optimal solution requires fewer processors, the TSP-based heuristic will fail to find it.

6.2 Greedy Heuristic

The greedy heuristic starts by selecting the fastest processor. Then, it iteratively includes a new node in the current solution ring. Assume that we have already selected a ring of r processors. For each remaining processor P_i , we search where to insert it in the current ring: for each pair of successive processors (P_j, P_k) in the ring, we compute the cost of inserting P_i between P_j and P_k in the ring. We retain the processor and the pair that minimize the insertion cost, and we store the value of T_{step} . This step of the heuristic has a complexity proportional to $(p - r) \cdot r$.

Finally, we grow the ring until we have p processors and we return the minimal value obtained for T_{step} . The total complexity is $\sum_{r=1}^p (p-r)r = O(p^3)$. Note that it is important to try all values of r , because T_{step} may not vary monotonically with r .

6.3 Platform Description

We experimented with two platforms, one located in ENS Lyon and the other in the University of Strasbourg, which are represented in Figure 2. The Lyon

platform is composed of 14 processors, whose cycle-times are described in Table 1. Table 2 shows the capacity of the links, i.e. the inverse of the bandwidth, between each processor pair (P_i, P_j) . The Strasbourg platform is composed of 13 processors, whose cycle-times are described in Table 3, while Table 4 shows the capacity of the links.

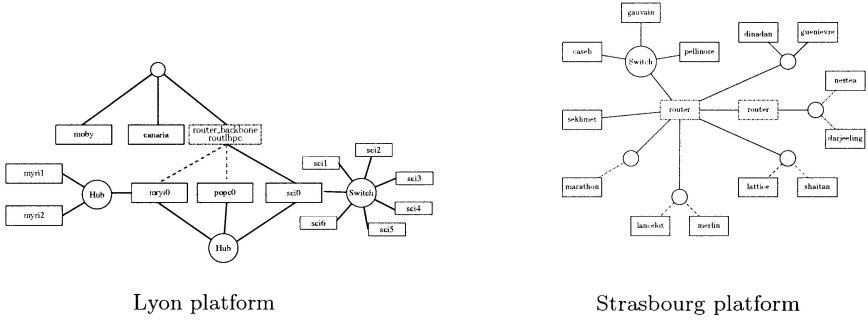


Fig. 2. Topology of the Lyon and Strasbourg platforms.

Table 1. Processor cycle-times (in seconds per megaflop) for the Lyon platform.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}
0.0291	0.00874	0.0206	0.0451	0.0206	0.0291	0.0206	0.0206	0.0206	0.0206	0.0206	0.0206	0.0206	0.0206

6.4 Results

For both topologies, we compared the greedy heuristic against the optimal solution obtained with ILP software, when available. Tables 5 and 6 show the difference between the greedy heuristic and the optimal solution (computed with PIP) on the Lyon and Strasbourg platforms. The numbers in the tables represent the minimal cost of a path of length q on the platform, i.e. the value of the objective function of the ILP program of Section 5.2 (multiplied by a scaling factor 6000, because PIP needs a matrix of integers).

PIP is able to compute the optimal solution only when all processors are involved (note that we used a machine with two gigabytes of RAM!). When all processors are involved, we also tried the LKH heuristic: for both platforms, it returns the optimal result. The conclusions that can be drawn from these experiments are the following:

- the greedy heuristic is fast (and seems efficient)
- the LKH heuristic is very reliable, but its application is limited to the case where all resources are involved
- ILP softwares rapidly fail to compute the optimal solution.

Table 2. Capacity of the links (time in seconds to transfer a 1 Mb message) for the Lyon platform.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}
P_0		0.198	1.702	1.702	0.262	0.198	1.702	1.702	0.262	0.262	0.262	0.262	0.262	0.262
P_1	0.198		1.702	1.702	0.262	0.198	1.702	1.702	0.262	0.262	0.262	0.262	0.262	0.262
P_2	1.702	1.702		0.248	0.248	1.702	0.248	0.248	0.248	0.248	0.248	0.248	0.248	0.248
P_3	1.702	1.702	0.248		0.248	1.702	0.248	0.248	0.248	0.248	0.248	0.248	0.248	0.248
P_4	0.262	0.262	0.248	0.248		0.262	0.248	0.248	0.248	0.248	0.248	0.248	0.248	0.248
P_5	0.198	0.198	1.702	1.702	0.262		1.702	1.702	0.262	0.262	0.262	0.262	0.262	0.262
P_6	1.702	1.702	0.248	0.248	0.248	1.702		0.248	0.248	0.248	0.248	0.248	0.248	0.248
P_7	1.702	1.702	0.248	0.248	0.248	1.702	0.248		0.248	0.248	0.248	0.248	0.248	0.248
P_8	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248		0.248	0.248	0.248	0.248	0.248
P_9	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248	0.248		0.248	0.248	0.248	0.248
P_{10}	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248	0.248	0.248		0.248	0.248	0.248
P_{11}	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248	0.248	0.248	0.248		0.248	0.248
P_{12}	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248	0.248	0.248	0.248	0.248		0.248
P_{13}	0.262	0.262	0.248	0.248	0.248	0.262	0.248	0.248	0.248	0.248	0.248	0.248	0.248	

Table 3. Processor cycle-times (in seconds per megaflop) for the Strasbourg platform.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
0.00874	0.00874	0.0102	0.00728	0.00728	0.0262	0.00583	0.016	0.00728	0.00874	0.0131	0.00583	0.0131

We considered the number p_{opt} of processors in the best solution found by the greedy heuristic as a function of the ratio W/H . As expected, when this ratio grows (meaning more computations per communication), more and more processors are used in the optimal solution, and the value of p_{opt} increases. Because the interconnection network of the Lyon platform involves links of similar capacities, the value of p_{opt} jumps from 1 (sequential solution) to 14 (all processors participate). The big jump from 1 to 14 is easily explained: once there is enough work to make a communication affordable, rather use many communications for the same price, thereby better sharing the load.

Table 4. Capacity of the links (time in seconds to transfer a one-megabit message) for the Strasbourg platform.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
P_0		0.048	0.019	0.017	0.019	0.147	0.151	0.154	0.147	0.048	0.017	0.016	0.151
P_1	0.048		0.048	0.048	0.048	0.147	0.151	0.154	0.147	0.017	0.048	0.048	0.151
P_2	0.019	0.048		0.019	0.019	0.147	0.151	0.154	0.147	0.048	0.019	0.019	0.151
P_3	0.017	0.048	0.019		0.019	0.147	0.151	0.154	0.147	0.048	0.017	0.018	0.151
P_4	0.019	0.048	0.019	0.019		0.147	0.151	0.154	0.147	0.048	0.019	0.019	0.151
P_5	0.147	0.147	0.147	0.147	0.147		0.151	0.154	0.147	0.147	0.147	0.147	0.151
P_6	0.151	0.151	0.151	0.151	0.151	0.151		0.154	0.151	0.151	0.151	0.151	0.151
P_7	0.154	0.154	0.154	0.154	0.154	0.154	0.154		0.154	0.154	0.154	0.154	0.154
P_8	0.147	0.147	0.147	0.147	0.147	0.147	0.151	0.154		0.147	0.147	0.147	0.151
P_9	0.048	0.017	0.048	0.048	0.048	0.147	0.151	0.154	0.147		0.048	0.048	0.151
P_{10}	0.017	0.048	0.019	0.017	0.019	0.147	0.151	0.154	0.147	0.048		0.018	0.151
P_{11}	0.016	0.048	0.019	0.018	0.019	0.147	0.151	0.154	0.147	0.048	0.018		0.151
P_{12}	0.151	0.151	0.151	0.151	0.151	0.151	0.151	0.154	0.151	0.151	0.151	0.151	

Table 5. Comparison between the greedy heuristic and PIP for the Lyon platform.

Processors \ Heuristics	3	4	5	6	7	8	9	10	11	12	13	14
Greedy	1202	556	1152	906	2240	3238	4236	5234	6232	7230	8228	10077
PIP	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	9059

Table 6. Comparison between the greedy heuristic and PIP for the Strasbourg platform.

Processors \ Heuristics	3	4	5	6	7	8	9	10	11	12	13
Greedy	1520	2112	3144	3736	4958	5668	7353	8505	10195	12490	15759
PIP	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	out of memory	14757

The interconnection network of the Strasbourg platform is more heterogeneous, and there the value of p_{opt} jumps from 1 (sequential solution) to 10, 11 and 13 (all processors participate).

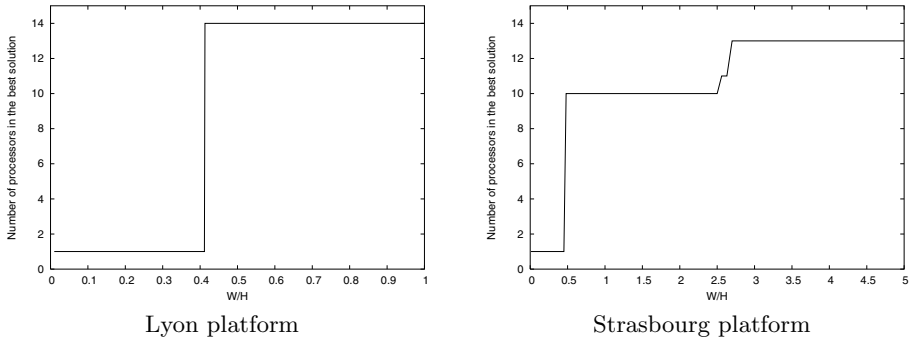


Fig. 3. Optimal number of processors for the Lyon and Strasbourg platforms according to the greedy heuristic.

7 Related Work

Due to the lack of space, see [9].

8 Conclusion

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load. Data and computations are not evenly distributed to processors. Minimizing communication overhead becomes a challenging task.

Load balancing techniques can be introduced dynamically or statically, or a mixture of both. On one hand, we may think that dynamic strategies are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependencies, in addition to communication costs and control overhead, may well lead to slow the whole process down to the pace of the slowest processors. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution. Furthermore, in the context of a scientific library, static allocations seem to be necessary for a simple and efficient memory allocation. We agree, however, that targeting larger platforms such as distributed collections of heterogeneous clusters, e.g. available from the metacomputing grid [5], may well enforce the use of dynamic schemes.

One major result of this paper is the NP-completeness of the SLICERING problem. Rather than the proof, the result itself is interesting, because it provides yet another evidence of the intrinsic difficulty of designing heterogeneous algorithms. But this negative result should not be over-emphasized. Indeed, another important contribution of this paper is the design of efficient heuristics, that provide a pragmatic guidance to the designer of iterative scientific computations. Implementing such computations on commodity clusters made up of several heterogeneous resources is a promising alternative to using costly super-computers.

References

1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
2. D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.
3. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988. Software available at <http://www.prism.uvsq.fr/~cedb/bastools/piplib.html>.
4. P. Feautrier and N. Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel PIP. Technical Report 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), Jan. 1990.
5. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.
7. K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. Software available at <http://www.dat.ruc.dk/~keld/research/LKH/>.
8. S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.
9. H. Renard, Y. Robert, and F. Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. Research Report RR-2003-12, LIP, ENS Lyon, France, Feb. 2003.

Impact of Job Allocation Strategies on Communication-Driven Coscheduling in Clusters^{*}

Gyu Sang Choi¹, Saurabh Agarwal², Jin-Ha Kim¹, Anydy B. Yoo³, and Chita R. Das¹

¹ Department of Computer Science and Engineering
{gchoi,sagarwal,jikim,das}@cse.psu.edu

² IBM India Research Labs
New Delhi – 110016, India
{saurabh.agarwal}@in.ibm.com

³ Lawrence Livermore National Laboratory
Livermore, CA 94551
{yoo2}@llnl.gov

Abstract. In this paper, we investigate the impact of three job allocation strategies on the performance of four coscheduling algorithms (SB, DCS, PB and CC) in a 16-node Linux cluster. The job allocation factors include Multi Programming Level (MPL), job placement, and communication intensity. The experimental results show that the blocking based coscheduling schemes (SB and CC) have better tolerance to different job allocation techniques compared to the spin based schemes (DCS and PB), and the local scheduling. The results strengthen the case for using blocking based coscheduling schemes in a cluster.

1 Introduction

Recently, several dynamic coscheduling algorithms have been proposed for improving the performance of parallel jobs on cluster platforms, which have gained wide acceptance from cost and performance standpoints [1][2][3]. These coscheduling algorithms rely on the communication behavior of the applications to schedule the communicating processes of a job simultaneously. Using efficient user-level communication mechanisms such as U-Net [4], Fast messages [5], and Virtual Interface Architecture (VIA) [6], these techniques are shown to be quite efficient. All prior coscheduling studies have primarily focuses on the scheduling of parallel processes assigned to a processor. They do not address the job allocation issue, *i.e.*, the mechanism to assign jobs to the required nodes of a cluster. Allocation is an integral part of a processor management technique and can have a significant impact on the overall system performance. In view of this,

^{*} This research has been supported by NSF grants CCR-9900701, CCR-0098149, CCR-0208734, and EIA-0202007.

several job allocation strategies have been proposed for batch and gang scheduling techniques to improve system performance [7]. In this paper, we investigate the impact of several job allocation strategies on the relative performance of four coscheduling techniques and native local scheduling using a Linux cluster. The allocation factors that may affect the performance include Multi Programming Level (MPL), communication intensity and job placement. We have developed a generic, scalable and re-usable framework for implementing the coscheduling techniques on a cluster platform. Three prior coscheduling algorithms, Spin-Block (SB)[1][2], Dynamic Coscheduling (DCS)[3] and Periodic Boost (PB)[2], and a newly proposed coscheduling algorithm, called Co-ordinated Coscheduling (CC)[8] are implemented using this framework on a Myrinet connected 16-node Linux cluster. We use four NAS parallel benchmarks to analyze the impact of job allocation strategies. Our experimental results reveal that the blocking based schemes CC and SB consistently show tolerance to most allocation metrics and provide the best performance for all workloads. The local scheduling, and the two spin based techniques (DCS and PB) are more vulnerable to various job allocation strategies. Further, node sharing due to various job placement techniques seems a viable option for the CC and SB algorithms as the application execution times are little affected by such sharing.

The rest of this paper is organized as follows. Section 2 describes the implementation of three prior coscheduling techniques along with our proposed coscheduling algorithm using a generic framework. The job allocation metrics and performance results are presented in Section 3 followed by the concluding remarks in the last section.

2 A Generic Framework for Implementing Coscheduling Algorithms

In this section, first a brief description of the four coscheduling algorithms is given followed by our generic framework for their implementation. All coscheduling algorithms rely primarily on one of two local events (*arrival of a message* and *waiting for a message*) to determine when and which process to schedule. For example, in the SB algorithm, a process waiting for a message spins for a fixed amount of time before blocking itself, hoping that the corresponding process is coscheduled at the remote node [1][2]. Dynamic coscheduling algorithm (DCS) uses incoming messages to schedule the process for which the messages are destined [3]. The underlying idea is that there is a high probability that the corresponding sender is scheduled at the remote node and thus, both processes can be scheduled simultaneously. In the PB scheme, a periodic mechanism checks the endpoints of the parallel processes in a round-robin fashion and boosts the priority of one of the processes with un-consumed messages based on some selection criteria [2].

Our new Co-ordinated Coscheduling (CC) scheme [8] is different in that it optimizes both sender and receiver side spinning to improve performance. With this scheme, the sender spins for a pre-determined amount of time waiting for an

acknowledge from Network Interface Card (NIC). The NIC sends the acknowledge after pushing the message to the wire. If a send is completed within the spin time, the sender remains scheduled hoping that its receiver will be coscheduled and a response can be received soon. If the sender does not receive the acknowledge from the NIC within the spin time, it is blocked and the scheduler chooses another process from its runqueue. As soon as the NIC completes the corresponding send, it wakes up the original sender process, and makes it ready to be coscheduled before the reply comes from the other end. On the receiver side, a process waits for a message arrival within the spin time. If a message does not arrive within this time, the process is blocked and registered for an interrupt from the NIC. The NIC firmware maintains per process message arrival information in a table and continuously updates the table by recording the cumulative number of incoming messages for the corresponding process. Every 10ms, the table information in the NIC is retrieved to find the process, which has the largest number of un-consumed incoming message, and the process is returned to the local scheduler to be run next. The *send_spin_time* and *recv_spin_time* are carefully calculated for maximal performance benefits.

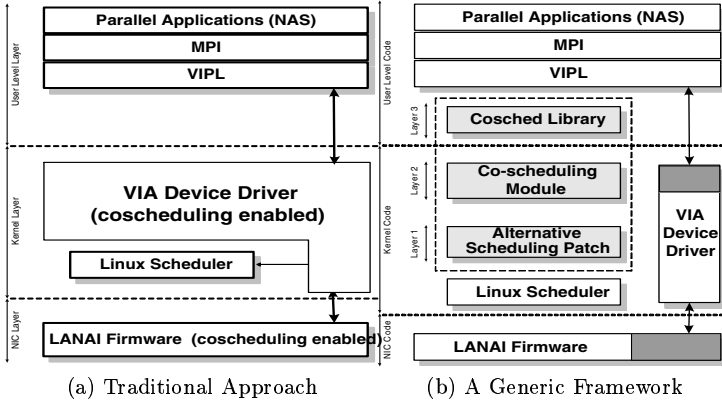


Fig. 1. Two Coscheduling Implementation Alternatives

Implementation of these schemes on a cluster needs significant modifications in the user-level communication layer such as VIA, NIC firmware and the device driver as shown in Figure 1 (a). Changing from one platform to another needs rework of the entire effort for each algorithm. In view of this, we present a generic, reusable framework that can be used to implement any coscheduling algorithm across various platforms, using well defined interfaces for the NIC, kernel and user layers [8]. This framework has been implemented on a Myrinet connected 16-node Linux cluster that uses industry standard VIA [6] as the user-level communication abstraction. In our framework (see Figure 1 (b)), an *alternative scheduling patch*, integrated in the local Linux scheduler, invokes the *coscheduling module* in which all the coscheduling algorithms used in this paper

are implemented. The *coscheduling module* chooses the next possible running process based on the underlying coscheduling algorithm and provides appropriate interfaces to the VIA device driver and NIC firmware. The selected process is returned to the Linux native scheduler and the native scheduler intelligently decides whether to execute or ignore the new process, based on the overall system load. Otherwise, the Linux native scheduler selects on other process from its run-queue. Further details of CC scheme and the framework are also described in [8].

Table 1. Workload mixes used in this study.

Category	Workload	Applications	Communication Intensity
Parallel, Homogeneous	<i>Wl1</i>	(1,2,4,6)EPs	low
	<i>Wl2</i>	(1,2,4,6)LUs	Medium
	<i>Wl3</i>	(1,2,4,6)MGs	High
	<i>Wl4</i>	(1,2,4,6)CGs	Very high
Parallel, Heterogeneous	<i>Wl5</i>	EPs + MGs	Low + High
	<i>Wl6</i>	LUs + CGs	Medium + Very High

3 Job Allocation Strategies

This section explores three strategies that affect the way parallel jobs are allocated on a cluster. Their relative impacts are explained in detail through a series of experiments in subsequent sub-sections¹. Before discussing the strategies, we explain the experimental testbed and the workload used in this study.

3.1 Experimental Platform and Workload

Our experimental testbed is a 16-node Linux (2.4.7-10) cluster, connected through a 16-port Myrinet [9] switch. Each node is an Athlon 1.76 GHZ uni-processor machine, with 1 GB memory and a PCI based on-board intelligent NIC [9], with 8 MB of on-chip RAM and a 133 MHZ Lanai 9.2 RISC processor. We have significantly enhanced and used the Berkeley’s VIA implementation (version 3.0) over Myrinet as our user-level communication layer and NERSC’s MVICH (MPI-over-VIA) implementation [10] as our parallel programming library.

For our parallel workload, we consider 4 applications from the NAS parallel benchmark suite [11] : EP, LU, MG and CG; with lowest to highest communication intensities, respectively. Using combinations of these 4 applications, we designed a set of 6 parallel workloads as shown in Table 1. *Wl1* through *Wl4* exhibit uniform, homogeneous characteristics with low, medium, high and very high

¹ In addition, we have analyzed the impact of CPU and I/O intensive several jobs on the performance of parallel jobs. The results are omitted here due to space limitation, but can be found in [8]

communication intensities respectively. *W15* and *W16* exhibit non-uniform, heterogeneous characteristics with mixed communication intensities. The numbers in the bracket represent the multi programming level of different applications. Total size of all applications, when run together, fits well within the memory (1GB), and hence, we incur no swapping overheads. Now, we examine various metrics that potentially influence the job-allocation decision using 4 coscheduling algorithms (DCS, PB, SB and CC).

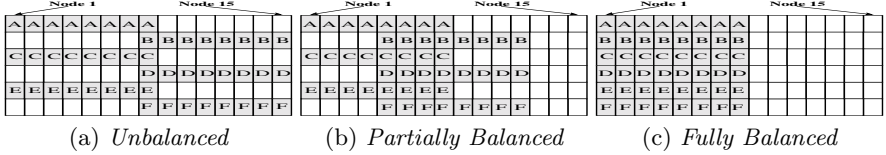


Fig. 2. Sharing Strategies for allocating multiple jobs: (a) allocates to use all 15 nodes, overloading just 1 node, (b) overloads 4 nodes and frees up 3 nodes and (c) overloads 8 nodes to free up 7 extra nodes.

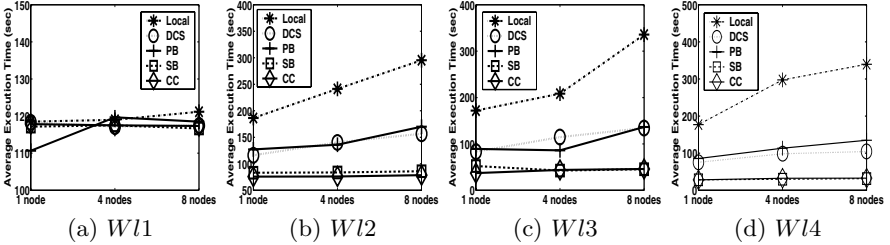


Fig. 3. Performance Analysis of sharing the number of nodes.

3.2 Effect of Job Placement

Typically, prior implementations of coscheduling algorithms have assumed the worst case scenario, where all parallel applications require maximum number of available nodes (16, in our case). Let us consider a more realistic case, where jobs have differing node requirements and not enough nodes are available to allocate all jobs independently. In such cases, we would like to see how sharing of multiple jobs can impact the overall performance. There are two basic issues to consider: (1) How many nodes are overloaded and (2) How many jobs are affected due to arrival of a new job?

For the first issue, considering the best, average and worst case of node sharing, we examine three job allocation techniques as shown in Figures 2 (a) through (c), respectively. We consider 6 class-A MG applications (each requires 8 nodes), and allocate them using three patterns (on available 15 nodes) as shown in Figure 2. Analyzing the results of these schemes under various workloads as shown in

Figures 3 (a) through (d), we find that only in the presence of a good coscheduling mechanism like CC [8] or SB [1][2], there is no difference on the average execution time of the 6 applications. Note that *Wl1* is a very low communication workload, hence all coscheduling schemes perform almost the same. We expect that with a good coscheduling algorithm, we can choose any of the three allocation schemes and still achieve the best results. This conclusion is important because it can help optimize upon total number of nodes to be shared, and hence, can directly increase the overall throughput.

For the second issue, we consider three important cases: (1) Incoming job affects only 1 job of its own size (best case) (2) Incoming job affects 1 job, but of larger size (Intermediate case) and (3) Incoming job affects multiple (atleast two) jobs (Worst case). The three allocations are shown in Figures 4 (a) through (c), respectively, when A and B are resident jobs and C is the incoming job. Results of these allocation techniques are quite as expected : Figure 4 (a) is the best strategy in all cases, irrespective of any coscheduling algorithm used, followed by (b) and (c) respectively. We do not show these results for space limitations.

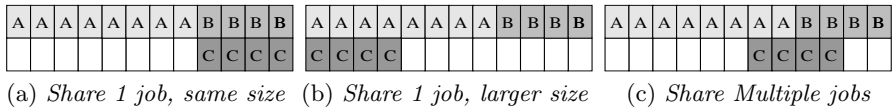


Fig. 4. Allocating a new job on nodes of a fully loaded machine

3.3 Effect of Multi-programming Level (MPL)

When coscheduling multiple jobs on cluster nodes, it is important to dynamically determine the maximum threshold number of parallel processes (*thN*) that can be successfully multi-tasked at a time. By successfully, we mean that the time taken by the *thN* processes when executed together should be less than or equal to the sum of the times they took, when executed in isolation. This metric is important from the allocation view-point because, if the allocator is aware of such a threshold, it can choose the nodes intelligently and avoid overloading on a node. Depending on the workload type of coscheduling scheme we use, such a threshold can vary. This is evidently seen in Figures 5 (a) through (d), where we plot the average execution time *per-application* of workloads *Wl1* through *Wl4*, respectively, as we increase the MPL. We see that blocking based schemes like CC [8] and SB [1][2] are most tolerant to increase in the MPL, as the execution time *per-application* remains nearly constant. On the other hand, spin-based schemes suffer (sometimes drastically), especially in communication intensive workloads like *Wl2*, *Wl3* and *Wl4*. This can be explained by the fact that as we increase the MPL, the likelihood of processes remaining coscheduled for a longer time gets lesser. This makes blocking and wakeup on demand a better option than spinning. We conclude from this result that in the presence of a good coscheduling algorithm (like CC or SB), we can allocate reasonably higher

number of jobs (tested upto MPL 6) without performance penalty. Such schemes not only reduce the response time per-job, but also increase the overall system throughput, because multi-programming frees up other nodes.

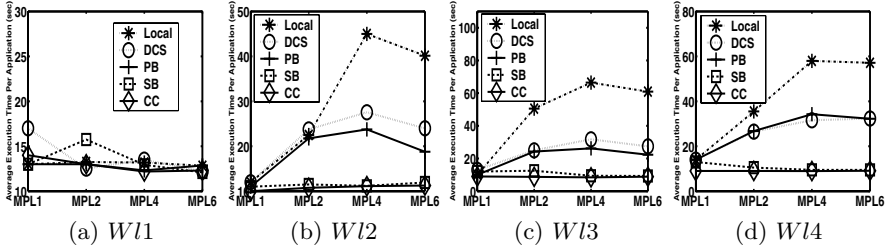


Fig. 5. Effect of MPL on the average execution time per application.

3.4 Effect of Variance in Communication Intensity

Another factor often neglected during job allocation is the communication intensity of the jobs. To see how it affects job allocation, we consider 2 types of workloads (*W15* and *W16*), each mixed with jobs of varying communication intensities, and allocate them in two different ways: homogeneous and mixed, as shown in Figures 6 (a) and (b), respectively. Figures 6 (c) and (d) show the impact of various coscheduling algorithms on these job allocation strategies for workloads *W15* and *W16*, respectively. We observe that for *W15*, mixed allocation is better, while for *W16*, homogeneous allocation is better. However, striking observation from both these results is that in the presence of a good coscheduling algorithm like CC and SB, both allocation schemes perform nearly the same. This leads us to the conclusion that with CC or SB, job allocator design gets simpler, as communication intensities have little impact on average job execution time.

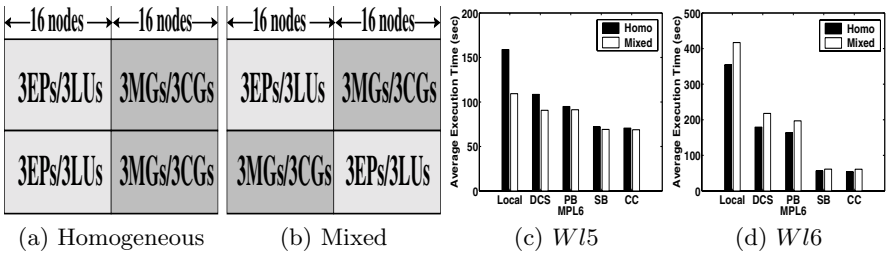


Fig. 6. Allocation strategies [(a),(b)] and their effect on various job mixes [(c),(d)].

4 Conclusion and Future Work

Although several communication-driven coscheduling algorithms have been proposed for improving the performance of parallel jobs in a cluster environment, the allocation techniques that impact the relative merits of these algorithms have barely been studied. In this experimental work on a Myrinet connected 16-node Linux cluster and real workloads, we have analyzed several critical factors that can influence job-allocation decisions, and hence affect the overall performance. These factors include MPL, job placement strategies and communication intensity of parallel jobs. Several important conclusions about coscheduling algorithms can be derived from our experimental results. First, when not enough nodes are available to execute jobs independently, node-sharing becomes critical for performance. We find that in the presence of coscheduling mechanisms like CC and SB, maximizing node-sharing is a good option as the per-application execution time remains nearly constant. However, with other mechanisms (Local, DCS, PB) the execution time increases with sharing, hurting the overall performance. Moreover, sharing should be done intelligently not to affect more than a single parallel job. The node sharing was tested upto a relatively high MPL of six, and with CC and SB, there was no adverse impact on the performance of any parallel job. Second, we find that mixing jobs of differing communication intensities does not hurt performance in the presence of blocking-based coscheduling algorithms like CC and SB. This strengthens the case for using blocking-based coscheduling algorithms even further.

References

1. A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with Implicit Information in Distributed Systems," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1998.
2. S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das, "A Closer Look at Coscheduling Approaches for a Network of Workstations," in *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 96–105, June 1999.
3. P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic Coscheduling on Workstation Clusters," in *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 231–256, March 1998. LNCS 1459.
4. T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.
5. S. Pakin, V. Karamcheti, and A. A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs," *IEEE Concurrency*, vol. 5, pp. 60–72, April-June 1997.
6. Compaq, Intel and Microsoft Corporations, "Virtual Interface Architecture Specification. Version 1.0," Dec 1997. Available from <http://www.vidf.org>.
7. Dror G. Feitelson, Larry Rudolph, "Mapping and scheduling in a shared parallel environment using distributed hierarchical control," in *International Conference on Parallel Processing*, vol. 1, pp. 1–8, 1990.

8. S. Agarwal, "A Generic Infrastructure for Coscheduling Mechanisms on Clusters," Dec 2002. M.S. Thesis. Available from <http://www.cse.psu.edu/~sagarwal/sagarwalMSThesis.pdf>.
9. N. J. Boden *et al.*, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29–36, February 1995.
10. National Energy Research Scientific Computing Center, "M-VIA: A High Performance Modular VIA for Linux," 2001. Available from <http://www.nersc.gov/research/FTG/via/>.
11. N. A. S. division., "The nas parallel benchmarks (tech report and source code)." Available from <http://http://www.nas.nasa.gov/Software/NPB/>.

Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids

Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro

Universidade Federal de Campina Grande, Departamento de Sistemas
e Computação, Av. Aprício Veloso s/n, Bodocongó, 58.109-970.
Campina Grande, PB, Brazil.
{danielps,walfredo,fubica}@dsc.ufcg.edu.br

Abstract. Scheduling independent tasks on heterogeneous environments, like grids, is not trivial. To make a good scheduling plan on this kind of environments, the scheduler usually needs some information such as host speed, host load, and task size. This kind of information is not always available and is often difficult to obtain. In this paper we propose a scheduling approach that does not use any kind of information but still delivers good performance. Our approach uses task replication to cope with the dynamic and heterogeneous nature of grids without depending on any information about machines or tasks. Our results show that task replication can deliver good and stable performance at the expense of additional resource consumption. By limiting replication, however, additional resource consumption can be controlled with little effect on performance.

1 Introduction

Recent years have seen increased availability of powerful computers and high-speed networks. This fact has made it possible to aggregate geographically dispersed resources for the execution of large-scale resource-intensive applications. This aggregation of resources has been called a *Computational Grid*. Grids may be composed by a wide variety of computers, visualization devices, storage systems and databases, scientific instruments, all connected through combinations of local and wide area networks. Furthermore, multiple users can simultaneously use these resources to execute a variety of large parallel applications. This characterizes grids as being heterogeneous and very dynamic.

Due to the wide distribution, heterogeneity and dynamicity of grids, loosely coupled parallel applications are better suited for execution on grids than tightly coupled applications. In particular, one can argue that Bag-of-Tasks (BoT) applications (i.e. applications whose tasks are completely independent) are the most suitable kind of application for current grid environments. Moreover, there are many important BoT applications, including data mining, massive searches (such as key breaking), parameter sweeps, Monte Carlo simulations, fractals calculations (such as Mandelbrot),

and image manipulation applications (such as tomographic reconstruction). In short, BoT applications are a class of relevant applications that are well suited for execution on grids.

However, scheduling BoT applications on grids is still an open problem. Good scheduling requires good information about the grid resources, which is often difficult to obtain. Known knowledge-free scheduling algorithms usually have worse performance than algorithms that have full knowledge about the environment.

We developed the *Workqueue with Replication* (WQR) algorithm to solve this problem. WQR delivers good performance without using any kind of information about the resources or tasks. It basically adds task replication to the classic *Workqueue* algorithm to cope with the dynamic and heterogeneous nature of *Computational Grids*. When a task is replicated, the first replica that finishes is considered as the valid execution of the task and the other replicas are cancelled. With this approach we minimize the effects of the dynamic machine load, machine heterogeneity and task heterogeneity, and do so without relying on information on machines or tasks. A way to think about WQR is that it allows us to trade some additional CPU cycles for the need of information about the resources on the grid.

The performance of WQR is similar or even better than solutions that have full knowledge about the environment (which is not feasible to obtain in practice), at the expense of consuming more cycles. In some scenarios, the additional cycles consumed by WQR are negligible. In the scenarios this is not the case, the extra cycles from replication can be controlled by limiting replication, a solution that shows little impact on the performance attained by WQR. Moreover, BoT applications often use cycles that would otherwise go idle cycles [7]. Thus, trading CPU cycles for the need of information can be advantageous in practice.

2 Scheduling *Bag-of-Tasks* (BoT) Applications on Grids

Scheduling applications on such a heterogeneous and dynamic environment like a *Computational Grid* is not a trivial task. Some characteristics that are intrinsic to grids should be considered during the scheduling, such as resource heterogeneity, the dynamic nature of the machine and network loads, the bandwidth, latency and topology of the network.

Of course, how difficult scheduling is, depends on the characteristics of the target application. Even scheduling BoT applications, which one could think is easy due to its simplicity, is not a trivial task. Scheduling BoT applications on grids is difficult due to the dynamic behavior and the intrinsic resource heterogeneity exhibited by most grids. Grid environments are typically composed by shared resources and thus the contention created by other applications running simultaneously on these resources causes delays and degrades the quality of service. Also, resources on grids are heterogeneous and may not perform the same way for all applications. Achieving good performance in this situation usually requires the use of good information to make the scheduling plan.

Unfortunately, due to grid's very wide distribution, it is usually difficult to obtain good information about the entire grid. There were some efforts to instrument the grid

to obtain dynamic information such as host load and network latency and bandwidth [5] [8] [14]. Some initial results are encouraging [8] [14], but making such monitoring scale to full grid size involves a number of obstacles [5]. Moreover, sometimes it is not possible to install monitoring systems at the user's will. Sites often place administrative restrictions on what can be ran and network traffic is filtered by firewalls. Finally, monitoring per se is not enough. In order to assure good scheduling, performance prediction based on monitored values is often desirable, what further complicates the issue.

Moreover, it is also difficult to obtain information about the tasks (execution time) to elaborate the scheduling plan. Sometimes the only way to obtain it is to run the task on a given processor. Of course this strategy only works in cases that all tasks have the same complexity or are very similar. A task can be executed in all processors and use its execution time on them to predict the execution time of the remaining tasks.

Despite the difficulties in obtaining good information about both grid resources and tasks, most efforts in scheduling applications composed by independent tasks assume good information is available (e.g. [1] [6] [9] [10]).

We developed a dynamic algorithm (WQR) for scheduling BoT applications on grid environments. This solution does not need any kind of information. WQR will be detailed in Section 2.4. To compare the performance of our solution we considered only dynamic algorithms because of their better suitability on grids. The algorithms that we chose to be compared are: *Dynamic Fastest Processor to Largest Task First (Dynamic FPLTF)*, *Sufferage* and *Workqueue*. These algorithms were chosen because they are well known and studied algorithms. *FPLTF* is a static scheduler that presents good performance on dedicated environments [10]. *Dynamic FPLTF* is the result of a modification we made on static *FPLTF* to make it adaptive and hence usable on grid environments. *Sufferage* has been shown to attain good performance on grids [1]. *Workqueue* was obviously chosen because our solution extends it.

2.1 Dynamic FPLTF

A good representative of static schedulers for Bag-of-Tasks applications is *Fastest Processor to Largest Task First (FPLTF)* [10]. However, the dynamicity and heterogeneity of resources present on grids make static schedulers not a good solution to be used on grids. To cope with this problem we changed *FPLTF*, making it dynamic. *Dynamic FPLTF* has a good ability to adapt to the dynamicity and heterogeneity of the environment. *Dynamic FPLTF* needs three types of information to schedule the tasks properly: Task Size, Host Load and Host Speed. Host Speed represents the speed of the host and its value is relative. A machine that has Host Speed = 2 executes a task twice faster than a machine with Host Speed = 1. Host Load represents the fraction of the machine that is *not* available to the application. Finally, Task Size is the time necessary for a machine with Host Speed = 1 to complete a task when Host Load = 0. In the beginning of the algorithm, the Time to Become Available (TBA) of each host is initialized with 0 and the tasks are sorted by size in a descending way. Therefore, the largest task is allocated first. A task is allocated to the

host that provides the better completion time CT ($CT = TBA + \text{Task Cost}$, where $\text{Task Cost} = (\text{Task Size} / \text{Host Speed}) / (1 - \text{Host Load})$).

When a task is allocated to a host, the TBA value corresponding to this host is incremented by Task Cost . Tasks are allocated until all machines of the grid are used. After that, the execution of the application begins. When a task finishes, all tasks that are not running are unscheduled and rescheduled again until all machines become used. This scheme continues until all tasks are completed. This strategy tries to minimize the effects of the dynamicity of the grid. A problem would arise if a lightly loaded machine becomes heavily loaded. This load variation would compromise the whole application since the tasks scheduled to this machine would execute much slower. The task rescheduling process corrects this problem by allocating larger tasks prioritizing the currently faster machines. This scheme leads to a good performance, as we shall see in Section 3.2, but requires too much information about the environment (Task Size , Host Load and Host Speed), making it hard to deploy in practice.

2.2 Sufferage

The idea behind *Sufferage* [1] is that a machine is assigned to a task that would “suffer” the most if that machine would not be assigned to it. The sufferage value of a task is defined by the difference between its second best and its best completion time (CT). The sufferage value of each task in the “bag of tasks” is calculated and a task is possibly assigned to the machine that gives its best completion time. If another task was previously assigned to the machine, the sufferage values of the task previously assigned and of the new task are compared. The task that stays in the host is the one that has the greater sufferage value, the other one returns to the “bag of tasks” to be assigned later.

Of course, the sufferage values of each task vary during the application execution due to the load dynamicity intrinsic in grid environments. To cope with that, we invoke *Sufferage* many times during the execution of an application. Every time a task finishes, all tasks that have not started yet are unscheduled and the algorithm is invoked again, using the current sufferage values. The algorithm runs again to schedule the remaining tasks, but this time with the updated load of the machines. This scheme is repeated until all tasks are completed. The problem of this algorithm is the same as *Dynamic FPLTF*; it needs too much information to calculate the completion times of the tasks.

2.3 Workqueue

Workqueue is a knowledge-free scheduler in the sense that it does not need any kind of information for task scheduling. Tasks are chosen in an arbitrary order in the “bag of tasks” and sent to the processors, as soon as they become available. After the completion of a task, the processor sends back the results and the scheduler assigns a new task to the processor. That is, the scheduler starts by sending a task to every available host. Once a host finishes its task, the scheduler assigns another task to the host.

The idea behind this approach is that more tasks will be assigned to the fast/idle machines while the slow/busy machines will process a small load. The great advantage here is that *Workqueue* does not depend on performance information. Unfortunately, *Workqueue* does not attain performance comparable to schedulers based on full knowledge about the environment, as we shall see in Section 3.2. The problem with *Workqueue* arises when a large task is allocated to a slow machine towards the end of the schedule. When this occurs, the completion of the application will be delayed until the complete execution of this task.

2.4 Workqueue with Replication (WQR)

Due to the difficulty on consistently obtaining good performance information about grid machines and application tasks, we have decided to implement a scheduling algorithm that is not based on performance information. We call our solution *Workqueue with Replication* (WQR), since it basically adds task replication to the *Workqueue* algorithm. Its performance is equivalent to solutions that have full knowledge about the environment (which are not feasible in practice), at the expense of consuming more cycles.

The WQR algorithm uses task replication to cope with the heterogeneity of hosts and tasks, and also with the dynamic variation of resource availability due to contention created by others users. The beginning of the algorithm execution is the same as *Workqueue* and continues the same until “the bag of tasks becomes empty”. At this time, in *Workqueue*, hosts that finish their tasks would become idle during the rest of the application execution. Using the replication approach, these hosts are assigned to execute replicas of tasks that are still running. Tasks are replicated until a predefined maximum number of replicas is achieved. When a task finishes, its replicas are cancelled. With this approach, we increase the performance on situations that tasks are delaying the complete execution because they were assigned to slow/busy hosts. When a task is replicated, there is a greater chance that a replica is assigned to a fast/idle host.

Note that the replication assumes that the tasks do not cause collateral effects (e.g. Database accesses by independent tasks can generate inconsistency). This assumption is reasonable when talking about grid environments because it is not common to have this kind of application on environments like that due to their widely distribution.

The good point of our solution is that it does not use information about hosts (speed, load) or task sizes. The negative point is that to achieve good performance it wastes CPU cycles with task replicas that are cancelled. That is, the CPU used was not useful for application processing. As we shall see, however, when the application granularity is not high, WQR wastes few cycles compared to the cycles need to execute the application (up to 40% more). Alas, with high application granularity the extra resource consumption of WQR can be significant (up to 105%, in our experiments). In these cases, replication can be limited, keeping cycle’s wasting below 50% and still attaining good performance.

In summary, our approach tries to minimize the problem of scheduling independent tasks in grid environments, while avoiding dependence on any kind of information like

task size and processor performance. This kind of information is helpful to make a good scheduling plan but it can be difficult to obtain due to the distributed nature of grids. Further, when it is possible to obtain this information, the predictions for computing and networking resources are not always accurate and reliable.

3 Performance Evaluation

The main objective of our experiments is to evaluate the performance of different scheduling algorithms. The experiments help to evaluate the influence of the grid machines heterogeneity (different speeds), the application tasks heterogeneity (size variation) and the granularity of application tasks (number of tasks per machine).

We performed approximately 8,000 experiments to compare the performance of different scheduling heuristics for BoT applications. Three other algorithms were evaluated together with the *Workqueue with Replication* approach: *Workqueue*, *Dynamic Fastest Processor to Largest Task First (Dynamic FPLTF)* and *Sufferage*. Each experiment consists of six simulations. All algorithms (*Workqueue*, *Dynamic FPLTF*, *Sufferage*, WQR 2x, WQR 3x and WQR 4x) are simulated with the same set of machines and tasks. 2x, 3x and 4x denote the maximum amount of replication WQR is allowed to do.

3.1 Simulation Environment

To run our experiments, we used the Simgrid toolkit [13]. This toolkit provides basic functions for the simulation of distributed applications in grid environments. In our work, we make the assumption that the network transfer times are negligible because we are dealing with applications that have small input/output data. This means either that the tasks are CPU bound or that large amounts of data have been previously staged in the grid, as it is common practice for data grid applications [4] [12]. Another point that has to be emphasized about our experiments is that the information-based algorithms (*Dynamic FPLTF* and *Sufferage*) are fed with perfect data about machines and tasks, something that is almost impossible in the real world.

All experiments have a fixed value for grid power (sum of processor capacities of all machines) and application size. The fixed value for the grid power is 1000. That, for example, could be 1000 machines with power 1. Machine power represents how fast the hosts can execute tasks. A host with power 2 can execute a "10 seconds task" in 5 seconds. The fixed value for the application size is 3600000 seconds. In a perfect world (where all machines are 100% free and all tasks finish simultaneously), this application could be completed in exactly 1 hour (3600 seconds) within a grid whose power is 1000. Note that, by fixing grid power and application size, differences in application completion time can be credited solely to the scheduler.

In order to simulate the heterogeneity of the machines in approximately five years according to the Moore's Law [3], the speed of the machines is taken from the uniform distribution $U(10-(hm/2), 10+(hm/2))$. The possible values for hm are 1, 2, 4, 8 and 16. The major purpose of this distribution is to keep the average speed of all machines in approximately 10. Thus, the scheduler is made the responsible for the appli-

cation performance because all grids used in the experiments have roughly the same number of machines. In particular, $hm = 1$ means that the grid is homogeneous (all machines have speed 10). When hm is equal to 2, the speed of machines varies accordingly to the uniform distribution $U(9, 11)$ (difference of 2 units, keeping the average speed in 10) and so on. Machines are added to the grid accordingly to each distribution until the sum of their speeds reaches the grid power which is always 1000 (roughly 100 machines).

The load on each host is simulated by traces obtained from NWS [14] measurements on actual systems. These traces contain the percentage of free CPU as a function of time. The simulator uses these traces to make machines, in the experiment, behave similarly to real machines.

The experiments were defined in such a way that it could create 20 different types of application to be evaluated. The types of applications are divided in four major groups, where the application granularity varies. The mean task size of these groups are 1000, 5000, 25000 and 125000 seconds, following an exponential scale.

To simulate the heterogeneity of tasks in each on of the four major groups, the size of tasks varies, but the mean task size remains the same within the group. The variation on task sizes can be of 0%, 25%, 50%, 75% and 100% relative to the mean tasks size of the group. The group of applications whose mean task size is 5000 seconds, for example, is subdivided in five groups. In one group, all tasks have the same size (5000 seconds, variation 0%), homogeneous tasks. In the group where the variation is 25%, the time of execution of each task varies accordingly to the uniform distribution $U(4375, 5625)$ (from 12,5% less than 5000 to 12,5% more than 5000 seconds). For the group which the variation is 50%, the task sizes follow the uniform distribution $U(3750, 6250)$ (from 25% less than 5000 to 25% more than 5000 seconds) and so on. Tasks are added to the application accordingly to each distribution until the sum of their sizes reaches the application size which is always 3,600,000 seconds.

With the experiments defined this way, we could evaluate not only the influence of grid resources heterogeneity and tasks heterogeneity, but also the granularity of the application tasks. Varying the mean sizes of tasks (1000, 5000, 25000 and 125000 seconds), the number of tasks of the applications is also changing as can be seen in Table 1.

Table 1. Granularity of Application Tasks.

Mean Sizes of Tasks (seconds)	# of Tasks	Tasks per Machine
1000	3.600	36
5000	720	7,2
25000	144	1,44
125000	29	0,29

The main objective of this variation on the tasks per machine relation is to evaluate the performance impact on each scheduler when there are much more tasks than ma-

chines (3600 tasks/100 machines) and gradually reduce this relation to a situation where there is more machines than tasks (29 tasks/100 machines).

3.2 Result Analysis

Figure 1A shows the mean execution time for each scheduling algorithm working on the four major application groups. Note that each data point summarizes the five levels of machines heterogeneity and tasks heterogeneity.

The tendency that can be noted in this figure is that, in situations of smaller grain, the schedulers tend to have closer performances. This is because, when there are many tasks per machine, dynamic schedulers can keep all processors busy most of the time. Only during the very end of application execution (when some machines go idle), this situation changes due to the load unbalance and harms performance. As these grains grow, however, the differences between the schedulers' behaviors come up. Accordingly to Figure 1A, WQR achieves better performance than the other schedulers on applications that have up to 25000 of mean task size. By growing the mean task size, the trend is that all scheduling algorithms performance become worse because higher application granularity raises difficulties on the scheduling process. The larger the task size, the greater chances are that the performance of the machine executing it degrades. This machine can be shared among other users making it so slow that the execution of the whole application becomes impaired. This fact indicates that, schedule a big task to the faster machine in the moment is not always the best choice.

This phenomenon affects *Sufferage* and *Dynamic FPLTF*, algorithms that use information about the environment and the applications to make their scheduling decisions. As can be seen in Figure 1A, as the mean task size grows these algorithms have a nearly linear fall in their performances.

Workqueue achieves the same level of *Sufferage* and *Dynamic FPLTF* performances in situations where the grains (mean task size) are small, up to 5000 seconds. With larger tasks (25000 and 125000 seconds), *Workqueue* gets worse. This happens because this algorithm does not have any mechanism to avoid the scheduling of a large task to a slow machine at the end of the scheduling plan. *Workqueue* is better when the number of tasks is considerable higher than the number of machines, more tasks per machine.

The three WQR versions achieve better performance than the others schedulers in situations where there are more tasks than machines on the grid, up to 25000 seconds of mean task size. This better performance can be credited to the replication strategy. With this strategy, even if a task is initially scheduled to a slow machine, this task can be replicated on a faster machine and finishes before the "original one". Furthermore, even if it does not happen, the impairment to the final execution time can be not so big because the tasks are not very large. For the applications where the tasks are larger and the relation tasks per machine is less than 1 (mean task size = 125000 seconds), WQR does not achieve a good performance. Moreover, *Sufferage* and *Dynamic FPLTF* achieve better performance than WQR. The fall of the performance can be credited to the high application granularity. In cases like that, a large task and also its replica(s) can be scheduled to slow machines, impairing the whole application execution.

Figure 1B exhibits the impact of the grid machines heterogeneity on the schedulers' performances. Similar as before, each data point summarizes the five levels of tasks heterogeneity and the four major types of applications.

Workqueue does not achieve a good performance compared to the others schedulers, but shows some constancy on its performance while the machines heterogeneity level is less or equal to 8. On level 16, its performance is considerably worse. This happens because, as higher is the heterogeneity, higher can be the difference of a fast machine and a slow one. Choosing a slow machine to execute a large task causes a great impact on application execution time.

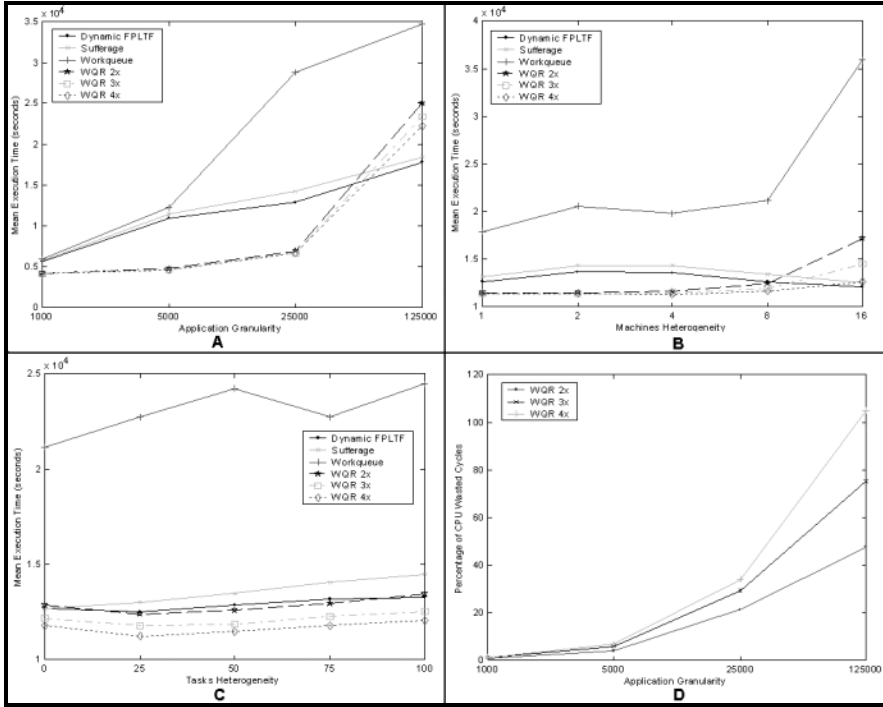


Fig. 1. (A) Schedulers Performance by Application Granularity. (B) Schedulers Performance by Machines Heterogeneity. (C) Schedulers Performance by Tasks Heterogeneity. (D) Percentage of Wasted CPU Cycles by Application Granularity

Although with better numbers, WQR displays a trend similar to *Workqueue*. Up to the level 8 of machines heterogeneity, WQR performance maintains practically unaltered. At the level 16 of machines heterogeneity, WQR suffers a fall of performance, but it is very slightly compared to the *Workqueue*'s fall. Nevertheless, WQR performance can be increased using more replicas, reducing the fall. This fall occurs due to the same reason of *Workqueue* but its effects are minimized due to the replication strategy used by WQR. Large tasks assigned to slow machines can be replicated to faster machines and finish before the "original ones".

The *Sufferage* and *Dynamic FPLTF* have an almost inverse behavior related to the machines heterogeneity variation. Growing the level of grid machines heterogeneity, the performance of these algorithms gets better. On level 16, both performances overcome WQR. This performance improvement achieved by these algorithms can be credited to the ability of choosing powerful machines to execute the larger tasks. Small tasks assigned to slow machines do not have a relevant impact on application execution time.

Figure 1C shows the impact of application tasks heterogeneity on the performance of the scheduling algorithms. On this chart, *Workqueue* confirms again its poorer performance quality compared to the remainder algorithms.

Sufferage and *Dynamic FPLTF* achieve closer results, but the second one suffers the least the impact of different sizes of tasks. The *Dynamic FPLTF* better results can be credited to its ability to order the application tasks by their sizes (the larger tasks are firstly assigned). Prioritizing “the task that suffers the most” to make the scheduling plan, the *Sufferage* algorithm not always prioritize larger tasks. The 2x version of WQR achieves a performance very similar to *Dynamic FPLTF*, proofing that it is a good alternative when it is not possible to obtain information about the environment and application. Increasing the replication level (3x and 4x), WQR still achieves better results. These two versions overcome the performance of the algorithms that use information about the environment and application to make the scheduling plan.

To achieve the great results presented above, WQR wastes CPU cycles with the replication, as shown in Figure 1D. The percentage of wasted CPU cycles is obtained from the division of the total cycles wasted with replicas by the total of cycles consumed (wasted and useful).

The cycles waste is considerably smaller for applications with 1000 and 5000 seconds of mean tasks sizes, achieving less than 5% of wasting. This percentage increases when the mean tasks sizes grows to 25000 seconds, but still the percentage of wasted cycles does not exceeds 40%. In the 125000 seconds category of applications, the percentage of wasted CPU cycles can exceed 100%. The increase of cycles wasting when the application granularity grows occurs due to the quick start of replication. When the application has small grains, the replication only starts after executing many tasks of the application and the fraction of time that the remaining tasks represent is small. When the application has large grains (e.g. 125000, more machines available than application tasks), the replication starts immediately wasting more cycles. Nevertheless, the replication can be limited to 2x and the percentage of wasting cycles can be maintained fewer than 50% as shown in the figure.

3.3 Implementation

We are finishing the process of implementing *Workqueue with Replication* (WQR) as part of MyGrid, a user-level tool for running BoT applications on grids [2][11]. So far, we have been able to observe that MyGrid’s overhead for task starting and cancellation is small. For tasks created “nearby” (the scheduler and the remote machine within the same local network), task overhead has not exceeded 7 seconds in 300 trials. For tasks created “far away” (the scheduler in Campina Grande, Brazil and the remote

machine in San Diego, USA), task overhead has not exceeded 17 seconds within 300 trials. Since the overhead has been small (compared to task size) and stable (has not changed much with the trials), we think it will not affect our results. In fact, since the overhead is stable, we could just include it in the task execution time, little changing our experiments.

4 Conclusions

In this paper we have proposed the *Workqueue with Replication* (WQR) task scheduling algorithm for computational grids. WQR targets Bag-of-Tasks applications, which are composed by independent tasks with no inter-task communication. WQR is similar to the classic *Workqueue* algorithm, but when hosts would otherwise go idle, they are used to compute replicas of the tasks that are still running. The idea here is that among a set of task replicas, the first one to finish is considered as being the normal execution of the task in question and the other copies are killed to make the hosts available.

WQR is not based on any kind of performance information, yet unlike other knowledge-free approaches, achieves good and consistent performance. This good performance is attained with additional increase in resource consumption to compensate the fact that WQR does not base its scheduling plan on information about machines and tasks. It is important to note that this additional resource consumption can be controlled by limiting replication, a solution that has little impact on performance. We feel that WQR is an important and practical contribution to a real problem. We are currently deploying WQR as part of MyGrid [2] [11]. Our hope is that this effort will enable more people to effectively compute on the grid.

Future work will be based on improvements to our grid model, including modeling file transfers present in I/O intensive applications that have not staged their data. A failure model will be included to encompass problems that make machines unavailable. We will also study the emergent behavior caused by multiple instances of our scheduler in the same grid. Of course, additional resource consumption imposes greater total load on the grid. On the other hand, individual applications finish earlier with WQR leaving a less loaded grid for applications that arrive in the future. Finally, we will try a simulation-based perfect algorithm to make a performance comparison with the replication approach. The idea is to determine a lower bound for our scheduling problem.

References

- [1] H. Casanova, A. Legrand and D. Zagorodnov et al. *Heuristics for Scheduling Parameter Sweep Applications in Grid Environments*. HCW. 2000.
- [2] W. Cirne, D. Paranhos and L.Costa et al. *Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach*. Submitted for publication. April 2003.
- [3] R. Dragan. *The Meaning of Moore's Law*.
<http://www.pcmag.com/article2/0,4149,4092,00.asp>. Online on February 14th 2003.
- [4] W. Elwasif, J. Plank and R. Wolski. *Data Staging Effects in Wide Area Task Farming Applications*. IEEE ISCC and the Grid, Brisbane, Australia, May 2001.

- [5] P. Francis, S. Jamin and V. Paxson et al. *An Architecture for a Global Internet Host Distance Estimation Service*. Proceedings of IEEE INFOCOM, 1999.
- [6] H. James, K. Hawick and P. Coddington. *Scheduling Independent Tasks on Metacomputing Systems*. The University of Adelaide. DHPC-066, 1999.
- [7] M. Litzkow, M. Livny, and M. Mutka. *Condor: A Hunter of Idle Workstations*. Proc. 8th International Conference of Distributed Computing Systems, pp. 104–111, 1988.
- [8] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. *A Resource Query Interface for Network-Aware Applications*. 7th IEEE HPDC, July 1998.
- [9] M. Maheswaran, S. Ali and H. Siegel et al. *Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems*. HCW, 1999.
- [10] D. Menascé, D. Saha and S. Porto et al. *Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures*. JPDC, pp. 1–18. 1995.
- [11] *MyGrid Web Page*. <http://dsc.ufcg.edu.br/mygrid/>. Online on February 14th 2003.
- [12] J. Plank, M. Beck and W. Elwasif et al. *The Internet Backplane Protocol: Storage in the network*. In NetStore '99: Network Storage Symposium. Internet2, October 1999.
- [13] *Simgrid*. <http://grail.sdsc.edu/projects/simgrid/>. Online on February 14th 2002.
- [14] R. Wolski. *Dynamically Forecasting Network Performance Using the Network Weather Service*. Cluster Computing, 1(1): 119–132, 1998.

Dynamic Load Partitioning Strategies for Managing Data of Space and Time Heterogeneity in Parallel SAMR Applications*

Xiaolin Li and Manish Parashar

The Applied Software Systems Laboratory
Department of Electrical & Computer Engineering
Rutgers University, Piscataway, NJ 08854, USA
{xlli, parashar}@caip.rutgers.edu

Abstract. This paper presents the design and experimental evaluation of two dynamic load partitioning and balancing strategies for parallel Structured Adaptive Mesh Refinement (SAMR) applications: the Level-based Partitioning Algorithm (LPA) and the Hierarchical Partitioning Algorithm (HPA). These techniques specifically address the computational and communication heterogeneity across refinement levels of the adaptive grid hierarchy underlying these methods. An experimental evaluation of the partitioning schemes is also presented.

Keywords: Dynamic Load Balancing, Parallel and Distributed Computing, Structured Adaptive Mesh Refinement, Scientific Computing

1 Introduction

Large-scale parallel/distributed simulations are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities in academia and industry. While the past decade has witnessed a significant boost in CPU, memory and networking technologies, the size and resolution of these applications and their corresponding computational, communication and storage requirements have grown at an even faster rate. As a result, applications are constantly saturating available resources. Dynamically adaptive techniques, such as Structured Adaptive Mesh Refinement (SAMR) [1], can yield highly advantageous ratios for cost/accuracy when compared to methods based upon static uniform approximations. SAMR provides a means for concentrating computational effort to appropriate regions in the computational domain. These techniques can lead to more efficient and cost-effective solutions to time dependent problems exhibiting localized features. Parallel implementations of these methods offer the potential for accurate solutions of physically realistic models of complex physical phenomena. However, the dynamics and space and time heterogeneity of the adaptive grid hierarchy underlying SAMR algorithms makes their efficient parallel implementation a significant challenge.

* Support for this work was provided by the NSF via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

Traditional parallel implementation of SAMR applications [3] [5] [7] have used dynamic partitioning/load-balancing algorithms that view the system as a flat pool of (usually homogeneous) processors. These approaches are based on global knowledge of the state of the adaptive grid hierarchy, and partition the grid hierarchy across the set of processors. Global synchronization and communication is required to maintain this global knowledge and can lead to significant overheads on large systems. Furthermore, these approaches do not exploit the hierarchical nature of the grid structure and the distribution of communications and synchronization in this structure.

This paper presents the design and experimental evaluation of dynamic load partitioning and balancing strategies for parallel SAMR applications that specifically address the computational and communication heterogeneity across refinement levels of the adaptive grid hierarchy underlying these methods. In this paper, we first analyze the space/time computational and communication behavior of parallel SAMR applications. We then present the design and experimental evaluation of two dynamic load partitioning and balancing strategies for parallel SAMR applications: Level-based Partitioning Algorithm (LPA) and Hierarchical Partitioning Algorithm (HPA). These algorithms are also presented in combination.

The rest of the paper is organized as follows. Section 2 gives an overview on SAMR technique and presents the computation and communication behavior of parallel SAMR applications. Section 3 presents the LPA and HPA dynamic partitioning and load balancing strategies. Section 4 presents the experimental evaluation of these partitioners. Section 5 presents conclusions.

2 Problem Description

SAMR techniques track regions in the domain that requires additional resolution and dynamically overlay finer grids over these regions. These methods start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are tagged and finer grids are overlaid on these tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting grid structure for the Structured Berger-Oliger AMR is a dynamic adaptive grid hierarchy [1].

2.1 Computation and Communication Behavior for Parallel SAMR

In the targeted SAMR formulation, the grid hierarchy is refined both in space and in time. Refinements in space create finer level grids which have more grid points/cells than their parents. Refinements in time mean that finer grids take smaller time steps and hence have to be advanced more often. As a result, finer grids not only have greater computational loads but also have to be integrated and synchronized more often. This results in space and time heterogeneity in the SAMR adaptive grid hierarchy. Furthermore, regridding occurs at regular intervals at each level and result in refined regions are created, moved

and deleted. Together, these characteristics of SAMR applications makes their efficient parallel implementation a significant challenge.

Parallel implementations of hierarchical SAMR applications typically partition the adaptive heterogeneous grid hierarchy across available processors, and each processor operates on its local portions of this domain in parallel. Each processor starts at the coarsest level, integrates the patches at this level and performs intra-level or ghost communications to update the boundaries of the patches. It then recursively operates on the finer grids using the refined time steps - i.e. for each step on a parent grid, there are multiple steps (equal to the time refinement factor) on the child grid. When the parent and child grid are at the same physical time, inter-level communications are used to inject information from the child to its parent. Dynamic re-partitioning and re-distribution is typically required after this step.

The overall performance of parallel SAMR applications is limited by the ability to partition the underlying grid hierarchies at runtime to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load. A critical requirement of the load partitioner is to maintain logical locality across partitions at different levels of the hierarchy and at the same level when they are decomposed and mapped across processors. The maintenance of locality minimizes the total communication and synchronization overheads.

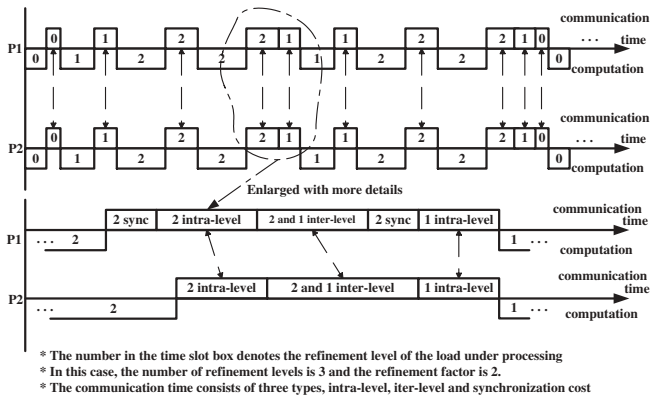


Fig. 1. Timing Diagram for Parallel SAMR Algorithm

The timing diagram (note that the timing is not drawn to scale) in Figure 1 illustrates the operation of the SAMR algorithms described above using a 3 level grid hierarchy. For simplification, only the computation and communication time behaviors of processors P1 and P2 are shown. The three components of communication overheads (listed in Section 2.2) are illustrated in the enlarged portion of the time line. This figure shows the exact computation and communication patterns for parallel SAMR implementations. Note that the timing diagram shows that there is one time step on the coarsest level (level 0) of the grid hierarchy followed by two time steps on the first refinement level and four time steps on the second level, before the second time step on level 0 can start. Also note

that the computation and communication for each refinement levels are interleaved. This behavior makes it quite challenging to partition the dynamic SAMR grid hierarchy to both balance load and minimize communication/synchronization overheads.

2.2 Communication Overheads for Parallel SAMR Applications

As described above and shown in Figure 1, the communication overheads of parallel SAMR applications primarily consist of three components: (1) *Inter-level communications* defined between component grids at different levels of the grid hierarchy and consist of prolongations (coarse to fine transfer and interpolation) and restrictions (fine to coarse transfer and interpolation). (2) *Intra-level communications* required to update the grid-elements along the boundaries of local portions of a distributed grid, consists of near-neighbor exchanges. These communications can be scheduled so as to be overlapped with computations on the interior region; (3) *Synchronization cost*, which occurs when the load is not well balanced among all processors. These costs may occur at any time step and at any refinement level due to the hierarchical refinement of space and time in SAMR applications.

Clearly, an optimal partitioning of the SAMR grid hierarchy and scalable implementations of SAMR applications requires careful consideration of the timing pattern as shown in Figure 1 and the three communication overhead components. Critical observations from the timing diagram in Figure 1 are that, in addition to balancing the total load assigned to each processor and maintaining parent child locality, we also have to balance the load on each refinement level and address the communication and synchronization costs within a level. The load partitioning and balancing strategies presented in the following section address these issues.

3 Dynamic Load Partitioning and Balancing Strategies

The dynamic partitioning algorithms presented in this paper are based on a core Composite Grid Distribution Strategy (CGDS) [7]. This domain-based partitioning strategy performs a composite decomposition

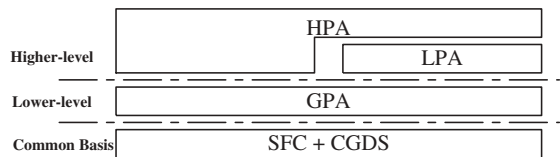


Fig. 2. Layers of Partitioning Algorithms

of the adaptive grid hierarchy using Space Filling Curve (SFC) [8]. Space filling curves are locality preserving recursive mappings from n -dimensional space to 1-dimensional space. CGDS uses SFCs, and partitions the entire SAMR domain into sub-domains such that each sub-domain keeps all refinement levels in the sub-domain as a single composite grid unit. Thus all inter-level communication are local to a sub-domain and the inter-level communication time is greatly reduced. The resulting composite grid unit list (GUL) for the overall domain must now be partitioned and balanced across processors.

A Greedy Partitioning Algorithm (GPA) is used to partition the global GUL to produces a local GUL for each processor. The key motivation for using the GPA scheme is

that it is fast and efficient as it scans the global GUL only once to try to equally distribute load among all processors. This is important as the number of composite grid units can be large and regridding steps can be quite frequent. This scheme works very well for homogeneous computational domain. However, for heterogeneous computational domains, it may cause large intra-level synchronization cost due to load imbalance for each refinement level.

To improve the performance of GPA scheme, we propose two efficient partitioning schemes, Level-based Partitioning Algorithm (LPA) and Hierarchical Partitioning Algorithm (HPA). As shown in Figure 2, GPA is a lower-level partitioning scheme which can work independently. LPA and HPA are higher-level partitioning schemes that work on the top of the lower-level scheme. Specially, HPA can work either on the top of LPA or directly on the top of GPA. Detailed description and analysis of LPA and HPA are presented in the following two subsections.

3.1 Level-Based Partitioning Algorithm

The level-based partitioning algorithm (LPA) essentially preprocesses a portion of the global GUL by disassembling it according to refinement levels, and feeds the resulting homogeneous composite GUL to GPA. The GPA then partitions this list to balance load. As a result of the preprocessing, the load on each refinement level is also balanced. This algorithm is as follows:

1. *Get the maximum refinement level $MaxLev$. Disassemble the global GUL into homogeneous GUL's according to grid unit's refinement depth, denoted by $gul\_array[lev]$. The load assigned in the previous iteration is denoted by $load\_array[np]$.*
2. *Loop for refinement level $lev = MaxLev$ to 0 reversely*
3. *Passing $gul\_array[lev]$ and $load\_array[np]$ to GPA to obtain local assignment.*
4. *In GPA, it will partition the load such that each processor get equal distribution on each refinement level.*

We observe that the LPA scheme partitions deep composite grid units before shallow grid units. The previous iteration has assigned some load on lower refinement level because we use the composite grid strategy. Since we cannot guarantee the perfect balance during the partition of each iteration, to compensate the possible imbalance introduced in higher level and equally partition the GUL on the lower level, we need to keep track of the load on lower level previously assigned. This is done using $load\_array[np]$. LPA takes full advantages of CGDS by keeping parent-children relationships in the composite grid and localizing inter-level communications. Furthermore, it balances the load on each refinement level which reduces the synchronization cost as shown by the experimental evaluation and demonstrated by the following example.

Consider partitioning a one dimensional grid hierarchy with two refinement levels, as shown in Figure 3. For this 1-D example, GPA partitions the composite grid unit list into two subdomains. These two parts contain exactly same load: the load assigned to $P0$ is $2 + 2 \times 4$ while the load assigned to $P1$ is 10 units. From the viewpoint of GPA scheme, the partition result is perfectly balanced. However, due to the heterogeneity of

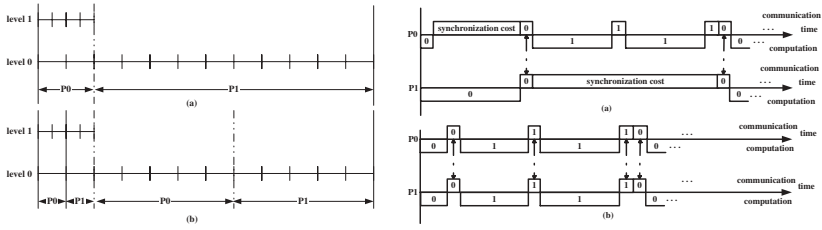


Fig. 3. Partitions of a 1-D Grid Hierarchy and Resulting Timing Diagrams (a) GPA (b) LPA

SAMR algorithm, this distribution leads to large synchronization costs as shown in the timing diagram of Figure 3 (a). The LPA scheme takes these synchronization costs at each refinement level into consideration. For this simple example, LPA will produce a partition as shown in Figure 3 (b) which results in the computation and communication behavior as shown in Figure 3 (b). As a result, there is an improvement in overall execution time and a reduction in communication and synchronization time.

3.2 Hierarchical Partitioning Algorithm

In most parallel implementations of SAMR, such as ParaMesh [5], SAMRAI [3] and GrACE [6], load partitioning and balancing is collectively done by all processors and all processors maintain a global knowledge of the total workload. These schemes have the advantage of a better load balance. However these approaches require the collection and maintenance of global load information which makes them expensive, specially on large systems. In HPA, after initially obtaining the global GUL, the top processor group partitions it and assign portions to each processor subgroup in a hierarchical manner. In this way, HPA further localizes the communication to subgroups, enables concurrent communication and reduces the global communication and synchronization costs.

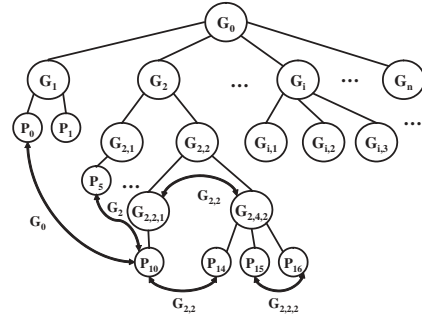


Fig. 4. A General Hierarchical Structure of Processor Groups

In HPA, after initially obtaining the global GUL, the top processor group partitions it and assign portions to each processor subgroup in a hierarchical manner. In this way, HPA further localizes the communication to subgroups, enables concurrent communication and reduces the global communication and synchronization costs.

Figure 4 illustrates a general hierarchical tree structure of processor groups, where, G_0 is the root level group (grouplevel=0) which consists of all processors. G_i is the i -th group at the group level 1. Note that only the leaves of the tree are processors. The communication between processors is accomplished through their closest common parent group. For example, for the processors P_{10} and P_{14} , they have common ancestor group G_0 , G_2 and $G_{2,2}$, where $G_{2,2}$ is their closest common ancestor. Thus communication between P_{10} and P_{14} is via the group $G_{2,2}$. Similarly, the communication between

processors P_0 and P_{10} is via the group G_0 . The detailed steps of HPA are as follows. More detailed description of HPA and its variants is presented in [4].

1. *Setup the processor group hierarchy according to group size and group levels.*
2. *Loop for group level $lev=1$ to num_group_level*
3. *Partition the global GUL into N_{lev} subdomains using GPA, where N_{lev} is the number of processor groups at this level.*
4. *Assign the load L_i on subdomain R_i to a group of processors G_i such that the number of processors NP_i in the group G_i is proportional to the load L_i , i.e., $NP_i = L_i / L_{sum} \times NP_{sum}$, where L_{sum} is the total size of load and NP_{sum} is the total number of processors in the parent group level.*
5. *Loop until reaching the leaves of the group tree hierarchy. Partition the load portion L_i using GPA and assign the appropriate portion to the individual processor in the group G_i , for $i = 0, 1, \dots, NP_j - 1$, where NP_j is the number of processors in the lowest group level.*

During the repartitioning phase, instead of performing global load balancing, HPA hierarchically checks for load imbalance. Starting from the lower level subgroups, if the imbalance is below some threshold, it only repartitions and redistributes load among the lowest subgroup, else if it is above the threshold, it checks for load imbalance in the next higher level group. It proceeds recursively up to the root processor group consisting of all processors in this way HPA thus reduces the amount of global synchronization required and enable incremental load balancing. Furthermore, HPA scheme exploits more communication parallelism through multiple concurrent communication channels among hierarchical groups.

4 Experimental Evaluation

The partitioning strategies are evaluated on the IBM SP2 cluster BlueHorizon at San Diego Supercomputer Center. The design of the adaptive runtime framework is driven by specific problems in enabling realistic simulations using AMR techniques. The 3-D Richtmyer-Meshkov instability application from computational fluid dynamics are used. RM3D has been developed by Ravi Samtaney as part of the virtual test facility at the Caltech ASCI/ASAP Center [2].

The input configurations are as follow: the base grid size is $128 \times 32 \times 32$, maximum 3 refinement levels, refinement factor is 2, granularity is 4, regrid every 4 time steps on each level, the total base level time steps are 100. Four partitioning schemes are used in the experiments, namely, GPA, LPA, HPA and HPA+LPA, running on a range of processors from 16 to 128. The comparison of the total execution time is shown in Figure 5. In this figure, the execution time for the different parameters are normalized against GPA on 16 processors (100%). We observe that execution time reduction using LPA scheme compared to GPA is about 48.8% on the average for these five system configurations. HPA scheme alone reduces the execution time by about 52.9% on the average. Applying HPA scheme on the top of LPA scheme, we gain further improvement reducing overall execution time by about 56% for 16 processors, 60% for 128 processors, and 57.3% on the average. These reductions in the overall execution times are due to a reduction

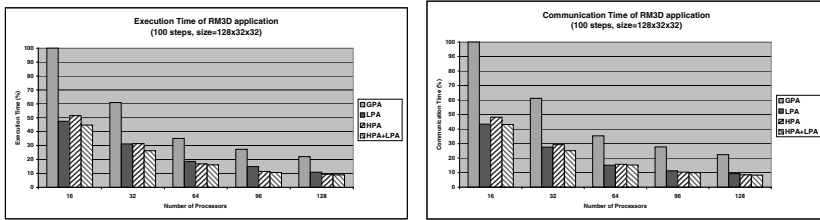


Fig. 5. Execution and Communication Time: GPA, LPA, HPA and HPA+LPA

in communication times as shown in Figure 5. The figure also shows that, HPA greatly reduces the global communication time and exploits more concurrent communications. For all cases, HPA+LPA delivers the best performance since it takes full advantages of HPA and LPA.

5 Conclusions

In this paper we presented the design and experimental evaluation of two dynamic load partitioning and balancing strategies for parallel Structured Adaptive Mesh Refinement (SAMR) applications: the Level-based Partitioning Algorithm (LPA) and the Hierarchical Partitioning Algorithm (HPA). These techniques specifically address the computational and communication heterogeneity across refinement levels of the adaptive grid hierarchy underlying these methods. The efficiency of LPA and HPA is experimentally validated using a real SAMR application (RM3D). The improvement over the GPA scheme is significant. The overall reduction of execution time is about 48.8% for LPA, 52.9% for HPA and 57.3% for HPA+LPA on the average.

References

1. M. Berger and J. Olinger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.
2. J. Cummings, M. Aivazis, R. Samtaney, R. Radovitzky, S. Mauch, and D. Meiron. A virtual test facility for the simulation of dynamic response in materials. *Journal of Supercomputing*, 23:39–50, 2002.
3. R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation – Practice & Experience*, 14(5):347–368, 2002.
4. X. Li and M. Parashar. Hierarchical partitioning techniques for structured adaptive mesh refinement applications. (to appear) *Journal of Supercomputing*, 2003.
5. P. MacNeice. Paramesh. <http://esdcd.gsfc.nasa.gov/ESS/macneice/paramesh/paramesh.html>.
6. M. Parashar. Grace. <http://www.caip.rutgers.edu/~parashar/TASSL/>.
7. M. Parashar and J. Browne. On partitioning dynamic adaptive grid hierarchies. In *29th Annual Hawaii Int. Conference on System Sciences*, pages 604–613, 1996.
8. H. Sagan. *Space Filling Curves*. Springer-Verlag, 1994.

An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm

Henan Zhao and Rizos Sakellariou

Department of Computer Science, University of Manchester, U.K.

Abstract. This paper considers the Heterogeneous Earliest Finish Time (HEFT) algorithm for scheduling the tasks of an application, represented by a directed acyclic graph, onto a bounded number of heterogeneous machines. We focus on the appropriate selection of the weight for the nodes and edges of the graph, and experiment with a number of different schemes for computing these weights. Our findings indicate that the length of the schedule produced may be affected significantly by the scheme used, and suggest that the mean value based approach used by HEFT may not be a particularly good choice.

1 Introduction

Among the scheduling algorithms for heterogeneous machines, the *Heterogeneous Earliest Finish Time* (HEFT) algorithm [3], a natural extension of the classical list scheduling algorithm for homogeneous systems to cope with heterogeneity, has been shown to produce shorter schedule lengths more often than other comparable algorithms. In classical list scheduling algorithms an application is viewed as a *directed acyclic graph* (DAG), where nodes (or tasks) represent computation and edges represent communication. By assigning a weight to each node (typically, the corresponding computation cost) and edge (typically, the corresponding communication cost), those algorithms prioritize the nodes to be scheduled on the basis of a value computed by a *rank function*; this is typically a function of the weights assigned to the nodes and edges of the graph. However, in a heterogeneous setting, the values typically used for the weights cannot be considered as constant any more: the computation cost of a task may vary, depending on the machine that this would run on; same, the communication cost may vary, depending on which machines are communicating. Although it has long been known, in homogeneous environments, that the choice of the rank function (and the values it returns) may affect the quality of the schedule produced, the proposers of HEFT have not examined the different (and additional with regard to homogeneous environments) possibilities that exist for computing the weights in a heterogeneous environment; they opted for the intuitively appealing mean values. In this paper, we consider a number of different options for computing the weights in HEFT. We find that, on average, the mean value option is not necessarily the most efficient choice, but, more importantly, the length of the schedules produced may differ significantly from one option to another.

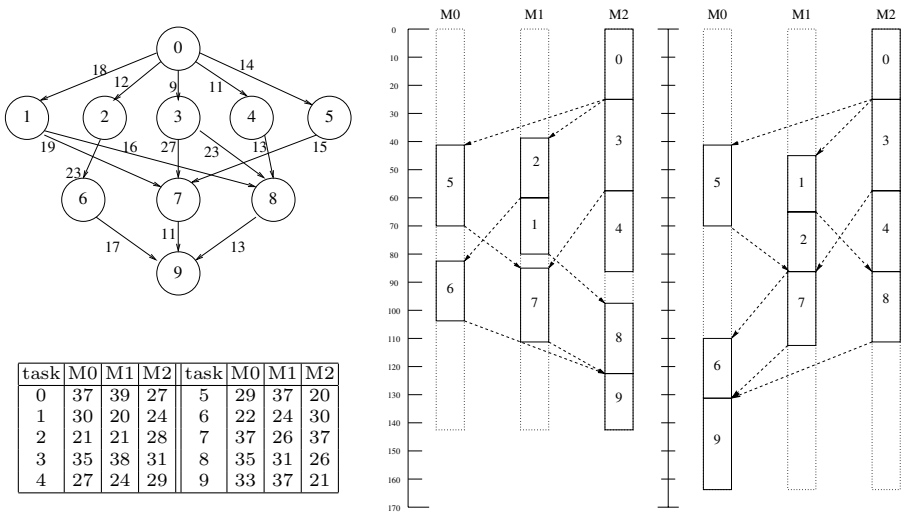


Fig. 1. A Motivating Example: Two different schedules for two different rank schemes.

2 Background and Motivation

The HEFT algorithm [3] works as follows. First, a weight is assigned to each node and edge of the graph, based on the average computation and communication, respectively. Then, the graph is traversed upwards and a rank value is assigned to each node; this is based on the sum of the weight of the node with the maximum value resulting from all possible summations that add the weight of an edge to an immediate successor node with the rank value of that successor node. Tasks are then scheduled, in order of their rank value, on the machine which gives the earliest finish time. A task may be scheduled in an idle slot between two already scheduled tasks on a machine as long as precedence constraints are preserved.

The use of the average computation and communication as weights in the graph has been an *ad hoc* choice in the HEFT algorithm. However, there are cases where the average value may not produce a good schedule. To illustrate this consider the graph shown in Figure 1 (example adopted from [3]). The communication cost between two nodes in the graph is given by the number next to each edge, except when the two nodes will run on the same processor, in which case it is zero. The computation cost of each task on three different machines is given by the table. If the nodes are prioritized using as a weight the mean values of the computation cost over all three machines (as in the original HEFT), then the schedule produced has a length equal to 164 (right-hand side of the figure). If, on the other hand, nodes are prioritized using as a weight the worst (i.e., maximum) values of the computation cost (over all three machines on which each node may run), the schedule produced has a length equal to 143. This significant improvement is the result of only a small difference in the priority order of the nodes; in the first case, the order is $\{0, 3, 5, 1, 2, 4, 7, 8, 6, 9\}$, whereas, in the second case, the order is $\{0, 3, 5, 2, 1, 4, 7, 8, 6, 9\}$. The question that motivated

this work is what is the behaviour of the mean value as an option to compute the weights over a large number of cases, and whether different schemes to compute the weights can lead to significant variations in the performance of HEFT.

3 Methodology

We have developed a program that implements the HEFT algorithm, allowing for different options for task prioritization. Its inputs are: a DAG representing an application; the number of machines of the heterogeneous platform, M ; the computation cost to execute each task of the DAG on each machine; the data transfer rate between each machine; and, the amount of data required to be transmitted from one task to another if these two tasks run on different machines.

When prioritizing tasks, we have considered both upward and downward ranking. Thus, the upward rank, $r_u(i)$, of a task i is recursively defined by

$$r_u(i) = f_1(w_i^0, \dots, w_i^m, \dots, w_i^{M-1}) + \max_{\forall j \in S_i} (f_2(c_{ij}^{00}, \dots, c_{ij}^{mm'}, \dots, c_{ij}^{M-1, M-1}) + r_u(j)),$$

where w_i^m is the computation cost of task i on machine m , $0 \leq m < M$, S_i is the set of the immediate successors of task i , and $c_{ij}^{mm'}$ is the communication cost between nodes i and j when i is executed by machine m and j by machine m' , $0 \leq m, m' < M$. The communication cost is derived by dividing the amount of data required to be transmitted between the tasks i and j by the data transfer rate between the two machines m and m' . It is assumed that when i and j are executed by the same machine (i.e., $m = m'$), the communication cost is zero. Furthermore, the function f_1 returns a value which is dependent on the computation cost of a given task on every machine, and the function f_2 returns a value which is dependent on the communication cost between two given tasks considering every combination of machines where the two given tasks may execute. Similarly, the downward rank, $r_d(i)$, of a task i is recursively defined by

$$r_d(i) = \max_{\forall j \in P_i} (f_1(w_j^0, \dots, w_j^m, \dots, w_j^{M-1}) + f_2(c_{ji}^{00}, \dots, c_{ji}^{mm'}, \dots, c_{ji}^{M-1, M-1}) + r_d(j)),$$

where P_i is the set of the immediate predecessors of task i .

Six approaches are used to compute a value for the functions f_1 , f_2 : (i) *Mean value* (denoted by the shorthand M) returns the average over all the input arguments; that is, f_1 returns the average computation cost of a task and f_2 returns the average communication cost between two tasks (this is the approach used in the original HEFT). (ii) *Median value* (ME) returns the median value over all the input arguments for both f_1 and f_2 . (iii) *Worst value* (W) returns the worst value (i.e., maximum computation cost) for f_1 and, for f_2 , the communication cost between the two machines on which each of the two communicating tasks has its highest computation cost. For example, in Figure 1, task 3 has its highest computation cost on machine M1 and task 8 has its highest computation cost on machine M0; then, the value returned by f_2 is specified by the amount of data transmitted between nodes 3 and 8 and the data transfer rate between machines

M0 and M1, i.e., c_{38}^{10} using the notation of the previous paragraph. (iv) *Best value* (B) returns the best value (i.e., minimum computation cost) for f_1 and, for f_2 , the communication cost as determined by the procedure described previously. (v) *Simple Worst value* (SW) returns the worst value for both f_1 and f_2 (i.e., maximum computation cost and maximum communication cost, respectively). (Recall that this approach was used to prioritize the nodes when the schedule of length equal to 143 was produced in the motivating example of Section 2) (vi) *Simple Best value* (SB) returns the best value for both f_1 and f_2 (i.e., minimum computation cost and minimum communication cost, respectively).

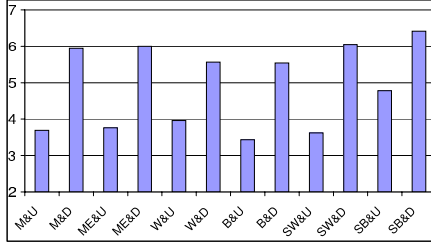
Since we use both upward and downward ranking with each of the above approaches, a total of $2 \times 6 = 12$ different schemes are considered.

4 Experimental Results and Discussion

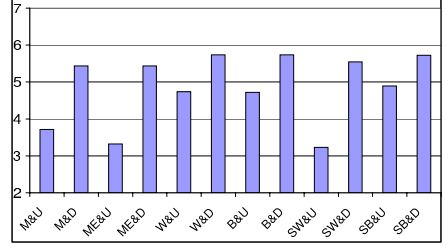
We compared the performance of the 12 different schemes in four sets of experiments using four metrics [1]. The *Average percentage degradation from the best* (APD) is the average (over all cases of a set of experiments) of the percentage of degradation of the schedule length of a particular scheme from the scheme returning the best solution. The *Number of best solutions* (NB) refers to the number of times a particular scheme was the *only one* that produced the shortest schedule. The *number of best solutions equal with another scheme* (NEB) counts those cases where a scheme produced the shortest schedule length but at least one scheme more achieved the same length. Finally, the *worst percentage degradation from the best* (WPD), is the maximum percentage degradation from the best of a given scheme over all cases of a particular set of experiments.

Two classes of DAGs were used in the experiments. In the first class, we randomly generate graphs having between 25 and 100 nodes as follows. Each graph has a single entry and a single exit node; all other nodes are divided into levels. Each level is created progressively and has a random number of nodes, which varies from two to half the number of the remaining (to be generated) nodes. In the second class of graphs, we used task graphs representing a Laplace equation solver, forms of which have been used in other related studies [2]. The versions considered in our experiments have between 25 and 400 nodes.

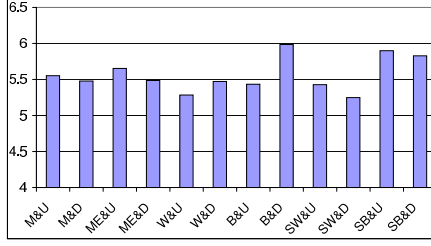
Two different approaches were used to specify the degree of heterogeneity of the machines. The first, referred by the name *random computation cost*, is similar to the approach used in [3], in that the computation cost for a given task and machine is selected randomly from a uniform distribution within a given range. The second approach attempts to take account of the observation that, in a heterogeneous environment, the factors affecting the execution of a given task on a given machine are related primarily to particular characteristics (e.g., CPU power) of the machine concerned. In this approach, a random number between 0.5 and 1 is first generated as a factor indicating the computational power of each machine. Then, the cost for a given task on a given machine is within 5% of the product of this number and a baseline computation cost for each task (selected randomly). This approach is referred by the name *proportional computation cost*.



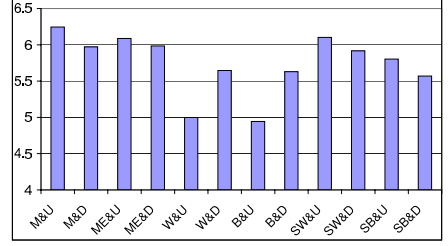
(a) Random DAGs, 25-100 tasks, 2-8 machines, random comp. cost



(b) Random DAGs, 25-100 tasks, 2-8 machines, proportional comp. cost



(c) Laplace, 25-400 tasks, 2-8 machines, random comp. cost



(d) Laplace, 25-400 tasks, 2-8 machines, proportional comp. cost

Fig. 2. Average percentage degradation from the best; 4 different sets of experiments.

All combinations above (2 families of graphs $\times$ 2 approaches for heterogeneity) lead to 4 different sets of experiments. For each set, we vary the task graph granularity so that a communication to computation ratio [1] between 0 and 4 is achieved; the number of available machines is also varied between 2 and 8. We consider a total of 2000 cases in each set of experiments.

The *APD* for each of the 12 schemes and for each set of experiments is shown in Figure 2. The shorthand denoting each scheme is a concatenation of: the shorthand of the approach used to compute the weights (see Section 3) and the suffix ‘&U’ or ‘&D’ depending on whether ranking is performed upward or downward, respectively. From the figure, the best 6 schemes in each set (starting with the best) are: (a) {B&U, SW&U, M&U, ME&U, W&U, SB&U}; (b) {SW&U, ME&U, M&U, B&U, W&U, SB&U}; (c) {SW&D, W&U, SW&U, B&U, W&D, M&D}; (d) {B&U, W&U, SB&D, B&D, W&D, SB&U}. Mean value schemes are outperformed in every experiment. In (a), M&U is worse than B&U by 7.6%; in (b), M&U is worse than SW&U by 14.8%; in (c), M&D is worse than SW&D by 4.4%; and, in (d), M&D is worse than B&U by 20.8%, while M&U is worst amongst all schemes. There is no clear pattern favouring one scheme; however, we notice that B&U is always among the best 4, and W&U is always among the best 5. Another observation is that upward schemes always outperform downward schemes for randomly generated DAGs, but not for Laplace.

Table 1 shows the values of the remaining three metrics, *NB*, *NEB*, *WPD*. On the basis of the value $NB + NEB$, the only scheme which is consistently

Table 1. *NB*, *NEB*, and *WPD* for each set of experiments in Figure 1.

	Experiment (a)			Experiment (b)			Experiment (c)			Experiment (d)		
	<i>NB</i>	<i>NEB</i>	<i>WPD</i>	<i>NB</i>	<i>NEB</i>	<i>WPD</i>	<i>NB</i>	<i>NEB</i>	<i>WPD</i>	<i>NB</i>	<i>NEB</i>	<i>WPD</i>
M&U	236	10	25.0	233	97	36.0	157	21	34.8	112	27	47.2
M&D	64	22	27.7	149	79	33.7	139	30	33.4	87	31	35.6
ME&U	186	72	26.4	200	185	21.1	133	13	34.8	121	31	32.4
ME&D	35	47	26.2	65	144	33.7	138	28	34.0	94	34	41.4
W&U	242	5	25.8	161	67	39.0	207	3	29.8	263	2	38.9
W&D	116	3	23.5	69	131	36.6	180	2	30.4	181	0	42.1
B&U	369	1	27.6	59	183	39.0	165	1	36.8	273	2	33.8
B&D	109	0	22.0	21	178	31.0	145	4	36.5	171	1	30.3
SW&U	242	72	26.4	225	193	21.1	167	17	34.8	117	24	32.0
SW&D	37	43	26.2	40	152	33.7	153	22	31.0	117	31	35.6
SB&U	159	5	29.2	19	189	39.0	190	0	34.4	191	1	42.2
SB&D	68	1	32.5	6	183	31.0	164	1	36.3	195	2	34.3

among the best 5 is W&U. Although the *WPD* metric, refers to a single bad case (one out of 2000), it is interesting to notice that the performance of any scheme may deviate from that of the best scheme significantly. At one extreme, M&U performed 47.2% worse than the best scheme in the case.

The main observations from the results presented can be summarized as follows: (i) There are significant differences between the performance of HEFT depending on the scheme used for computing the weights. (ii) There is no evidence to suggest that the use of mean values should be preferred; instead, our results seem to indicate the opposite. (iii) Upward ranking appears to perform in many cases, but not always, better than downward ranking. (iv) Based on the relative ranking of the performance of B&U and W&U with respect to other schemes, there is some evidence in favour of these two schemes.

Thus, the performance of HEFT can be improved by checking the schedule produced by each scheme (or some of them) and taking the best. This would increase the cost of the algorithm, but it may be a trade-off worth making. Future research could focus on ways to improve the scheduling process by making it less sensitive to the different schemes for ranking nodes.

References

1. Y.-K. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59, pp. 381–422, 1999.
2. A. Radulescu and A.J.C. van Gemund. Fast and Effective Task Scheduling in Heterogeneous Systems. *9th Heterogeneous Computing Workshop*, pp. 229–238, 2000.
3. H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3), pp. 260–274, March 2002.

Performance-Based Dynamic Scheduling of Hybrid Real-Time Applications on a Cluster of Heterogeneous Workstations¹

Ligang He, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd

Department of Computer Science, University of Warwick
Coventry, United Kingdom CV4 7AL
{ligang.he, saj, dps, grn}@dcs.warwick.ac.uk

Abstract. It is assumed in this paper that periodic real-time applications are being run on a cluster of heterogeneous workstations, and new non-periodic real-time applications arrive at the system dynamically. In the dynamic scheduling scheme presented in this paper, the new applications are scheduled in such a way that they utilize spare capabilities left by existing periodic applications in the cluster. An admission control is introduced so that new applications are rejected by the system if their deadlines cannot be met. The effectiveness of the proposed scheduling scheme has been evaluated using simulation; experimental results show that the system utilization is significantly improved.

1 Introduction

In cluster environments the nodes are rarely fully utilized [1]. In order to make use of the spare computational resources, scheduling schemes are needed to judiciously deal with the hybrid execution of existing and newly arriving tasks [2]. The work in this paper addresses the issue. This work has two major contributions. First, an optimal approach for modeling the spare capabilities of clusters is presented. Second, based on the modeling approach, a dynamic scheduling framework is proposed to allocate newly arriving independent *non-periodic real-time applications (NPA)* to a heterogeneous cluster on which *periodic real-time applications (PRA)* are running.

2 System Modeling

A cluster of heterogeneous workstations is modeled as $P = \{p_1, p_2, \dots, p_m\}$, where p_i is an autonomous workstation [4]. Each workstation p_i is weighted w_i , which represents the time it takes to perform one unit of computation. Each workstation has a set of *PRAs*. On a workstation with n *PRAs*, the i -th periodic real-time application PRA_i ($1 \leq i \leq n$) is defined as (S_i, C_i, T_i) , where S_i is the PRA_i 's start time, C_i is its execution

¹ This work is sponsored in part by grants from the NASA AMES Research Center (administered by USARDC, contract no. N68171-01-C-9012), the EPSRC (contract no. GR/R 47424/01) and the EPSRC e-Science Core Programme (contract no. GR/S03058/01).

time (in *time units*) on the workstation, and T_i is the PRA_i 's period. An execution of PRA_i is called a *periodic application instance (PAI)* and the j -th execution is denoted as PAI_{ij} . PAI_{ij} is ready at time $(j-1)*T_i$, termed the ready time (R_{ij} , $R_{ij}=S_i$), and must be complete before $j*T_i$, termed the deadline (D_{ij}). All $PAIs$ must meet their deadlines and are scheduled using an Early Deadline First (*EDF*) policy. The i -th arriving NPA , NPA_i , is modeled as (a_i, cv_i, d_i) , where a_i is NPA_i 's arrival time, cv_i is its computational volume and d_i is its deadline. The execution time of NPA_i on workstation k is denoted as $c^k(cv_i)$.

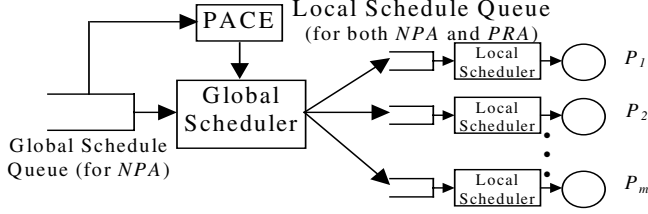


Fig. 1. The scheduler model in the heterogeneous cluster environment

Fig.1 depicts the scheduler model in the heterogeneous cluster. It is assumed that the PRA s are running in the workstations. All NPA s arrive at the global scheduler, where they wait in a *global schedule queue (GSQ)*. Each NPA from the *GSQ* is globally scheduled and, if accepted, sent to the *local scheduler* of the designated workstation. At each workstation, the *local scheduler* receives the new NPA s and inserts them into a *local schedule queue (LSQ)* in order of increasing deadlines. The local scheduler schedules both NPA s and PRA s' $PAIs$ in the *LSQ* uniformly using *EDF*. The local schedule is preemptive. In this scheduler model, the *PACE* toolkit [3] accepts NPA s, predicts their execution time on each workstation in real-time and returns the predicted time to the global scheduler [5].

3 Scheduling Analysis

A function constructed of idle time units, denoted as $S_i(t)$, is defined in Equ.1. P_{ij} is the sum of execution time of the $PAIs$ that must be completed before D_{ij} . P_{ij} can be calculated as Equ.2, where S_k is PRA_k 's start time.

$$S_i(t) = D_{ij} - P_{ij} \quad D_{ij} < t \leq D_{ij} \quad 1 \leq i \leq n, j \geq 1 \text{ where } D_{i0} = 0 \quad (1)$$

$$P_{ij} = \sum_{k=1}^n \left\lfloor \alpha / T_k \right\rfloor * C_k, \text{ where, } \alpha = \begin{cases} D_{ij} - S_k & D_{ij} > S_k \\ 0 & D_{ij} \leq S_k \end{cases} \quad (2)$$

$$S(t) = \min \{ S_i(t) | 1 \leq i \leq n \} \quad (3)$$

In the function $S_i(t)$, the time points, except zero, at which the function value increases, are called *Jumping Time Points (JTP)*. If the number of time units that are used to run NPA s between time 0 and any *JTP* is less than $S_i(JTP)$, the deadlines of all $PAIs$ of PRA_i can be guaranteed. Suppose n PRA s ($PRA_1, \dots, PRA_r, \dots, PRA_n$) are running

on the workstation, then the function of idle time units, denoted as $S(t)$, for the PRA set can be derived from the individual $S_i(t)$ ($1 \leq i \leq n$), shown in Equ.3. $S(t)$ suggests that idle time units of $S(JTP)$ are available in $[0, JTP]$.

If a NPA arrives and starts running at any time t_o , the remaining idle time slots in $[t_o, JTP]$, denoted as $S(t_o, JTP)$, is calculated in Theorem 1. Some additional notation is introduced to facilitate the description. $PA(t_o)$ is a set of $PAIs$ whose deadlines are no more than time t_o , defined in Equ.4. $LA(t_o)$ is a set of $PAIs$ whose deadlines are more than t_o , but whose ready times are less than t_o , defined in Equ.5. $P(t_o)$ and $L(t_o)$ are the number of time units in $[0, t_o]$ that are used for running the $PAIs$ in $PA(t_o)$ and $LA(t_o)$, respectively. $P(t_o)$ can be calculated as Equ.6. Let $JTP_1, JTP_2, \dots, JTP_k$ be a sequence of $JTPs$ after t_o in $S(t)$ and JTP_1 the nearest to t_o . $LA_k(t_o)$, defined in Equ.7, is a set of $PAIs$, whose deadlines are more than t_o , but no more than JTP_k . $L_k(t_o)$ is the number of time units in $[0, t_o]$ that are used to run the $PAIs$ in $LA_k(t_o)$.

$$PA(t_o) = \{PAI_{ij} \mid D_{ij} \leq t_o\} \quad (4)$$

$$LA(t_o) = \{PAI_{ij} \mid R_{ij} < t_o < D_{ij}\} \quad (5)$$

$$P(t_o) = \sum_{k=1}^n \left[\alpha / T_k \right] * C_k, \text{ where } \alpha = \begin{cases} t_o - S_k & t_o > S_k \\ 0 & t_o \leq S_k \end{cases} \quad (6)$$

$$LA_k(t_o) = \{PAI_{ij} \mid R_{ij} < t_o < D_{ij} \text{ and } D_{ij} \leq JTP_k\} \quad (7)$$

Theorem 1. $S(JTP_k)$ and $S(t_o, JTP_k)$ ($0 < t_o < JTP_k$) satisfy the following equation:

$$S(t_o, JTP_k) = S(JTP_k) - t_o + P(t_o) + L_k(t_o) \quad (8)$$

Proof: $PAIs$ whose deadlines are less than JTP_k must be completed in $[0, JTP_k]$. Their total workload is $P(JTP_k)$ (see Equ.4 and 6). The workload of $P(t_o)$ and $L_k(t_o)$ has to been finished before t_o , so the workload of $P(JTP_k) - P(t_o) - L_k(t_o)$ must be done in $[t_o, JTP_k]$. Hence, the maximal number of time units that can be spared to run $NPAs$ in $[t_o, JTP_k]$, i.e. $S(t_o, JTP_k)$, is $(JTP_k - t_o) - (P(JTP_k) - P(t_o) - L_k(t_o))$. Thus, the following exists: $S(t_o, JTP_k) = JTP_k - P(JTP_k) - t_o + P(t_o) + L_k(t_o)$. In addition, $JTP_k - P(JTP_k) = S(JTP_k)$, and therefore Equ.8 also holds.

Theorem 2 reveals the distribution property of the remaining time units before t_o after running $PAIs$ in $PA(t_o)$ as well as $NPAs$. $I_p^{t_0}(t_s, t_0)$ represents the number of time units left in $[t_s, t_0]$ after executing $PAIs$ in $PA(t_o)$; $I_{P,A}^{t_0}(f, t_0)$ represents the number of time units left in $[f, t_0]$ after executing both $PAIs$ in $PA(t_o)$ and also $NPAs$.

Theorem 2. Let the last NPA before t_o be completed at time f , then there exists such a time point t_s in $[f, t_0]$, that a) there are no idle slots in $[f, t_s]$, b) either $PAIs$ in $PA(t_o)$ retain the same execution pattern in $[t_s, t_0]$ as the case when no $NPAs$ are run before t_o , or all $PAIs$ in $PA(t_o)$ are completed before t_s , and c) t_s can be determined by Equ.9.

$$I_p^{t_0}(t_s, t_0) = I_{P,A}^{t_0}(f, t_0) \quad (9)$$

Proof: The execution of the last *NPA* may delay the execution of *PAIs* in $PA(t_0)$. The delayed *PAIs* may delay other *PAIs* in $PA(t_0)$ further. The delay chain will however cease when the delayed *PAIs* no longer delay other *PAIs*, or all the *PAIs* in $PA(t_0)$ are complete. Since all *PAIs* in $PA(t_0)$ must be complete before t_0 , such a time point, t_s , must exist that satisfies Theorem 2.2. Since there are unfinished workloads before t_s , Theorem 2.1 also exists. Equ. 9 is a direct derivation from Theorem 2.1 and 2.2.

Since *PAIs* in $PA(t_0)$ running in $[t_s, t_0]$ retain the original execution pattern (as though there were no preceding *NPAs*), it is possible to calculate the remaining time units in $[t_s, t_0]$ after running these *PAIs*. Consequently, $L_k(t_0)$ in Equ.8 can be calculated.

4 Scheduling Algorithms

If a *NPA* starts execution at t_0 , using Equ.8, the global scheduler can calculate how many idle time units there are between t_0 and any *JTP* following t_0 , which can be used to run the *NPA*. Therefore, it can be determined before which *JTP* the *NPA* can be completed. Consequently, the *NPA*'s finish time in any workstation p_j can be determined, which is shown in Algorithm 1. If the *NPA*'s finish time on any workstation in the heterogeneous cluster is greater than its deadline, the *NPA* will be rejected. The admission control is shown in Algorithm 2. When more than one workstation can satisfy the *NPA*'s deadline, the system selects the workstation on which the *NPA* will have the earliest finish time. After deciding which workstation the *NPA* should be scheduled to, the global scheduler re-sets the *NPA*'s deadline to its finish time on that workstation and sends the *NPA* to it. The global dynamic scheduling algorithm is shown in Algorithm 3. When the local scheduler receives the new allocated *NPAs* or the *PAIs* are ready, it inserts them into the *LSQ*. Each time a task (*NPA* or *PAI*) is fetched by the local scheduler from the head of the *LSQ* and the task is then executed. As the modeling analysis suggests in Section 3, a *NPA* cannot be finished earlier in the workstation on which the new task is scheduled. Otherwise, some *PAI*'s deadline on that workstation must be missed. In this sense, the modeling approach is optimal.

Algorithm 1 Calculating the finish time of NPA_i starting at t_0 in workstation p_j (denoted as $ft^j(NPA_i)$)

1. $c^j(cv_i) \leftarrow NPA_i$'s execution time;
2. Calculate $P(t_0)$; Get t_s ;
3. Get the first *JTP* after t_0 ;
4. Call Algorithm 1 to calculate corresponding $L_k(t_0)$;
5. Calculate $S(t_0, JTP)$ using Equ.8;
6. **while** ($S(t_0, JTP) < c^j(cv_i)$)
7. $OJTP \leftarrow JTP$; Get the next *JTP*;
8. Calculate $S(t_0, JTP)$ by Equ.8;
9. **end while**
10. $ft^j(NPA_i) \leftarrow OJTP + c^j(cv_i) - S(t_0, OJTP)$;

Algorithm 2 Admission Control

1. $PC \leftarrow \emptyset$ when a new NPA_i arrives;
2. **for** each workstation p_j ($1 \leq j \leq m$) **do**
3. call Algorithm 1 to calculate $ft^j(NPA_i)$;

```

4.  if ( $ft^j(NPA_i) \leq d_i$ ) then  $PC = PC \cup \{p_j\}$ ;
5.  end for
6.  if  $PC = \emptyset$  then reject  $NPA_i$ ;
7.  else accept  $NPA_i$ ;

```

Algorithm 3 the Global dynamic scheduling algorithm

```

1.  if global schedule queue  $GSQ = \emptyset$  then
    wait until a new  $NPA$  arrives;
2.  else
3.    get a  $NPA$  from the head of  $GSQ$ ;
4.    call Algorithm 3 to judge if accept or reject it;
5.    if accept the  $NPA$  then
6.      select workstation  $p_j$  by response-first policy;
7.      set the  $NPA$ 's deadline to be its finish time;
8.      Dispatch the  $NPA$  to workstation  $p_j$ ;
9.    end if
10. end if
11. go to step 1;

```

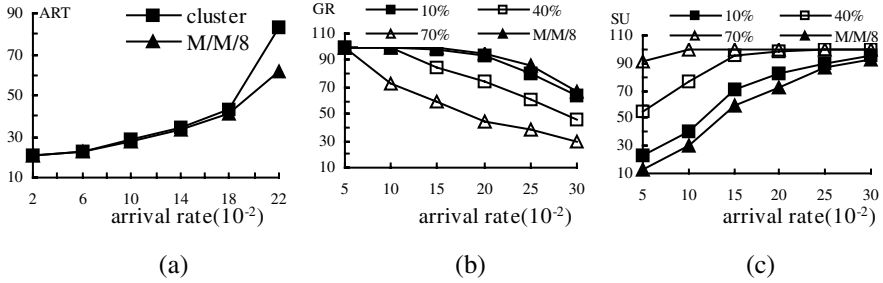


Fig. 2. (a) Comparison of ART of NPA s between our scheduling scheme and an M/M/8 queuing model, 10% PRA, (b) Effect of workload on GR, (c) Effect of workload on SU, $MAX\_DR/MIN\_DR=1.0/0$, $MAX\_CV/MIN\_CV=25/5$, $MAX\_W/MIN\_W=4/1$

5 Performance Evaluation

The performance of the global scheduling algorithm is also evaluated through extensive simulation. Workstation p_i 's weight w_i is chosen uniformly between MAX_W and MIN_W . NPA s arrive at the cluster following a Poisson process with the arrival rate of λ . The NPA_i 's computational volume cv_i is uniformly chosen between MAX_CV and MIN_CV , and the NPA_i 's deadline is chosen as follows: $arrival-time + \min\{c^k(cv_i)\} + cv_i * nw * dr$ ($1 \leq k \leq m$), where, nw is the geometric mean of the weight of all workstations, and dr is chosen uniformly between MAX_DR and MIN_DR . Three levels of PRA workload, light, medium and heavy, are generated for each workstation, which provides 10%, 40% and 70% system utilization, respectively. Three metrics are measured in the simulation experiments: *Guarantee Ratio* (GR), *System Utilization* (SU) and *Average Response Time* (ART). The GR is defined as the

percentage of jobs guaranteed to meet their deadlines. The *SU* of a cluster is defined as the fraction of busy time for running tasks to the total time available in the cluster. A *NPA's Response Time* is defined as the difference between its arrival time and the finish time. The *ART* is the average response time for all *NPA*s.

Fig.2.a displays the *ART* of *NPA*s as a function of λ in a cluster of 8 workstations each running 10% *PRA* workload. The *Guarantee Ratio (GR)* of *NPA*s is fixed to be 1.0. An M/M/8 queuing model is used to compute the ideal bound for the *ART* of the same *NPA* workload in the absence of *PRA*s. As can be observed from Fig.2.a, the *ART* obtained by this scheduling scheme is very close to the ideal bound except that λ is greater than 0.18. This suggests that the scheduling scheme can make full use of the idle slots and new *NPA*s are completed at the earliest possible time.

Fig.2.b and Fig.2.c illustrate the impact of *NPA* and *PRA* workload on *GR* and *SU*, respectively. It can be observed from Fig.2.b that *GR* decreases as λ increases or *PRA* workload increases, as would be expected. A further observation is that the curve for 10% *PRA*, as well as the curve for 40% *PRA* when λ is less than 0.1, is very close to that for the M/M/8 queuing model; which again shows the effectiveness of the scheduling scheme in utilizing idle capacities. Fig.2.c demonstrates that *SU* increases as λ increases. This figure shows that utilization of the cluster is significantly improved compared with the original *PRA* workload.

6 Conclusions

An optimal modeling approach for spare capabilities in heterogeneous clusters is presented in this paper. Furthermore, a dynamic scheduling scheme is proposed to allocate newly arriving real-time applications on the cluster by utilizing the modeling results.

References

1. K. Hwang and Z. Xu.: Scalable Parallel Computing: Technology, Architecture, Programming. McGraw Hill, 1998.
2. J. P. Lehoczky and S. Ramos-Thuel.: An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. Proc. of Real-Time Systems Symposium, 1992, pp. 110–123.
3. G.R. Nudd, D.J.Kerbyson et al.: PACE-a toolset for the performance prediction of parallel and distributed systems. International Journal of High Performance Computing Applications, Special Issues on Performance Modelling, 14(3), 2000, 228–251.
4. X Qin and H Jiang.: Dynamic, Reliability-driven Scheduling of Parallel Real-time Jobs in Heterogeneous Systems. In Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001), Valencia, Spain, September 3–7, 2001.
5. D.P. Spooner, SA. Jarvis, J. Cao, S. Saini and GR. Nudd.: Local Grid Scheduling Techniques using Performance Prediction. IEE Proc., Comput. Digit. Tech., 2003, 150, (2), pp. 87–96.

Recursive Refinement of Lower Bounds in the Multiprocessor Scheduling Problem

Satoshi Fujita, Masayuki Masukawa, and Shigeaki Tagashira

Department of Information Engineering
Graduate School of Engineering, Hiroshima University

Abstract. This paper proposes a new method to derive a refined lower bound on the makespan in the multiprocessor scheduling problem. The result of experiments implies that the proposed method really improves the performance of the underlying branch-and-bound scheme when it is applied at the root in the search tree; e.g., we could achieve a speedup of at least 7000 times in the best case.

1 Introduction

In this paper, we propose a new method to refine the lower bound on the makespan in the multiprocessor scheduling problem (MSP). The proposed method is an extension of *Fernandez's bound* (abbreviated as FB hereafter) that is known to provide the best bound among others [1], and can be computed in a quadratic time [2]. The key idea of the extension is a *recursion*; i.e., in determining the as soon as possible start time and the as late as possible completion time of each task, that should be as precisely identified as possible to obtain a sharp bound under FB, we recursively calculate lower bounds on the makespan for several sub-instances of the given instance. The proposed method was implemented as a part of a branch-and-bound (B&B) algorithm for solving MSP, and was experimentally evaluated. The result of experiments implies that the proposed method really improves the performance of the underlying B&B scheme when it is applied at the root vertex in the search tree (note that an optimal lower bound at a child vertex in the search tree is not smaller than a lower bound at the parent vertex). We found that the amount of speedup caused by the proposed method is at least 7000 times in the best case, and it could solve difficult instances within few seconds that could not be solved under conventional schemes within a day.

2 Model

Let $G = (V(G), E(G))$ be a directed acyclic graph with node set $V(G)$ and edge set $E(G)$, where a node in $V(G)$ represents a task that should be sequentially and non-preemptively executed on a processor, and an edge in $E(G)$ represents a precedence constraint between tasks. Terms “node” and “task” are used interchangeably. Let $|V(G)| = n$. A node with no predecessors is referred to as an

entry node, and a node with no successors is referred to as an exit node. G is assumed to contain exactly one entry node s , and one exit node t . In this paper, we consider a parallel computer system consisting of m identical processors, where the time for communication is negligible compared with the time for executing a task. Let $w(u)$ denote the execution time of task u . In this paper, we assume that $w(u)$ is an integer, and without loss of generality, assume $w(s) = w(t) = 0$. Let $t_{cp}(G)$ denote the length of a longest directed path in G , where the length of a path is defined as the summation of execution times of tasks on the directed path (including terminal nodes). In what follows, we often denote $V(G)$, $E(G)$, and $t_{cp}(G)$ as V , E , and t_{cp} , respectively, if the underlying graph is clear from the context. The **multiprocessor scheduling problem** (MSP), that we will study in this paper, is the problem of finding a feasible schedule with a minimum makespan for given G and P .

3 Lower Bound

3.1 Known Techniques

An outline of the lower bounding technique shown in [1] is described as follows. Suppose that we want to calculate the lower bound for partial solution x . Let $\mathcal{S}_T$ be the set of feasible schedules for x in which the entry node s starts at time 0 and the exit node t completes by time T . Given time interval $[\theta_1, \theta_2]$, for $0 \leq \theta_1 < \theta_2 \leq T$, let $R(\theta_1, \theta_2, T)$ denote the minimum amount of intersection of a schedule in $\mathcal{S}_T$ with interval $[\theta_1, \theta_2]$, over all selections of the schedule. Then, a lower bound on the number of processors required for completing any schedule in $\mathcal{S}_T$ within T time units is given by $m_L(T) \stackrel{\text{def}}{=} \max_{0 \leq \theta_1 < \theta_2 \leq T} [R(\theta_1, \theta_2, T)/(\theta_2 - \theta_1)]$. If $m \geq m_L(t_{cp})$, we cannot improve the lower bound on the makespan from t_{cp} ; however, if $m < m_L(t_{cp})$, we could improve it in the following two different ways: In the first method, called FB [1], the lower bound on the makespan increases to at least $t_{cp} + q$, where $q \geq \max_{0 \leq \theta_1 < \theta_2 \leq t_{cp}} \left\{ -(\theta_2 - \theta_1) + \frac{R(\theta_1, \theta_2, t_{cp})}{m} \right\}$. The second method, called FBB [2], uses binary search to find a minimum T such that $m_L(T) \leq m$. Note that FBB provides a better bound than FB in general.

3.2 Proposed Method

Let C be a set of integers ($\subset \{0, 1, \dots, T\}$) consisting of the as soon as possible completion time (ASCT) and the as late as possible start time (ALST) of all tasks $u \in V$, provided that the entry task starts at time 0 and the exit task completes by time T . In [2], we showed that C contains a pair (θ_1, θ_2) that minimizes $\frac{R(\theta_1, \theta_2, T)}{\theta_2 - \theta_1}$ (resp. $-(\theta_2 - \theta_1) + \frac{R(\theta_1, \theta_2, t_{cp})}{m}$); i.e., it is enough to examine all combinations of elements in C to calculate FBB (resp. FB). In addition, we showed in [2] that an examination of all combinations takes only $O(n^2)$ time, i.e., the examination of a single combination takes a constant time in average. In FBB and FB, the elements of C are determined based on the length of the longest path to the entry and/or exit nodes. However, we could refine the preciseness of

the values by recursively applying lower bounding techniques that are used in calculating a lower bound from given set of integers C , since the length of the longest path is a trivial lower bound that could generally be improved by using FB and FBB [1,2].

The proposed method is based on the above intuition. More concretely, in determining the ASCT of a task u , 1) it first considers a subgraph G' of G consisting of all predecessors of u containing the entry task s , and 2) it calculates a lower bound on the makespan of G' provided that there are m processors in the system (the ALST of a task u can be determined in a similar manner). The refinement of ASCT and ALST is carried out according to a linear order of the tasks, say $s, u_1, u_2, \dots, u_{n-2}, t$, that is consistent with the given precedence constraints; i.e., ASCT is refined from s to t , and ALST is refined from t to s . Any (trivial) lower bounds could be used as their initial values. It is worthy to note that when examining ASCT of task u_i , we can use refined ASCT's for tasks $s, u_1, u_2, \dots, u_{i-1}$, and conversely, when examining ALST of task u_i , we can use refined ALST's for tasks $u_{i+1}, u_{i+2}, \dots, u_{n-2}, t$. Such a refinement is repeatedly applied until no updates can be observed.

4 Experiments

4.1 Preliminaries

In order to evaluate the quality of the approach, we implemented the proposed recursive method as a part of B&B algorithm for solving MSP. In the resultant B&B algorithm, referred to as REC hereafter, the recursive application of FBB is attempted only at the root vertex, and at the other vertices, it determines the lower bound by using FBB. The specification of the machine used in the experiments is as follows: CPU: PentiumIII 850MHz, memory: 1GB, and OS: FreeBSD4.4.

In the experiments, we used STG (standard task graphs) suite [3] as the benchmark set. Since our main objective is to exactly solve relatively small instances, among several task graphs contained in the STG suite, we focus on 360 small ones with less than 1000 tasks. In addition, we discriminate 107 "difficult" instances by executing conventional FBB scheme for 10 minutes; The number of difficult instances that could not be solved in 10 minutes by the conventional scheme is: 14 for $m = 2$; 38 for $m = 4$; and 55 for $m = 8$. Note that among those 107 instances, only 31 instances can be solved within 24 hours by the conventional scheme, and in all of those solved instances, the lower bound at the root vertex coincides with the final solution; i.e., for those instances, we could not improve the lower bound at least at the root vertex. However, it does not imply that the improvement of lower bounds is meaningless. In fact, there are several instance in which a "decision" is made at a high level of the search tree; in such cases, a better lower bound really plays an important role in pruning branches at as higher level of the search tree as possible.

Table 1. Instances whose lower bound could be improved by REC.

	ID	FBB	REC	UB	difference
$m = 2$	50-0007	264	265	272 $\rightarrow$ 265	7 $\rightarrow$ 0
	100-0009	560	567	576 $\rightarrow$ 569	9 $\rightarrow$ 2
	300-0009	1478	1479	1484 $\rightarrow$ 1481	6 $\rightarrow$ 2
	500-0009	2793	2798	2808 $\rightarrow$ 2805	10 $\rightarrow$ 7
	700-0009	3589	3614	3632 $\rightarrow$ 3630	18 $\rightarrow$ 16
	900-0008	2558	2569	2577 $\rightarrow$ 2571	8 $\rightarrow$ 2
$m = 4$	300-0010	444	446	455 $\rightarrow$ 449	9 $\rightarrow$ 3
	500-0002	724	744	748 $\rightarrow$ 745	4 $\rightarrow$ 1
	500-0003	1361	1363	1369 $\rightarrow$ 1367	6 $\rightarrow$ 4
	700-0000	975	979	981 $\rightarrow$ 981	2 $\rightarrow$ 2
	700-0003	1922	1949	1955 $\rightarrow$ 1952	6 $\rightarrow$ 3
	700-0011	2034	2048	2091 $\rightarrow$ 2085	43 $\rightarrow$ 37
	900-0003	2433	2548	2554 $\rightarrow$ 2552	6 $\rightarrow$ 4
	900-0010	1368	1376	1395 $\rightarrow$ 1390	19 $\rightarrow$ 14
	900-0011	3051	3058	3084 $\rightarrow$ 3079	26 $\rightarrow$ 21
$m = 8$	500-0023	652	654	671 $\rightarrow$ 671	17 $\rightarrow$ 17
	700-0023	923	930	959 $\rightarrow$ 944	29 $\rightarrow$ 14

4.2 REC

Table 1 shows a list of (difficult) instances whose lower bound at the root vertex can be improved by REC. Note that those instances can not be solved within 24 hours by FBB. As is shown in the table, the use of REC really reduces the gap between upper and lower bounds; e.g., the gap becomes zero for one instance, one for one instance, and two for four instances. In addition, it achieves a large amount of improvement of the lower bound in several instances. However, a much more important fact we have to point out is that there *exists* an instance that could be solved by using REC. For that instance, i.e., graph 50-0007 for $m = 2$, REC can output an optimum solution in 10 seconds, while it takes more than 24 hours under FBB (under REC, the lower bound at the root can be found in 50 ms, and an optimum solution 265 can be found within 10 seconds). That is, at least for this instance, we could achieve more than 7920 times speedup by using the proposed method.

5 Concluding Remarks

In this paper, we proposed a recursive method to derive a refined lower bound on the makespan in the multiprocessor scheduling problem. An important open problem is to apply the recursive method to several vertices besides the root without increasing the computation cost.

References

1. E. B. Fernandez and B. Bussell. "Bounds on the number of processors and time for multiprocessor optimal schedules," IEEE Trans. Comput., C-22(8):745–751, 1973.
2. S. Fujita, M. Masukawa, S. Tagashira. "A Fast Branch-and-Bound Algorithm with an Improved Lower Bound for Solving the Multiprocessor Scheduling Problem," Proc. ICPADS 2002, pages 611–616.
3. URL: <http://www.kasahara.elec.waseda.ac.jp/schedule/index.html>

Efficient Dynamic Load Balancing Strategies for Parallel Active Set Optimization Methods*

I. Pardines¹ and Francisco F. Rivera²

¹ Dept. Arquitectura de Computadores y Automática, Univ. Complutense,
28040 Madrid, Spain

`inmap1@dacya.ucm.es`

² Dept. Electrónica y Computación, Univ. Santiago de Compostela,
15782 Santiago de Compostela, Spain

`fran@dec.usc.es`

Abstract. In this paper three strategies are described to restore dynamically the load balancing in parallel active set optimization algorithms. The efficiency of our proposals is shown by comparison with other heuristics described in related works, such as the classical Bestfit and Worstfit methods. The computational cost due to the load unbalancing in the parallel code and the communication overheads associated with the most efficient load balancing strategy are analyzed and compared in order to establish whether the distribution is convenient or not. Experimental results on a distributed memory system, the Fujitsu AP3000, highlight the accuracy of our estimations.

1 Introduction

Active set methods are based on a prediction (called working set) of the active constraints at the solution. The working set may change at each iteration, adding or deleting one constraint, which means adding or deleting a column of the Hessian matrix [3]. The addition is performed according to a cyclic distribution of columns among the processors. Therefore in this operation the load balance is maintained. However, deleting columns are a source of imbalance when the problem variables and its computations are distributed in a parallel implementation.

Our parallel algorithm works with a triangular matrix R (being $R^T R$ the Hessian approximation of the system) that is divided into triangular and rectangular blocks [6]. The computations of a rectangular block are executed in parallel whereas the computations of the triangular blocks are sequential. After the execution of each rectangular block a synchronization between all the processors of the system is needed. Initially, the system is completely balanced, but after a number of iterations it is possible that columns, belonging to the same rectangular block, are deleted. When the load unbalancing decreases the efficiency of the parallel code, an effective strategy to redistribute the columns between the processors is essential.

* This work was supported by CICYT under grants TIC 2002/750 and TIC 2001-3694-C02.

2 Dynamic Load Balancing Strategies

The load redistribution problem described in this paper is a *NP*-hard problem similar to the Multiple Knapsack problems [5]. A set of processors, called donors (D), have extra loads in terms of the number of columns. The value of the unbalancing load w_i of each donor i is stored in an array called weight array (W). In the system there are other processors, called receivers (R), with under the average load, so they have the capacity to receive load. The value of this lack of load c_j per receiver j is saved in an array called capacity array (C). W and C are organized in a decreasing order.

Similar redistribution problems appeared in the literature. These problems try to minimize the total number of messages [4]. However, the objective of our methods is to balance the system minimizing the maximum number of receptions and the maximum number of sent messages per processor. So, we have to deal with a combinatorial optimization problem that can be formalized as follows:

$$\begin{aligned}
 & \text{minimize} \quad \max\{\max \sum_{i \in D} x_{ij} \quad \forall j \in R, \max \sum_{j \in R} x_{ij} \quad \forall i \in D\} \\
 & \text{subject to} \quad \sum_{i \in D} w_i(j) x_{ij} = c_j, \quad j \in R; \\
 & \quad \quad \quad \sum_{j \in R} w_i(j) = w_i, \quad \forall i \in D; \\
 & \quad \quad \quad x_{ij} \in \{0, 1\}, \quad i \in D, j \in R,
 \end{aligned} \tag{1}$$

where $\sum_{i \in D} x_{ij}$ is the number of messages received by processor j , $\sum_{j \in R} x_{ij}$ is the number of messages sent by processor i , and $w_i(j)$ is the load that processor i sends to processor j . Finally, x_{ij} is 1 if there is a message between donor i and receiver j , and 0 otherwise.

Three strategies are proposed to solve this optimization problem: the Load Donor Strategy (LDS), the Load Receiver Strategy (LRS) and the Load Hybrid Strategy (LHS), which are introduced in the next sections.

2.1 Load Donor Strategy

Two different stages are distinguished in this greedy method. Firstly, equal weights and capacities are searched in W and C , respectively. The perfect situation is based on finding a weight w_i identical to a capacity c_j . So, just one message between the corresponding donor and receiver processors is necessary.

In the second stage, the weight and capacity arrays are swept making matches between processors trying to balance the remaining computational load. In the process a suitable capacity is searched for each element of the weight array. There are two situations:

1. The value of the first element of W is greater than all the elements of C ($w_1 > c_j, \forall j \in R$). In this case, the processor with load w_1 sends $c_j = \max_{k \in R} \{c_k\}$ columns to the processor with the greatest capacity. The remaining load, $w_1 - c_j$ columns, has to be stored again in the proper position of W . The objective is to minimize the maximum number of sends per processor. If a donor redistributes as many columns as possible in a single message, the number of messages needed to achieve the load balancing tends to be minimum.

2. Element w_1 is smaller than c_1 . In this case, a capacity c_j that equals w_1 is searched. If this capacity exists, a message is established. On the other hand, if there is not such capacity, the selection will be made among the processors with capacity c_j that verify,

$$c_j - w_1 > \min_{k \in D} \{w_k\} . \quad (2)$$

In this way, very small capacities that imply short messages are avoided. So, the possibilities that a donor has to divide its load into a large number of messages is reduced, thereby minimizing the maximum number of sends. The receiver, verifying equation 2, will be that which has already received the minimum number of messages. The aim is to avoid an increase in the maximum number of receptions per processor. In case equation 2 is not satisfied, the receiver processor will be the one with the greatest capacity. This process is repeated until there are no entries to match in W and C .

2.2 Load Receiver Strategy

This method is basically the same as the LDS strategy, but the priority is to minimize the maximum number of receptions per processor. Therefore, the technique is the same as the one described in section 2.1, swapping the roles of the weight and capacity arrays.

2.3 Load Hybrid Strategy

The idea behind this proposal is to combine the previous strategies. The objective is to minimize the maximum number of both sends and receptions per processor. So, it will be the selected heuristic for our parallel algorithm. The first stage is the same in the three methods, searching for equal weights and capacities in W and C . In the second stage, two cases are distinguished:

1. Element w_1 of W is smaller than the greatest capacity (c_1). In this case, the LDS strategy is applied. Therefore, a capacity c_j such as $c_j - w_1 > \min_{k \in D} \{w_k\}$ is searched. The processor with such a capacity that has received the minimum number of messages is selected.
2. Element w_1 is greater than capacity c_1 . Then, the LRS strategy is used to minimize the maximum number of receptions. A processor with the minimum number of sent messages and a weight w_i verifying the condition $w_i - c_1 > \min_{k \in R} \{c_k\}$ is searched. This processor will send its extra load to the processor that needs c_1 columns to achieve the load balancing.

3 Estimated Cost of the Load Hybrid Strategy

In order to make the parallel algorithm efficient, the computational cost due to a load unbalancing and the cost due to the application of the LHS strategy have to

be estimated. Then, the parallel algorithm can decide whether the unbalancing effect is strong enough to apply the load balancing method or not.

If the system is unbalanced, the execution time of the parallel algorithm is determined by the processor with the greatest computational load. As the parallel code needs synchronizations after the computation of each rectangular block, the study of the imbalance can be reduced to these blocks. So, the cost due to imbalance per block can be defined as the time that the most overloaded processor needs to compute the excess of columns over the balancing situation (C_{max}). Considering that each column inside a block is composed by αP elements [6], being P the number of processors of the system, and $\alpha \in \mathbb{N}$ a parameter. The computational cost per block due to imbalance is,

$$\text{unbalancing}(\text{block}) = C_{max} \cdot (a_u + b_u \cdot \alpha P) . \quad (3)$$

The cost of the LHS strategy can be established as

$$\text{cost}(\text{LHS}) = T1 + T2, \quad (4)$$

where $T1$ is the execution time of the LHS heuristic, and $T2$ is the communication time, defined as the time spent by the processor with the maximum number of sends or receptions obtained by the LHS heuristic.

$T1$ depends on the number of times that W and C are swept. As the number of messages can not be known until the strategy is applied, we have decided to assume the worst case to model it, i.e. when the number of messages is equal to $P-1$. Therefore, as each iteration means a message, the estimated cost will be,

$$T1 = \text{cost}(\text{iter}) \cdot (P - 1) . \quad (5)$$

To compute $T2$, it is necessary to know the maximum number of sends or receptions and the cost of a message. The number of messages per processor is not known a priori. However, a study over a wide set of examples has been made, with the result that for the LHS strategy the more probable value for the maximum number of sent messages or receptions is 3. So, we assume that the number of messages per processor is 3. Therefore, the communication time will be modeled as,

$$T2 = 3 \cdot \text{cost}(\text{message}) . \quad (6)$$

The cost of a message depends on its size. Different linear fits $a_{men} + b_{men} \cdot n$, being n the message size, have been obtained for the AP3000 [1]. Then, the cost of a message can be estimated when its size is known. Assuming that the excess load is C_{max} , and being 3 the maximum number of messages that a processor sends or receives, each message will have an estimated size of,

$$n = (\text{elements/column}) \cdot (\text{columns}) \cdot (\text{bytes/element}) = \alpha P \cdot \left\lceil \frac{C_{max}}{3} \right\rceil \cdot 8 . \quad (7)$$

The complexity of the algorithm is $O(P \log P)$.

4 Results

A wide set of 1740 synthetic examples was used to study the efficiency of the proposed strategies. A comparison with the Bestfit (BF) and Worstfit (WF) heuristics, and a greedy method called *Pairs Inside While Loop* (PIWL) [4] can be seen in Table 1. For each method, a column with the total number of times, over the 1740 examples, that the best solution of the optimization problem is obtained is shown. In most of the cases, this solution is achieved for various heuristics at the time. However, in some occasions, only one method obtains the best result. The number of times that this happens is shown in parenthesis. Note that as the number of processors increases, our strategies achieve the best solution most number of the times, outperforming dramatically WF and BF methods. The heuristic that most times achieves the minimum value of the objective function is highlighting in bold.

Table 1. Number of times over the 1740 examples that each heuristic obtain the best solution of the optimization problem

P	LDS	LRS	LHS	PIWL	BF	WF
4	290	290	290	290	290	281
8	230	226(6)	232	231	193(10)	181(15)
16	239(5)	247(14)	242	238	84(6)	81(9)
32	230(19)	234(31)	227	226	27(2)	26(3)
64	202(27)	214(32)	211(4)	190(1)	28(4)	16
128	179(40)	179(32)	202(10)	173(5)	24(3)	15
TOTAL	1370	1390	1404	1348	646	600

On the other hand, the accuracy of the estimation cost of our load balancing strategy has been validated in the AP3000 with 8 processors. The quality of the estimation of the unbalancing cost and the LHS strategy is verified for different sizes of the rectangular blocks. As an example, results for $\alpha = 4$ and $\alpha = 10$ are shown in Fig. 1.

Analyzing the results, we can conclude that for small values of α , the estimated cost of the LHS strategy differs from the heuristic real cost in case of great imbalance. However, as α increases, the LHS strategy predicts more accurately the real heuristic behavior. The reason is that in a balanced situation, each processor has α columns. So, for small α (Fig. 1(a)), great imbalance (high C_{max} values) implies that many processors lose its load completely. Therefore, the greatest loaded processor has to send probably more than 3 messages to restore balance. For large values of α (Fig. 1(b)), it is reasonable to estimate the total number of messages as 3. However, as our estimation is highly accurate for small imbalances, independently of α , the intersection point of the estimated imbalance and LHS strategy cost lines is very close to the real intersection point. The intersection is achieved for small unbalancing values, being even smaller as

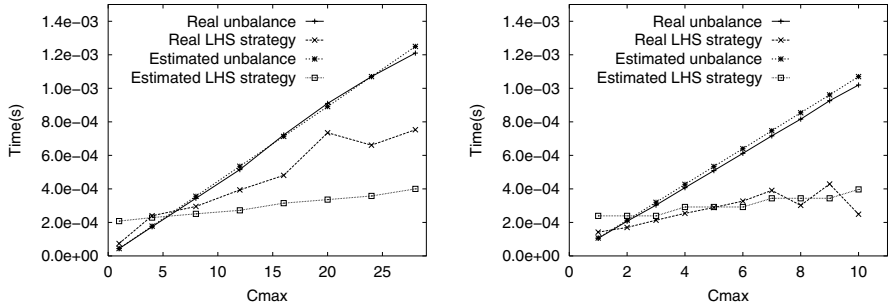


Fig. 1. Comparison between real and estimated cost of the LHS strategy for different block sizes. (a) $\alpha = 4$, (b) $\alpha = 10$

α or P increases, because in this case the computational load is greater and the imbalance will be more significant and expensive. Therefore, it is convenient to execute the load redistribution in almost every case.

5 Conclusions

In this work three load balancing strategies are proposed. Our methods focus on minimizing the maximum number of sends and receptions in order to reduce the communication cost. Improvements with respect to WF, BF and PIWL heuristics have been achieved. An estimation of the cost due to imbalances and of the cost of our LHS strategy has been established to know when it is better to redistribute the computational load. The comparison between them and the estimated results shows the high accuracy of the prediction. Moreover, it is proved that the load redistribution will be realized in practically all the cases.

References

1. Blanco, V. et al.: Performance of Parallel Iterative Solvers: a Library, a Prediction Model and a Visualization Tool. *Journal of Information Science and Engineering*, Vol. 8 **5** (2002) 763–785.
2. Gill, Philip E., Murray, Walter, Wright, Margaret H.: *Practical Optimization*. Academic Press, London (1981).
3. Gill, Philip E., Murray, Walter, Wright, Margaret H.: *Numerical Linear Algebra and Optimization*. Volume 1. Addison-Wesley Publishing Company, California (1991).
4. Haglin, David J., Ford, Rupert W.: The Message-Minimizing Load Redistribution Problem. *Journal of Universal Computer Science*, Vol. 7 **4** (2001) 291–306.
5. Martello, Silvano, Toth, Paolo: *Knapsack Problems: Algorithms and Computer Implementations*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York (1990).
6. Pardines, I., Rivera, F.F.: Parallel Quasi-Newton Optimization on Distributed Memory Multiprocessors. *Parallel Computing: Advances and Current Issues*. Proceedings of ParCo2001. Imperial College Press, London (2002) 338–345.

Cooperating Coscheduling in a Non-dedicated Cluster^{*}

Francesc Giné¹, Francesc Solsona¹, Porfidio Hernández², and Emilio Luque²

¹ Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain.
`{sisco,francesc}@eup.udl.es`

² Departamento de Informática, Universitat Autònoma de Barcelona, Spain.
`{p.hernandez,e.luque}@cc.uab.es`

Abstract. Our research is focussed on keeping both local and parallel jobs together in a time-sharing NOW. In such systems, new scheduling approaches are needed to allocate the underloaded resources among parallel jobs. In this paper, a distributed system, named Cooperating CoScheduling, is presented to assign CPU and memory resources efficiently by applying resource balancing and dynamic coscheduling between parallel jobs. This is shown experimentally in a PVM-Linux NOW.

1 Introduction

Time-slicing scheduling techniques [1,4] exploit the unused resources of a Network Of Workstations (NOW), by running both parallel and local jobs together. In this framework, the coscheduling of parallel jobs becomes a critical issue. Some studies [6,5] have shown the good performance of dynamic coscheduling techniques when many local users compete with a single parallel job. Dynamic coscheduling deals with minimizing the communication waiting time of parallel processes. With this aim, when parallel tasks are dynamically coscheduled, the arrival of a message will cause the dispatch of the target task. Unfortunately, dynamic coscheduling performance can slow down drastically when the number of parallel jobs competing against each other (MultiProgramming Level (MPL)) is increased. This is due to the fact that if the resources are not well-balanced between the competing parallel tasks, the completion of each process of a parallel job will require a different amount of time, and the entire job must wait for the slowest process before it can synchronize.

A new technique, called Cooperating CoScheduling (CCS), is presented in this paper. CCS improves dynamic coscheduling techniques by balancing the CPU and memory resources between parallel jobs throughout the NOW. Each node manages its resources efficiently by combining local – basically communication events, local user logging and computational capabilities–, and foreign runtime information (i.e. the local user logging onto a remote node) provided by their cooperating nodes. CCS is implemented in a PVM-Linux NOW.

^{*} This work was supported by the MCyT under contract TIC 2001-2592 and partially supported by the Generalitat de Catalunya – Grup de Recerca Consolidat 2001SGR-00218.

2 CCS: Cooperating CoScheduling

Our framework is a non-dedicated NOW, where all nodes are under the control of the CCS scheme. In each node, a time-sharing scheduler is assumed with task preemption based on the ranking of tasks according to their priority. The scheduler works by dividing the CPU time into *epochs*. In every epoch, each task is assigned a specific time slice and priority. In order to give preference to I/O-bound processes, the priority of each task is inversely proportional to the CPU used during the last epoch. A Memory Management Unit is assumed with demand-paging virtual memory. A SPMD model is assumed for parallel jobs. We assume that each node applies a *dynamic coscheduling* technique [5]. This means that every local scheduler increases the receiving task priority according to the number of packets in its reception queue, even causing CPU preemption of the task being executed inside.

The need to preserve local job performance limits the resources assigned to parallel tasks. This may lead to non-desirable situations (see fig. 1(a)). A parallel job (J_1) is executed on $node_1$ and $node_2$ with 25% of resources (CPU and Memory), while the local user has the remaining 75% available. Simultaneously, another parallel job (J_2) is executed with full resources on $node_4$ and $node_5$. Meanwhile, $node_3$ assigns half of the resources to each parallel job following a fairness policy. This uneven taxing of resources creates a situation where some processes of the parallel program run slower than others. However, if $node_3$ had knowledge about the status of J_1 and J_2 jobs in the remaining nodes (cooperating nodes), it could take better decisions by giving J_2 more resources than J_1 . Thus, the situation shown in fig. 1(b) would be achieved. In this case, J_1 has the same amount of resources in each node, while J_2 has more resources available than in the previous case. This way, the NOW resources are better balanced.

This example illustrates the three main aims of the CCS algorithm, namely: a) the *performance of local jobs*; b) the *uniform allocation of the CPU and memory resources* assigned to each parallel job throughout the cluster; and c) the *coscheduling* of communicating parallel jobs.

2.1 Cooperating CoScheduling Mechanism

This section explains how CCS achieves the objectives set out above.

Job interaction. We use priority as a mechanism to limit the interference of parallel jobs in the resources (CPU and memory) needed by local tasks. In the CPU, when $node_i$ detects the presence of a local user, it will assign a time slice to parallel and locals tasks equal to $DEF\_QUANTUM_i * L$ and $DEF\_QUANTUM_i * (1 - L)$, respectively. $DEF\_QUANTUM_i$ is the base time slice of $node_i$ and L is the minimum percentage of resources assigned to parallel jobs. The minimum term is related to the fact that if parallel tasks require a bigger percentage of CPU than L , then they will be able to use the portion allocated to local tasks, whenever local tasks are not using this. It is assumed that L has the same value across the cluster. In order to quantify the heterogeneous nature of a NOW, we have used the Power Weight [1] ($W_i = BOGO_i / BOGO_{max}$),

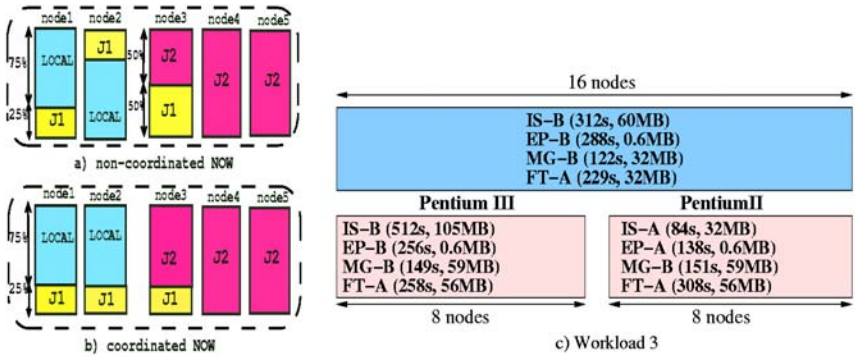


Fig. 1. (a and b) Example of applying CCS. (c) Workload 3.

where $1 \leq i \leq n$ (number of nodes) and $BOGO_{i/max}$ is a measurement of the computational capabilities of the i th and fastest node, respectively. Note that this measurement is provided by some o.s (p.e. Linux) in the boot period. Thus, the base time slice of each node is set relative to its Power Weight ($DEF\_QUANTUM_i = DEF\_QUANTUM/W_i$), where $DEF\_QUANTUM$ is the base time slice of the local o.s. (p.e. Linux=210ms).

The memory (M_i) is divided into two pools: one for the local tasks (M_i^L) and the other for parallel jobs ($M^D = \min_{i=1}^n(M_i) * L$). Note that the same minimum memory size (M^D), will be guaranteed to parallel tasks in each node. As in the CPU case, if parallel tasks require a bigger memory size than M^D then they will be able to use the portion allocated to local tasks, whenever local tasks are not using this. However, in the nodes where swapping is activated and the memory requirements of the parallel tasks exceed their assigned part (M^D), the parallel task with most pages mapped in memory (*swapping-task*) will be stopped. Consequently, the resident size of the remaining tasks will be increased and the swapping activity will also be reduced [2].

Cooperating Scheme. In order to balance resources throughout the cluster, the cooperating nodes should interchange status information. The key question is which information should be interchanged. An excess of information could slow down the performance of the system, increasing its complexity and limiting its scalability. For this reason, only the following four events are notified:

1. *LOCAL*: when there is local user activity in a *node*, it sends the identifier of the parallel jobs executing in such node to all its cooperating nodes.
2. *NO_LOCAL*: if the local activity finishes, the node sends a notification message to its cooperating nodes.
3. *STOP*: when *swapping-task* is stopped, the node sends the identifier of the penalized job to its cooperating nodes.
4. *RESUME*: when the *swapping-task* restarts, the node informs its cooperating nodes of this event.

When a cooperating node receives one of the above events, it will reassign the resources according to the following rules. (1) The scheduler with more than one parallel tasks under its management will assign a quantum equal to $(DEF\_QUANTUM_i * L)$ or zero to those tasks notified by *LOCAL* or *STOP* events, respectively. (2) In order to ensure that a parallel job is not slowed down by any local user across the cluster, a node will only reassign more resources to a task notified by a *NO_LOCAL* event when the number of *LOCAL* received events coincides with the *NO_LOCAL* received events. (3) If a node receives a *STOP* notification for a parallel task, this task will be chosen as *swapping-task*, whenever the memory is exhausted. (4) If a node with overloaded memory receives the *STOP* event for a task different than the *swapping-task*, the kernel will stop the notified task and the *swapping-task* will be resumed. (5) If a node with overloaded memory receives a *RESUME* event associated with the *swapping-task*, the kernel will restart the notified task and a new *swapping-task* will be chosen. Applying these rules, we show experimentally (see extended version [3]) that the cluster reaches a stable state.

3 Experimentation

CCS was implemented and tested in a PVM (v.3.4) - Linux (v.2.2.15) cluster. A complete description of the CCS implementation can be found in [3]. Our cluster was composed of eight Pentium II and eight Pentium III with 128 and 256 MB of memory and a Power Weight (W_i) of 0.43 and 1, respectively. All of them were connected through a Fast Ethernet network.

The performance of the parallel jobs was evaluated by running four PVM benchmarks from the NAS suite with class A and B: *IS* and *FT*, *MG* and *EP*. Three workloads (*Wrk1*, *Wrk2* and *Wrk3*) with different job sizes and *Multi-Programming Level* (MPL) were defined. *Wrk1* is characterized by a MPL equal to 1, while *Wrk2* and *Wrk3* are 2. *Wrk1* continuously executes a benchmark of size 16 chosen randomly according to a discrete uniform distribution. *Wrk2* continuously executes two size 16 jobs with the constraint that parallel tasks fit in their assigned memory portion (M^D). *Wrk3* executes one size 16 job together with two size 8 jobs (see fig. 1(c)) without any memory constraint. Additionally, fig. 1(c) shows the execution time and memory requirements of every benchmark on a dedicated system. Each workload was executed for 4 hours.

The local workload was carried out by running one synthetic benchmark, called *local*. This allowed the CPU load, memory requirements and network traffic used by the local user to be fixed. According to the monitoring of real local users [3], we defined two different local user profiles: a user with high CPU requirements, denoted as *CPU*, and a user with a high requirements for interactivity, denoted as *Xwin*. These parameters were set at (0.65, 35MB¹ and 0KB/s) and (0.20, 84MB¹ and 3KB/s) for *CPU* and *Xwin* users, respectively. In order to simulate the high variability of local user behavior, the local execution time was modeled by a two stage hyper-exponential distribution with means,

¹ Memory requirements do not include kernel memory.

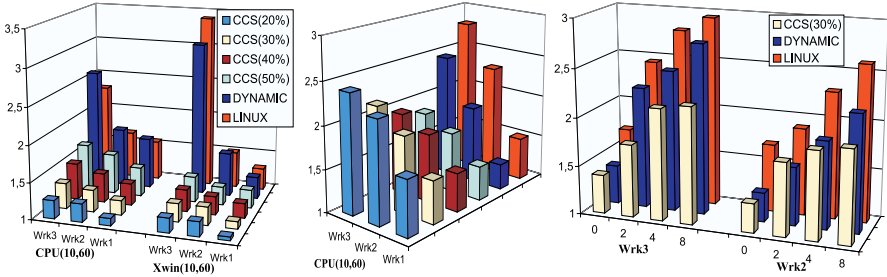


Fig. 2. (left) Slowdown of local and (center) parallel tasks with four local users. (right) Slowdown of parallel tasks varying the number of local CPU users.

by default, of 10min and 60min and a weight of 0.6 and 0.4 for each stage. At the end of its execution, the *local* benchmark returns the system call latency, memory and network bandwidth and wall-clock execution time.

Three environments were evaluated: the plain LINUX, the DYNAMIC and the CCS schedulers. Note that the DYNAMIC scheduler applies exclusively a dynamic coscheduling technique without any restrictions of the resources used by parallel jobs and any exchanging of information between cooperating nodes.

The performance of the parallel and local jobs was validated by means of the *Slowdown* metric (which is the average of 5 different executions). The next section shows a summary of the extended version of the experimentation [3].

3.1 Experimental Results

Fig. 2 (left) shows the slowdown of *local* according to the percentage of resources assigned to parallel tasks (L). L was varied between 20% to 50%. This trial was repeated twice: once with only CPU users and again with *Xwin* users. We can see that *LINUX* and *PREDICTIVE* gave the worst performance for local tasks. The peak obtained with the *Wrk3* is because the *Wrk3* memory requirements exceed the main memory size of some nodes with local activity and as a consequence, swapping is performed without guaranteeing any minimal memory size to local tasks. With the exception of the *Wrk3* case, it is worth pointing out the low intrusion for the *Xwin* users. This is due to the priority given by Linux to I/O processes. The impact of CCS rises slightly when the MPL is increased to 2 (*Wrk2* and *Wrk3*), although this is always limited to below values of 1.25 and 1.6 for *Xwin* and CPU users respectively, even in the worst case for local tasks ($L = 50\%$). For the rest of the trials, the local workload is exclusively made up of CPU local users (the most unfavorable case for parallel performance).

Fig. 2 (center) shows the slowdown in the performance of parallel jobs under the same conditions explained above. In general, *LINUX* obtains the worst results due mainly to the absence of coscheduling between communicating tasks. *DYNAMIC* obtains the best results for *Wrk1* while *CCS* obtains the best results for *Wrk3*. This degradation for *DYNAMIC* in the *Wrk3* case is due to the fact

that high or moderate page fault frequency in one node can drop parallel job performance drastically, vastly exceeding in this way the coscheduling benefits. In the case of *Wrk2*, *CCS* obtains better results than *DYNAMIC* for $L \geq 30$. This is due to the fact that the resources are better balanced under *CCS* and thus, all PVM jobs can progress across the NOW in a coordinated way. Fig. 2 (right) shows the slowdown on *Wrk2* and *Wrk3* when the number of local users was varied between 0 and 8. For the *Wrk2* case, *CCS* maintains a nearly constant slowdown in a non-dedicated environment, while *DYNAMIC* performance is slightly degraded when local users are increased. This is due to the absence of a mechanism for balancing resources across the NOW. Regarding the *Wrk3* case, *CCS* obtains the best results with this workload because *CCS* can distribute NOW resources better between parallel jobs of different sizes.

4 Conclusions and Future Work

In this paper, a new technique, named Cooperating CoScheduling (*CCS*), is presented. The main aim of *CCS* is to exploit the unused computing capacity of a non-dedicated cluster without disturbing local jobs excessively. This is achieved by means of combining balancing of resources and coscheduling between parallel jobs. In doing so, each *CCS* node assigns its resources dynamically, based on a combination of local and foreign runtime information provided by its own o.s. and its cooperating nodes, respectively. Thus, local decisions are taken in a coordinated way across the NOW. The performance of *CCS* has been tested in a PVM-Linux cluster and compared with other coscheduling policies. The results obtained demonstrated its good behavior, substantially reducing the response time of local tasks and improving, in some cases, the slowdown of parallel tasks with respect to other coscheduling policies.

Our next work is directed towards extending *CCS* to be applied together with a space-slicing technique. Our aim is to execute multiple parallel programs in every cluster partition by means of applying the *CCS* technique in each one.

References

1. X. Du and X. Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1997.
2. F. Giné, F. Solsona, P. Hernández, and E. Luque. Minimizing paging tradeoffs applying coscheduling techniques in a linux cluster. *LNCS*, 2565:593–607, 2002.
3. F. Giné, F. Solsona, P. Hernández, and E. Luque. Cooperating coscheduling in a non-dedicated cluster. extended version. Technical Report DIEI-03-RT-1, Universitat de Lleida, 2003.
4. K.D. Ryu and J.K. Hollingsworth. Linger longer: Fine-grain cycle stealing for networks of workstations. In *Proceedings Supercomputing'99 Conference*, 1999.
5. P.G. Sobalvarro, S. Pakin, W.E. Wehl, and A.A. Chien. Dynamic coscheduling on workstation clusters. *LNCS*, 1459:231–256, 1998.
6. F. Solsona, F. Giné, P. Hernández, and E. Luque. Predictive coscheduling implementation in a non-dedicated linux cluster. *LNCS*, 2150:732–741, 2001.

Predicting the Best Mapping for Efficient Exploitation of Task and Data Parallelism^{*}

Fernando Guirado¹, Ana Ripoll², Concepció Roig¹, Xiao Yuan², and Emilio Luque²

¹ Universitat de Lleida, Dept. of CS
{fernando,roig}@eup.udl.es,

² Universitat Autònoma de Barcelona, Dept. of CS
{a.ripoll,e.luque}@cc.uab.es, xiaoyuan@aows10.uab.es

Abstract. The detection and exploitation of different kinds of parallelism, task parallelism and data parallelism often leads to efficient parallel programs. This paper presents a simulation environment to predict the best mapping for the execution of message-passing applications on distributed systems. Using this environment, we evaluate the performance of an image processing application for the different parallelizing alternatives, and we propose the ways to improve its performance.

1 Introduction

Several applications from scientific computing enclose different kinds of potential parallelism: task parallelism and data parallelism [1] [2]. The exploitation of parallelism is specially important in clusters of distributed machines that remain a challenging environment in which to achieve a good performance. One reason of this is the difficulty of predicting the performance of an application for different mappings.

For instance, in image processing, an application that is composed of a chain of parallel tasks which act on a stream of input data sets, can be mapped onto a parallel machine on several ways. Figure 1 shows an application with four tasks and several combinations of data and task parallel mappings using four processors:

- Task parallel mapping (Figure 1(b)). Each processor executes a subset of application tasks. Then all the images are processed by all the processors. This kind of mapping requires an automatic mechanism to assign tasks to processors.
- Pure data parallel mapping (Figure 1(c)), where all the tasks execute on all processors. Then, in this case, the stream of images is divided into a set of substreams, where each substream is processed in one processor.

^{*} This work was supported by the MCyT under contract 2001-2592 and partially sponsored by the Generalitat de Catalunya (G. de Rec. Consolidat 2001SGR-00218).

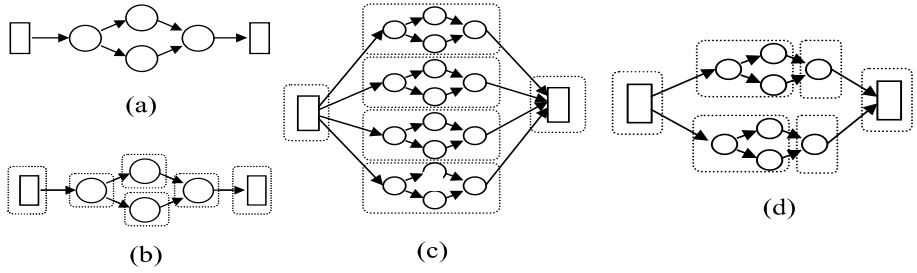


Fig. 1. (a) Parallel application. (b) Task parallel mapping. (c) Data parallel mapping. (d) Mix of task and data parallel mapping.

- Mix of task and data parallel mapping (Figure 1(d)). Replicated copies of a task parallel mapped application are assigned on different sets of processors.

The goal of this research is to present a simulation environment to predict the best mapping for parallel applications. This approach assumes that the execution behaviour of the program is not strongly dependent on the properties of the input data. With this environment we evaluated the execution of an image processing application called BASIZ (Bright and Saturated Image Zones) in a cluster of PC's. The results show that the mixed task and data parallel mapping is the most convenient for minimizing the latency of each processed image, while an execution based on a pure data parallel mapping is the best solution to achieve a good rate in the throughput.

The remaining sections are organized as following. Section 2 exposes the simulation environment. Section 3 presents the analysis of performance evaluation carried out using this environment for BASIZ application. Finally, Section 4 outlines the main conclusions.

2 Simulation Environment

In this section we present the simulation environment used in our approach, that is based on the tool called ESPPADA [3], and it is designed to provide automatic support for mapping evaluation. ESPPADA simulates the execution of message-passing applications on distributed systems. The input of the simulator is the task-dependency model of the application, that captures for each task the computation phases where the task performs sequential computation and the communication events with their adjacent tasks. The simulator can operate with applications defined with arbitrary task interaction pattern. This allows the evaluation of different implementations for the program in order to identify the most efficient one.

The underlying system is modelled by defining the characteristics of the nodes and the interconnection network. The user has the ability to define a specific processing speed for each node and different parameters for the interconnection

network. The execution simulation of a program is carried out with a specific allocation of tasks to processors. The following automatic mapping strategies are included based on different task graph models [4]: (a) the ETF heuristic based on the classical TPG graph, (b) the CREMA heuristic based on the classical TIG model and, (c) the MATE heuristic which is based on a new task graph model, proposed by the authors, called TTIG [5].

As a case study, we now develop the task-dependency model of the BASIZ application. This is a C+PVM application that performs the detection of the zones with most brightness and colour intensity within a given image. The program behaviour of BASIZ was automatically synthesized with the tool AMEEDA [4]. This behaviour was obtained by a tracing mechanism when running the application on a single processor and for a single image. Figure 2 shows the task-dependency model that was obtained for BASIZ, that is composed of 21 tasks ($T_0, \dots, T_{20}$) that are arranged in a pipeline structure with seven steps. The highest-cost computation phase was represented with a cost of 1000 computation units, and the remaining of computation phases had assigned a cost relative to this. The communication costs are expressed in Kbytes.

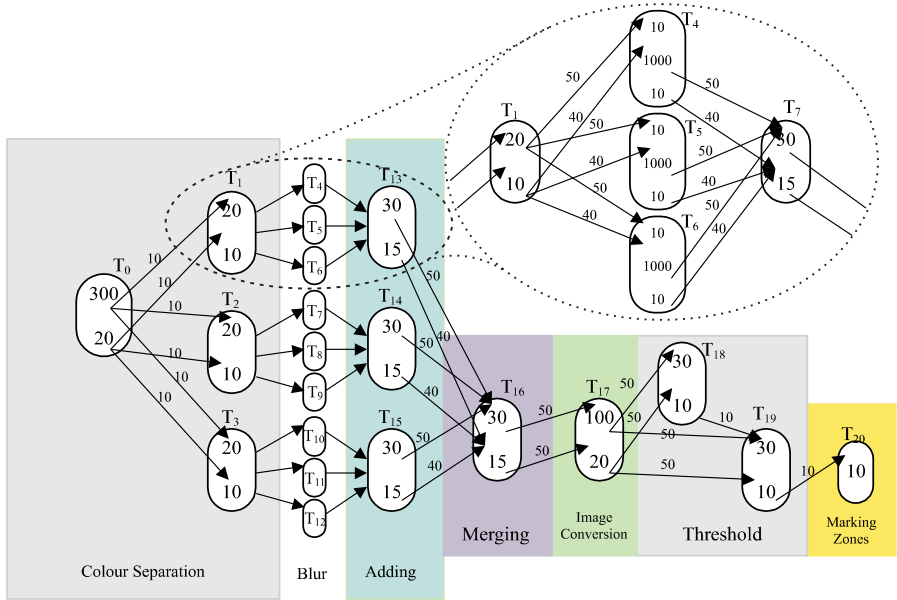


Fig. 2. Task-dependency model of BASIZ application.

3 Performance Evaluation

In this section, we present the usability of the simulation environment, to carry out performance evaluation of parallel applications under different mapping al-

ternatives. The task assignment corresponding to task and mixed parallel mappings were carried out with the MATE strategy reported in [5]. This experimentation process was performed with the application BASIZ.

The performance evaluation was carried out when the BASIZ application was executed by varying the number of frames of the stream of images and the number of processors. The distribution of frames was based on a cyclic fashion.

Since the application processes a stream of images, there are two distinct criteria for judging the quality of a mapping: latency and throughput. The latency is the time taken to process an individual image, while the throughput is the aggregate rate at which the images are processed. We first verified that the task-dependency model of BASIZ is representative of its behaviour by running the application for 80 and 160 frames in a cluster of PC's with different mapping alternatives. We could see that the predicted values of latency and throughput that were obtained by executing the task-dependency model in ESPPADA are very close to the measured when executing the application in the real platform.

To carry out a more extended evaluation of performance, based on these measures of quality, we focus on the study of the two following aspects: (a) scalability of performance measures when the problem size is augmented and, (b) identifying possible improvements of the application that better exploit parallelism.

3.1 Study of Scalability

The study of the application's scalability was carried out with the task-dependency model of BASIZ, by simulating its execution for a stream of images with 1280 frames. In this simulation, we used a number of processors varying from 16 to 128 and the two following mapping alternatives: (a) data parallel mapping and, (b) mixed task and data parallel mapping with groups of 8 processors. Pure task parallel mapping was not used, since the number of application tasks is not sufficient to require the use of all the processors. Figure 3 shows the latency and the throughput obtained in these conditions.

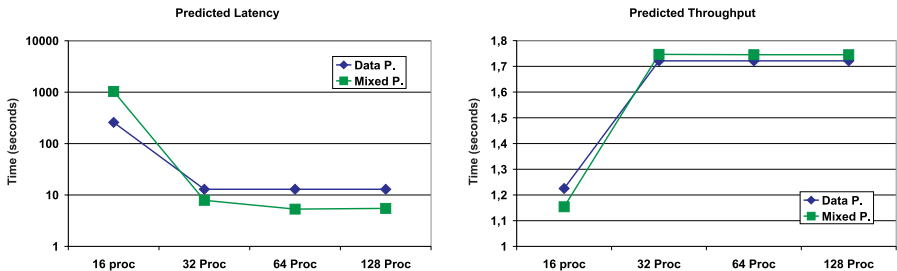


Fig. 3. Scalability of latency and throughput.

As can be observed, both measures become stable when the application is executed with more than 32 processors. This is due to different reasons, depen-

ding on the mapping. In the case of data parallel mapping, minimum latency is determined by the time needed to process an image through the internal pipe of BASIZ that corresponds to the sum of the cost of all the application tasks. When the number of processors is augmented, a smaller number of frames is distributed to each processor and the time to receive each frame increases. Thus, the pipes process only one frame at a time and, as a consequence, the frames are processed at the maximum rate. for this reason, latency and throughput have the same value from 32 to 128 processors, and these values cannot be improved when executing in a data parallel fashion.

In the case of mixed task and data-parallel mapping, each processor executes only the tasks it has assigned. The minimum latency for this mapping is determined by the critical path of the application (i.e. the path with greatest sum of computation and communication costs). Therefore, it can be observed that the mixed parallelism is able to reduce minimum latency with respect to the data parallel mapping, when this minimum is achieved, the throughput also becomes constant.

3.2 Improving Performance

In order to improve performance, it is not possible to have better results for the throughput. In the case of latency, this can be improved only with mixed parallel mapping by reducing the critical path of the application and being able to have a more balanced distribution of tasks. This was achieved by splitting the tasks with the greatest execution cost (tasks T4,...,T12 in the task-dependency model of Figure 2) into three balanced tasks each. Figure 4(a) shows how the latency is reduced with respect to the other mappings, when executing the modified model, which is labeled as Bal.Mix.P.

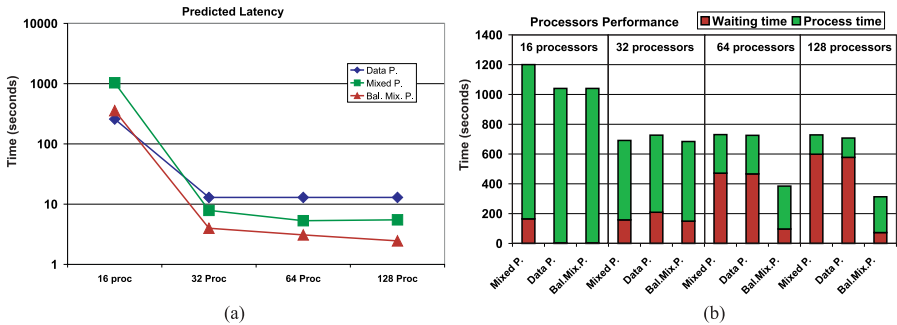


Fig. 4. (a) Diminution of latency for the balanced model of BASIZ. (b) Internal utilization of processors for different mappings.

For all these executions on the simulator, we studied the manner in which the processors work internally on all the mapping solutions. Figure 4(b) shows the

medium waiting time and medium processing time for each mapping, varying the number of processors. It can be observed that when the number of processors is increased, the waiting time for data and mixed parallel mappings also increases. For data parallel mapping, this can be explained because the frames need more time to be delivered to each processor. In the case of mixed parallel mapping, the tasks of the blurring process (tasks T4,...,T12) yield an unbalanced distribution that forces certain processors to wait while other processors are executing these tasks. The proposed solution of splitting these hard-cost tasks avoids this unbalanced distribution and reduces the length of the critical path. As a consequence, it is able to reduce medium waiting time in all cases.

4 Conclusions

The simultaneous exploitation of task and data parallelism can lead to significantly faster programs than the sole exploitation of data parallelism. In this work, we have presented a simulation environment that performs the execution of parallel applications to predict the best mapping alternatives, based on task parallelism, data parallelism and a combination of both. A mapping evaluation process was carried out for an image processing application called BASIZ, composed of different tasks that are arranged in a pipeline structure.

We studied the scalability of latency and throughput for BASIZ. We have shown that the mixed task and data parallel mapping is the best one to minimize the latency. Additionally, we have shown that the application does not scale when more than 32 processors are used. In order to improve latency, a new implementation alternative was proposed. The effectiveness of this alternative was evaluated in the developed environment, and it has shown to produce better performance results.

References

1. Rauber T. and Rünger G.: Load Balancing Schemes for Extrapolation Methods. *Concurrency: Practice and Experience*. vol. 9. n. 3, pp: 181–202. 1997.
2. Subhlok J. and Vongran G.: Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations. *J. Par. Distr. Computing*. vol. 60. pp 297–319. 2000.
3. Guirado F.: ESPPADA: Simulation Environment for Parallel Programs in Distributed Architectures. Master thesis (in Spanish). Computer Science Dep. Universidad Autónoma de Barcelona. 2001.
4. Yuan X., Roig C., Ripoll A., Senar M.A., Guirado F. and Luque E.: AMEEDA: A General-Purpose Mapping Tool for Parallel Applications on Dedicated Clusters. *Proc. Euro-Par'02. LNCS* vol. 2400. pp: 248–252. 2002.
5. Roig C., Ripoll A., Senar M.A., Guirado F. and Luque E.: A New Model for Static Mapping of Parallel Applications with Task and Data Parallelism. *IEEE Proc. of IPDPS-2002 Conf.* ISBN: 0-7695-1573-8. Apr. 2002.

Dynamic Load Balancing for I/O- and Memory-Intensive Workload in Clusters Using a Feedback Control Mechanism

Xiao Qin, Hong Jiang, Yifeng Zhu, and David R. Swanson

Department of Computer Science and Engineering
University of Nebraska – Lincoln, Lincoln, NE, USA.
{xqin, jiang, yzhu, dswanson}@cse.unl.edu

Abstract. One common assumption of the existing models of load balancing is that the weights of resources and I/O buffer size are statically configured. Though the static configuration of these parameters performs well in a cluster where the workload can be predicted, its performance is poor in dynamic systems where the workload is unknown. In this paper, a new feedback control mechanism is proposed to improve the overall performance of a cluster with I/O-intensive and memory-intensive workload. The mechanism dynamically adjusts the resource weights as well as the I/O buffer size. Results from a trace-driven simulation show that this mechanism is effective in enhancing the performance of a number of existing load-balancing schemes.

1 Introduction

Dynamic load balancing schemes in a cluster can improve system performance by attempting to assign work, at run time, to machines with idle or under-utilized resources. Several distributed load-balancing schemes have been presented in the literature, primarily considering CPU [1], memory [2], or a combination of CPU and memory [3]. Although these load-balancing policies have been very effective in increasing the utilization of resources in distributed systems, they have ignored one type of resource, namely disk I/O. The impact of disk I/O on overall system performance is becoming significant as more and more jobs with high I/O demand are running on clusters. Therefore, we have developed two load balancing algorithms to improve the read and write performance of a parallel file system [4][5].

Very recently, surdeanu et. al [6] developed a load balancing model that considers I/O, CPU, and memory resources simultaneously. In this model, three resource weights were introduced to reflect the significance of resources in a cluster and, as an assumption, the weights of system resources are statically configured. To release this assumption that does not hold for clusters with dynamic workloads, this paper proposes a feedback control mechanism to judiciously configure the resource weights in accordance with the dynamic workloads. Moreover, the new mechanism is able to improve the buffer utilization of each node in the cluster when its workload is I/O- and/or memory-intensive.

2 Adaptive Load Balancing Scheme

2.1 Weighted Average Load-Balancing Scheme

We consider the issue of a feedback control method in a cluster, $M = \{M_1, \dots, M_n\}$, connected by a high-speed network. Since jobs may be delayed because of sharing resources with other jobs or being migrated to remote nodes, the slowdown imposed on a job j is defined as:

$$slowdown(j) = \frac{time_{WALL}(j)}{time_{CPU}(j) + time_{IO}(j)} \quad (1)$$

where $time_{WALL}(j)$ is the total time the job spends running, accessing I/O, waiting, or migrating, and $time_{CPU}(j)$ and $time_{IO}(j)$ are the times spent by j on CPU and I/O, respectively, without any resource sharing.

For a newly arrived job j at a node i , load balancing schemes attempt to ship it to a remote node with the lightest load if node i is heavily loaded, otherwise job j is admitted into node i and executed locally. We propose a weighted average load-balancing scheme, or WAL-FC, where a feedback control mechanism is incorporated. Each job is described by its requirements for CPU, memory, and I/O, which are measured by Seconds, Mbytes, and number of I/O accesses per ms, respectively. For a newly arrived job j at a node i , WAL-FC balances the system load in five steps. First, the load of node i is updated by adding job j 's load. Second, a migration is initiated if node i is overloaded. Third, a candidate node k with the lowest load is chosen. To avoid useless migrations, the load discrepancy between node i and k has to be greater than the load induced by job j . If a candidate node is not available, no migration will be carried out. Fourth, WAL-FC determines if job j 's migration is able to potentially reduce the job's slowdown. Finally, job j is migrated to the remote node k , and the loads of nodes i and k are updated.

WAL-FC calculates the weighted average load index of each node i , which is defined as the weighted average of CPU and I/O load, thus:

$$load_{WAL} = W_{CPU} \times load_{CPU}(i) + W_{IO} \times load_{IO}(i), \quad (2)$$

where $load_{CPU}(i)$ is CPU load defined as the number of running jobs and $load_{IO}(i)$ is the I/O load defined as the summation of the individual implicit and explicit I/O load contributed by jobs assigned to node i . It is noted that the memory load is expressed by the implicit I/O load imposed by page faults. Let $l_{page}(i, j)$ and $l_{IO}(i)$ denote the implicit and explicit I/O load of job j assigned to node i , respectively, then, $load_{IO}(i)$ can be defined as:

$$load_{IO}(i) = \sum_{j \in M_i} l_{page}(i, j) + \sum_{j \in M_i} l_{IO}(i). \quad (3)$$

When the node's available memory space is larger than or equal to the memory demand, there is no implicit I/O load imposed on the disk. Conversely, when the memory space of a node is unable to meet the memory requirements, page faults may lead to a high implicit I/O load.

2.2 A Feedback Control Mechanism

To retain high performance, a feedback control mechanism is employed to automatically adjust the weights of resources. The high level view of the architecture for the mechanism is presented in Fig.1, where the architecture comprises a load-balancing scheme, a resources-sharing controller, and a feedback controller. The slowdown of a newly completed job and the history slowdowns are fed back to the feedback controller, which then determines the required control action ΔW_{IO} . ($\Delta W_{IO} > 0$ means the IO-weight needs to be increased, and otherwise the IO-weight should be decreased.) Since the sum of W_{CPU} and W_{IO} is 1, the control action ΔW_{CPU} can be obtained accordingly.

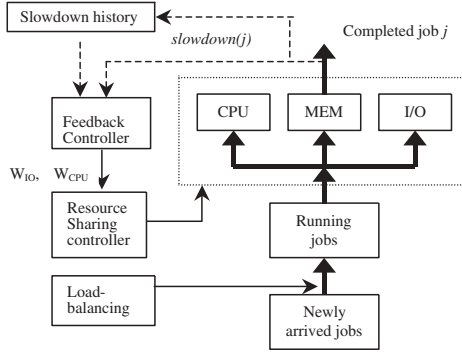


Fig. 1. Architecture of the feedback control mechanism

The feedback controller attempts to manipulate W_{CPU} and W_{IO} in the following three steps. First, the controller calculates the slowdown s_j of the newly completed job j . Second, s_j is stored in the history table, which reflects a specific pattern of the recent slowdowns. The average value of the slowdowns (s_{avg}) in the history table is computed. Finally, the performance is considered being improved if $s_{avg} > s_j$, therefore W_{IO} is increased if it has been increased by the previous control action, otherwise W_{IO} is decreased. Similarly, $s_{avg} < s_j$ means that the performance has been worsened, suggesting that W_{IO} has to be increased if the previous control action has reduced W_{IO} , and vice versa.

Besides configuring the weights, the feedback control mechanism is utilized to dynamically manipulate the buffer size of each node. The main goals are: (1) improve the buffer utilization and the buffer hit rate; and (2) reduce the number of page faults. In Fig.1, the feedback control generates action $\Delta bufSize$ in addition to ΔW_{CPU} and ΔW_{IO} . Specifically, $s_{avg} > s_j$ means the performance is improved by the previous control action, thereby increasing the buffer size if it has been increased by the previous control action, otherwise the buffer size is reduced. Likewise, $s_{avg} < s_j$ indicates that the latest buffer control action leads to a worse performance, implying that the buffer size has to be increased if the previous control action has reduced the buffer size, and vice versa.

3 Experiments and Results

To evaluate the performance of the proposed scheme, We have evaluated the performance of the following load-balancing policies:

- (1) CM: the CPU-memory-based policy [3] without using buffer feedback controller(BFC).
- (2) IO: the IO-based policy without using the BFC algorithm. The IO policy uses a load index that represents only the I/O load.
- (3) WAL: the Weighted Average Load-balancing scheme without BFC [6].
- (4) CM-FC: the CPU-memory-based policy in which BFC is incorporated.
- (5) IO-FC: the IO-based load-balancing policy to which BFC is applied.
- (6) WAL-FC: the Weighted Average Load-balancing scheme with BFC.

In addition, the above schemes are compared with the performance of the following two policies: the non-load-balancing policy without BFC(NLB) and the non-load-balancing policy that employs BFC(NLB-FC).

3.1 Simulation Model

To study dynamic load balancing, Harchol-Balter and Downey [1] implemented a simulator for a distributed system with six nodes. Zhang et. al [3] extended this simulator by incorporating memory recourses. We have modified these simulators, implementing a simple disk model and an I/O buffer model. The traces used in the simulation are modified from [1][3], and it is assumed that the I/O access rate is randomly chosen in accordance with a uniform distribution. The simulated system is configured with parameters listed in Fig.2.

Parameters	Value	Parameters	Value
CPU Speed	800 MIPS	Page Fault Service Time	8.1 ms
RAM Size	640 MByte	Seek and Rotation time	8.0 ms
Initial Buffer Size	160MByte	Disk Transfer Rate	40MB/Sec.
Context switch time	0.1 ms	Network Bandwidth	1Gbps

Fig. 2. Data Characteristics

Disk accesses of each job are modeled as a Poisson process. Data sizes of the I/O requests in each job are randomly generated based on a Gamma distribution with the mean size of 250KByte and the standard deviation of 50KByte. Since buffer can be used to reduce the disk I/O access frequency, we approximately model the buffer hit probability for job j running on node i as follows:

$$hit\_rate(i, j) = \begin{cases} \frac{r_j}{r_j+1} & \text{if } d_{buf}(i, j) \geq d_{data}(j), \\ \frac{r_j}{r_j+1} \times \frac{d_{buf}(i, j)}{d_{data}(j)} & \text{otherwise,} \end{cases} \quad (4)$$

where r_j is the data re-access rate, $d_{buf}(i, j)$ is the buffer size allocated to job j , and $d_{data}(j)$ is the amount of data job j retrieves from or stores to the disk,

given a buffer with infinite size. Buffer is a resource shared by multiple jobs in the node, and the buffer size a job can obtain in node i at run time depends on the jobs' I/O access rate and mean data size of I/O accesses.

Figure 3 shows the effects of buffer size on the buffer hit probabilities of the NLB, CM and IO policies. When buffer size is smaller than 150MByte, the buffer hit probability increases almost linearly with the buffer size. The increasing rate of the buffer hit probability drops when the buffer size is greater than 150MByte, suggesting that further increasing the buffer size can not significantly improve the buffer hit probability when the buffer size approaches to a level at which a large portion of the I/O data can be accommodated in the buffer.

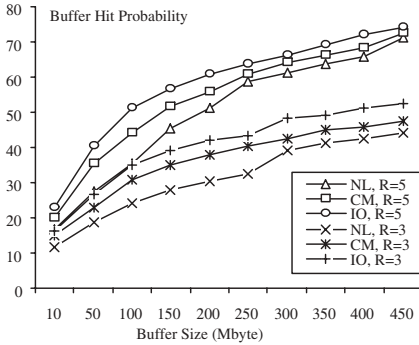


Fig. 3. Buffer Hit Probability. Page-fault rate is 4.0No./ms, I/O access rate is 2.2No./ms, R is data re-access rate.

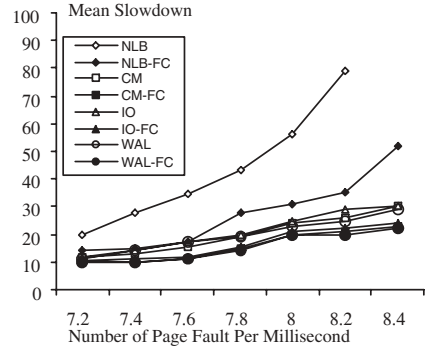


Fig. 4. Mean slowdowns as a function of the page-fault rate. I/O access rate of 1.5No./ms

3.2 Memory and I/O Intensive Workload

This section attempts to show an interesting case where the workload is both memory and I/O intensive. The I/O access rate is set to 1.5No./ms, and the page fault rate is from 7.2No./ms to 8.4No./ms.

Not surprisingly, Figure 4 shows that load-balancing schemes achieve significant improvements. Moreover, the performances of CM, IO, and WAL are close to one another, whereas the performance of CM-FC is similar to those of IO-FC and WAL-FC. This is because the trace comprises memory-intensive and I/O-intensive jobs. Hence, while CM and CM-FC take advantage of balancing CPU-memory load, IO and IO-FC can enjoy benefits of balancing I/O load.

A second observation is that, under the memory and I/O intensive workload, CM-based policies achieve higher level of improvements over NLB than IO-based schemes. The reason is that when memory and I/O demands are high, the buffer sizes in a cluster are unlikely to be changed, as there is a memory contention

among memory-intensive and I/O-intensive jobs. Thus, the buffer sizes finally converge to a value that minimizes the mean slowdown.

Third, the result shows that the feedback control mechanism can further improve the performance of the load-balancing schemes. For example, WAL-FC further decreases the slowdown of WAL by up to 27.3%, and IO-FC further decreases the slowdown of IO by up to 24.7%. This result suggests that, to sustain a high performance in clusters, compounding a feedback controller with an appropriate load-balancing policy is desirable and strongly recommend.

4 Conclusion

In this paper, we have proposed a feedback control mechanism to dynamically adjust the weights of resources and the buffer sizes in a cluster. The proposed algorithm minimizes the number of page faults for memory-intensive jobs while improving the buffer utilization of I/O-intensive jobs. A trace-driven simulation demonstrates that our approach is effective in enhancing performance of existing dynamic load-balancing algorithms. Future study in this area is two-fold. First, we plan to study the stability of the proposed feedback controller. Second, we will study how quickly the controller converges to the optimal value.

Acknowledgments. This work was partially supported by an NSF grant (EPS-0091900), a Nebraska University Foundation grant (26-0511-0019), and a UNL Academic Program Priorities Grant. Work was completed using the Research Computing Facility at University of Nebraska-Lincoln.

References

1. Harchol-Balter, M., Downey, A.: Exploiting process lifetime distributions for load balancing. *ACM Transactions on Computer Systems* **15** (1997) 253–285
2. Acharva, A., Setia, S.: Availability and utility of idle memory in workstation clusters. In: *Proceedings of the ACM SIGMETRICS Conf. on Measuring and Modeling of Computer Systems*. (1999)
3. Zhang, X., Qu, Y., Xiao, L.: Improving distributed workload performance by sharing both cpu and memory resources. In: *Proceedings of the 20th Int'l Conf. on Distributed Computing Systems*. (2000)
4. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.: Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs). In: *Proc. of the 3rd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*. (2003) 730–735
5. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.: Scheduling for improved write performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs). In: *the Fourth LCI International Conference on Linux Clusters*. (2003)
6. Surdeanu, M., Modovan, D., Harabagiu, S.: Performance analysis of a distributed question/answering system. *IEEE Trans. on Parallel and Distributed Systems* **13** (2002) 579–596

An Experimental Study of k -Splittable Scheduling for DNS-Based Traffic Allocation

Amit Agarwal^{1*}, Tarun Agarwal¹, Sumit Chopra^{2*}, Anja Feldmann³,
Nils Kammenhuber³, Piotr Krysta⁴, and Berthold Vöcking⁵

¹ IIT Delhi, India

² Department of Computer Science, Hansraj College, University of Delhi, India.

³ Department of Computer Science, Technische Universität München, Germany.

⁴ Max-Planck-Institut für Informatik, Saarbrücken, Germany.

⁵ Department of Computer Science, Universität Dortmund, Germany.

Abstract. The Internet domain name system (DNS) uses rotation of address lists to perform load distribution among replicated servers. We model this kind of load balancing mechanism in form of a set of request streams with different rates that should be mapped to a set of servers. Rotating a list of length k corresponds to breaking streams into k equally sized pieces. We compare this and other strategies of how to break the streams into a bounded number of pieces and how to map these pieces to the servers.

One of the strategies that we study computes an *optimal k -splittable allocation* using a scheduling algorithm that breaks streams into at most $k \geq 2$ pieces of possibly different size and maps these pieces to the servers in such a way that the maximum load over all servers is minimized. Our experimental study is done using the network simulator SSFNet. We study the average and maximum delay experienced by HTTP requests for various traffic allocation policies and traffic patterns. Our simulations show that splitting data streams can reduce the maximum as well as the average latency of HTTP requests significantly. This improvement can be observed even if streams are simply broken into k equally sized pieces that are mapped randomly to the servers. Using allocations computed by machine scheduling algorithms, we observe further significant improvements.

1 Introduction

The Internet's Domain Name System (DNS) is responsible for resolving host names like "www.tum.de" into IP addresses. This service is also being used to perform load distribution among distributed Web servers. Busy sites are replicated over multiple servers, with each server running on a different end system having its own IP address. Thus a set of IP addresses is associated with one canonical hostname contained in the DNS database. When clients make a DNS query for such a host name, the server returns the entire set of addresses, thereby rotating the order of addresses within each reply. Because a client typically sends its HTTP request to the server listed first, the traffic is distributed among all replicated servers (cf. [7]).

* This work was done while the author was visiting the MPI in Saarbrücken.

One can view the requests that are directed to the same URL as traffic streams. The rotation of address lists basically splits these streams into equally sized pieces. Suppose the request streams are formed by a sufficiently large number of clients. Then the arrivals of requests can be described by a stochastic process. In the scenario that we consider there are n streams and m identical servers. Let λ_j denote the rate of stream $j \in [n]$, i. e., the expected number of requests in some specified time interval. Under this assumption, rotating a list of k servers corresponds to splitting stream j into k substreams each of which having rate λ_j/k . The following slightly more sophisticated stochastic splitting policy would possibly achieve a better load balancing. Suppose, the DNS attaches a vector $p_1^j, \dots, p_k^j$ with $\sum_i p_i^j = 1$ to the list of each stream j . In this way, every individual request in stream j can be directed to the i th server on this list with probability p_i^j . This policy breaks Poisson stream j into k Poisson streams of rates $p_1^j \lambda_j, \dots, p_k^j \lambda_j$, respectively.

The focus of this paper is not on how exactly to implement such strategies within the DNS implementation. Instead, our goal is to study the impacts of different allocation strategies a few steps ahead of current DNS implementations, including calculating an optimal k -split allocation. The optimal k -split allocation is computed using a variant of a recently presented algorithm for the k -splittable machine scheduling problem. Suppose a set of jobs (data streams) $[n] = \{1, \dots, n\}$ of sizes $\lambda_1, \dots, \lambda_n$ need to be scheduled on a set of identical machines (web servers) $[m] = \{1, \dots, m\}$. Every job can be broken into at most k pieces. The goal is to map job pieces to machines so that the makespan, i. e. the maximum load over all machines, is minimized.

Scheduling with bounded splittability was introduced by Shachnai and Tamir [9]. They prove that the problem is NP-hard and present approximation algorithms. Krysta et al. [6] presented an exact algorithm for the k -splittable machine scheduling problem with running time $O(m^{m+m/k} + n)$. Thus the problem can be solved in polynomial time for any fixed number of machines. It is well known that the classical unsplittable scheduling problem is NP-hard for 2 machines, so the result from [6] proves that bounded splittability reduces the problem's complexity for a fixed number of machines.

We remark that the algorithms in [6,9] not only work for identical but also for machines of different speeds. In this paper, we focus on the special case of identical machines. Note that the above and all following bounds on the running time refer to algorithms that compute feasible allocations for any given upper bound on the makespan. These algorithms, in turn, can be used to compute the optimal makespan by applying binary search techniques.

When implementing the algorithm from [6], we learned that the exponential term in the running time let us only compute optimal assignments for relatively small numbers of machines. For some randomly generated instances with less than 40 machines the algorithm did not terminate in reasonable time. It was obvious, however, that, in the case of identical machines, one could exploit symmetries to speed up the algorithm. Clearly, because of the NP-hardness of the problem, the best one can hope for is to soften the exponential influence of the number of the machines. The theoretical contribution of this paper is an upper bound of $O((2m)^{m/(k-1)+1}(m/k) + n)$ on the running time of an exact algorithm for the k -splittable scheduling problem on identical machines. Observe that this result yields the first complete tradeoff on the influence of different degrees of

splittability on the running time ranging from polynomial time in the case of $k = \Omega(m)$ to exponential time when $k = O(1)$.

In Section 2, we present this improved algorithm. Furthermore, we give some experimental results for different input distributions showing that the improved algorithm can be used to compute optimal solutions for several hundred machines very efficiently. In Section 3, we present our comparative study of different traffic allocation strategies applied to a simple network topology in which we measure average and maximum delays of HTTP requests. For a brief summary of these results, we refer the reader to the Conclusions presented in Section 4.

2 An Improved Algorithm for k -Splittable Scheduling

In this section, we present an improved algorithm for k -splittable scheduling on identical machines. We refer the reader to [6] for the original algorithm.

Let $x_{i,j}$ denote the fraction of job j that goes to machine i , let k_j be the current splittability of j . Of course, at the beginning we have $\forall j : k_j = k$. The *bulkiness* of a job j is defined as $\lambda_j / (k_j - 1)$. Assuming that a value z for the makespan (maximum load) is given, we look for an algorithm that either computes an assignment satisfying $\max_{i \in [m]} \sum_{j \in [n]} \lambda_j x_{ij} \leq z$ or returns that there is no z . Given such an algorithm, the original optimization problem can be solved in a polynomial number of iterations by applying binary search techniques over the rational numbers [5,8]. The binary search works despite the possibility of irrational splits because the value of the optimal solution can be proved to be rational [6].

2.1 The Improvement

We change the algorithm only in a small aspect: we reduce search space size by exploiting symmetries. Considering all machines with the same remaining capacity as belonging to one equivalence class, there is no need to backtrack over machines in the same class. In the case of identical machines, all *empty machines* (remaining capacity = z) fall into the same equivalence class. In contrast, *partially filled machines* (remaining capacity in $(0, z)$) typically build equivalence classes of size one. For simplicity of the implementation, we thus only exploit the symmetry among empty machines. When picking the jobs in the order of their bulkiness, we distinguish three cases (j denotes the job's index):

- 1) If $k_j \geq 2$ then we saturate one of the machines, selecting a machine from the class of the empty machines first and then trying all machines with reduced capacity.
- 2) If $k_j = 1$ then we put the remaining piece job j to a machine i with $c_i \geq \lambda_j$, first trying all partially filled machines and then one of the empty machines.
- 3) If $\frac{\lambda_j}{k_j - 1} \leq \min_{i \in [m]} (c_i)$ then the McNaughton phase is initiated.

Obviously, our modification does not affect the algorithm's correctness. However, one might object that exploiting symmetries cannot help too much as the assignment of jobs destroys these symmetries rapidly. Nevertheless, the following theorem states that the running time is improved dramatically.

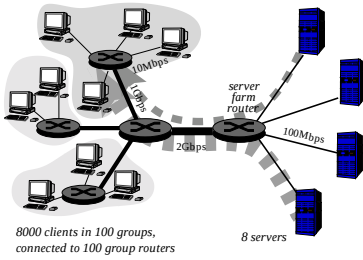


Fig. 1. Topology of simulated network.

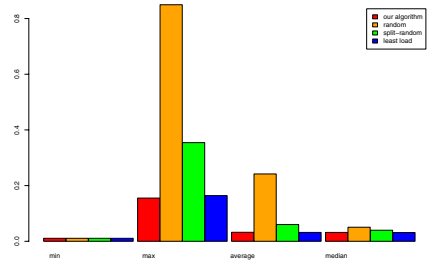


Fig. 2. Experienced response delays

Theorem 1. *The running time of the improved algorithm for the k -splittable traffic allocation problem is $O((2m)^{m/(k-1)+1}(m/k) + n)$, for every $k \in \{2, \dots, m\}$.*

The proof of this theorem is provided in [11]. Observe that the theorem yields a complete tradeoff on the influence of the different degrees of splittability on the running time. In particular, the following result was not known before.

Corollary 1. *For any constant $c > 0$ and $k \geq \max\{2, \frac{m}{c}\}$, k -splittable scheduling on identical machines is solvable in polynomial time.*

We implemented both the original algorithm and its modification to compare their running times. An evaluation showed that also in practice, the modified algorithm performs significantly better than the original one. For details, we refer the reader to [11].

3 Evaluation of Server Load Balancing Simulations

To check how well our algorithms performs in the server farm load balancing scenario, we compared their results against three heuristics, i. e., *random allocation* (assign each job to a random machine), *random k -split allocation* (split each job into k equal parts, assign each part to a random machine) and *least-load* (“LL” — sort the jobs by their rates, split each job into k equal parts, assign each part to the machine with the smallest load at the time of scheduling).

For various k values, we created 10 random job assignment problems with uniform machine capacities. This was done for uniform, equally-distributed, exponentially distributed and Pareto-distributed job sizes, the latter for various α values. From the solutions gained by the algorithm (or the heuristics, respectively), we created network setups where each machine is represented by a web server and each job is represented by a group of uniform clients. The number of clients in a group reflects the workload of that job. Job assignment is modelled in a probabilistic way: Each group is given a probability vector that defines the probabilities under which the clients will issue their requests to what server. The network setups then were simulated using *SSFNet* [10].

We started using relatively simple network setups and experimented with various parameters, e. g. the link delays and random distributions of the object sizes, and slowly moved on to complex scenarios. We refer the reader to [11] for further details.

Among other evaluations, we calculated the delays between a request being issued and the first byte of the response arriving at the client. Finally, we used a realistic web workload model for the client's inter-request times, sizes of the objects being served, etc., which was derived from [1] and [3]. The network topology used is shown in Fig. 1.

An evaluation of some simulation runs with this setup is depicted in Fig. 2. We see that our algorithm clearly wins over the randomized schemes. It also wins over the simple LL heuristics, however the gain is rather narrow. Although Fig. 2 being an example, we note that these are tendencies also present in other simulation runs [11].

4 Conclusions

We studied delay and bandwidths experienced by HTTP requests for various traffic allocation policies and traffic patterns. Our simulation results show that splitting data streams into a small number of pieces can reduce the maximum as well as the average latency of HTTP requests significantly.

Besides to random allocations, we devised and studied allocation strategies that allocate pieces of jobs to machines with the objective to minimize the maximum load over all servers. These strategies lead to a clear improvement in the maximum as well as the average latency of HTTP requests. It turned out, however, that the differences between the latencies obtained by the simple LL heuristic and the latencies obtained by an optimal k -splittable allocation are small.

There are several questions left open by our analysis. Certainly, some more test runs with the very large network setup would be interesting. It also could make sense to make the simulation environment even more realistic by studying even more complex network topologies. A second interesting topic are dynamic allocation schemes that use splittability to realize a smooth rather than an abrupt adaptation to dynamically changing traffic patterns. These are topics that we plan to investigate in future work.

References

1. P. Barford, M. Crovella. *Web Server Workload Characterization: The Search for Invariants*. Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems 1996, pp. 126–137
2. M. Crovella, A. Bestavros. *Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes*. IEEE/ACM Transactions on Networking'96.
3. A. Feldmann, A. Gilbert, P. Huang and W. Willinger. *Dynamics of IP traffic: a study of the role of variability and the impact of control*. Proceedings of SIGCOMM'99.
4. M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, 1979.
5. St. Kwek and K. Mehlhorn. *Optimal search for rationals*. Information Processing Letters, 86:23–26, 2003.
6. P. Krysta, P. Sanders and B. Vöcking. *Scheduling and Traffic Allocation for Tasks with Bounded Splittability*. Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS), to appear, 2003.
7. J. F. Kurose, K. W. Ross. *Computer Networking: a top-down approach featuring the Internet*. Addison-Wesley, 2001.

8. C. H. Papadimitriou. *Efficient search for rationals*. Information Processing Letters, 8:1–4, 1979.
9. H. Shachnai and T. Tamir. *Multiprocessor Scheduling with Machine Allotment and Parallelism Constraints*. Algorithmica, 32(4): 651–678, 2002.
10. Scalable Simulation Framework Research Network (SSFNet). <http://www.ssfnet.org/>
11. A. Agarwal, T. Agarwal, S. Chopra, A. Feldmann, N. Kammenhuber, P. Krysta, B. Vöcking. *An Experimental Study of k -Splittable Scheduling for DNS-Based Traffic Allocation*. Technical Report TUM-I0304, Technische Universität München, 2003.

Scheduling Strategies of Divisible Loads in DIN Networks

Ligang Dong, Lek Heng Nghoh, and Joo Geok Tan

Institute for Infocomm Research
21 Heng Mui Keng Terrace, Singapore 119613
{donglg, lhn, tjg}@i2r.a-star.edu.sg

Abstract. The DIN network is a special network, in which data can “live” by circulating. Circulating data can be rapidly accessed by a large number of hosts connected to the network. Scheduling strategies of processing divisible loads among hosts in DIN networks is studied to minimize the total processing time. We derive the closed-form solution of the processing time. Moreover, our experiment shows that the performance of our scheduling strategy can reduce the process time of a load up to previous 60%.

1 Introduction

In a distributed computing system, a load is often computed/processed among several cooperative computer hosts/processors, so that the processing time of the entire load can be minimized. One of the main methods is to design efficient scheduling strategies [6]. In detail, through the mathematical modelling based on the parameters of networks and hosts, the optimal size and the starting time of load fractions to be distributed among the hosts are calculated. The divisible load may be either arbitrarily divisible or grain-based (i.e., there is a minimum size of data that can be computed independently). We consider only arbitrarily divisible load in this paper. In the past few years, there is a rich development on scheduling divisible loads because of the wide availability of divisible loads, e.g., military and space information, image/vision data, matrix vectors, queries. These data can be treated to be divisible in quite a certain extent. Early study results (from 1988 to 1996) can be found in [1][4]. A compendium of relative papers is available in [5].

In this paper, we study the scheduling strategy of divisible loads in a special network - DIN networks (shown in Fig. 1). The motivation of proposing DIN networks is to make use of the expected proliferation of optical bandwidth and to solve the bottleneck problem of accessing highly popular data from the server I/O. The bottleneck includes “narrow” I/O bandwidth and “long” access time. We let data circulate in a fiber loop, called *DIN loop*. Through a modified router, called *DIN node*, data can be written into or read from the DIN loops. Since servers, which have relatively slow response and small throughput, are not included in the access process, *more* users can *rapidly* access data from DIN networks.

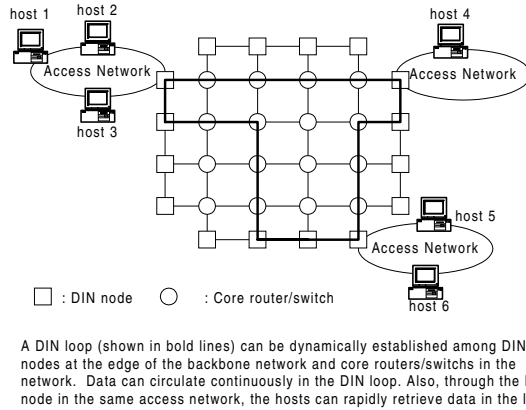


Fig. 1. Structure of DIN networks

Previous scheduling strategy still can be used for DIN networks. However, we propose a novel scheduling strategy that has much better performance than before. Our scheduling strategy has two features. In past scheduling strategy, the allocated fraction of the load is not processed until entire allocated load fraction is received. In other words, when the data is being received, the host does not process data. In our strategy, we let the host process data and receive data *simultaneously*. This strategy can be realized when the hosts are equipped with *front-ends* to communicate [1]. This is the first feature. The second feature is that, unlike traditional scheduling strategy, the originating host of the load does not *directly* transmit the load fractions to hosts. Instead, the load is transmitted to and stored in DIN loops. If one host would like to process the load, then this host can retrieve one load fraction from the DIN loop. Because the originating host of the load *does not* need to partition the load, the originator of the load does not need to know where are the hosts and their parameters (e.g., computing ability of hosts).

2 Analysis of Scheduling Strategy in Processing Arbitrarily Divisible Load

Our scheduling strategy is described as follows. The divisible load is assumed to originate at a host (referred as original host (OH)), which injects the load (in the form of data) into a DIN loop through a DIN node. Thus, the load will *circulate* in the loop. If a host would like to process a part of the load, then this host can retrieve a load fraction via a DIN node, which is in the same access work with the host. In this way, the load is distributed among m hosts, every of which processes a part of the load. For instance, the host p_i processes data with a total size of α_i .

The notation used in this analysis can be summarized as follows. m : Number of hosts that compute/process the load. α_i : Total amount of the load computed/processed in the host p_i . $\sum_{i=1}^m \alpha_i = 1$. E_i : Time taken to compute/process a unit load by the host p_i . C : Time taken to transmit a unit load to the loop in DIN by OH. θ : A constant additive communication overhead associated with a communication process by OH.

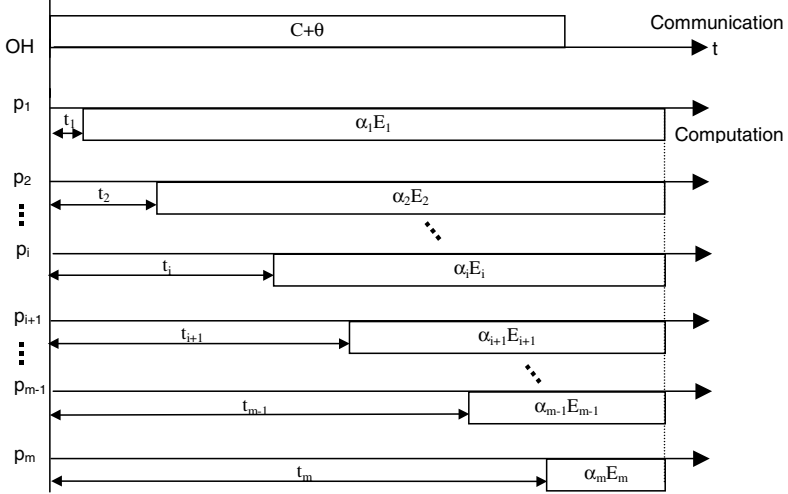


Fig. 2. Timing diagram for processing the arbitrarily divisible load with m hosts

We show the timing diagram for our scheduling strategy in Fig. 2. The traditional scheduling strategy can be found in [6]. In our scheduling strategy, we propose a *starting condition* of hosts: The host p_i cannot start the computation unless there are available data with the size of $i\alpha_0$ in the system. Here, *available data* are referred to the data that have been transferred to the DIN loop and have not computed. Actually, this starting condition is not necessary in a real system. However, with this condition, our scheduling can be analyzable. From Fig. 2, we easily calculate t_i, α_i for $i = 1, \dots, m$. The detail is ignored because of the limited space. Finally, we obtain the processing time as $T = t_m + \alpha_m E_m$.

3 Performance Comparison

We now compare the processing time of the traditional scheduling strategy [6] (referred as Strategy I) and the proposed scheduling strategy (Strategy II) through the numerical experiment. Here, our numerical experiments follow the approach in [6]. Input parameters and their default values for the numerical experiment are as follows: $\alpha_0 = 0.01$, $\theta = 0.01$, $C = 0.3$, E_i is uniform distributed in $[0.5, 1.5]$. We carry out an extensive simulation. Owing to limited space, we

Table 1. Comparison between the proposed strategy and the traditional strategy

$C = 0.3$							$C = 0.4$						
m	T^I	T^{II}	T^{II}/T^I	X^I	X^{II}	β	m	T^I	T^{II}	T^{II}/T^I	X^I	X^{II}	β
1	1.539	1.242	80%	0.000	0.000	0.013	1	1.639	1.243	75%	0.000	0.000	0.014
2	0.833	0.598	71%	0.008	0.003	0.004	2	0.914	0.600	65%	0.008	0.005	0.006
3	0.637	0.418	65%	0.027	0.014	0.006	3	0.714	0.422	59%	0.028	0.027	0.013
4	0.542	0.328	60%	0.059	0.044	0.012	4	0.619			0.062	9.923	3.963
5	0.455	0.316	69%	0.106	0.496	0.139	5	0.537			0.115		
6	0.430			0.191	0.464	-0.007	6	0.514			0.217		
7	0.416			0.307			7	0.502			0.362		
8	0.402			0.456			8	0.493			0.559		
9	0.397			0.701			9	0.491			0.911		

only show part results here. Tables 1 is the scheduling result when $C = 0.3$ and $C = 0.4$. From Table 1, firstly, if Strategies I and II adopt the same hosts to compute the load, Strategy II has a 20% – 40% improvement in comparison with Strategy I. Secondly, when the number of hosts is not limited, Strategy I is apt to make use of more hosts than Strategy II for the computation. Even in this case, Strategy II shows better performances than Strategy I.

4 Conclusion

Our main contribute is to propose an efficient and dynamic scheduling strategy of divisible loads for DIN networks. Although past scheduling strategies can be used, our scheduling strategy has better performances. Two features of our scheduling strategy can be summarized as follows. First, because of simultaneous receiving and processing of data, the processing time is reduced up to 60% of previous scheduling result. Second, we make full use of the feature of DIN networks, i.e., high throughput and short access time. We let DIN loops store the load to be processed, so that the load can be rapidly, dynamically balanced among the hosts.

References

1. V. Bharadwaj, D. Ghose, V.Mani, and T.G. Robertazzi, "Scheduling Divisible Loads in Parallel and Distributed Systems," *IEEE Computer Society Press*, Los Alamitos CA, Sep. 1996.

2. V. Bharadwaj and N. Viswanadham, "Sub-Optimal Solutions Using Integer Approximation Techniques for Scheduling Divisible Loads on Distributed Bus Networks," *IEEE Transactions on Systems, Man, and Cybernetics: Part A*, Vol.30(6), pp.680–691, Nov. 2000.

3. V. Bharadwaj, D. Ghose, and T.G. Robertazzi, "A New Paradigm for Load Scheduling in Distributed Systems," Accepted for publication in a *Special Issue in Cluster Computing*, to appear spring 2003.

4. M. Drozdowski, "Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems," Politechnika Poznanska, Book No.321, Poznan, Poland, 1997.
5. Website about divisible load scheduling, <http://www.ee.sunysb.edu/~tom/dlt.html>.
6. B. Veeravalli, Xiaolin Li, and Chi Chung Ko, "On the influence of start-up costs in scheduling divisible loads on bus networks," *IEEE Transactions on Parallel and Distributed Systems*, Vol.11(12), Dec. 2000.

Topic 4

Compilers for High Performance

Michael Gerndt, Chau-Wen Tseng, Michael O'Boyle, and Markus Schordan
Topic Chairs

Every new generation of computer architectures designed with high-performance microprocessors offers new potential gains in performance over the previous generation. The overall complexity of high-performance systems increases with every new generation. This makes it increasingly difficult to realize the full potential of a high-performance system and produce efficient code. Compilers play a pivotal role in addressing this critical issue.

Therefore we need to deal with all subjects concerning the automatic parallelization and the compilation of programs, from general-purpose platforms to specific hardware accelerators. This includes language aspects, program analysis, program transformations and optimizations for all resource utilizations.

The papers presented this year cover the spectrum of approaches to compilation for high performance with contributions to parallelization, vectorization, cache optimization, and classic compiler optimizations. The authors present valuable insights and experiences, and the resulting discussions should continue the important progress in this area.

Out of 12 submitted papers to this topic, 6 have been accepted as regular papers (50%), 2 as short papers (17%), and 4 contributions have been rejected (33%). For every paper 4 reviews have been received, in total 48 reviews.

We would like to take the opportunity of thanking all contributing authors as well as all reviewers for their work.

Partial Redundancy Elimination with Predication Techniques

Bernhard Scholz¹, Eduard Mehofer², and Nigel Horspool³

¹ Institute of Computer Languages, Vienna University of Technology
Argentinierstr. 8/4, A-1040 Vienna, Austria
`scholz@complang.tuwien.ac.at`

² Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
`mehofer@par.univie.ac.at`

³ Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
`nigelh@uvic.ca`

Abstract. Partial redundancy elimination (PRE) techniques play an important role in optimizing compilers. Many optimizations, such as elimination of redundant expressions, communication optimizations, and load-reuse optimizations, employ PRE as an underlying technique for improving the efficiency of a program.

Classical approaches are conservative and fail to exploit many opportunities for optimization. Therefore, new PRE approaches have been developed that greatly increase the number of eliminated redundancies. However, they either cause the code to explode in size or they cannot handle statements with side-effects.

First, we describe a basic transformation for PRE that employs predication to achieve a complete removal of partial redundancies. The control flow is not restructured, however, predication might cause performance overhead. Second, a cost-analysis based on probabilistic data-flow analysis decides whether a PRE transformation is profitable and should be applied. In contrast to other approaches our transformation is strictly semantics preserving.

1 Introduction

Partial redundancy elimination (PRE) is an important optimization technique in compilers for improving the efficiency of programs. The objective of PRE is to avoid unnecessary re-computations of values at runtime. The PRE transformation replaces the computations by accesses to temporary variables, where the temporary variables are initialized at suitable program points.

PRE is a key technology for compilers. In the literature, several optimizations can be found which employ PRE as the underlying technique. For example, PRE has been successfully applied in compilers for high-performance systems. Communication optimizations [KBC⁺99] and dynamic redistributions [KM02]

employ PRE as underlying optimization technique. PRE is also used for optimizations in RISC compilers. An important example of a RISC optimization is the load-reuse analysis [BGS99] that also uses PRE as underlying technique.

However, it has been observed that classical approaches [KRS94] are too cautious and fail to take advantage of many opportunities for optimization. To improve the effectiveness of PRE, new approaches were developed that use speculation [HH97,GBF98] and code duplication [Ste96]. Speculation uses profile information and inserts additional computations (i.e. speculative computations) in the program. In contrast, code duplication copies code and identifies information carrying paths. However, both techniques raise concerns: (1) speculation is not always applicable due to possible side-effects in expressions, and (2) code duplication may cause an explosion in size.

In this paper we introduce a novel transformation for PRE that achieves a complete removal of all partially redundant expressions without duplicating code. Our PRE technique uses predication. A predicate controls whether the computation in the temporary variable is valid or not. If the computation is destroyed, the predicate is updated. Computations are conditionally executed. If the temporary variable does not contain a valid value, re-computation is performed and the predicate is updated again. The transformation is guided by a cost-analysis based on probabilistic data-flow analysis [MS01,SM02] that decides whether a computation needs to be predicated. The features of our PRE approach are that (1) the control flow graph is not re-structured, (2) all partial redundancies can be removed, and (3) our transformation is semantics preserving.

The paper is organized as follows. In Section 2 we motivate PRE with predication. In Section 3 the optimization is shown in detail. The basic transformation is given and a cost analysis determines whether the transformation is beneficial to the program performance or not. Section 4 surveys related work. Finally, a summary is given in Section 5.

2 Motivation

Consider the example in Figure 1. In the first control flow graph in Figure 1(a), the computation on the left-hand branch of the inner-loop is partially redundant. The expression a/b is evaluated twice if the left-branch is executed in the first iteration of the loop and if the left-branch is subsequently executed. A traditional PRE algorithm [KRS94] is not able to eliminate this partial redundancy because the statement $b:=f(b)$ on the right-branch destroys the computation of the expression.

Now consider the second graph in Figure 1(b), which shows a complete removal of partial redundancies of the example in Figure 1(a) by employing code duplication as introduced in [Ste96]. The transformation nearly duplicates all nodes in the control flow graph. After applying the transformation two versions of a node exist in the graph. One node represents the state where the computation is not available, while the other one represents the node where the computa-

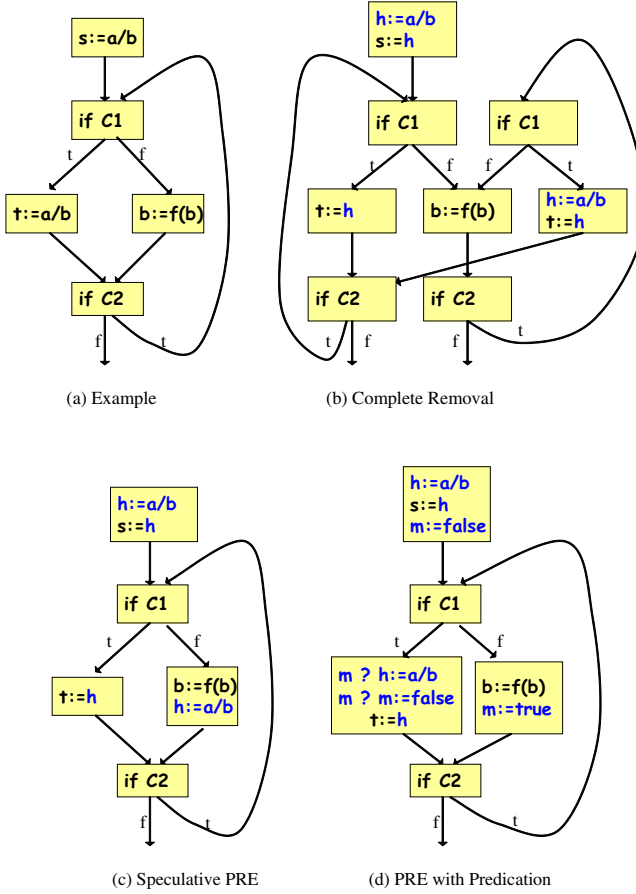


Fig. 1. Motivating Example

tion is available. Whenever the computation is destroyed, control is transferred from a node where the computation is assumed to be available to a node where it is not. Conversely, evaluation of the expression causes a control transfer to a node where it is assumed to be available.

In the first node of the example in Figure 1(b), the expression a/b is computed and stored in h . The loop on the left-hand side does not destroy the computation of a/b and therefore can re-use the value of a/b inside the loop. Whenever the computation is destroyed by $b := f(b)$, the loop on the right-hand side is entered. The program stays in the loop on the right-hand side until a/b is re-computed. Then, the computation is available in h again, and the loop on the left side is re-entered.

Not only does the control flow graph become irreducible¹, but the number of nodes has nearly doubled. In general, the code growth is exponential in the number of expressions and, therefore, the approach is not viable in practice. To alleviate this problem, Bodik et al. [BGS98] have introduced an approach that limits code growth under the guidance of profile information. However, much code would still be duplicated.

In Figure 1(c) an approach is illustrated that uses speculation [HH97,GBF98]. Let us assume that the left-branch is executed more often than the right one. Expression a/b inside the loop can only be hoisted to the first node if the expression is re-computed after destroying the computation in the right-branch. However, the expression a/b is not a safe computation since it can raise a “division by zero” exception. For example, along a subsequent execution of the right branch the computation of the expression a/b is inserted in the program which does not exist in the original program. If, on some execution of the program b happens to be zero, the error would occur. Hence, the insertion on the right branch might destroy program semantics.

The intuition behind our predication approach is straightforward, as demonstrated in Figure 1(d). Instead of duplicating code, our approach achieves complete removal of partial redundancies by introducing a predicate. The predicate reflects the unavailability of the expression in h . If the predicate holds, the expression a/b is not available in h and must be re-computed, otherwise the computation in h can be re-used. In comparison to speculative PRE, our approach is safe. No additional computations (other than predicate tests) are inserted in the program.

For re-computing the value of a/b , we use predication as shown in the example of Figure 1(d). Predication is a conditional execution of a statement depending on the logical value of a predicate. For example, the assignment in $m ? h := a/b$ on the left-branch is only executed if predicate m holds. After re-computing the expression, the predicate is set to **false** with $m ? m := \text{false};$. When a statement destroys the value of expression a/b , the predicate needs to be invalidated as shown in the node on the right branch inside the loop.

The transformation is not free of cost and neither are all computations of the expression replaced by predicated evaluations. E.g., in the first node we do not transform the computation of a/b to a predicated one because there is no preceding computation which can be exploited. In addition, the predicated computation inside the loop will only make sense if the left branch is executed more frequently than the right branch. Therefore, the transformation needs to be guided by a cost analysis based on profile information that decides whether a predicated computation would improve program performance or not.

3 PRE with Predication

In this section, we develop our predicated PRE approach. The optimization consists of an analysis part which identifies profitable predication opportunities

¹ For some optimizations irreducible control flow graphs have a negative impact.

followed by a subsequent transformation step. We start with a description of the transformation first. Subsequently we develop a cost model and describe how the transformation is guided by it. Finally, we present the analysis required for the cost model.

3.1 Basic Transformation

The key idea of predicated PRE is to save the value of an expression in a temporary and to maintain a predicate which indicates whether the saved value is still valid or not. Whenever the saved value is valid, subsequent computations of the expression can be eliminated by loading the saved value.

The transformation is shown in Fig. 2. Basically every assignment $u := exp$ has to be replaced by the sequence $h_{exp} := exp; u := h_{exp}$ where h_{exp} denotes a temporary which is associated with term exp and is used to hold the value of the last computation of exp . Additionally, we have predicate m_{exp} which is associated with the term exp as well and which indicates whether temporary h_{exp} can be reused or the term exp has to be recomputed. The important point is that $h_{exp} := exp$ is not executed each time. The term exp is evaluated only if necessary, i.e. it is evaluated if the predicate m_{exp} is true as indicated by the predicated assignment $m_{exp} ? h_{exp} := exp$, as shown in case 1 of Fig. 2. Next m_{exp} is set to false to suppress unnecessary recomputations. However, if a variable occurring in exp is modified, the value held in h_{exp} cannot be reused any more, and the term exp would need to be recomputed. This situation is described in case 2 of Fig. 2. There it is assumed that variable v occurs in term exp . Thus assigning a new value to v requires a recomputation of term exp , and this is enforced by setting the predicate m_{exp} to true.

Case 1: **Compute**

Case 2: **Destroy**

$$u := exp; \Rightarrow \begin{cases} m_{exp} ? h_{exp} := exp; \\ m_{exp} ? m_{exp} := \text{false}; \\ u := h_{exp}; \end{cases} \quad v := \dots \Rightarrow \begin{cases} v := \dots \\ m_{exp} := \text{true}; \end{cases}$$

Fig. 2. Basic Transformation

It is important to stress that contrary to other PRE approaches our transformation is strictly semantic preserving, i.e. we do not introduce new computations on any path which were not present in the original code. If an evaluation of an expression raises an exception, that exception would be raised at exactly the same point in the computation. However we add additional assignments which have to be executed and may degrade a program's performance. Hence, an analysis of the effect on performance is inevitable.

3.2 Cost Model

For each appearance of a term exp in the program we must decide whether it is profitable to perform the transformation or not. A transformation for term exp at some program point pays off if

$$OrigComp > PredicatedComp \quad (1)$$

where *OrigComp* denotes the computational costs of *exp* in the original code and *PredicatedComp* denotes the computational costs of *exp* in the transformed code. Obviously, *PredicatedComp* is dominated by the number of times the value stored in the temporary variable can be reused compared to the number of times the term has to be recomputed. If p denotes the probability that the stored value of *exp* is valid at some program point, then *PredicatedComp* is given by

$$PredicatedComp = p \times Reuse + (1 - p) \times Recompute \quad (2)$$

with *Reuse* denoting the costs if m_{exp} is set to false and the stored value can be reused, and with *Recompute* denoting the costs associated with a recomputation. By combining Equation (1) and (2) we obtain

$$p > \frac{Recompute - OrigComp}{Recompute - Reuse} \quad (3)$$

Equation (3) specifies a lower bound for probability p . The execution times of *Recompute*, *OrigComp*, and *Reuse* can be measured or predicted and the value of p determined. Whenever $p_n > p$ holds at some program point n , it is profitable to apply the transformation. Otherwise, if $p_n \leq p$, the performance of the program may be degraded.

Let us consider again our motivating example with a program run π which enters the loop and 2 times takes the left branch, 1 time the right branch, and finally 6 times the left branch and terminating the loop. Let us further assume that the execution times for term a/b are given as follows: *Recompute* = 110ns, *OrigComp* = 100ns, and *Reuse* = 10ns. Thus we get $p > (110ns - 100ns)/(110ns - 10ns) = 1/10$. In our motivating example, the term a/b occurs in nodes 1 and 3. Since p_n denotes the probability that term a/b is valid at program point n , p_n can be calculated easily by the ratio:

$$p_n = \frac{\text{nr. of times } a/b \text{ is available at } n}{\text{nr. of times } n \text{ occurs in } \pi} \quad (4)$$

Node 1 is executed once and term a/b is never available, thus we get $p_1 = 0/1 = 0$. Since p_1 is not greater than p , a decision to perform the transformation is negative. On the other hand, node 3 is executed 8 times and term a/b is available and reused 7 times which results in $p_3 = 7/8$. Now $p_3 > p$ holds and the transformation should therefore be performed for node 3.

Calculation of the probabilities p_n is crucial to our cost model. An important observation is that the definition of probability p_n in Equation (4) is identical to the definition of probabilistic partially available expressions.

3.3 Probabilistic Partially Available Expression Analysis

Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. Probabilistic data flow systems compute

a range, i.e. a probability, with which a dataflow fact will hold at some program point [Ram96,MS01]. In probabilistic dataflow systems, control flow graphs annotated with edge probabilities are employed to compute the probabilities of dataflow facts. Usually, edge probabilities are determined by means of profile runs based on representative input data sets. These probabilities denote heavily and rarely executed branches and are used to weight dataflow facts when propagating them through the control flow graph.

An expression e is called *partially available* at a program point n , if there is at least one path from the entry node to n containing a computation of e and with no subsequent assignments to any variable used in e on that path. Such a path contains an unnecessary recomputation of e which can be avoided by *partial redundancy elimination* techniques.

Central to our cost model is the definition of p_n . Combining probabilistic data flow analysis with partial availability, we arrive at a suitable definition of p_n : p_n represents the probability that an expression e is available at program point n . If n is reached N times during a program's execution, and e is available on A of those N occasions, then p_n is estimated as A/N .

Our probabilistic data flow framework [MS01] takes as input profiles with edge probabilities of the control flow graphs and the dataflow equations for the dataflow problem. The dataflow equations for partial availability are defined in the usual way, and are shown in Figure 3. Profiling information is easily obtained with our GNU gcc environment by specifying appropriate compiler options. Thus we can obtain estimates for p_n values with minimal programming effort and with little extra compilation time (detailed experiments with SPEC95 have been published in [MS01]).

$$\mathbf{N-PAVAIL}(n) = \begin{cases} \text{false} & \text{if } n = \text{start node} \\ \bigvee_{m \in \text{pred}(n)} \mathbf{X-PAVAIL}(m) & \text{otherwise} \end{cases}$$

$$\mathbf{X-PAVAIL}(n) = \text{LocAvail}(n) \vee \mathbf{N-PAVAIL}(n) \wedge \overline{\text{LocBlock}(n)}$$

where

$\text{LocAvail}_{exp}(n)$ = There is an expression exp which is available at the end of n .

$\text{LocBlock}_{exp}(n)$ = The expression exp is blocked by some instruction of n .

Fig. 3. Partial Availability Analysis

3.4 Refinements of the Basic Transformation

First, the predicate can be omitted from assignments when the expression is known not to be available. More formally, if exp is not partially available at program point n , but on at least one path starting from n there exists a (partially) redundant occurrence of exp , the predicates in front of the assignments can be omitted (cf. Fig. 2, case 1). This simplification has been applied for assignment $s := a/b$ at node 1 in our motivating example.

Second, maintenance of the predicate can be omitted if the predicate is not used. Assume that variable v occurs in term exp and variable v is assigned a new value at program point n . If there is no subsequent use of that new value of v in a predicated assignment of exp on any path emanating from n , the assignment invalidating the reuse of the temporary h_{exp} can be omitted.

4 Related Work

Classical partial redundancy elimination techniques [KRS94] cannot always remove redundant expressions since static analysis approaches are too conservative. In [BGS98] it is reported that the number of dynamically eliminated expressions can be doubled by employing more sophisticated approaches. However, although a simple algorithm [Ste96] can achieve a complete removal of all partial redundancies, the approach causes code growth which is exponential in the number of expressions and is therefore not viable in practice. Speculative approaches [HH97, GBF98] do not restructure the control flow graph and insert additional computations into the control flow graph. They achieve nearly the same optimization results as complete removal. However, a major disadvantage of speculative PRE is that computations with side-effects cannot be handled. In [BGS98] a combination of speculative and code duplication is given. To limit code growth and to select the appropriate PRE technique, profile information is taken into account.

Our approach has the potential to remove all redundancies, though removal is performed only when it would be profitable. Predicates indicate whether the computation is available in temporaries or not. The approach is similar to memoization techniques for functional languages. However, no lookups in a memoization table are required since only the last computation of an expression is stored in a temporary and a predicate controls whether the computation is valid or not.

With our approach, a probabilistic data flow analysis is required to determine whether a transformation is profitable or not. Ramalingam [Ram96] pioneered the field of probabilistic data flow analysis which computes the probability of a dataflow fact. The approach yields an approximate solution which can differ from the accurate solution. In [MS01], that approach was improved by utilizing execution history for estimating the probabilities of the dataflow facts. For calculating the deviations of the probabilistic approaches from the accurate solution, the notion of an abstract run [MS00] was developed. An abstract run accurately calculates the frequencies; however the computational complexity is proportional to the program path length and is thus not feasible in practice. To compute an accurate solution in acceptable time, a novel approach [SM02] based on whole program paths was developed.

5 Summary

We have presented a partial redundancy elimination approach based on probabilistic data flow analysis and predication. Our basic transformation achieves a complete removal of all redundancies. Contrary to other approaches, the control

flow graph is not restructured and the optimization is strictly semantics preserving, i.e. computations with possible side effects are handled correctly. However, predication can cause additional costs. Hence, cost-analysis controls the PRE transformation.

References

- [BGS98] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa, *Complete removal of redundant expressions*, Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI) (Montreal, Canada), 17–19 June 1998, pp. 1–14.
- [BGS99] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa, *Load-reuse analysis: Design and evaluation*, ACM SIGPLAN Notices **34** (1999), no. 5, 64–76.
- [GBF98] Rajiv Gupta, David A. Berson, and Jesse Z. Fang, *Path profile guided partial redundancy elimination using speculation*, Proceedings of the 1998 International Conference on Computer Languages, IEEE Computer Society Press, 1998, pp. 230–239.
- [HH97] R. Nigel Horspool and H. C. Ho, *Partial redundancy elimination driven by a cost-benefit analysis*, Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering (ICSSE'97) (Herzliya, Israel), IEEE Computer Society, June 1997, pp. 111–118.
- [KBC⁺99] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, *A global communication optimization technique based on data-flow analysis and linear algebra*, ACM Transactions on Programming Languages and Systems **21** (1999), no. 6, 1251–1297.
- [KM02] J. Knoop and E. Mehofer, *Distribution assignment placement: Effective optimization of redistribution costs*, IEEE Transactions on Parallel and Distributed Systems **13** (2002), no. 6, 628–647.
- [KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen, *Optimal code motion: Theory and practice*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 4, 1117–1155.
- [MS00] E. Mehofer and B. Scholz, *Probabilistic data flow system with two-edge profiling*. Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), ACM SIGPLAN Notices **35** (2000), no. 7, 65–72.
- [MS01] ———, *A novel probabilistic data flow framework*, International Conference on Compiler Construction (CC 2001) (Genova, Italy), Lecture Notes in Computer Science (LNCS), Vol. 2027, Springer, April 2001, pp. 37–51.
- [Ram96] G. Ramalingam, *Data flow frequency analysis*, Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96) (Philadelphia, Pennsylvania), May 1996, pp. 267–277.
- [SM02] B. Scholz and E. Mehofer, *Dataflow frequency analysis based on whole program paths*, Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT-2002) (Charlottesville, VA), September 2002.
- [Ste96] Bernhard Steffen, *Property oriented expansion*, Proc. Int. Static Analysis Symposium (SAS'96), Aachen (Germany) (Heidelberg, Germany), Lecture Notes in Computer Science (LNCS), vol. 1145, Springer-Verlag, September 1996, pp. 22–41.

SIMD Vectorization of Straight Line FFT Code

Stefan Kral, Franz Franchetti, Juergen Lorenz, and Christoph W. Ueberhuber

Institute for Applied Mathematics and Numerical Analysis,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
`skral@complang.tuwien.ac.at`,
`http://www.math.tuwien.ac.at/ascot`

Abstract. This paper presents compiler technology that targets general purpose microprocessors augmented with SIMD execution units for exploiting data level parallelism. FFT kernels are accelerated by automatically vectorizing blocks of straight line code for processors featuring two-way short vector SIMD extensions like AMD's 3DNow! and Intel's SSE 2. Additionally, a special compiler backend is introduced which is able to (i) utilize particular code properties, (ii) generate optimized address computation, and (iii) apply specialized register allocation and instruction scheduling.

Experiments show that automatic SIMD vectorization can achieve performance that is comparable to the optimal hand-generated code for FFT kernels. The newly developed methods have been integrated into the codelet generator of FFTW and successfully vectorized complicated code like real-to-halfcomplex non-power-of-two FFT kernels. The floating-point performance of FFTW's scalar version has been more than doubled, resulting in the fastest FFT implementation to date.

1 Introduction

Major vendors of general purpose microprocessors have included short vector single instruction multiple data (SIMD) extensions into their instruction set architecture (ISA) to improve the performance of multimedia applications by exploiting data level parallelism. The newly introduced instructions have the potential for outstanding speed-up, but they are difficult to utilize using standard algorithms and general purpose compilers.

Recently, a new software paradigm emerged in which optimized code for numerical computation is generated automatically [5,6]. For example, FFTW has become the de facto standard for high performance FFT computation. However, the current version of FFTW does not utilize short vector SIMD extensions.

This paper presents compiler technology for automatically vectorizing numerical computation blocks as generated by automatic performance tuning systems like FFTW, SPIRAL, and ATLAS. The blocks to be vectorized may contain load and store operations, index computation, as well as arithmetic operations.

Moreover, the paper introduces a special compiler backend which generates assembly code optimized for short vector SIMD hardware. This backend is able

to utilize special features of straight line code, generates optimized address computation, and applies additional performance boosting optimization.

The newly developed methods have been integrated into FFTW's codelet generator yielding FFTW-GEL, a short vector SIMD version of FFTW featuring an outstanding floating-point performance (see Figure 1).

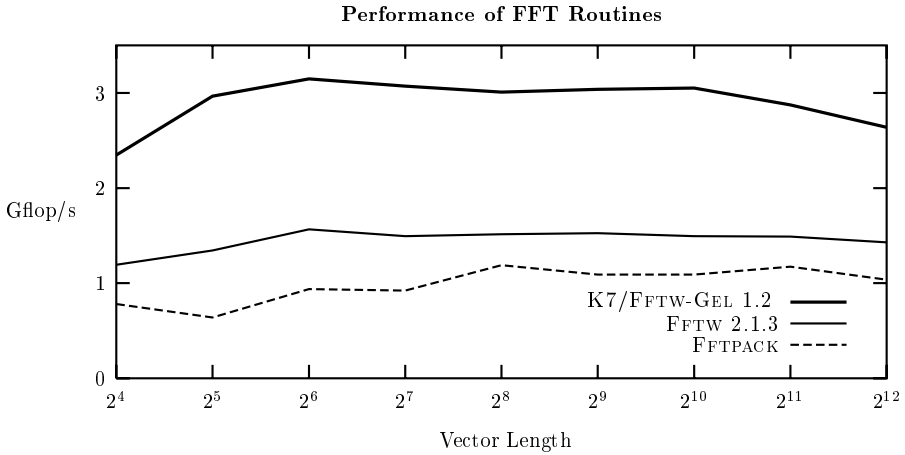


Fig. 1. Floating-point performance of the newly developed K7/FFTW-GEL (3DNow!) compared to FFTPACK and FFTW 2.1.3 on a 1.53 GHz AMD Athlon XP 1800+ carrying out complex-to-complex FFTs in single-precision.

Related Work. Established vectorization techniques mainly focus on loop-constructs satisfying certain constraints. For instance, Intel's C++ compiler and Codeplay's VECTOR C compiler are able to vectorize loop code for both integer and floating-point short vector extensions provided that arrays are accessed in a very specific way. The upgrade [8] to the SUIF compiler vectorizes loop code for the integer short vector extension MMX.

A SPIRAL based approach to portably vectorize discrete linear transforms utilizing structural knowledge is presented in [3].

Intel's math kernel library (MKL) and performance primitives (IPP) both support SSE and SSE 2 available on Pentium III and 4 processors and the Itanium processor family.

Synopsis. Section 2 describes FFTW-GEL, the new short vector SIMD version of FFTW. Section 3 describes the machine independent vectorization of basic blocks and related optimization techniques. Section 4 introduces the newly developed backend optimization techniques. The methods of Sections 3 and 4 are the core techniques used within the codelet generation of FFTW-GEL. Section 5 presents numerical results for the 3DNow! and the SSE 2 version of FFTW-GEL applied to complex-to-complex and real-to-halfcomplex FFTs.

2 FFTW for Short Vector Extensions: FFTW-GEL

FFTW-GEL¹ is an extended version of FFTW that supports two-way short vector SIMD extensions (see Figure 2). In FFTW-GEL the scalar computing cores of FFTW have been replaced by their short vector SIMD counterparts. Thus, the overall structure and all features of FFTW remain unchanged, only the computational speed is increased significantly. Details can be found in [2].

FFTW. The first effort to automatically generate and optimize FFT code using actual run times as an optimization criterion resulted in FFTW [4]. In many cases FFTW runs faster than other publicly available FFT codes like the industry standard FFTPACK².

FFTW is based on a recursive implementation of the Cooley-Tukey FFT algorithm and uses dynamic programming with measured run times as cost function to find a fast implementation for a given problem size on a given machine. FFTW consists of four fundamental parts: (i) the planner, (ii) the executor, (iii) the codelet generator **genfft**, and (iv) codelets.

Planner and executor provide for the adaptation of the FFT computation to the target machine at runtime while the actual computation is done within routines called codelets. Codelets are generated by the codelet generator **genfft**, a special purpose compiler.

Within the codelets a variety of FFT algorithms [9] is used, including the Cooley-Tukey algorithm, the split radix algorithm, the prime factor algorithm, and the Rader algorithm.

The methods presented in this paper were used to extend FFTW's codelet generator to produce vectorized codelets. Due to the structure of FFT algorithms, codelets feature a certain degree of parallelism allowing for the application of the new vectorization technique but preventing the application of standard methods.

These vectorized codelets are compatible with FFTW's framework and thus can be used instead of the original codelets on machines featuring two-way short vector SIMD extensions. That way FFTW's core routines are sped up while all features of standard FFTW remain supported.

Short Vector SIMD Extensions. Examples of two-way short vector SIMD extensions supporting both integer and floating-point operations include Intel's streaming SIMD extensions SSE 2 and AMD's 3DNow! family.

Double-precision short vector SIMD extensions paved the way to high performance scientific computing. However, special code is needed as conventional scalar code running on machines featuring these extensions utilizes only a small fraction of the potential performance.

Short vector SIMD extensions are advanced architectural features which are not easily utilized for producing high performance codes. Currently, two approaches are commonly used to utilize SIMD instructions.

¹ available from <http://www.fftw.org/~skral>

² available from <http://www.netlib.org/fftpack>

Vectorizing Compilers. The application of compiler vectorization is restricted to loop-level parallelism and requires special loop structures and data alignment. Whenever the compiler cannot prove that a loop can be vectorized optimally using short vector SIMD instructions, it has to resort to either emitting scalar code or to introducing additional code to handle special cases.

Hand-Coding. Compiler vendors provide extensions to the C language (data types and intrinsic or built-in interfaces) to enable the direct use of short vector SIMD extensions from within C programs. Of course, assembly level hand-coding is unavoidable if there is no compiler support.

FFTW-GEL. Both of the latter approaches are not directly utilizable to generate vectorized FFTW codelets that may act as a replacement for the respective standard codelets. Due to its internal structure, FFTW cannot be vectorized using compilers focusing on loop vectorization: (i) arrays are accessed at strides that are determined at runtime and may be not unit stride, preventing loop level vectorization, and (ii) a large class of codelets does not even contain loops. FFTW’s automatically generates code featuring large basic blocks (up to thousands of lines) of numerical straight line code. Accordingly, such basic blocks consisting of straight line code have to be vectorized rather than loops.

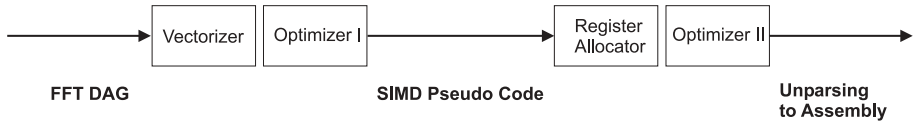


Fig. 2. The basic structure of FFTW-GEL. FFT DAGs generated by the codelet generator `genfft` are vectorized and subjected to advanced optimization.

Vectorization of Straight Line Code. Within the context of FFTW, codelets do the bulk of the computation. As codelets feature a function body consisting sometimes of thousands of lines of automatically generated straight line code without loops, the vectorization of straight line code is an indispensable requirement in this context. The challenge addressed by vectorizing straight line code is to extract parallelism out of this sequence of operations while maintaining data locality and utilizing special features of the target architecture’s short vector SIMD extensions.

Straight Line Code Assembly Backend. The straight line code of FFTW’s codelets features several characteristics that can be exploited by special backend optimization: (i) All input array elements of a codelet are loaded into a temporary variable exactly once, (ii) each output array element is stored exactly once, and (iii) all index computations are linear (base address + index $\times$ stride).

The main goal in optimized index computation is to avoid expensive integer multiplication by computing the required multiples of the stride by clever utilization of integer register contents and implicit operations supported by the

target instruction set (e. g., load/store instructions involving some effective address calculation or the “load effective address” instruction in the x86 ISA). The optimal cache utilization algorithm is used for register allocation. Furthermore, straight line code specific optimization techniques are applied.

3 Two-Way SIMD Vectorization of Straight Line Code

An important core technology used in FFTW-GEL is the two-way SIMD vectorization of numerical straight line code. This process extracts vector instructions out of the stream of scalar floating-point operations by joining scalar operations together. These artificially assembled vector instructions allow to completely vectorize rather complex codes. However, some of these instructions may not be contained in the target architecture’s instruction set and thus have to be emulated by a sequence of available vector instructions.

The vectorization engine performs a non-deterministic matching against a directed acyclic graph (DAG) of scalar floating-point instructions to automatically extract two-way SIMD parallelism out of a basic block.

Additional optimization stages are required to translate SIMD pseudo-instructions into target instructions and to produce a topologically sorted sequence of SIMD instructions. This sequence is subsequently fed into the back-end described in the following section for further optimization and unparsing to assembly language.

3.1 The Vectorization Engine

The goal of the vectorization process is to replace all scalar floating-point instructions by SIMD instructions. To achieve this goal, pairings of scalar floating-point instructions (and their respective operands) are to be found yielding in each case one SIMD floating-point instruction. In some cases, additional SIMD instructions like swaps may be required. A given pair of scalar floating-point instructions can only be fused if it can be implemented by the available SIMD instructions without changing the semantics of the DAG.

The vectorization algorithm works as follows. (1) Pick two scalar instructions with minimal depth whose output variables are already paired. (2) Combine these and pair their input variables. A set of rules defines feasible pairs of scalar instructions and the resulting pairings of scalar variables for each alternative SIMD implementation. It has to be asserted globally that every scalar variable occurs in at most one pairing. Alternatives are saved on a stack.

Initially no scalar instructions are paired. Steps (1) and (2) are iterated until either all scalar instructions are vectorized, or the search process fails. Upon failure, the last alternative saved on the stack is chosen, resuming the search at that point. This implements a depth-first search with chronological backtracking.

The pairing ruleset is chosen to enable a large enough number of alternative SIMD implementations for pairs of scalar operations to enable vectorization while keeping the search space reasonably small.

The benefits of the vectorization process are the following: *(i)* Scalar floating-point instructions are replaced by SIMD instructions. Thus, the instruction count is nearly halved. *(ii)* The fusion of memory access operations potentially reduces the effort for calculating effective addresses. *(iii)* The wider SIMD register file allows for more efficient computation.

3.2 Vectorization Specific Optimization

After finishing the vectorization stage, further optimization is performed in a separate stage. This procedure keeps the implementation of the vectorizer simple and saves time needed for the vectorization process by postponing additional optimization until the backtracking search has succeeded.

The pseudo SIMD instructions generated by the vectorization engine are directly fed into the optimization stage. A rewriting system is used to simplify groups of target-architecture independent instructions as well as target-architecture specific combinations of instructions.

A first group of rewriting rules is applied to *(i)* minimize the number of instructions, *(ii)* eliminate redundancies, *(iii)* reduce the number of source operands, *(iv)* eliminate dead code, and *(v)* perform constant folding. Some rules shorten the critical path lengths of the data dependency graphs by exploiting specific properties of the target instruction set. Finally, some rules reduce the number of source operands necessary to perform an operation, thus effectively reducing register pressure.

A second group of rules is used to rewrite pseudo instructions into combinations of instructions actually supported by the target architecture.

In a third and last step the optimization process topologically sorts the DAG that will be translated subsequently. The respective scheduling algorithm was designed to minimize the lifetime of variables in the scheduled code by improving the locality of variable accesses. The scheduling process of FFTW-GEL extends the scheduler of FFTW 2.1.3 by additional postprocessing to further improve locality of variable accesses in some cases.

4 Backend Optimization for Straight Line Code

The backend translates a scheduled sequence of SIMD instructions into assembly language. It performs optimization in both an instruction set specific and a processor specific way. In particular, register allocation and the computation of effective addresses is optimized with respect to the target instruction set. Instruction scheduling and avoidance of address generation interlocks is done specifically for the target processor.

4.1 Instruction Set Specific Optimization

Instruction set specific optimization takes into account properties of the target microprocessor's instruction set.

Depending on whether a RISC or a CISC processor is used, additional constraints concerning source and target registers have to be satisfied. For instance, many CISC style instruction sets (like Intel's x86) require that one source register must be used as a destination register.

The utilization of memory operands, as provided by many CISC instruction sets, results in locally reduced register pressure. It decreases the number of instructions as two separate instructions (one load- and one use-instruction) can be merged into one (load-and-use). The code generator of FFTW-GEL uses this kind of instructions whenever some register content is used only once.

Many machines include complex instructions that are combinations of several simple instructions like a "shift by some constant and add" instruction. The quality of the code that calculates effective addresses can often be improved by utilizing this kind of instructions.

Register Allocation. FFTW codelets are (potentially very large) sequences of straight line C code. Knowing the complete code structure and data dependencies, it is possible to perform register allocation according to Belady's optimal caching algorithm [1].

Experiments have shown that this policy is superior to the ones used in mainstream C compilers (like the GNU C compiler, Compaq's C compiler for Alpha, and Intel's C compiler) for this particular type of code.

Efficient Calculation of Effective Addresses. FFTW codelets operate on arrays of input and output data. Both input and output arrays may not be stored with unit stride in memory. Thus access to element `in[i]` may result in a memory access at address `in + i*sizeof(in)*stride`. Both `in` and `stride` are parameters passed from the calling function. For a codelet of size N , all elements `in[i]` with $0 \leq i < N$ are accessed exactly once. As a consequence, the quality of code dealing with variable array strides is crucial for achieving high performance. In particular, all integer instructions apart from parameter passing occurring in FFTW codelets are used for calculating effective addresses.

To achieve an efficient computation of effective addresses, FFTW-GEL follows the following guidelines: (i) General multiplications are avoided. While it is tempting to use an integer multiplication instruction (`imul`) for general integer multiplication in effective address calculation, it is usually better to use equivalent sequences of simpler instructions (addition, subtraction or shift operations) instead. Typically, these instructions can be decoded faster, have shorter latencies, and are all fully pipelined. (ii) The load effective address instruction (`lea`) is a cheap alternative to integer multiplication on x86 compatible processors. It is used for combining up to two adds and one shift operation into a single instruction. (iii) Only integer multiplications are rewritten whose result can be obtained using less than n other operations, where n depends on the latency of the integer multiplication instruction on the target architecture.

In FFTW-GEL, the actual generation of code sequences for calculating effective addresses has been intertwined with the register allocation process. Whenever code for the calculation of an effective address is to be generated, the (locally) shortest possible sequence of instructions is chosen (determined by depth-

first iterative deepening), trying to reuse effective addresses that have already been calculated and that are still stored in some integer register.

4.2 Processor Specific Optimization

Typically, processor families supporting one and the same instruction set have different instruction latencies and throughput. Processor specific optimization has to take into account the execution properties of a *specific* processor. The code generator of FFTW-GEL uses two processor specific optimizations: (i) Instruction scheduling, and (ii) avoidance of address generation interlocks.

Instruction Scheduling. The instruction scheduler of FFTW-GEL tries to find an ordering of instructions optimized for the target processor. During the scheduling process it uses information about critical-path lengths of the underlying data dependency graph in a heuristic way. The basic principles of the instruction scheduler are illustrated in [7].

The instruction scheduler is extended by a relatively simple processor model that is used for estimating whether a sequence of instructions can be issued within the same cycle and executed in parallel by means of super-scalar execution. By using this prediction, the scheduler tries to resolve tie situations arising in the scheduling process.

In addition, the instruction scheduler serves as runtime estimator and is used for controlling optimizations that could either increase or decrease performance.

Avoiding Address Generation Interlocks. Although many modern super-scalar microprocessors allow an out-of-order execution, most architecture definitions require that memory accessing instructions must be executed in-order.

An address generation interlock (AGI) occurs whenever a memory operation that involves some expensive effective address calculation directly precedes a memory operation that involves inexpensive address calculation or no address calculation at all. Then the second access is stalled by the computation of the first access' effective address.

The backend tries to avoid AGIs by reordering instructions immediately *after* instruction scheduling. This is done in a separate stage to simplify the design of the instruction scheduler.

5 Experimental Results

Numerical experiments were carried out to demonstrate the applicability and the performance boosting effects of the newly developed techniques.

FFTW-GEL was investigated on two IA-32 compatible machines: (i) An Intel Pentium 4 running at 1.8 GHz and featuring two-way double-precision SIMD extensions (SSE 2), as well as (ii) an AMD Athlon XP 1800+ running at 1.53 GHz and featuring two-way single-precision SIMD extensions (3DNow! professional).

Performance data are displayed in pseudo-flop/s, i. e., $5N \log N/T$ for complex-to-complex and $2.5N \log N/T$ for real-to-halfcomplex FFTs. This unit of

measurement is a scaled inverse of the run time T and thus preserves run time relations and gives an indication of the absolute floating-point performance [5].

The speed-up achievable by vectorization (ignoring other effects like smaller code size, wider register files, etc.) is limited by a factor of two on both test machines. However, the additional backend optimization further improves the performance of the generated codes. All codes were generated using the GNU C compiler 2.95.2 and the GNU assembler 2.9.5. Both complex-to-complex FFTs for power-of-two problem sizes and real-to-halfcomplex FFTs for non-powers of two were evaluated. Experiments show that neither FFTW's executor nor the codelets are vectorized by Intel's C++ compiler. Thus, the newly developed vectorizer of FFTW-GEL is more advantageous than traditional compiler vectorization by a big margin.

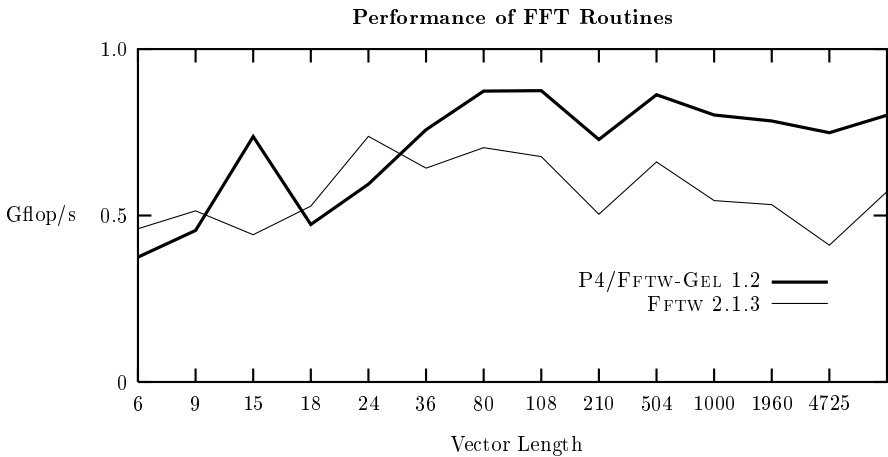


Fig. 3. Floating-point performance of the newly developed P4/FFTW-GEL (SSE 2) compared to FFTW (double-precision) on an Intel Pentium4 running at 1.8 GHz for real-to-halfcomplex FFTs.

AMD 3DNow! Figure 1 (on page 252) shows the performance of FFTPACK, FFTW 2.1.3, and K7/FFTW-GEL on the Athlon XP in single-precision. The run-times displayed refer to powers of two complex-to-complex FFTs whose data sets fit into L2 cache. K7/FFTW-GEL utilizes enhanced 3DNow! which provides two-way single-precision SIMD extensions. FFTW is up to two times faster than FFTPACK, the industry standard in non-hardware-adaptive FFT software.

FFTW-GEL is about twice as fast as FFTW which demonstrates that the performance boosting effect of vectorization and backend optimization is outstanding.

Intel SSE2. Figure 3 shows the performance of FFTW 2.1.3 and P4/FFTW-GEL on the Pentium4 in double-precision arithmetic. Runtimes were measured for non-powers of two with real-to-halfcomplex FFTs whose data sets fit into L2

cache. P4/FFTW-GEL utilizes the two-way double-precision SIMD operations of SSE 2. For most problem sizes FFTW-GEL is faster than FFTW 2.1.3 (up to 60 %). Note that real-to-halfcomplex FFTs are notoriously hard to vectorize (especially for non-powers of two) due to the much more complicated algorithmic structure compared to complex-to-complex FFT algorithms.

Conclusion

This paper presents a set of compilation techniques for automatically vectorizing numerical straight line code. As straight line code is in the center of all current numerical performance tuning software, the newly developed techniques are of particular importance in scientific computing.

Impressive performance results demonstrate the usefulness of the newly developed techniques which can even vectorize the complicated code of real-to-halfcomplex FFTs for non-powers of two.

Acknowledgement. We would like to thank Matteo Frigo and Steven Johnson for many years of fruitful cooperation and for making it possible for us to access non-public versions of FFTW.

Additionally, we would like to acknowledge the financial support of the Austrian science fund FWF.

References

1. Belady, L.A.: A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal* **5** (1966), 78–101
2. Franchetti F., Kral, S., Lorenz, J., Ueberhuber, C.: AURORA Report Series on SIMD Vectorization Techniques for Straight Line Code. Technical Report AURORA TR2003-01, TR2003-10, TR2003-11, TR2003-12, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology (2003)
3. Franchetti, F., Püschel, M.: A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms In: *Proc. International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Comp. Society Press, Los Alamitos (2002), 20–26
4. Frigo, M.: A Fast Fourier Transform Compiler. In: *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. ACM Press, New York (1999), 169–180
5. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: *ICASSP 98. Volume 3*. (1998) 1381–1384
6. Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V., Püschel, M., Veloso, M.M.: SPIRAL: Portable Library of Optimized Signal Processing Algorithms (1998)
7. Muchnick, S.S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco (1997)
8. Sreraman, N., Govindarajan, R.: A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming* **28** (2000) 363–400
9. Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia (1992)

Branch Elimination via Multi-variable Condition Merging^{*}

William Krehling¹, David Whalley¹, Mark Bailey², Xin Yuan¹,
Gang-Ryung Uh³, and Robert van Engelen¹

¹ Florida State University, Tallahassee FL 32306, USA,
{krehling, whalley, yuan, engelen}@cs.fsu.edu

² Hamilton College, Clinton NY 13323, USA,
mbailey@hamilton.edu

³ Boise State University, Boise ID 83725, USA,
uh@cs.boisestate.edu

Abstract. Conditional branches are expensive. Branches require a significant percentage of execution cycles since they occur frequently and cause pipeline flushes when mispredicted. In addition, branches result in forks in the control flow, which can prevent other code-improving transformations from being applied. In this paper we describe profile-based techniques for replacing the execution of a set of two or more branches with a single branch on a conventional scalar processor. First, we gather profile information to detect the frequently executed paths in a program. Second, we detect sets of conditions in frequently executed paths that can be merged into a single condition. Third, we estimate the benefit of merging each set of conditions. Finally, we restructure the control flow to merge the sets that are deemed beneficial. The results show that eliminating branches by merging conditions can significantly reduce the number of conditional branches performed in non-numerical applications.

1 Introduction

Conditional branches occur frequently in programs, particularly in non-numerical applications. Branches are an impediment to improving performance since they consume a significant percentage of execution cycles, cause pipeline flushes when mispredicted, and can inhibit the application of other code-improving transformations. Techniques to reduce or eliminate the number of executed branches in the control flow have the potential for significantly improving performance.

Sometimes a set of conditions can be merged together. Consider Fig. 1(a), which shows conditions being tested in basic blocks 1 and 3. The wider transitions between blocks shown in figures in this paper represent the more frequently executed path, which occurs in Fig. 1 when conditions *a* and *b* are both satisfied. Figure 1(b) depicts the two conditions being merged together. If the merged

^{*} This research was supported in part by NSF grants CCR-9904943, EIA-0072043, CCR-0208892, CCR-0105422, and by DOE grant DEFG02-02ER25543.

condition is true, then the original conditions need not be tested. Note merging conditions results in the elimination of both comparison and branch instructions.¹ The elimination of the fork in the control flow between blocks 2 and 4 may enable additional code-improving transformations to be performed. If the merged condition is not satisfied, then the original conditions are tested. Figure 1(c) shows that branches can become redundant after merging conditions. In this case, condition *b* must be false if $(a \ \&\& \ b)$ is false and (a) is true. Thus, the branch in block 3 can be replaced by an unconditional transition to block 6. We call this the *breakeven* path since the same number of branches will be executed. We only apply the condition merging transformation when we estimate that the total instructions executed will be decreased.

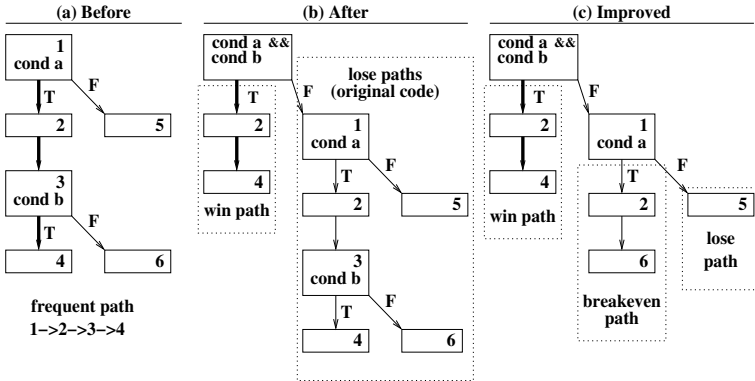


Fig. 1. Merging Three Conditions

In this paper we describe techniques to replace the execution of a set of two or more branches with a single branch. The performance improvements were obtained automatically by our compiler on a conventional scalar processor.

2 Related Work

There are numerous techniques that have been used to decrease the number of conditional branches executed. Loop unrolling has long been used to reduce execution of the conditional branch associated with a loop termination condition[1]. Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and duplicates the loop in each of the two destinations of the

¹ Blocks 2 and 3 in Fig. 1(a) could have been represented as a single block. Throughout the paper we represent basic blocks containing a condition as having no other instructions besides a comparison and a conditional branch so the examples may be more easily illustrated.

branch[2]. Superoptimizers have been used to find a bounded sequence of instructions that have the same effect as a conditional branch[3]. Conditional branches have been avoided by using static analysis and code duplication[4,5]. Conditional branches have been coalesced together into an indirect jump from a jump table[6]. Sequences of branches have been reordered to allow the sequence to be exited earlier, which reduces the number of branches executed[7,8]. There has been recent work on eliminating branches using ILP architectural features. If conversion uses predicated execution to eliminate branches by squashing the result of an instruction when a predicate is not satisfied[9]. Another technique eliminates branches by performing if-conversion as if the machine supported predicated execution, and then applies reverse-if conversion to return the code to an acyclic control flow representation[10]. This technique shares some similarities to ours, however our approach differs in several aspects, including that we can merge conditions involving multiple variables. A technique called control CPR (Critical Path Reduction) has been used to merge branches in frequent paths by performing comparisons that set predicate bits in parallel and testing multiple predicates in a single bypass branch[11,12]. Control CPR not only reduces the number of executed branches, but also enables other code-improving transformations. Among the techniques mentioned, control CPR is the most similar to the work that we describe in this paper, since both approaches merge conditions by using path-based profile data. There are several difference in the two approaches, including that our approach can be used on a conventional scalar processor without any ILP architectural support.

3 Merging Conditions

We have developed techniques that merge conditions where either single variables or multiple variables are compared to invariant values.

3.1 Merging into a Single Equivalent Condition

We use logical operations to efficiently merge conditions that compare multiple variables to 0 or -1 into a single equivalent condition. Consider the flow graph shown in Fig. 2(a). The frequent path checks if the two variables are equal to zero. Figure 2(b) depicts the two conditions being merged together using a bitwise OR operation. Only when both variables are equal to zero will the result of the OR operation be zero. If the merged condition is not satisfied, then testing the condition $v2 == 0$ is unnecessary since it cannot be true.

The number of instructions required to merge conditions involving multiple variables is proportional to the number of different variables being compared. Figure 3 shows the general code generation strategy we used for merging such a set of conditions. $r[1] \dots r[n]$ represent the registers containing the values of n different variables. $r[t]$ represents a register containing the temporary results. When merging conditions comparing n multiple variables, $n - 1$ logical operations and a single branch replace $2n$ instructions (n pairs of comparisons and branches).

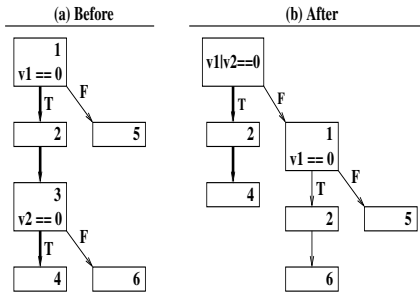


Fig. 2. Merging Conditions That Test If Different Variables Are Equal to Zero

```

r[t] = r[1] | r[2];
r[t] = r[t] | r[3];
...
IC = (r[t] | r[n]) ? 0;
PC = IC != 0, <off-trace target>;

```

Fig. 3. Code Generated for the Merged Condition That Checks If n Variables Are Equal to Zero

The top portion of table 1 shows how sets of conditions comparing multiple variables ($v1 \dots vn$) to 0 and -1 can be merged into a single condition. The first column shows the original conditions, the second column shows the conditions after merging, and the third column shows the static percentage each rule was applied, when merging sets involving multiple variables. Rule 1 has been illustrated in Figures 2 and 3. Rule 2 uses the SPARC ORNOT instruction to perform a bitwise NOT on an operand before performing a bitwise OR operation. A word containing the value -1 has all of its bits set in a two's complement representation. Thus, if the operand was a -1 , then the result of the bitwise NOT would be 0 and at that point the first rule can be used. The merged condition in rule 3 performs a bitwise AND operation on the variables. A negative value in a two's complement representation has its most significant bit set. Only if the most significant bit is set in all of the variables will the most significant bit be set in the result of the bitwise AND operation. If the most significant bit is set in the result, then the result value will be negative. The merged condition in rule 4 performs a bitwise OR operation on the variables. A nonnegative value in a two's complement representation has its most significant bit clear. Only if the most significant bit is clear in all the variables will the most significant bit be clear in the result of the bitwise OR operation. Rules 5 and 6 perform a bitwise NOT on an operand, which flips the most significant bit along with the other bits in the value. This allows $<$ and $>=$ tests to be merged together.

Notice that > 0 and $<= 0$ tests are not listed in the table. A > 0 test would have to determine that both the most significant bit is clear and that one or more of the other bits are set. These types of tests cannot be efficiently performed using a single logical operation on a conventional scalar processor.

Additional opportunities for condition merging become available when sets of conditions that cross loop boundaries are considered. It would appear that in Fig. 4(a) there is no opportunity for merging conditions. However, Fig. 4(b) depicts that after loop unrolling there are multiple branches that use the same relational operator. Our compiler merges sets of conditions across loop iterations.

Table 1. Rules for Merging Conditions Comparing Multiple Variables Into a Single Equivalent Condition

Rule	Original Conditions	Merged Condition	% Applied
1	$v1 == 0 \ \&\&..\&\& \ vn == 0$	$(v1 \mid .. \mid vn) == 0$	42.7%
2	$v1 == 0 \ \&\&..\&\& \ vn == -1$	$(v1 \mid .. \mid \sim vn) == 0$	0.0%
3	$v1 < 0 \ \&\&..\&\& \ vn < 0$	$(v1 \ \&..\& \ vn) < 0$	0.0%
4	$v1 >= 0 \ \&\&..\&\& \ vn >= 0$	$(v1 \mid .. \mid vn) >= 0$	4.5%
5	$v1 < 0 \ \&\&..\&\& \ vn >= 0$	$(v1 \ \&..\& \ \sim vn) < 0$	0.9%
6	$v1 >= 0 \ \&\&..\&\& \ vn < 0$	$(v1 \mid .. \mid \sim vn) >= -1$	0.0%
7	$v1 \neq 0 \ \&\&..\&\& \ vn \neq 0$	$(v1 \ \&..\& \ vn) \neq 0$	18.2%
8	$v1 \neq c1 \ \&\&..\&\& \ vn \neq cn$	$(v1 \ \&..\& \ vn) \ \& \ \sim (c1 \mid .. \mid cn) \neq 0$	15.5%
9	$v1 \neq c1 \ \&\&..\&\& \ vn \neq cn$	$\sim (v1 \mid .. \mid vn) \ \& \ (c1 \mid .. \mid cn) \neq 0$	16.4%
10	$v1 < c1 \ \&\&..\&\& \ vn < cn$	$(v1 \mid .. \mid vn) \ u < \min(c1 \mid .. \mid cn)$	1.8%

(a) Original Code	(b) After Loop Unrolling
<pre>for (i=0; i < 10000; i++) if (a[i] < 0) x;</pre>	<pre>for (i=0; i < 10000; i += 2){ if (a[i] < 0) x; if (a[i+1] < 0) x; }</pre>

Fig. 4. Increasing Merging Opportunities by Unrolling Loops

3.2 Merging into a Sufficient Condition

Our compiler uses logical operations to efficiently merge conditions that compare multiple variables into a single sufficient condition. In other words, the success of the merged condition implies the success of the original conditions. However, the failure of the merged condition does not imply the original conditions were false.

Tests to determine if a variable is not equal to zero occur frequently in programs. We can determine if two or more variables are guaranteed to be not equal to zero by performing a bitwise AND operation on the variables and checking if the result does not equal to zero, as shown in rule 7 of Table 1.

One may ask how often such conditions can be successfully merged in practice. Consider the code in Fig. 5(a), where two pointer variables, $p1$ and $p2$, are tested to see if they are both non-null. In most applications, a pointer variable is only null in an exceptional case (e.g. end of a linked list). It is extremely likely that two non-null pointer values will have one or more corresponding bits both set due to address locality. Figure 5(b) shows code with a merged condition, testing the pointers. Note that if the merged condition is not satisfied, then the entire original set of branches still needs to be tested.

We are also able to merge conditions that check if multiple variables are all not equal to a specified list of constants. One method we used is to check if

(a) Before	(b) After
<code>if (p1 != 0)</code>	<code>if (p1 & p2 != 0){</code>
<code>w;</code>	<code>w;</code>
<code>else</code>	<code>y;</code>
<code>x;</code>	<code>}else{</code>
<code>if (p2 != 0)</code>	<code>if (p1 != 0)</code>
<code>y;</code>	<code>...</code>
<code>else</code>	<code>if (p2 != 0)</code>
<code>z;</code>	<code>...</code>
	<code>}</code>

Fig. 5. Checking If Different Variables Are Not Equal to Zero

any bits set in all of the variables are always clear in all of the constants. Rule 8 of Table 1 depicts how this is accomplished, where $c1 \dots cn$ are constants. A bitwise AND operation is performed on all of the variables to determine which bits are set in all of these variables. The complement of the bitwise OR of the constants is taken, which results in the bits being set that are clear in all of the constants. Note that determining which bits in the constants are always clear is determined at compile time. If any bits set in all of the variables are clear in all of the constants, then it is known that all of the variables will not be equal to all of the constants.

We are also able to merge conditions checking if multiple variables are less than (or less than or equal to) constants. Rule 10 in Table 1 depicts how this is accomplished, where $c1 \dots cn$ have to be positive constants and the $u <$ in the merged condition represents an unsigned less than comparison. If the result of the bitwise OR on the variables is less than the minimum constant, then the original conditions have to be satisfied. The unsigned less than operator is necessary since one of the variables could be negative and the result of the bitwise OR operation would be treated as a negative value if a signed less than operator is used.

Our compiler merges conditions comparing multiple variables to values that are not constants. Consider the original loop and unrolled loop shown in Fig. 6(a) and Fig. 6(b), respectively. It would appear there is no opportunity for merging conditions. However, x is loop invariant. Thus, the bits that are set in both $a[i]$ and $a[i + 1]$ can be ANDed with $\sim x$ to determine if the array elements are not equal to x , as shown in Fig. 6(c). The expression $\sim x$ is loop invariant and the compiler will move it out of the loop when loop-invariant code motion is performed. Likewise, the loads of the two array elements in the *else* case will be eliminated after applying common subexpression elimination. Note we are also able to merge conditions checking if multiple variables are not equal to multiple loop invariant values.

(a) Original Code	(c) After Condition Merging
<pre> for (i=0; i < 10000; i++) if (a[i] == x) num++; </pre>	<pre> for (i=0; i < 10000; i += 2){ if ((a[i] & a[i+1]) & ~x) continue; else{ if (a[i] == x) num++; if (a[i+1] == x) num++; } } </pre>
(b) After Loop Unrolling	
<pre> for (i=0; i < 10000; i += 2){ if (a[i] == x) num++; if (a[i+1] == x) num++; } </pre>	

Fig. 6. Merging Conditions That Check If Multiple Variables Are Not Equal to Loop-Invariant Values

4 Applying the Transformation

After finding all of the sets of conditions that can be merged, the compiler sorts these sets in the order of the highest estimated benefit first. There are two reasons for merging the most beneficial sets first. (1) Merging may require the generation of loop-invariant expressions that can be moved out of the loop after applying loop-invariant code motion. This transformation requires the allocation of registers for which there are only a limited number available on a target machine. (2) Merging conditions changes the paths of blocks that connect them, then the estimated benefit is invalid after the first set is merged and the second set of conditions will not be merged.

After merging sets of conditions, we apply a number of code-improving transformations in an attempt to improve the modified control flow. For instance, Fig. 1(c) shows that merging conditions simplifies the control flow in the *win* and *breakeven* paths. Our general strategy was to generate the control flow in a simple manner and rely on other code-improving transformations to improve the code. For instance, the code shown in Fig. 6(c) can be improved by applying loop-invariant code motion and common subexpression elimination.

5 Results

Condition merging on multiple variables reduced the average number of branches executed by 11.54% and reduced the total number of instructions executed by an average of 3.93%. Figure 7 displays the results of condition merging when the techniques that merged conditions involving single variables were combined with the techniques involving multiple variable conditions. All the benchmarks were run on the UltraSPARC II. When merging multiple variable conditions, the branches executed were reduced by an average of 11.54%. For space considerations the techniques that merge single variable conditions are not shown

here but are discussed, in detail, in a technical report[13]. The average number of branches executed when using both techniques was reduced by 15.81% while the average number of instructions executed was reduced by 5.74%. Execution time was reduced an average of 3.43%. While most of the reduction was directly due to fewer executed comparisons and branches, occasionally the compiler was able to apply code-improving transformations on the modified control flow to obtain additional improvements. *Sort* had unusually large benefits since most of the execution was spent in tight loops where conditions could be merged. The execution time for a few of the benchmarks got worse after the condition merging techniques were applied. The increase in execution time could be caused by several factors that could include inaccurate performance estimates due to subsequent optimizations and unforeseen effects due to multiple issue, instruction caching and branch prediction.

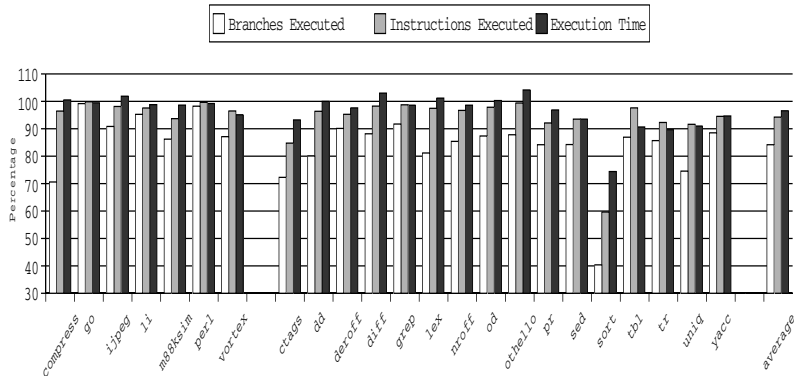


Fig. 7. Effect on Branches, Instructions, and Execution Time on the UltraSPARC II

Applying all the techniques resulted in a 6.99% increase in static code size after condition merging. This static increase compares favorably with the 5.74% decrease in instructions executed. The code size would have increased more if we had not required that the paths we inspected comprise at least 0.1% of the total instructions executed.

On many machines there is a pipeline stall associated with every taken branch. Merging conditions resulted in an average 27.90% reduction in the number of taken branches for the test programs. This reduction was due in part to decreasing the number of executed branches. The transformation also makes branches within the *win* path more likely to fall through since the sequence of blocks representing each frequent path is now contiguous in memory. Overall, the average number of control transfers (taken branches, unconditional jumps, calls, returns, and indirect jumps) was reduced by 20.18%.

6 Future Work

One area to explore is the use of more aggressive analysis to detect when speculatively executed loads would not introduce exceptions. We conservatively merged sets of conditions that required loads to be speculatively executed only when it was obvious that these loads would not introduce new exceptions. By performing more aggressive analysis, we should be able to detect more sets of conditions to be merged. This will be particularly useful for merging conditions that cross loop boundaries.

Past studies on control CPR have not investigated the impact on branch prediction[11,12]. A more thorough investigation of the effect condition merging has on branch misprediction is needed to discover the reasons why additional mispredictions sometimes occur. It may be possible to perform additional optimizations that may reduce the number of mispredictions.

We also found that the merging of one set of conditions may inhibit the merging of another set. Code is duplicated and the paths within a function are modified when conditions are merged. This code duplication invalidates the path profile data on which condition merging is based. Thus, merging a set of conditions is not currently allowed whenever the control flow changed in the subpaths associated with the set of conditions to be merged. With careful analysis it may be possible to infer new path frequency measurements for these duplicated paths.

7 Conclusions

In this paper we described a technique to perform condition merging on a conventional scalar processor. We replaced the execution of two or more branches with a single branch by merging conditions. Path profile information is gathered to determine the frequency that paths are executed in the program. Sets of conditions that can be merged are detected and the benefit of merging each set is estimated. The control flow is then restructured to merge the sets of conditions deemed beneficial. The results show that significant reductions can be achieved in the number of branches performed for non-numerical applications. In addition, we showed benefits in the number of instructions executed, and execution time.

There contributions presented in this paper are twofold. First, we were able to merge conditions comparing multiple variables to constants or invariant values through innovative use of instructions on a conventional scalar processor. Second, we have shown that these techniques may still provide benefits even when the *win* path is not the most frequently executed. In these cases it may be possible to generate and merge conditions along a *breakeven* path.

We believe that condition merging may be useful in other settings. Condition merging may be a very good fit for run-time optimization systems, which optimize frequently executed paths during the execution of a program. Condition merging may also be useful for low power embedded systems processors where architectural support for ILP is not available.

References

1. Dongarra, J.J., Hinds, A.R.: Unrolling loops in FORTRAN. *Software, Practice and Experience* **9** (1979) 219–226
2. Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In Rustin, R., ed.: *Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, NJ, USA (1971) 1–30. Transformations.
3. Granlund, T., Kenner, R.: Eliminating branches using a superoptimizer and the GNU C compiler. In Fraser, C.W., ed.: *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, ACM Press (1992) 341–352
4. Mueller, F., Whalley, D.B.: Avoiding conditional branches by code replication. In: *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, ACM Press (1995) 56–66
5. Bodík, R., Gupta, R., Soffa, M.L.: Interprocedural conditional branch elimination. In: *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. Volume 32, 5 of ACM SIGPLAN Notices, New York, ACM Press (1997) 146–158
6. Uh, G.R., Whalley, D.B.: Coalescing conditional branches into efficient indirect jumps. In: *Proceedings of the International Static Analysis Symposium*. (1997) 315–329
7. Yang, M., Uh, G.R., Whalley, D.B.: Improving performance by branch reordering. In: *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, ACM Press (1998) 130–141
8. Yang, M., Uh, G.R., Whalley, D.B.: Efficient and effective branch reordering using profile data. Volume 24. (2002) 667–697
9. Hennessy, J., Patterson, D.: *Computer Architecture: A Quantitative Approach*. Second edn. Morgan Kaufmann Publishers Inc., San Francisco, CA (1996)
10. Warter, N.J., Mahlke, S.A., Hwu, W.M.W., Rau, B.R.: Reverse If-Conversion. In: *Proceedings of the Conference on Programming Language Design and Implementation*. (1993) 290–299
11. Schlansker, M., Kathail, V.: Critical path reduction for scalar programs. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, Michigan, IEEE Computer Society TC-MICRO and ACM SIGMICRO, IEEE Computer Society Press (1995) 57–69
12. Schlansker, M., Mahlke, S., Johnson, R.: Control CPR: A branch height reduction optimization for EPIC architectures. In: *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, ACM Press (1999) 155–168
13. Krehling, W., Whalley, D., Bailey, M., Yuan, X., Uh, G.R., van Engelen, R.: Branch elimination by condition merging. Technical report, Florida State University (2003) URL: <http://websrv.cs.fsu.edu/research/reports/TR-030201.ps>.

Exploiting On-Chip Data Transfers for Improving Performance of Chip-Scale Multiprocessors

G. Chen¹, M. Kandemir¹, I. Kolcu², and A. Choudhary³

¹ Pennsylvania State University, PA 16802, USA

² UMIST, Manchester M60 1QD, UK

³ Northwestern University, Evanston, IL 60208, USA

Abstract. As compared to a complex single processor based system, on-chip multiprocessors are less complex, more power efficient, and easier to test and validate. In this work, we focus on an on-chip multiprocessor where each processor has a local memory (or cache). We demonstrate that, in such an architecture, allowing each processor to do off-chip memory requests on behalf of other processors can improve overall performance over a straightforward strategy, where each processor performs off-chip requests independently. Our experimental results obtained using six benchmark codes indicate large execution cycle savings over a wide range of architectural configurations.

1 Introduction

A natural memory architecture for an on-chip multiprocessor is the one in which each processor has a private (on-chip) memory space (in form of either a software-controlled local memory or a hardware-controlled cache). Consequently, assuming the existence of an off-chip (shared) memory, from a given processor's perspective, we have a two-level memory hierarchy: on-chip local memory and off-chip global-memory. One of the most critical issues in exploiting this two-level memory hierarchy is to reduce the number of off-chip accesses. Note that reducing off-chip memory accesses is beneficial not only from the performance perspective but also from the energy consumption viewpoint. This is because energy consumption is a function of the effective switching capacitance [2], and large off-chip memory structures have much larger capacitances as compared to relatively smaller on-chip memory structures.

In this paper, we focus on such a two-level memory architecture, and propose and evaluate a compiler-directed optimization strategy which reduces the frequency and volume of the off-chip memory traffic. Our approach targets array-intensive applications, and is based on the observation that, in an on-chip multiprocessor, inter-processor communication is cheaper (both performance and energy wise) than off-chip memory accesses. Consequently, we develop a strategy which reduces off-chip memory accesses at the expense of increased on-chip

data traffic. The net result is significant savings in execution cycles. Our approach achieves its objective by not performing off-chip memory accesses based on data requirements of each individual processor but based on the layout of the data in the off-chip memory. In other words, the total data space (array elements) that will be required considering all processors is traversed (accessed) in a layout-efficient manner (instead of allowing each processor access off-chip memory according to its own needs). The consequence of this “collective” memory access strategy is that each processor gets some data which are, potentially, required by some other processor. In the next step, the processors engage in a many-to-many communication (i.e., they access each others local memories) and, after this on-chip communication step, each processor gets the data it originally wanted.

We implemented the necessary compilation techniques for this two-step optimization strategy using an experimental compiler, and collected results from this implementation. Experimental data obtained using six application codes clearly demonstrate the effectiveness of our strategy. In order to measure the robustness of our approach, we also conducted experiments with different architectural parameters (due to the space limitation, the results are omitted in this paper). Based on our experimental data, we believe that the proposed strategy is a much better alternative to a more straightforward strategy, where each processor performs independent off-chip memory accesses. Our experimental results indicate large execution cycle savings over a wide range of architectural configurations. The results also indicate that the proposed approach outperforms an alternate technique that employs classical locality-oriented loop optimizations.

2 Architecture Abstraction

In this paper, we focus on an architectural abstraction depicted in Figure 1. In this abstraction, the on-chip multiprocessor contains k CPU-local memory pairs. There is also an off-chip (shared) global memory. There are two buses: an on-chip bus that connects local memories to each other, and an off-chip bus that connects local memories to the off-chip global memory. The other components of the on-chip multiprocessor (e.g., the clock circuitry, ASICs, signal converters, etc.) are omitted as they are not the focus of this work. It should be mentioned that many architectural implementations described by prior research are similar to this abstraction.

In this architecture, a data item required by CPU _{j} ($1 \leq j \leq k$) can be in three different locations: local memory _{j} , local memory _{j'} of CPU _{j'} (where $1 \leq j' \leq k$ and $j' \neq j$), and the global memory. The performance of a given application executing on such an architecture is significantly influenced by the location of the data requested. Typically, from the viewpoint of a given processor, accessing its local memory is much faster than accessing the global memory (and, in fact, considering current trends, we can expect this gap to be widened in the future). As compared to accessing its own local memory, accessing the local memory of some other processor (called “remote local memory,” or “remote memory” for

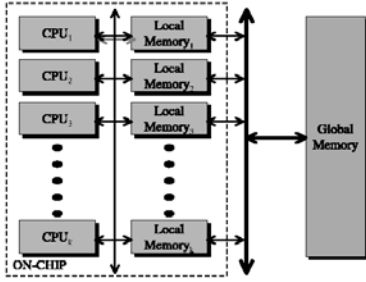


Fig. 1. On-chip multiprocessor abstraction.

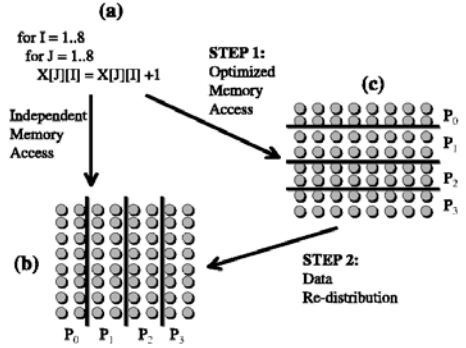


Fig. 2. Independent memory access vs. optimized memory access and data re-distribution.

short) is also more costly; however, such remote memory accesses are still much less expensive in comparison to global memory accesses. Therefore, an important issue in compiling applications for running on this chip-scale multiprocessor environment is to ensure that most of data requests of individual processors are satisfied from their local memories or remote memories. The goal of this paper is to present and evaluate a compiler-directed optimization strategy along this direction. It should be emphasized that, in this paper, when we mention “on-chip communication,” we mean remote memory access. Consequently, when we say “processors engage in all-to-all (or many-to-many) communication,” we mean that they access each other’s local memory. We also use the terms “off-chip memory” and “global memory” interchangeably.

3 Our Approach

Performance of an application executed on an on-chip multiprocessor is closely tied with the data access pattern. This is because data access pattern is the primary factor that determines how memory is accessed. Therefore, it is critically important that an application exhibits a good access pattern, in which data is accessed with temporal or spatial locality to the highest extent possible. Accesses with locality tend to frequently reuse the data in the same transfer unit (block), and this, in turn, reduces the number of off-chip memory accesses.

However, a straightforward coding of many array-intensive applications does not lead to good memory access patterns. As an example, consider the simple loop structure given in Figure 2(a). Assuming that we would like to parallelize the outer loop across four processors (P_0 , P_1 , P_2 , and P_3) in our on-chip multiprocessor, Figure 2(b) shows the data access pattern exhibited when each processor performs off-chip accesses independently. It can be seen from this access pattern that, assuming a row-major array layout in memory, each processor touches only a small amount of sequential data. In other words, the spatial locality exhibited

by each processor is very low. To quantify this, let us conduct a simple calculation to see the memory access cost of this loop nest. Assuming that the granularity of the data transfers between the off-chip memory and on-chip memory is 4 array elements and that the array is stored in a row-major format in the off-chip memory, each processor will make 8 off-chip requests in the access pattern depicted in Figure 2(b). The total volume of the data transferred by each processor is, therefore, 32 (and, note that only 16 of these are really useful). One simple strategy that can be adopted by an optimizing compiler is to interchange the loops. While this may help improve locality in some cases, in many loops, data dependences can prevent loop interchange (or similar loop transformations [4]).

In this example, our two step solution operates as follows. We first consider the data layout in memory (which is row-major) and allow each processor access the data using the best possible access pattern (as depicted in Figure 2(c)). Now, each processor has some data which are, originally, required by some other processor. In the next step, the processors engage in an all-to-all (intra-chip) communication (a special case of many-to-many communication), and data is redistributed accordingly. We can calculate the cost of this approach as follows. The number of off-chip memory accesses per processor is only 4 (as all 16 elements that need to be read are consecutive in the off-chip memory). In addition, the total data volume (transferred from the off-chip memory) is 16 (i.e., no unneeded data is transferred). Note that, as compared to the independent memory access scenario, here we reduce the number of off-chip accesses (per processor) by half (which can lead to significant execution cycles savings).

However, in this optimized strategy, we also need to perform inter-processor data re-distribution (i.e., the second step) so that each processor receives the data it originally wanted. In our example, each processor needs to send/receive data elements to/from the other three processors. Let us assume that performing an off-chip data transfer takes C_1 cycles and performing an on-chip communication (i.e., local memory-to-local memory transfer) takes C_2 cycles, the independent memory access strategy costs $8C_1$ (per processor) and our optimized strategy costs $4C_1 + 3C_2$ (again, per processor). Assuming C_1 is 50 cycles and C_2 is 10 cycles, our approach improves data access cost by 42.5% (170/400). This small example (with its simplistic cost view) shows that performing off-chip data accesses on behalf of other processors can actually improve the performance of the straightforward (independent) memory access strategy.

It should be noted that, depending on whether the local memory space is a software-managed local memory or hardware-managed cache, it might be necessary (in our approach) to interleave off-chip data accesses with on-chip communication. If the local memory is software-managed and is large enough to hold the entire set of elements that will be transferred from the off-chip memory, it is acceptable to first perform all off-chip accesses and, then, perform on-chip communication. On the other hand, if the on-chip local memory is not large enough to hold all the required elements, then we need to “tile” the computation. What tiling means in this context is that we first transfer (from the off-chip memory) some data elements (called “data tile”) to fill the local memory, then perform

Benchmark	Brief Description	Cycles
<i>Atr</i>	Network IP address translator	11,095,718
<i>SP</i>	Computing the all-nodes shortest paths on a given graph	13,892,044
<i>Encr</i>	Secure digital signature generator and verifier	30,662,073
<i>Hyper</i>	Multiprocessor communication activity simulator	18,023,649
<i>Wood</i>	Color-based visual surface inspector	37,021,119
<i>Usonic</i>	Feature-based object estimation	46,728,681

Fig. 3. Brief Description of our benchmarks.

on-chip communication, and then perform the computation on this data tile (and write back the tile to the global memory if it is modified). After this, we transfer the next set of elements (i.e., the next data tile) and proceed the same way, and so on. It should be observed that if the on-chip memory space is implemented as a conventional cache, tiling the computation might be very important as there is no guarantee that all the data transferred from the off-chip memory will still remain in the cache at the time we move to the on-chip communication phase.

Note that, in a straightforward off-chip memory access strategy, each processor accesses its portion of an array without taking into consideration of the storage pattern of the array (i.e., how the array is laid out in memory). In contrast, in our approach, the processors first access the data using an access pattern, which is the “same” as the storage pattern of the array. This helps reduce the number of off-chip memory references. After that, the data are redistributed across processors so that each array element arrives in its final destination. In general, the proposed strategy will be useful when access pattern and storage pattern are different.

4 Experiments

4.1 Setup

We used six applications to test the success of our optimization strategy. The brief descriptions of these benchmarks are presented in Figure 3. While we have also access to pointer-based versions of these applications, in this study, we only used their array-based versions. These applications are coded in such a way that their input sizes can be set by the programmer. In this work, the total data sizes manipulated by these applications range from 661.4KB to 1.14MB.

We simulated execution of an on-chip multiprocessor with local and global memories using a custom simulator. Each processor is a simple, single-issue architecture with a pipeline of five stages. Our simulator takes an architectural description and an executable code. The architectural description indicates the number of processors, the capacities of on-chip and off-chip memories, and their relative access latencies. We refer to the relative access latencies of local, remote, and global memories as the “access ratio”. For example, an access ratio of 1:10:50 indicates that a remote memory access is ten times as costly as a local memory

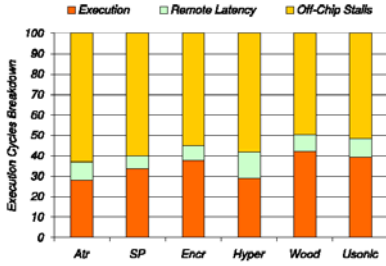


Fig. 4. Execution cycle breakdown for the default mode of execution.

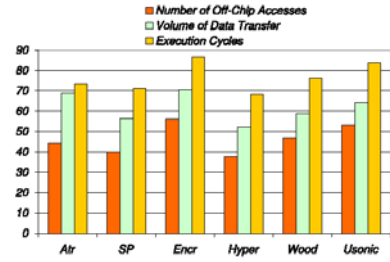


Fig. 5. Normalized number of off-chip accesses, transfer volume, and overall execution cycles.

access, and that a global memory access is five times as costly as a remote memory access. We ran the optimized code on a Sun UltraSparc machine. By default, we assume that the system has 8 processors (running at 250MHz) on the chip. Each processor has 4KB local memory. These processors share a 32MB global memory. The access ratio (i.e, the relative access latencies of local, remote and global memories) is 1:10:50.

In order to evaluate the effectiveness of our approach, we compare it to a “default mode of execution.” In this mode, each processor performs independent off-chip memory accesses. However, before going to the off-chip memory, it first checks its local memory, and then checks all remote memories. Consequently, if the requested data item is available in either local memory or remote memories, the off-chip memory access is avoided. As discussed earlier in detail, our approach tries to improve over this default mode of execution by performing off-chip memory accesses in a coordinated fashion. In this strategy, each processor performs off-chip accesses in a way which is most preferable from the data locality viewpoint. After this, processors exchange data so that each data item arrives in its final destination. The last columns of Figure 3 gives the execution cycles obtained under this default mode of execution. Execution cycles reported in the rest of this paper are values normalized with respect to these numbers. In comparing our optimization strategy to the default mode of execution, we used three different metrics. The first metric is the number of off-chip accesses. The second metric is the transfer volume, which is the amount of data transferred from the off-chip memory. As discussed earlier in the paper, our approach is expected to transfer fewer data items (as it exploits spatial locality in off-chip accesses) than the default mode of execution. While reductions in these two metrics indicate the effectiveness of our approach, the ultimate criterion is the reduction in overall execution cycles, which is our third metric.

4.2 Results

Figure 4 gives the execution cycle breakdown when each processor accesses the global memory independently (default mode of execution). In this graph, execution cycles are divided into three parts: (1) cycles spent in executing code

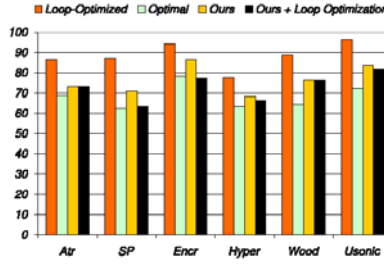


Fig. 6. Normalized execution cycles of different versions.

(including local memory accesses); (2) cycles spent in performing remote memory accesses; and (3) cycles spent in performing global memory accesses (i.e., off-chip accesses). Our first observation is that, on the average, 56.3% of cycles are expended in waiting for off-chip data to arrive. This observation clearly underlines the importance of reducing the number of off-chip data requests. In comparison, the cycles spent in remote memory accesses constitute only 8.75% of total cycles.

The graph in Figure 5 shows the number of off-chip memory accesses, transfer volume, and execution cycles resulting from our strategy, each normalized with respect to the corresponding quantity incurred when the default execution mode is used. We see that the average reductions in the number of off-chip memory accesses and transfer volume are 53.66% and 38.18%, respectively. These reductions, in turn, lead to 23.45% savings in overall execution cycles. As a result, we can conclude that our approach is much more successful than the default mode of execution. Execution cycle savings are lower in *Encr* and *Usonic* as the original off-chip access patterns exhibited by these applications are relatively good (as compared to the access pattern of the other applications under the default mode of execution).

One might argue that classical loop transformations can be used for achieving similar improvement to those provided by our optimization strategy. To check this, we measured performance of the loop optimized versions of our benchmark codes. The loop transformations used here are linear transformations (e.g., loop interchange and scaling), iteration space tiling, and loop fusion. To select the best tile size, we employed the technique proposed by Coleman and McKinley [3]. The first bar of each benchmark in Figure 6 gives the execution cycles of this version, normalized with respect to the original codes (note that both the versions use the default mode of execution). The average improvement brought by this loop-optimized version is 11.52%, which is significantly lower than the average improvement provided by our approach, which is 23.45% (the results of our approach are reproduced as the third bar in Figure 6). The reason that our approach performs better than classical loop transformations is related to data dependences [4]. Specifically, a loop transformation cannot be used if it violates the intrinsic data dependences in the loop being optimized. In many of the loops in our benchmark codes (specifically, in 45% of all loop nests), data dependences

prevent the best loop transformation from being applied. However, this result does not mean that our optimization strategy cannot take advantage of loop transformations. To evaluate the impact of loop optimizations on the effectiveness of our approach, in our next set of experiments, we measured the execution cycles of a version that combines our approach with loop optimizations. The loop optimizations employed are the ones mentioned earlier. Specifically, we first restructured the code using loop transformations such that spatial locality is improved as much as possible. We then applied our approach in performing off-chip data accesses. The last bar in Figure 6 for each application gives the normalized execution cycles for this version. One can observe that average improvements due to this enhanced version is 26.98% (a 3.53% additional improvement over our base strategy). It is also important to see how close our optimization comes to the optimal in reducing execution cycles. In order to see this, we performed experiments with a hand-optimized version that reduces the off-chip activity to a minimum. It achieves this using a global optimization strategy which considers all nests in the application together and by assuming an “infinite amount” of on-chip storage. In other words, for each data item, if possible, an off-chip reference is made only once (and since we have unlimited on-chip memory, no data item needs to be written back – even if it is modified). The second bar for each application in Figure 6 presents the behavior of this strategy. We see that the average execution cycle improvements is around 31.77%, indicating that our approach still has a large gap to close to reach the optimal.

5 Concluding Remarks

On-chip multiprocessing can be aimed at systems that require high performance in either instruction-level parallelism (ILP) or thread-level parallelism, since additional threads can be run simultaneously on other CPU cores within the chip, and therefore, is expected to have a clear advantage over traditional architectures that concentrate on ILP. In this paper, we demonstrate how a compiler can optimize off-chip data accesses in an on-chip multiprocessor based environment by allowing collective off-chip accesses, whereby each processor performs off-chip accesses on behalf of other processors. Our results clearly show the benefits of this approach.

References

1. S. P. Amarasinghe et al. Multiprocessors from a Software Perspective. *IEEE Micro Magazine*, June 1996, pages 52–61.
2. A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
3. S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
4. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

An Energy-Oriented Evaluation of Communication Optimizations for Microsensor Networks

I. Kadayif¹, M. Kandemir¹, A. Choudhary², and M. Karakoy³

¹ Pennsylvania State University, University Park, PA 16802, USA

² Northwestern University, Evanston, IL 60208, USA

³ Imperial College, London SW7 2AZ, UK

Abstract. Wireless, microsensor networks have potential for enabling a myriad of applications for sensing and controlling the physical world. While architectural/circuit-level techniques are critical for the success of these networks, software optimizations are also expected to become instrumental in extracting the maximum benefits from the performance and energy behavior angles. In this paper, focusing on a sensor network where the sensors form a two-dimensional mesh, we experimentally evaluate a set of compiler-directed communication optimizations from the energy perspective. Our experimental results obtained using a set of array-intensive benchmarks show significant reductions in communication energy spent during execution.

1 Introduction and Motivation

A networked system of inexpensive and plentiful microsensors, with multiple sensor types, low-power embedded processors, and wireless communication and positioning ability offers a promising solution for many military and civil applications. As noted in [7], technological progresses in integrated, low-power CMOS communication devices and sensors make a rich design space of networked sensors viable. Recent years have witnessed several efforts at the architectural and circuit level for designing and implementing microsensor based networks [2]. While architectural/circuit-level techniques are critical for the success of these networks, software optimizations are also expected to become instrumental in extracting the maximum benefits from the performance and energy behavior angles.

Optimizing energy consumption of a wireless microsensor network is important not only because it is not possible (in some environments) to re-charge batteries on nodes, but also because software running on sensor nodes may consume a significant amount of energy. In broad terms, we can divide the energy expended during operation into two parts: computation energy and communication energy. To minimize the overall energy consumption, we need to minimize the energy spent in both computation and communication. While power-efficient customized wireless protocols [5] are part of the big picture as far as minimizing

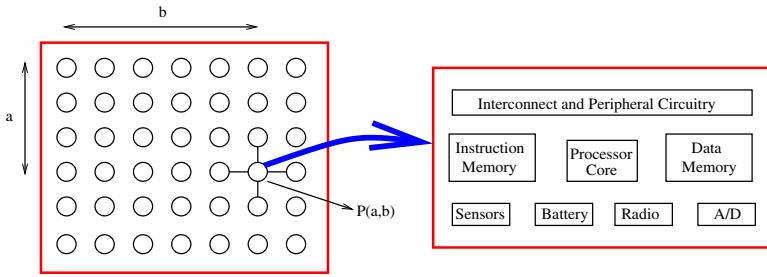


Fig. 1. Assumed sensor network architecture and blocks in a sensor node.

communication energy is concerned, we can also employ application level and compiler level optimizations.

In this paper, we focus on a wireless microsensor network environment that processes array-intensive codes. Array-intensive codes are very common in many image and signal processing applications [3]. Focusing on a sensor network where the nodes form a two-dimensional mesh, we present a set of source code level communication optimization techniques and evaluate them from the energy perspective.

2 Architecture and Language Support

The left part of Figure 1 shows the sensor network architecture assumed in this work. Basically, we assume that the sensor nodes are distributed over a two-dimensional space (area), and the distance between neighboring sensors are close to uniform throughout the network. In our study, each sensor node is assumed to have the capability of communicating with its four neighbors. Each node in this network can be identified using coordinates a (in the vertical dimension) and b (in the horizontal dimension), and can be denoted using $P(a,b)$. Consequently, its four neighbors can be identified as $P(a-1,b)$, $P(a+1,b)$, $P(a,b-1)$, and $P(a,b+1)$. Note that this nearest-neighbor communication style matches very well with many real-world uses of microsensors, as the neighboring nodes are typically expected to share data to carry out a given task. It should also be noted, however, that, although not discussed here, our software framework is able to handle general node-to-node or broadcast types of communications as well.

The right part of Figure 1 illustrates the major components of a given sensor node. Each node in our network contains sensor(s), A/D converter, battery, processor core, instruction and data memories, radio, and peripheral circuitry. Since sensor networks can be deployed in time-critical applications, we do not consider a cache architecture. Instead, each node is equipped with fast (SRAM) instruction and data memories. The software support for such an architecture is critical. Our compiler takes an input code written in C and parallelizes it across the nodes in the network. After parallelization, each sensor node executes the

same code (parameterized using parameters **a** and **b**) but works on a different portion of the dataset (e.g., a rectilinear segment of a multidimensional array of signals). In other words, the sensor nodes effectively exploit data parallelism. Our compiler parallelizes each loop nest in the code using the data decomposition supplied by the programmer. For a given array, a data decomposition specifies how the elements of the array are decomposed (distributed) across the memories of sensor nodes.

In order to express communication at the source language level, we assume that two communication primitives are available. The first primitive is of the form:

send DATA to P(c,d).

When executed by a sensor node **P(a,b)**, this primitive sends data (denoted by **DATA**) to the sensor node **P(c,d)**. The data communicated can be a single array element, an entire array, or (in many cases) an array region. We assume that this data is received by the node **P(c,d)** when it executes our other primitive:

receive DATA from P(a,b).

In order for a communication to occur, each **send** primitive should be matched with a corresponding **receive** primitive. All communication protocol-related activities are assumed to be captured within these primitives. If the size of the message indicated in these calls is larger than the packet size, the message is divided into several packets. This activity occurs within the **send** and **receive** routines.

3 Communication Optimizations

Our focus is on array-based applications that can benefit from parallel execution in a microsensor-based environment. In such applications, given an array of signals, typically, each processor is responsible for processing a portion of the array. This operation style matches directly to an environment where each sensor node is collecting some data from the portion of an area covered by it and processing the collected data. In this paper, we use the communication optimizations summarized in Table 1. Details of these optimizations can be found elsewhere [10,8]. In contrast, in a naive communication strategy, each data item is communicated only when it is needed and no two communication messages are combined to improve bandwidth.

4 Experimental Setup

4.1 Benchmark Codes

We use a set of array-intensive benchmark programs in our experiments. The salient characteristics of the codes in our experimental suite are summarized in Table 2. Each array element is assumed to be 4-bit wide. All code modifications have been performed using the SUIF compiler [1].

Table 1. Communication optimizations evaluated in this study.

Optimization	Brief Explanation
Message Vectorization	Hoisting communication that occurs due to a single reference to outer loop position, and combining them.
Message Coalescing	Combining communications that occur due to the different references to the same array
Message Aggregation	Combining communications that occur due to references to the different arrays
Inter-Nest Optimization	Eliminating communication required in a nest if it has already been performed in a previous nest
Computation/Communication Overlap	Eliminating the time spent in communication by doing useful computation while waiting for data to arrive

Table 2. Benchmarks used in the experiments. The second, and third columns give, respectively, the total input size, and the number of arrays in the code.

Benchmark	Input Size	Number of Arrays	Brief Description
3-step-log	295.08 KB	3	Motion Estimation
adi	271.09 KB	6	Alternate Direction Integral
full-search	98.77 KB	3	Motion Estimation
hier	97.77 KB	7	Motion Estimation
mxm	464.84 KB	3	Matrix Multiply
parallel-hier	295.08 KB	3	Motion Estimation
tomcatv	174.22 KB	9	Mesh Generation
jacobi	312.00 KB	2	Stencil-like Computation
red-black SOR	156.00 KB	1	Stencil-like Computation

4.2 Modeling Energy Consumption

We separate the system energy into two parts: computation energy and communication energy. Computation energy is the energy consumed in processor core (datapath), instruction memory, data memory, and clock network. In this work, we focus on a simple, single-issue, five-stage pipelined processor core which is suitable for employing in a sensor node. We use SimplePower [11], a publicly-available, cycle-accurate energy simulator, to model the energy consumption in this processor core and its clock generatin unit (PLL). We assume that each node has an instruction memory and a data memory (both are SRAM). The energy consumed in these memories is dependent primarily on the number of accesses and memory configuration (e.g., capacity, the number of read/write ports, and whether it is banked or not). We modified the Shade simulation environment [6] to capture the number of references to instruction and data memories and used the CACTI tool [12] to calculate the per access energy cost. The data collected from Shade and CACTI are then combined to compute the overall energy consumption due to memory accesses.

As our communication energy component, we consider the energy expended for sending and receiving data. The radio in the sensor nodes is capable of both sending data and, at the same time, sensing incoming data. We assume that if the radio is not sending any data, it does not spend any energy (omitting the energy expended due to sensing). After packing data, the processor sends the data to

Table 3. Computation and communication energy breakdown for our applications. All energy values are in microjoules.

Benchmark		Communication Energy			Computation Energy				
		Col.	Row	Block	IMem.	DMem.	Clock	Dpath	Total
3-step-log	v	6867	5922	12064	266503	58376	27411	69894	422185
	v+c	5449	5449	5449					
	v+c+a	3559	3559	3559					
adi	v	173196	173196	82886	38204	7734	8037	19541	73516
	v+c	173196	173196	82886					
	v+c+a	165636	165636	36894					
full-search	v	127615	123835	189985	343946	84058	31132	84654	543788
	v+c	110605	110605	110605					
	v+c+a	108715	108715	108715					
hier	v	191422	185752	284977	24961	5945	1902	5373	38179
	v+c	165907	165907	165907					
	v+c+a	164017	164017	164017					
mxm	v	431700	431700	286080	16609	2653	2576	5734	27572
	v+c	367440	367440	146976					
	v+c+a	367440	367440	146976					
parallel-hier	v	63807	61917	94992	168742	36775	15326	40626	261468
	v+c	55302	55302	55302					
	v+c+a	55302	55302	55302					
tomcatv	v	153780	61512	58934	18877	3788	3195	7844	33706
	v+c	153780	61512	42494					
	v+c+a	142440	57732	33623					
jacobi	v	72839	72839	33671	75264	12016	11858	27568	126706
	v+c	72839	72839	33671					
	v+c+a	72839	72839	33671					
red-black-SOR	v	76619	76619	39719	73732	11472	12288	28160	125652
	v+c	76619	76619	39719					
	v+c+a	76619	76619	39719					

the other processor via radio. The radio needs a specific startup time to start sending/receiving message. We used the radio energy model presented by [9] to account for communication energy. We assumed 160 sensor nodes participating in parallel execution of the application.

5 Results

5.1 Energy Breakdown

Table 3 gives the energy breakdown for our benchmarks for three different data decompositions and communication optimizations. Since the computation energies for different decompositions and different communication optimizations are almost the same we report only one computation energy value for each benchmark. The column decomposition (in Table 3) refers to a decomposition, whereby each sensor node owns a column-block of each array; the other decompositions correspond to cases where the arrays involved in the computation are decomposed in row-block and block-block manner across sensor nodes. Also, we consider three different communication optimizations: message vectorization (denoted *v*), message vectorization + message coalescing (denoted *v+c*), and message vectorization + message coalescing + message aggregation (denoted *v+c+a*). Unless stated otherwise, all energy numbers given are for dynamic energy only (i.e., they do not include the leakage energy consumption).

Table 4. Communication energy with naive communication. All energy values are in microjoules.

Benchmark	Column	Row	Block
3-step-log	173200144	173200144	173200144
adi	2429856	2429856	971942
full-search	360833632	360833632	360833632
hier	22719154	22719154	22719154
mxm	15819376	15819376	25311000
parallel-hier	28866690	28866690	28866690
tomcatv	2024880	809952	570206
jacobi	1036739	1036739	414695
red black SOR	1036739	1036739	414695

We can make several observations from the numbers reported in Table 3. First, we see that in some benchmarks the computation energy dominates the communication energy, whereas in some benchmarks it is the opposite. This is largely a characteristic of the access pattern exhibited by the application under a given (data decomposition, communication optimization) pair. Based on these results, we can conclude that communication energy constitutes a significant portion of the overall energy budget. The second observation is that the communication optimizations save significant amount of energy. For example, **v+c** improves the communication energy of the **hier** benchmark by 13.7% (over the only message-vectorized version) when column decomposition is used. As another example, the **v+c+a** version in **tomcatv** saves 7.2% communication energy over the **v+c** version. In fact, as can be seen from our results, in some cases, an optimization can even shift the energy bottleneck from communication to computation. For example, in the **adi** benchmark, when block decomposition is used, the communication energy of the **v+c** version is larger than the computation energy. However, when we use the **v+c+a** version, the communication energy becomes less than the computation energy. Third, we can observe that data decomposition in some benchmarks makes difference in communication energy. For example, the block decomposition performs much better than column and row decompositions in **adi**. In fact, for this benchmark code, while communication energy dominates computation energy in column and row decompositions, communication energy is approximately half of the computation energy if the message optimizations are applied in conjunction with block decomposition. It should be stressed that working with the naive communication strategy described earlier (that is, not using any communication optimization) may result in intolerable communication energies. To illustrate this, Table 4 shows the communication energy consumption when naive communication is employed. Comparing these values with those in Table 3 emphasizes the difference between optimizing and not optimizing communication energy.

5.2 Impact of Overlapping Communication with Computation

The results presented so far were obtained under the assumption that a processor sits idle during communication. In fact, we assumed that the processor, when

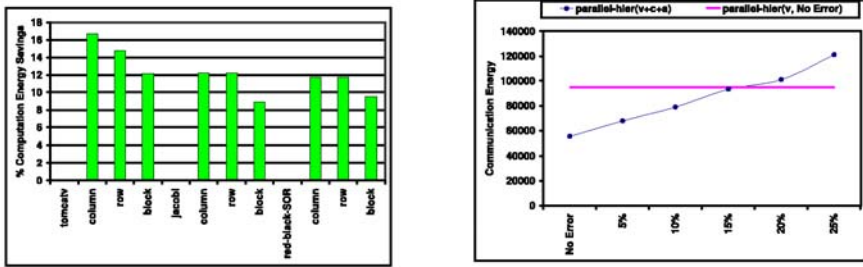


Fig. 2. Left: Percentage savings in computation energy due communication/computation overlapping. Right: Impact of communication error.

waiting for the data to be sent, places itself (or a tiny OS does this for it) into a low-power mode, where energy consumption is negligible. This is not very realistic as processor and memories typically consume some amount of leakage energy [4] during these waiting periods. Obviously, the longer the waiting period, the higher the leakage energy consumption. Consequently, we can reduce the leakage energy consumption by reducing the time that the processor waits idle for the communication to be completed. This can be achieved by overlapping the computation with communication. To evaluate the energy savings due to this optimization, we applied it to three benchmarks from our experimental suite, and measured the percentage computation energy savings. The results shown in the left graph in Figure 2 indicate that our approach can save around 20% of original computation energy of the $v+c+a$ version. In these experiments, we have assumed that the leakage energy consumption per cycle of a component is 20% of the per access dynamic energy consumption of the same component. Since current trends indicate that leakage energy consumption will be more important in the future [4], this optimization can be expected to be more useful with upcoming process technologies.

5.3 Communication Error

Communication errors in a sensor network may cause a significant energy waste due to re-transmission. To see whether our optimizations better resist against this increase in energy consumption, we performed another set of experiments where we measured the energy consumption under a random error model. Since the results with most of the benchmarks are similar, here we focus only on **parallel-hier**. The graph given in the right part of Figure 2 shows the communication energy consumption of the $v+c+a$ version of this application assuming different error rates. For comparison purposes, we also give the communication energy consumption (in microjoules) of the v version without error. The results shown indicate that the $v+c+a$ version starts to consume more energy than the error-free v version only beyond a 15% error rate. In other words, an applica-

tion powered by our optimization can tolerate more errors (than the original application) under the same energy budget.

6 Conclusions and Future Work

Advances in CMOS technology and microsensors enabled construction of large, power-efficient networked sensors. The energy behavior of applications running on these networks is largely dictated by the software support employed such as operating systems and compilers. Our results presented in this paper indicate that high-level communication optimizations can be vital if one wants to keep the communication energy consumption of microsensor network under control.

References

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.
2. G. Asada, M. Dong, T. S. Lin, F. Newberg, G. Pottie, and W. J. Kaiser. Wireless integrated network sensors: low power systems on a chip. In *Proc. ESSCIRC'98*, Hague, Netherlands, Sept 1998.
3. F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, June, 1998.
4. A. Chandrakasan et al. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
5. S.-H. Cho and A. Chandrakasan. Energy-efficient protocols for low duty cycle wireless microsensor networks. In *Proc. ICASSP'2001*, May 2001.
6. B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proc. the 1994 ACM Sigmetrics Conference on the Measurement and Modeling of Computer Systems*, May 1994.
7. J. Hill et al. System architecture directions for network sensors. In *Proc. ASPLOS*, 2000.
8. M. Kandemir et al. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, November 1999.
9. E. Shih et al. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor network. In *Proc. the 7th Annual International Conference on Mobile Computing and Networking*, July 16-22, 2001, Rome Italy.
10. C-W. Tseng. An optimizing Fortran D compiler for MIMD distributed-memory machines. *Ph.D. Thesis, Rice COMP TR93-199*, Dept. of Computer Science, Rice University, January 1993.
11. N. Vijaykrishnan et al. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. the International Symposium on Computer Architecture*, June 2000.
12. S. Wilton and N. P. Jouppi. CACTI: an enhanced cycle access and cycle time model. *IEEE Journal of Solid-State Circuits*, pp. 677–687, 1996.

Increasing the Parallelism of Irregular Loops with Dependences[★]

David E. Singh¹, María J. Martín², and Francisco F. Rivera¹

¹ Dept. of Electronics and Computer Science, Univ. Santiago de Compostela, SPAIN,
`{david,fran}@dec.usc.es`

² Dept. of Electronics and Systems, Univ. A Coruña, SPAIN
`mariam@udc.es`

Abstract. In this work, we present new proposals based on the owner-compute rule for the parallelization of irregular loops with dependences. The parallel code increases the available parallelism through the distribution of the statements inside each iteration instead of the whole iterations of the loop. Additionally, our proposal presents as main features the re-ordering of the layout of the indirection entries, optimizing data locality, and the efficient load balancing. Inspector and executor phases are fully parallel, without synchronizations and uncoupled, allowing the reuse of the information of the inspector. Experimental results on a SGI O2000 system prove that our approach exhibits a high performance, even when compared to well-known parallelization strategies.

1 Introduction

In general, we broadly classify irregular loops into two different families: loops without data dependences among iterations, and loops that present dependences, *doacross* loops. In this work, we focus on the parallelization of irregular *doacross* loops in CC-NUMA shared memory machines. In particular, we restrict ourselves to a generic structure of loops like the one shown in Figure 1(a) where S indirection arrays x_i of size N_x ($1 \leq i \leq S$) perform arbitrary read or write operations in the accessed array a . In this figure, $\odot$ represents a generic operator that can be different for each indirection. According to the value of the indirection arrays, any kind of dependence can arise. For instance, assuming that $N_x = 3$, $S = 2$, $x_1 = \{4, 1, 4\}$ for read accesses to a and $x_2 = \{1, 2, 2\}$ for write accesses to a , there is a true dependence between the second and the first iterations, and an output dependence between the second and third iterations.

In order to parallelize these kind of loops, different strategies can be applied. One of the most popular approaches is the inspector-executor strategy. The goal of the inspector-executor algorithms is to maximize parallelism and minimize the overhead of the analysis stage. One of the first inspector-executor approaches was proposed by Zhu and Yew [1]. In this proposal loop iterations are divided into subsets called wavefronts, which contain iterations that can be executed in

[★] This work was funded by CICYT under project TIC 2001-3694.

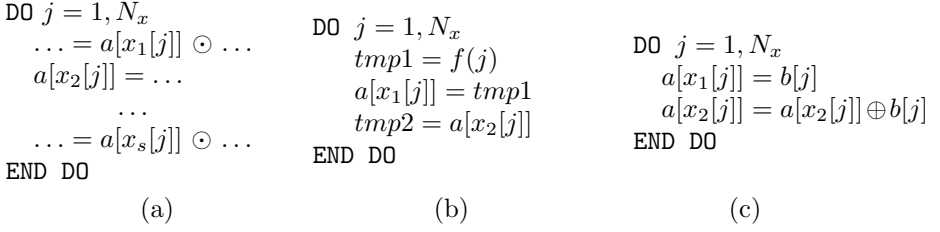


Fig. 1. Irregular loops: (a) template, (b) statement grouping, (c) multiple reordering.

parallel. This approach has two limitations: first, the inspector and the executor are tightly coupled and the inspector is not reused across invocations of the same loop. Second, the execution of consecutive reads to the same array entry is serialized. Midkiff and Padua [2] improve this strategy by allowing concurrent reads of the same entry. Saltz, et al [3] propose an alternative solution but restricted to loops with no output dependences. Their inspector and executor are uncoupled. Leung and Zahorjan [4] extend this work to consider output dependences and propose parallel versions for the inspector.

All these proposals exploit iteration-level parallelism, that is, loop iterations are distributed among the available processors and iteration-level synchronizations are performed to guarantee a correct execution order. A different approach can be found in the CYT method [5]. This method is also based on the distribution of the iterations among the processors but it checks for dependences in the operation-level instead of in the iterations level. Operation-level synchronizations are included to fulfill dependences. The main advantage of this algorithm is the extraction of a higher degree of parallelism. A comparison between strategies based on iteration-level and operation-level parallelism is presented in [6]. This work shows in an empiric way that operation-level methods outperform iteration-level methods. In [7], we presented two operation-level strategies for parallelizing *doacross* loops and improving data locality.

In this work, we propose new parallelization techniques that reduces synchronizations and improves even more data locality. Our techniques are based on the distribution of the statements instead of the iterations which increases the available parallelism. They are applicable to *doacross* loops where data dependences between statements are exclusively due to accesses to one array, named *a* in our example (Figure 1(a)). We define statement $stmt_i^j$ of a loop, as the i^{th} line of the body executed in the j^{th} iteration. In the previous example, dependences arise from statement $stmt_1^j$ to $stmt_2^j$ and from statement $stmt_2^j$ to $stmt_3^j$. Often real codes have statements that do not present indirect accesses. For instance, let's consider the loop of Figure 1(b). For each iteration j , statement $stmt_1^j$ does not have accesses to *a* but it presents a true dependence with $stmt_2^j$. In these cases, a fusion of statements can be performed, assuming that there are statements with more than one line of code. In the example, statement $stmt_1^j$ can include, for each iteration j , the first two lines, whereas $stmt_2^j$ refer to the third line of the body of the loop. In this way, we have a loop with two statements per iteration.

<pre> DOALL $p = 1, N_p$ DO $j = 1, N_x$ IF($x_1[j] \in my\_block[p]$) exec: $stmt_1^j$ IF($x_2[j] \in my\_block[p]$) exec: $stmt_2^j$... IF($x_s[j] \in my\_block[p]$) exec: $stmt_s^j$ END DO END DOALL </pre>	<pre> DOALL $p = 1, N_p$ DO $l = 1, N_{slc} - 1$ DO $i = 1, S$ DO $j = \rho_i^{slc}[l][p], \rho_i^{slc}[l + 1][p] - 1$ exec: $stmt_i^j$ END DO END DO END DO END DOALL </pre>
(a)	(b)

Fig. 2. Parallel executors: (a) parallel loop applying the owner compute rule, (b) parallel loop applying the slice sort strategy.

Our final proposal is based on the inspector-executor paradigm. Initially, on run-time, the access patterns are analyzed and the dependence information is collected. Then, in the executor stage, the $S \times N_x$ statements are distributed among the processors. Additionally, the information provided by the inspector is also applied to improve data locality and load balance.

2 Parallelization Strategy

We will use the loop of Figure 1(a) as case of study, where some of the statements inside of the loop could be irregular reduction operations. For this kind of irregular loops we can formulate the following property.

Property 1. Given two statements $stmt_i^j$ and $stmt_i^{j'}$, that perform generic accesses to the same entry of a . If there is not an intermediate statement that accesses to this entry, and $j' > j$, then $stmt_i^{j'}$ can be executed anytime after $stmt_i^j$. $\square$

Then, according to the previous property, the order of the statements can be changed as long as the dependences are maintained. The following sections describe different approaches that exploit this property.

2.1 Owner-Compute Rule Strategy

As starting point, we propose a parallel executor based on the owner-compute rule. The parallel code corresponding to Figure 1(a) is shown in Figure 2(a), where N_p is the number of available processors. Each one has assigned an interval of entries of array a which is denoted as $my\_block[p]$. In the loop execution, each processor traverses all the iterations of the loop checking for local accesses to a in each statement. Given that the statements are accessed in the same order than the original loop, the dependences are preserved. The main advantage of this proposal is the locality in the accesses to a , and its main drawback is the

overhead introduced by the additional checks. In the next section, we propose a new strategy called *slice sort* based on the owner-compute rule, but focused on several additional objectives. First, our technique obtains a better exploitation of the memory hierarchy. Our proposal performs a reordering of the indirection arrays to obtain good locality in the indirection accesses, and to reduce false sharing. Several previous papers [8,9] proof the importance of this factor in the performance of the parallel execution, specially on CC-NUMA shared memory machines. Second, in order to obtain a good performance of the parallel code, a correct load balance must be guaranteed. Our proposal optimizes dynamically the load balance. And third, the checks introduced by the owner-compute rule strategy are eliminated.

2.2 Slice Sort Strategy

The slice sort strategy consists of an inspector that performs the analysis and scheduling of the loop, and an executor that properly runs the parallel code. The purpose of the inspector is the determination, in run-time, of the optimal arrangement of the statements of an irregular loop. Our proposal is based on the owner-compute rule, each processor p is assigned to an interval of a . Only statements accessing to entries of a inside the considered block will be executed. Additionally, the indirection arrays are reordered so that, for each one, each processor accesses consecutive entries, improving further more data locality.

Inspector stage. The inspector stores the statements to be executed for each processor, as well as the execution order needed to fulfill the dependences. This information is stored into a structure of N_{slc} sets called slices. Each slice contains only the statements that can be executed in the considered block without data dependences. The concept of slice is similar to the definition of wavefronts [1] but now the slices are defined for individual statements instead of iterations. The basic parallel inspector is shown in Figure 3.

Initially, in the *analyzing stage*, each processor p traverses all the entries of each indirection, and analyze which of them perform accesses in its assigned interval of a . The private arrays ρ_i^{row} store the number of accesses that are performed in each entry of a by the statements $stmt_i^j$ ($1 \leq j \leq N_x$). For a given statement that performs an access to a , the minimum index of slice in which it can be assigned is obtained by function $slice()$. Let's assume that ρ^{row} is the set of all $\rho_i^{row} \forall i$, and since the accesses are analyzed in the proper order of execution, we have:

$$slice(\rho^{row}, x_i[j]) = \sum_{k=1}^S \rho_k^{row}[x_i[j]] \quad (1)$$

With this values, the shared array ρ_i^{slc} is properly updated. For each processor, the array counts the number of statements $stmt_i^j$ assigned to each slice. Note that any kind of dependence is maintained, even Read after Read dependences.

Subsequently, in the *shifting stage*, the ρ^{slc} arrays are realigned. This operation allows to write, in the next stage, the reordered array in different and

```

DOALL   $p = 1, N_p$ 
    DO   $j = 1, N_x$                                      % Analyzing stage
        DO   $i = 1, S$ 
            IF   $(x_i[j] \in my\_block[p])$ 
                 $\rho_i^{row}[x_i[j]] ++$ 
                 $\rho_i^{slc}[slice(\rho^{row}, x_i[j]), p] ++$ 
            END IF
        END DO
    END DO
    DO   $k = N_{slc}, 1$                                      % Shifting stage
        DO   $i = 1, S$ 
             $\rho_i^{slc}[k, p] = accum(x_i, p) + \sum_{j=1}^{k-1} \rho_i^{slc}[j, p]$ 
        END DO
    END DO
    DO   $j = N_x, 1$                                      % Sorting stage
        DO   $i = S, 1$ 
            IF   $(x_i[j] \in my\_block[p])$ 
                 $\rho_i^{slc}[slice(\rho^{row}, x_i[j]), p] --$ 
                 $x_i^{out}[\rho_i^{slc}[slice(\rho^{row}, x_i[j]), p]] = x_i[j]$ 
                 $\rho_i^{row}[x_i[j]] --$ 
            END IF
        END DO
    END DO
END DOALL

```

Fig. 3. Parallel inspector algorithm.

contiguous memory regions. Using function $accum()$, the number of entries of each indirection computed by each processor is obtained. More formally, defining $\|\cdot\|$ as the cardinality operator, for a given indirection x_i and processor p :

$$accum(x_i, p) = \|\{x_i[j] \mid x_i[j] \in my\_block[p']\} \mid \forall p' < p \quad \forall j \in [1, N_x]\| \quad (2)$$

Once this value is computed, the entries of ρ^{slc} arrays are updated adding the offset of previous processors and previous entries in the same slice. Finally, in the *sorting stage*, the reordered indirection arrays x_i^{out} are generated. Once more, all the accesses are analyzed, but this time in reverse order. Therefore arrays ρ_i^{slc} point to the last element of each slice, and they are decremented as they are being processed. For each entry of x_i , the position in the new array x_i^{out} is given by the current value of ρ_i^{slc} . Note that this procedure allows us to reuse the information of the analyzing stage.

Executor stage. Figure 2(b) shows the parallel executor. As each processor only computes the accesses to the region of a assigned to it, therefore no synchronizations are required. Since all the statements within a slice do not present dependences, they can be executed in parallel. Note that using slices the dependences for each processor are preserved.

The workload can be dynamically balanced changing the size of the block of a assigned to each processor. The workload associated to the p^{th} processor is:

$$load[p] = \sum_{i=1}^S (C_i \parallel x_i[j] / x_i[j] \in my\_block[p], \forall j \in [1, N_x] \parallel) \quad (3)$$

Where C_i represents the computational cost of each statement. Our algorithm obtains, with this equation, the whole cost of the loop. The ideal fraction of workload associated to each processor is obtained dividing the total workload by the number of processors. Therefore, contiguous blocks of entries of a that have the same workload are computed using equation 3. The whole process is done in parallel.

3 Performance Evaluation

We used various tests in order to evaluate the efficiency of our proposal. First, we evaluated the complexity and memory requirements of the inspector. Then, we compared the performance of our proposal with other state-of-the-art approaches, using synthetic and real benchmarks.

The complexity of our inspector is proportional to the number of entries of the indirection arrays. Specifically, it is proportional to the sum of the complexity of its three stages: $\mathcal{O}(N_x S + N_{slc} S + N_x S)$. Typically, $N_x \gg N_{slc}$, so we have $\mathcal{O}(2N_x S)$. If the irregular code presents poor load balance, then a variable block distribution is advisable, so we have to take into account the cost of the load balancer that is $\mathcal{O}(N_x S)$.

In terms of the memory overhead of the executor, we have distinguished two situations. The first one, shown in Figure 1(a), deals only with the indirection arrays that depend on the loop index. In this case, the overhead of the executor is due to ρ^{slc} arrays and it is $N_{slc} N_p S$. The second situation differs in that if more than one array depend of the loop index, it will have to be replicated. Figure 1(c) shows an example where array b is acceded by two statements. As

```

DO  j = 1, N_x
  DO  k = 1, W
    dummy work for stmt1
  END DO
  tmp1 = tmp1 + a[x1[j]]
  DO  k = 1, W
    dummy work for stmt2
  END DO
  a[x2[j]] = tmp2
END DO

```

(a) Synthetic code

Matrix	N_x	N_a	CP	N_{slc}
<i>gemat1</i>	23684	4929	4938	4928
<i>gemat12</i>	16555	4929	49	44
<i>mbeacxc</i>	24960	496	487	485
<i>beaflw</i>	26701	507	500	495
<i>psmigr_2</i>	270011	3140	2626	2294
<i>25600_U</i>	12800	25600	9	7
<i>25600_90_10</i>	12800	25600	45	35
<i>51200_U</i>	25600	51200	11	9
<i>51200_90_10</i>	25600	51200	46	30

(b) Benchmark matrices

Fig. 4. Synthetic benchmark.

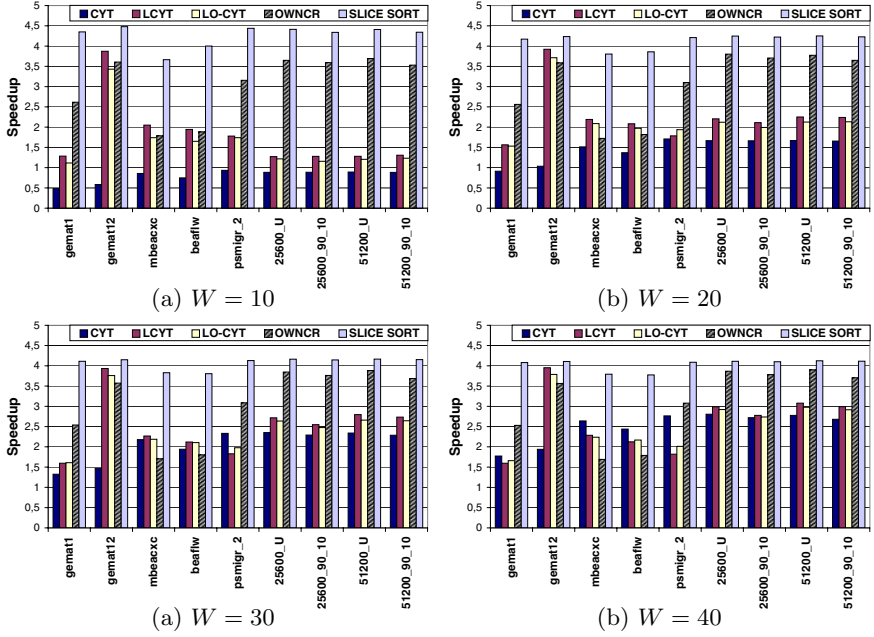


Fig. 5. Speedups for different workloads on 8 processors for $W = 10$.

a rule, we have that when one array is acceded by several statements, private copies for each one must be created and each copy must be sorted in the same way that the corresponding indirection array. This can be done in the inspector without any further computational cost. In this way, and for the example, we ensure that when entry $a[x_i^{out}[j]]$ is computed, the correct value of $b_i^{out}[j]$ will be used.

Next, our proposals are compared with the CYT, LCYT and LO-LCYT strategies [5,7]. The target machine is a SGI O2000 with up to eight R10K processors. The benchmarks are written in Fortran, and parallelized using OpenMP directives. As starting point, and following similar approaches taken by other authors [5,10], we have used the synthetic loop of Figure 4(a) in order to evaluate the efficiency of each strategy. We assume that each iteration of the loop has associated two statements; each one includes one irregular access and its associated dummy work. W determines the amount of workload associated to each one. To simplify the study, we assume the same W for both statements.

We have used sparse matrices extracted from real programs [11] as indirection arrays, as well as artificial access patterns that can be classified as uniform and non-uniform. For the uniform ones, denoted with the termination “ U ”, all the array elements have the same probability of being accessed, whereas in the non-uniform, 90% of references access to 10% of array elements. The table shown in Figure 4(b) contains the main features of the access patterns. N_x is the number of entries of each indirection, N_a is the size of a , and CP is the Critical Path. For the CYT, LCYT and LO-LCYT algorithms the N_x iterations are distributed

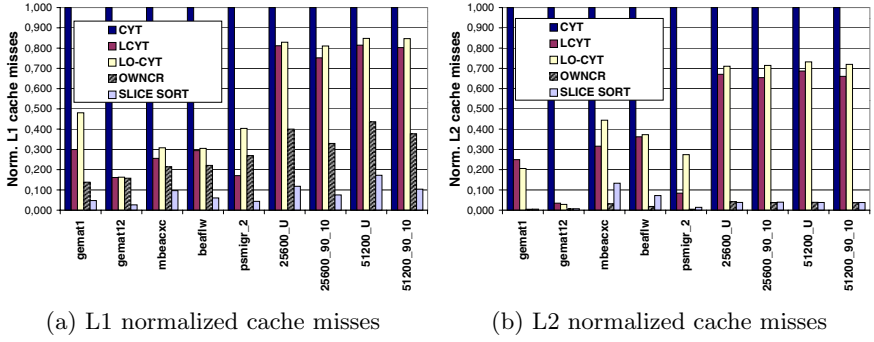


Fig. 6. Cache behaviour on 8 processors for $W = 10$.

among the p processors. The CP ($1 \leq CP \leq N_x$) is the length of the largest dependence chain in the loop and gives an estimate of how parallel the loop is. If $CP = 1$ the loop is fully parallel whereas if $CP = N_x$ the loop sequential. For the slice sort strategy, synchronizations are only needed when all the iterations of each slice are processed, so N_S gives an estimate of how parallel the loop is.

Note that the Critical Path is slightly bigger than the number of slices N_{slc} . This is due to the fact that previous proposals take into account dependences between iterations instead of single statements. For instance, considering the loop of Figure 1(a) with $N_x = 3$, $S = 2$, $x_1 = \{1, 2, 3\}$ and $x_2 = \{2, 3, 4\}$, the associated critical path is 3 because each iteration depends on the previous one. However, considering single statements, only two slices are required to preserve the dependences, that is, a better exploitation of the available parallelism is achieved using the slice sort strategy.

Figure 5 shows the speedups for the executor on 8 processors and different workloads. A block distribution of the entries of a was used for the owner-compute rule approach. This technique presents good data locality in the write accesses to a . It has good performance with synthetic matrices because they are well balanced. However, this performance exhibits an important drop down for less balanced real patterns. In all the cases, the slice sort strategy obtains the best results due to an equilibrated load balance and a good data locality. Figure 6 shows the number of L1 and L2 cache misses, normalized with respect to the CYT strategy, for 8 processors. Note the important reduction obtained with our proposals. Additionally, synchronizations between processors and runtime checks are eliminated. The performance of our proposal decreases as W increases due to a smaller influence of the locality improvements.

Table 1 shows the overall performance of the slice sort strategy taking into account the overhead of the inspector. Specifically, it contains the number of times the inspector needs to be reused to be faster than the others. For example, for matrix *gemat1*, two iterations of our executor are faster than two iterations of the owner-compute rule approach, including the overhead of the inspector of our proposal. A zero entry means that our proposal is faster than the other proposal even from the first iteration and taking into account the overhead of

Table 1. Threshold in iterations for outperforming the rest of the proposals.

Matrix	W=10				W=30			
	OWNCR	CYT	LCYT	LO-LCYT	OWNCR	CYT	LCYT	LO-LCYT
<i>gemat1</i>	2	0	0	0	0	0	0	0
<i>gemat12</i>	5	0	5	2	2	0	4	2
<i>mbeacxc</i>	1	0	1	0	0	0	0	0
<i>beaflw</i>	1	0	0	0	0	0	0	0
<i>psmigr_2</i>	5	0	1	1	2	0	0	0
<i>25600_U</i>	7	0	0	0	6	0	0	0
<i>25600_90_10</i>	7	0	0	0	5	0	0	0
<i>51200_U</i>	9	0	0	0	8	0	0	0
<i>51200_90_10</i>	7	0	0	0	4	0	0	0

the inspector. Note that, in general, the threshold is not high, and decreases as W increases. This threshold is more important for the owner-compute rule approach because it does not use an inspector.

Table 2. Input data and speedups for the PLTMG benchmark.

Characteristics				Speedup on 4 processors			Speedup on 8 processors		
Matrix	N_x	N_a	N_{slc}	OWNCR	ARRAYEXP	SLCSRT	OWNCR	ARRAYEXP	SLCSRT
<i>test1</i>	7500	1300	9	0.65	0.65	1.88	0.75	0.72	2.32
<i>test2</i>	21090	3595	9	0.55	0.80	2.25	0.76	0.91	3.21
<i>test3</i>	90480	15240	9	0.79	1.10	2.91	1.93	1.46	2.52

Table 3. Input data and speedups for the BDNA benchmark.

Characteristics				Speedup on 4 processors		Speedup on 8 processors	
Matrix	N_x	N_a	N_{slc}	OWNCR	SLCSRT	OWNCR	SLCSRT
<i>test1</i>	1520	5830	1	1.1	2.1	1.5	1.7

We also have evaluated the efficiency of our proposal with two real applications. The first one was extracted from the *mtxmlt* subroutine of PLTMG Edition 7.1 [12]. This software solves elliptic partial differential equations in general regions of the plane. Although data dependences related to this loop can be overridden by the array expansion parallelization technique, we have found two reasons to include it in our benchmarks. First, it allows us to compare our proposal with other high-competitive approaches. Second, as in other applications, it is mandatory to execute the loop following the sequential order. In an out-of-order execution, like array expansion, the final result can be slightly different from the original. Our proposals assures the correct results. We made tests for three different problem sizes. Their main features are shown in Table 2. This table also shows the obtained speedups for the executor on 4 and 8 proces-

sors. Note that the slice sort approach is the only one that obtains significant speedups.

The second benchmark was extracted from the Perfect Club Benchmarks [13]. Specifically, it corresponds to the loop 711 of the *correc* routine of the BDNA application. Given that access pattern is unknown at compiler-time, dependence analysis have to be performed at run-time. In this stage, our inspector determines that for the input data considered the loop is fully parallel. In [5,10] this loop was tested for the CYT algorithm, and poor speedups were obtained. LCYT and LO-LCYT can not be applied as more than one write access is considered. Results obtained with our proposals for the executor stage are shown in Table 3. Due to the small size of the problem, the available parallelism is low, even though our proposal obtains significant speedups.

References

1. C-Q. Zhu and P-C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Trans. on Software Engineering*, 13(6):726–739, (1987).
2. S P Midkiff and D A Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, (1987).
3. J H Saltz, R Mirchandaney, and K Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, (1991).
4. S-T Leung and J Zahorjan. Restructuring Arrays for Efficient Parallel Loop Execution. Technical Report 94-02-01, Department of Computer Science and Engineering, University of Washington, (1994).
5. D-K. Chen, J. Torrellas, and P-C. Yew. An Efficient Algorithm for the Run-Time Parallelization of DOACROSS Loops. In *Supercomputing Conference*, pages 518–527, Washington DC, (1994).
6. C. Xu. Effects of Parallelism Degree on Run-Time Parallelization of Loops. In *31st Hawaii Int'l Conference on System Sciences*, Kohala Coast, HI, (1998).
7. María J. Martín, David E. Singh, Juan Touriño, and Franciso F. Rivera. Exploiting Locality in the Run-Time Parallelization of Irregular Loops. In *Proceedings of the 31th International Conference on Parallel Processing.*, pages 17–22, (2002).
8. H. Han and C-W. Tseng. Improving Locality for Adaptive Irregular Scientific Codes. In *13th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 173–188, Yorktown Heights, NY, (2000).
9. J. M. Mellor-Crummey, D. B. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *ACM Int'l Conference on Supercomputing*, pages 425–433, Rhodes, Greece, (1999).
10. C. Xu and V. Chaudhary. Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):433–450, (2001).
11. Iain S. Duff, Roger G. Grimes, and John G. Lewis. *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*. Boeing Computer Services, (1992).
12. R. E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 7.0*. SIAM, Philadelphia, (1994).
13. M. Berry et al. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *Intl. Journal of Supercomputer Applications*, 3(3):5–40, (1989).

Finding Free Schedules for Non-uniform Loops

Volodymyr Beletskyy and Krzysztof Siedlecki

Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st.,
71-210 Szczecin, Poland, fax. (+4891) 487-64-39
{vbeletskyy, ksiedlecki}@wi.ps.pl

Abstract. An algorithm, permitting us to build free schedules for arbitrary nested non-uniform loops, is presented. The operations of each time schedule can be executed as soon as their operands are available. The algorithm requires exact dependence analysis. To describe and implement the algorithm and to carry out experiences, the dependence analysis by Pugh and Wonnacott was chosen where dependences are found in the form of tuple relations. The algorithm can be applied for both non-parameterized and parameterized loops. The algorithm proposed has been implemented and verified by means of the Omega project software.

1 Introduction

The larger number of transformations has been developed to expose parallelism in loops, minimize synchronization, and improve memory locality in the past, for example, [3]–[8], [11]. Most of those transformations permit us to extract parallelism from both uniform and non-uniform loops. But the question is how much loop parallelism these approaches extract.

The general problem of the loop parallelism detection is as follows. Dependences in loops can be represented in different ways, with approximations in general, or with an exact representation when possible. For each dependence representation based on approximations (level of dependences, uniform dependences, polyhedral representation of dependences), optimal algorithms exist, but for exact affine dependences, it is not known what are the loop transformations that extract maximal parallelism [5].

Following Vivien [14], we imply that an algorithm extracting parallelism is optimal if it finds all parallelism: 1) that can be extracted in its framework; 2) that is contained in the representation of the dependences it handles; 3) that is contained in the program to be parallelized (not taking into account neither the dependence representation used nor the transformations allowed).

This paper presents an algorithm permitting us to find free schedules for arbitrary nested non-uniform loops. This algorithm is optimal and find all parallelism in a loop represented by the instances of statements.

Our approach is based on non-linear schedules that can be implemented with procedures using a set of routines for manipulating linear constraints over integer variables, Presburger formulas, integer tuple relations and sets, and the code

generation routines for generating code to scan the points in the union of a number of convex polyhedra.

2 Background and Definitions

In this section, we in brief attach well-known knowledge to better explain our algorithm for finding free schedules for non-uniform loops.

In this paper, we deal with affine loop nests where lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known constants.

Following work [14], we refer to a particular execution of a statement for a certain iteration of the loops, that surround this statement, as an operation.

Two operations J and I are dependent if both access the same memory location and if at least one access is a write. We refer to I and J as the source and destination of the dependence, respectively, provided that I accesses the same memory location earlier than J ($I \prec J$).

Definition 1. An affine loop nest is non-uniform if it originates non-uniform dependence relations represented by an affine function f that expresses the dependence sources I in terms of the dependence destinations J ($I=f(J)$) or the converse.

Definition 2[13]. A dependence analysis is exact if for any affine dependence it detects a dependence if and only if one exists.

To describe the algorithm and carry out experiences, we chose the dependence analysis proposed by Pugh and Wonnacott [13] where dependences are presented with dependence relations.

Definition 3[13]. A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of the dependence and the values of the symbolic constants.

Definition 4[5]. A free schedule assigns operations as soon as their operands are available, that is, mapping $\sigma: I \rightarrow Z$ such that

$$\sigma(p) = \begin{cases} 0 & \text{if there is no } p' \in I \text{ s.t. } p' \prec p \\ 1 + \max(\sigma(p'), p' \in I, p' \prec p) & . \end{cases} \quad (1)$$

The free schedule is the "fastest" schedule possible. Its total execution time is

$$T_{free} = 1 + \max(q_{free}(p), p \in I) . \quad (2)$$

The algorithm proposed in this paper is applicable for loops that meet the requirements of the dependence analysis by Pugh and Wonnacott [13].

To follow the material of this paper, the reader should be familiar with the operations on tuple relations such as: union, difference, range, domain, application as well as existentially quantified variables. This knowledge is comprised in [9].

3 Free Schedules for Arbitrary Nested Loops

We will refer to the source of a dependence as the fair dependence source if it is not a destination of any other dependence.

We define the length of a chain of synchronization between a pair of dependent operations as $N-1$, where N is the maximal number of the operations which this chain connects, that is, it is the maximal number of the direct value based dependences [13] originated by this pair of the dependent operations.

The idea of the algorithm presented in this section is as follows. We divide all operations for each statement into two sets containing the independent and dependent operations (sources and destinations of dependences), respectively. For the second set, we firstly find those operations for which all operands are available. They form the operations of layer $Lay[0]$ to be executed firstly. Next, we eliminate from the second set the operations of $Lay[0]$ and find again those operations for which all operands are available. We repeat this process until there are no operations in the second set. The operations of the first set can be combined with the operations of arbitrary levels.

In imperfectly nested loops, statements may have different domains, and, in general, to find free schedules, we should deal with each statement independently. To generate resulting code, we should find free schedules for all statements independently, and next combine the same layers (including operations belonging to the same schedule time), originated by different statements, into one resulting layer.

Algorithm 1. Find the free schedule for an arbitrary nested loop

1. Find all cross-iteration dependences as well as the dependences originated by statements within the loop body. Build two sets of the dependences for each statement j , $j=1,2,\dots,n$. The first one, $S1_j$ includes the dependence relations whose destinations are the instances of statement j . Let the relations of $S1_j$ be $R1_{kj}$, $kj=1,2,\dots,n_j$, n_j is the number of the dependence relations in $S1_j$. The second set, $S2_j$ includes the dependence relations whose sources are originated with statement j .

For each statement j do:

2. Find the sources of the dependences as the domains of the relations, belonging to set $S2_j$, and unite them into one set I_j .
3. Find the destinations of the dependences as the ranges of the relations, belonging to set $S1_j$, and unite them into one set J_j .
4. Find the independent statement instances IND_j , that is, those that do not belong to any pair of the dependences as follows $IND_j := IS_j - I_j - J_j$, where IS_j is the set representing the iteration space of statement j .
5. Find all fair dependence sources FS_j as the difference between I_j and J_j . Sets FS_j form the layer of the operations ($Lay[0]_j$) which should be executed firstly.
6. Find the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length one or more as follows $L1_{kj} := R1_{kj}(FS(R1_{kj}))$, $kj=1,2,\dots,n_j$, where $R1_{kj}$ are found in step 1, $FS(R1_{kj})$ are the sets of the fair dependence sources that are the sources of the dependences represented with relations $R1_{kj}$. Unite sets $L1_{kj}$ into one set $L1_j$.
7. Find the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length two or more as follows $D1_{kj} := R1_{kj}(J(R1_{kj}))$, $kj=1,2,\dots,n_j$, where $J(R1_{kj})$ are the sets of the dependence destinations that are the

dependence sources represented with relations $R1_{kj}$. Unite sets $D1_{kj}$ into one set $D1_j$.

8. Find the operations belonging to the first layer as $Lay[1]_j := L1_j - D1_j$.

End for each

9. Find the second and remaining layers of the dependence destinations as follows:

$i=2$;

Loop:

For each j

$Jj := Jj - Lay[i-1]_j$;

$Li_{kj} := R1_{kj}(Lay[i]_j(R1_{kj}))$,

$kj=1,2,...,n_j$;

unite sets Li_{kj} into one set L_{ij} ;

$Di_{kj} := R1_{kj}(J(R1_{kj}))$,

$kj=1,2,...,n_j$;

unite sets Di_{kj} into one set D_{ij} ;

$Lay[i]_j := L_{ij} - D_{ij}$;

End for each

if each $Lay[i]_j == \text{False}$ then the end; else $i=i+1$; goto Loop;

– elimination of the dependence destinations belonging to layer $i-1$ and originated by the instances of statement j ;

– finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i or more, $Lay[i]_j(R1_{kj})$ are the sets representing the operations of layer i and are the sources of the dependences represented with relations $R1_{kj}$;

– finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length $i+1$ or more;

– finding the dependence destinations that are linked with the fair dependence sources by a chain of synchronization of length i ;

The independent operations IND_j can be combined with arbitrary layers. In our implementation, we unite them with the fair dependence sources to build $Lay[0]_j$.

4 Parameterized Loops

The algorithm presented permits us to find free schedules not only for the loops with the known loop bounds before compilation but also for parameterized loops.

Following the algorithm presented, we can try to find the given number m of parameterized layers for a loop. If there are no operations in a certain layer $i \leq m$, this means that the loop is characterized by the constant number of layers and each of them can be presented in a parameterized form. The procedure described permits us to reveal whether the number of layers is irrelevant to the size of a loop. If such a loop is

detected, code may be generated with a complexity that does not depend on the loop size.

But the main problem that requires further research is how to find the number of layers for parameterized loops in the general case.

If the number of layers does not depend on the size of a loop, we can present the free schedule in a symbolic way and generate corresponding code with a complexity that depends on the loop but not on the volume of the computation it describes.

When there exist only two layers under a free schedule, this means that there are no operations that simultaneously are the sources and destinations of dependences. Such a case can be very easily reviled. We should form the set of the dependence sources and the set of the dependence destinations originated with all loop statements and next find the intersection of these sets. If the result is FALSE, this means that there exist only two layers of operations under the free schedule.

It is worth to note that in the case when a loop exposes the constant number of layers, its free schedule may be much faster than the best affine schedule. Consider the following example [7]:

Table 1. Example of parallelizing the parametrized loop

Sequential code	Parallel code, our algorithm
<pre>for (i=0; i≤2*n; i++) a[i]=a[2*n-i];</pre>	<pre>par for (i=0; i≤n-1; i++){ a[i]=a[2*n-i]; a[2*n-i]=a[i]; } if (n ≥ 1) a[n] = a[2*n-n];</pre>

The best affine schedule for this loop is $i/2$ [7], that is, the number of the layers yielded with this schedule equals to n . The algorithm presented finds only two layers for this loop. The first and second statements in the parallel code presented in Table 1 originate the fair dependence sources (Lay[0]) and the first layer (Lay[1]), respectively; the condition statement represents the independent operations.

5 Related Work and Conclusion

The approaches presented in [1], [12] build an explicit graph of a subset of the iteration space, with each node representing the instance of a statement. Free schedules can be found by searching the graph, but the problem regarding boundary cases exists. We use tuple relations as an abstraction for data dependences. This permits us to reach all the same merits that discussed in [10]: handle non-uniform dependences and multidimensional loops without having to make special checks in boundary conditions. But, in contrast to work [10], our technique does not require the calculation of the transitive closure of relations.

Concerning parallelism detection, the following facts are known. If the level of dependences is the only available representation, then Allen and Kennedy's algorithm is known to be optimal [2]. For the case of a single statement with uniform dependences, linear scheduling (optimized Lamport's hyperplane method) is asymptotically optimal [4]. For the case of several statements with uniform

dependences, the previous result has been extended in work [8] to show that a linear schedule plus shifts leads to finding optimal parallelism. For the case of the polyhedral approximations of dependences, the method described in [3] is optimal. For affine dependences, the most powerful algorithm is Feautrier's one [6]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, among all possible affine schedules, it is optimal [14]. For affine dependences, the index splitting approach is known [7]. But it is a heuristic procedure, and it is not known how much index splitting is necessary, hence it is not known how many different code structures must be generated to reach optimality.

The technique presented in this paper is a first step to understand the problem of free schedules for loops with affine dependences. It permits us to find free schedules for both non-parameterized and parameterized loops but it does not answer what in general is the number of layers under the free schedule for a parameterized loop.

References

- [1] Chen, D.-K.: Compiler optimizations for parallel loops with fine-grained synchronization. Technical report TR-1863, Department of Computer Science, University of Illinois at Urbana-Champaign, (1994)
- [2] Darte and Vivien, F.: On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, Special issue on Optimizing Compilers for Parallel Languages, (1997) 12:83–112
- [3] Darte and Vivien, F.: Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. *International Journal of Parallel Programming*, 25(6), (1997) 447–496
- [4] Darte and Khachiyan, L. and Robert, Y.: Linear Scheduling is Nearly Optimal. *Parallel Processing Letters*, 1(2), (1991) 73–81
- [5] Darte, A., Robert, Y., Vivien, F.: *Scheduling and Automatic Parallelization*. Birkhäuser Boston, (2000)
- [6] Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), (December 1992)
- [7] Feautrier, P., Griebel, M. and Lengauer, C.: Index Set Splitting. *International Journal of Parallel Programming*, 28(6), (2000) 607–631
- [8] Patrick Le Gouëslier d'Argence: Affine Scheduling on Bounded Convex Polyhedral Domains is Asymptotically Optimal. *TCS* 196(1–2), (1998) 395–415
- [9] Kelly, W., Maslov V., Pugh, W., Rosser, E., Shpeisman, T. and Wonnacott, D.: *The Omega Library Interface Guide*. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, (March 1995)
- [10] Kelly, W., Pugh, W., Rosser, E. and Shpeisman, T.: Transitive Closure of Infinite Graphs and its Applications. *International Journal of Parallel Programming*, v. 24, n. 6, (December 1996) 579–598
- [11] Lim, W., Lam, M. S.: Maximizing parallelism and minimizing synchronization with affine transforms. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (January 1997)
- [12] Midkiff, S.P. and Padua, D.A.: A comparison of four synchronization optimization techniques. In *Proc. 1991 IEEE International Conf. on Parallel Processing*, (August 1991) II-9–II-16
- [13] Pugh, W., Wonnacott D.: An Exact Method for Analysis of Value-based Array Data Dependences. *Workshop on Languages and Compilers for Parallel Computing*, (1993)
- [14] Vivien F.: On the optimality of Feautrier's scheduling algorithm. In *Proceedings of the EUROPAR'2002*, (2002)

Replicated Placements in the Polyhedron Model

Peter Faber, Martin Griebel, and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D-94030 Passau, Germany
{faber,griebel,lengauer}@fmi.uni-passau.de

Abstract. Loop-carried code placement (LCCP) is able to remove redundant computations that cannot be recognized by traditional code motion techniques. However, this comes possibly at the price of increased communication time. We aim at minimizing the communication time by using replicated storage.

1 Introduction

The basic idea of the polyhedron model is to create a mathematical model of the computation to be done. This representation can be transformed in order to carry out code optimizations or automatic parallelization. In earlier work, we developed an optimization technique called *loop-carried code placement* (LCCP), which is able to remove redundant computations that are executed in different iterations of a loop. However, this improvement may come at the price of increased communication time. This is because intermediate results have to be stored in arrays which may introduce communications – actually a problem that also occurs in other programs (where intermediate results are stored).

Communication time may depend on various factors. An effective placement method should take into account as many specific traits of the underlying system as possible. Such specific traits may include certain communication patterns for which efficient communication code can be generated and should have the ability to use replicated data storage in order to decrease communication time.

Our present aim is to model as many of these traits as flexibly as possible with regard to some underlying cost model. The idea is then to “plug in” a cost function that evaluates the current placements and choose the placements that incur the least cost. Section 2 gives a short example of the use and limits of LCCP. In Section 3, we discuss the basic model used in our method. Section 4 describes the method of computing a placement in detail.

2 Motivation

Primarily, we are concerned with improving the results of an LCCP transformation, although the use of replicated placements is not limited to this scenario. In Example 1, a code fragment is given that is very well suited for applying LCCP.

Example 1. Let us consider the following code fragments:

<pre> !HPF\$ INDEPENDENT DO j=2,N-2 !HPF\$ INDEPENDENT DO k=1,N L(k,j)= (y(j) *2)*k& -(y(j+2)*2)/k& +(y(j+1)*2)+k& +(y(j-1)*2) END DO END DO </pre>	<pre> !HPF\$ INDEPENDENT DO j=1,N tmp(j)=y(j)*2 END DO !HPF\$ INDEPENDENT DO j=2,N-2 !HPF\$ INDEPENDENT DO k=1,N L(k,j)= tmp(j) *k & -tmp(j+2)/k & +tmp(j+1)+k & +tmp(j-1) END DO END DO </pre>

The (synthetic) code fragment on the left contains four computations of $y(j)*2$. The same value is computed repeatedly for different iterations of both the k -loop and the j -loop. LCCP can transform this code fragment into the one on the right (the actual transformation depends on the scheduler used). In the transformed code, $y(j)*2$ is calculated once and then assigned to a temporary.

Let us suppose that L is (BLOCK,BLOCK)-distributed. Depending on the distribution of y , different distributions of `tmp` may be beneficial: most probably, an alignment with y is a good approach; if y is replicated, it is probably more useful to align with L .

3 The Model

In this section, we discuss our model in closer detail. We give a representation for replicated storage in our model, and finally show how to adapt dependence information to this representation.

3.1 Base Language

Our method depends on the presence of an initial data placement. If we restrict ourselves to HPF programs, the user has already defined some data placement. The result of our method is a set of further placements that can be used in an HPF (or HPF-style) program for intermediate computations whose results are stored in arrays. Therefore, our method can be viewed as a compilation phase within an HPF compiler, or possibly as a preprocessor.

3.2 Occurrence Instances

We consider occurrence instances with dependence relations between them as described in our earlier work [FGL01b] (and in a technically expanded version

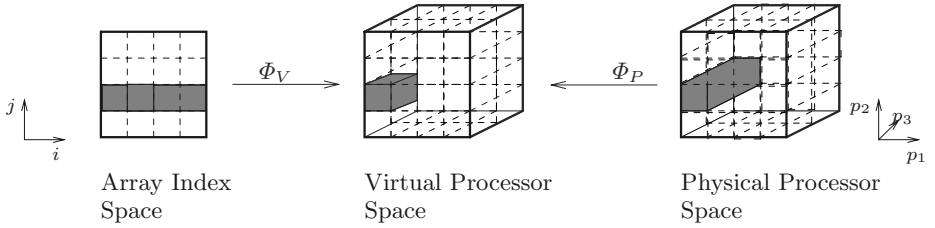


Fig. 1. Mappings representing a distribution: Φ_V maps array elements to a virtual processor, Φ_P maps physical processors to a virtual processor.

[FGL01a]), which is based on the polyhedron model [Lam74]. Thus, every computation executed can be viewed as a function application $f(e_1, \dots, e_q, \dots, e_r)$, where expressions $e_1, \dots, e_l$ are only read, while $e_{q+1}, \dots, e_r$ are only written, and each function application is assigned a unique number – its occurrence. This means that our method works on the expression level (and not – as usual – on the coarser level of statements). In the polyhedron model, placement functions defining where to execute a given occurrence are affine functions in the indices of loops surrounding the occurrence. We view these n -dimensional affine functions as linear functions in an $(n+1)$ -dimensional space using homogenous coordinates.

We employ a dependence analysis as the one of Feautrier [Fea91] to produce a dependence graph on occurrence instances which we call an *occurrence instance (dependence) graph* (OIG).

Actually, the usual input for our method will not be an OIG, but its reduced form, a *condensed occurrence instance graph* (COIG) as defined in our earlier work [FGL01b]. In a COIG, equivalent occurrence instances are reduced to a single representative (a single occurrence instance). However, for the work presented here, this distinction is unimportant.

3.3 How to Model Replication

Our aim is to use replication (of data and computations) to reduce communication costs. The replicated mapping of a point $\alpha \in \mathbb{Z}^{n+1}$ to a subset of $\mathbb{Z}^m$ can be described as follows: we introduce a template T (corresponding to $\mathbb{Z}^m$) and a pair of affine functions $\Phi = (\Phi_V, \Phi_P)$. α is then mapped to T via Φ_V , while all the points of $\mathbb{Z}^m$ to which α should ultimately be mapped are themselves mapped to that same template element by Φ_P (see Figure 1).

The set of processors on which an occurrence instance α is to be stored is:

$$\Phi(\alpha) = \Phi_P^{-1} \circ \Phi_V(\alpha)$$

A consistency condition is that the image of Φ_V be a subset of the image of Φ_P . For the placements given as input, we can safely assume this; the placements we compute in Section 4 satisfy this property by construction.

This approach corresponds to the description of replicated data in HPF, as sketched in Figure 1. An array, represented by the square with dimensions i and j on the left, is to be placed on a three-dimensional processor grid on the right.

3.4 Dependences in the Presence of Replication

In order to estimate the costs for communication induced by the target program, we view the dependences with respect to space coordinates – i.e., after application of the placement relation $\Phi = (\Phi_V, \Phi_P)$. Since neither of the two functions defining the placement relation is necessarily invertible, it is not immediately clear how to represent dependences in the target program.

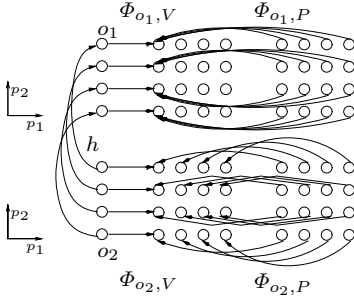


Fig. 2. Dependence from instances of occurrence o_1 to instances of occurrence o_2 .

$(\Phi_{o2,V}, \Phi_{o2,P})$ define a placement for the instances of o_2 , as depicted in the lower part of Figure 2. The instances of o_2 are only allocated on the first column of the processor space by their placement.

This means that all instances of occurrence o_2 , each of which depends on a different instance of o_1 , may “choose” the source processor from which to load the data needed for computation.

With the names taken from Figure 2, the new dependence relation is $h' = \Phi_{o1,P}^{-1} \circ \Phi_{o1,V} \circ h \circ \Phi_{o2,V}^{-1} \circ \Phi_{o2,P}$. We observe that there are two sets involved in the computation of h' :

- $\Phi_{o1,P}^{-1} \circ \Phi_{o1,V}(\alpha)$: the possible sources of α (copies of the same value).
- $\Phi_{o2,V}^{-1} \circ \Phi_{o2,P}(\beta)$: the set of occurrence instances to be executed on the processor with space coordinates β .

If $\Phi_{o1,P}^{-1} \circ \Phi_{o1,V}(\alpha)$ is a set, we may choose any point within that set for the representation of our dependence. Therefore, we can account for $\Phi_{o1,P}^{-1}$ by using a generalized inverse that yields a single point.

However, $\Phi_{o2,V}^{-1} \circ \Phi_{o2,P}$ has to be represented as a set, e.g., by using two mappings to represent the relation h' , just as with placement relations.

4 The Placement Method

With this description of replicated placements and resulting dependence relations, we can compute further placements. In order to allow almost any cost

¹ Note that occurrence instances may represent either, computations or data.

Figure 2 shows an example of a dependence given by an h-transformation h as introduced by Feautrier [Fea91]. In the upper part of the figure, four instances of an occurrence o_1 are mapped to a virtual processor grid by $\Phi_{o1,V}$; physical processors are mapped to the same grid by $\Phi_{o1,P}$. The h-transformation maps a target occurrence instance α to its source occurrence instance $h(\alpha)$. The instances of occurrence o_1 are stored in a replicated fashion (all processors with the same p_1 coordinate own a copy of an occurrence instance).¹ Just as the mappings $(\Phi_{o1,V}, \Phi_{o1,P})$ define a placement relation for the instances of o_1 , the mappings

model to be “plugged in”, we choose a naïve approach that basically considers all possible placements and selects the cheapest solution (names as in Figure 2):

1. For all sets of occurrence instances: propagate a placement candidate from source o_1 to target o_2 ($\Phi_{o_1} \circ h$) and from target to source ($\Phi_{o_2} \circ h^{-1}$).
2. For each combination of candidate placements for the occurrence instances:
 - a) Compute the dependence information according to the current placements. Candidate placements Φ_1, Φ_2 can be combined into a new placement Φ_3 placing all occurrence instances to both sets by asserting $\Phi_{3,P}^{-1} \circ \Phi_{3,V}(\alpha) \supseteq \Phi_1(\alpha) \cup \Phi_2(\alpha)$ for all α .
 - b) Compute a cost for the given dependence information, select the combination of placements that incurs the lowest cost.

As noted elsewhere [FGL01b], it is not necessary to compute a placement for *all* occurrence instances of a program. Which sets do need placements can be deduced from the dependence information.

Placements are ultimately propagated from data. Therefore, we have to consider the transitive closure of the dependence relation in Step 1. Then we compute placement relations that can be expressed by two affine mappings. If an occurrence o_2 depends on an occurrence o_1 with h-transformation h , the placements Φ_{o_1}, Φ_{o_2} should satisfy

$$\Phi_{o_2,P}^{-1} \circ \Phi_{o_2,V}(\alpha) \subseteq \Phi_{o_1,P}^{-1} \circ \Phi_{o_1,V}(h(\alpha)) \quad (1)$$

If placement relations are propagated from source to target, this can be guaranteed by computing the Φ_{o_2} as: $\Phi_{o_2,P} := \Phi_{o_1,P}, \Phi_{o_2,V} := \Phi_{o_1,V} \circ h$.

If the placement is propagated from target to source, the computation is a bit more complex since we have to consider dependences to several different targets. However, we can compute a placement relation that ensures condition (1) by using replication: the smallest subspace for which replication has to occur to satisfy condition (1) can be computed as the solution of a linear equality system.

5 Preliminary Results

Let us reconsider Example 1. We use a block distribution for both dimensions of L , and a replicated distribution for y along the first dimension of L (with $y(j)$ aligned to $L(*, j)$). We compiled the code using the ADAPTOR compiler and measured the runtime on 16 nodes of an SCI-connected PC-cluster. We applied a cost model that only distinguishes between communication patterns (local data [cost 1], shifts [cost 10], array-triplets [cost 100], general accesses [cost 1000]). According to that cost model, we had to choose replicated storage for `tmp` in our LCCP-optimized code. This resulted in about 10% improvement wrt. the case had we chosen a distribution that can be expressed by a function such as an alignment with L (which already performed better than the original code).

6 Related Work

The dHPF compiler uses a similar approach to take advantage of replication for computation partitioning [MCAB⁺02]. It can generate code for a union `ON HOME` specifications, which can therefore be propagated from a dependence target to its sources. This statement-driven approach leaves the control structure constant.

In contrast, our method is based on a fine-grained polyhedron model, i.e., we represent calculations of expressions inside loops as integer polyhedra and dependences as relations between these polyhedra. The control structure is later regenerated from the geometric description by loop scanning [QR00].

During the last decade, several placement algorithms based on the polyhedron model have been proposed, such as the placement algorithm by Feautrier [Fea94]. However, these methods do not consider replication.

7 Future Work

We are currently implementing our method as part of our code restructurer LooPo. We will conduct experiments on a substantial set of benchmarks when a stable implementation is available. In addition, we are working on an accurate cost model for our main HPF compiler, ADAPTOR. Finally, an important optimization is the partitioning of the iteration space into a form for which efficient code can be generated; loop transformations in the polyhedron model typically ignore the cost of enumerating loops, which is not always precise enough.

Acknowledgements. This work is supported by the DAAD through project PROCOPE and by the DFG through project *LooPo/HPF*.

References

- [Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.
- [Fea94] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
- [FGL01a] P. Faber, M. Griebel, and C. Lengauer. A closer look at loop-carried code replacement. In *Proc. GI/ITG PARS'01*, PARS-Mitteilungen Nr.18, pages 109–118. Gesellschaft für Informatik e.V., November 2001.
- [FGL01b] P. Faber, M. Griebel, and C. Lengauer. Loop-carried code placement. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing*, LNCS 2150, pages 230–234. Springer, 2001.
- [Lam74] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.
- [MCAB⁺02] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarría-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the Rice dHPF compiler. *Concurrency and Computation: Practice and Experience*, 14(8–9):741–767, July/August 2002.
- [QR00] F. Quilleré and S. Rajopadhye. Code generation for automatic parallelization in the polyhedral model. *Int. J. Parallel Programming*, 28(5):469–498, 2000.

Topic 5

Parallel and Distributed Databases, Data Mining, and Knowledge Discovery

Bernhard Mitschang, David Skillicorn, Philippe Bonnet, and Domenico Talia
Topic Chairs

Database systems and knowledge discovery tools are two key technologies for storing, querying, and mining large volumes of data available today. The number of people and organizations that use data analysis techniques in their daily activities is increasing significantly and in several application domains the need of processing and mining large data sets is becoming a standard task. However, these intensive data consuming applications suffer from performance problems and single database sources. Introducing data distribution and parallel processing help to overcome resource bottlenecks and to achieve guaranteed throughput, quality of service, and system scalability. High-performance computers supported by high speed networks and intelligent data management middleware offer parallel and distributed databases and knowledge discovery systems a great opportunity to support cost-effective every day applications.

Data processing and knowledge discovery on large amounts of data can benefit from the use of parallel computers both to improve performance and quality of data selection. Implementation of data mining tools on high-performance parallel computers allows for analyzing massive databases in a reasonable time. Faster processing also means that users can experiment with more models to understand complex data. Furthermore, high performance makes it practical for users to analyze greater quantities of data. Distribution of data sources and data mining tasks is another paramount issue that the increasing decentralization of human activities and large availability of connection facilities are making more and more critical.

This year, 14 papers discussing some of those issues were submitted to this topic and this was more than in previous years. Each paper was reviewed by at least three reviewers and, finally, we were able to select 3 regular papers and 3 short ones. The accepted papers discuss very interesting issues such as database replication on PC clusters, pipelined queries in large databases, parallel clustering, join site selection methods, parallel suffix arrays, and sensor database systems.

We would like to take the opportunity of thanking the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the referees with their highly useful comments, whose efforts have made this conference, and Topic 5 possible.

A Parallel Algorithm for Incremental Compact Clustering^{*}

Reynaldo Gil-García¹, José M. Badía-Contelles², and Aurora Pons-Porrata¹

¹ Universidad de Oriente, Santiago de Cuba, Cuba
`{gil,aurora}@app.uo.edu.cu`

² Universitat Jaume I, Castellón, Spain
`badia@icc.uji.es`

Abstract. In this paper we propose a new parallel clustering algorithm based on the incremental construction of the compact sets of a collection of objects. This parallel algorithm is portable to different parallel architectures and it uses the MPI library for message-passing. We also include experimental results on a cluster of personal computers, using synthetic data generated randomly and collections of documents. Our algorithm balances the load among the processors and tries to minimize the communications. The experimental results show that the parallel algorithm clearly improves its sequential version with large sets of data.

1 Introduction

Clustering algorithms are widely used for document classification, clustering of genes and proteins with similar functions, event detection and tracking on a stream of news, image segmentation and so on. Given a collection of n objects characterized by m features, clustering algorithms try to construct partitions or covers of this collection. The similarity among the objects in the same cluster should be maximum, whereas the similarity among objects in different clusters should be minimum. The clustering algorithms have three main elements, namely: the representation space, the similarity measure and the clustering criterion.

In many applications, the collection of objects is dynamic, with new items being added on a regular basis. An example of these applications is the event detection and tracking of streams of news. Classic algorithms need to know all the objects in order to perform the clustering and so, each time we modify the set of objects, it is necessary to cluster the whole collection again. Thus, we need algorithms able to update the clusters each time a new object is added to the data without rebuilding the whole set of clusters. This kind of algorithms are called incremental.

Many recent applications involve huge data sets that cannot be clustered in a reasonable time using one processor. Moreover, in many cases the data cannot

^{*} This work was partially supported by the Spanish CICYT projects TIC 2002-04400-C03-01 and TIC 2000-1683-C03-03.

be stored in the main memory of the processor and it is necessary to access the much slower secondary memory. A solution to this problem is to use parallel computers that can deal with large data sets and reduce the time of algorithms that can be very expensive. Parallel versions of some clustering algorithms have been developed, such as *Kmeans* [1], *MAFIA* [2], or *GLC* [3].

In this paper we propose a parallel incremental algorithm that finds the clusters as the compact sets in a collection of objects. This algorithm distributes the same number of objects in each processor and then balances the workload to perform in each one. The parallel algorithm was tested in a cluster of personal computers connected through a Myrinet network. Experimental results show a good behaviour of the parallel algorithm that clearly reduces the sequential time. Moreover, we have achieved near linear speedups when the collection of objects and the number of features are large.

The remainder of the paper is organized as follows. Section 2 describes the main features of the incremental compact algorithm. Section 3 includes the parallel algorithm. Section 4 shows the experimental results. Finally, conclusions are presented in Section 5.

2 Incremental Compact Algorithm

Incremental clustering algorithms update the clusters every time a new object arrives. They could be less efficient than the non-incremental algorithms if the whole collection is initially known. However, incremental algorithms are useful if they are more efficient when new objects are added in a regular basis. Some incremental algorithms have been developed, including, *Single-Pass* [4], *incremental K-Means* [5] and *Stars* algorithm [6]. The main drawback of these algorithms is that the obtained clusters depend on the arrival order of the objects. This is an undesirable characteristic because the clusters do not depend on the arrival order of the objects, but they only depend on the internal features of the data.

The incremental compact clustering algorithm [7] incrementally constructs the existing compact sets in a collection of objects. This algorithm is based on graphs and it produces disjoint clusters. Two objects are β_0 -similar if their similarity is greater or equal to β_0 , where β_0 is an user-defined parameter. We call maximum β_0 -similarity graph ($max-S$) to the oriented graph whose vertices are the objects to cluster and there is an edge from vertex o_i to vertex o_j , if o_j is the most β_0 -similar object to o_i . The compact sets are the connected components of the $max-S$ graph disregarding the orientation of the edges.

The compact algorithm stores the maximum β_0 -similarity of each object and the set of objects connected to it in the $max-S$ graph. Every time a new object (o) arrives, its similarity with each object of the existing clusters is calculated and the graph is updated. The arrival of o can change the current compact sets, because some new clusters may appear and others that already exist may disappear. Therefore, after updating the $max-S$ graph, the compact sets are rebuilt starting from o and the objects in the compact sets that become

unconnected. The compact sets that do not include objects connected with o remain unchanged.

During the graph updating task the algorithm constructs the following sets:

- *ClustersToProcess*: A cluster is included in this set if it has any object o' that satisfies the following conditions: 1) o is the most β_0 -similar to o' and the objects that were its most β_0 -similar are not anymore. 2) o' had at least two most β_0 -similar objects or o' is the most β_0 -similar to at least another object in this cluster. This set includes the clusters that could lose its compactness when the objects with the previous characteristics are removed from the cluster. Thus, these clusters must be reconstructed.
- *ObjectsToJoin*: An object o' is included in this set if it satisfies: 1) o is the most β_0 -similar to o' and the only object that was the most β_0 -similar to o' is not anymore. 2) o' is not the most β_0 -similar to any object of its cluster. The objects in this set will be included in the same compact set as o .
- *ClustersToJoin*: A cluster is included in this set if it is not in *ClustersToProcess* and it has at least one object o' that satisfies one of the following conditions: 1) o' is the most β_0 -similar object to o . 2) o is one of the most β_0 -similar objects to o' , that is, o is connected to o' and no edge of o' is broken in the $max - S$ graph. All the objects in *ClustersToJoin* will be included in the same compact set than o .

The main steps of the algorithm are the following:

1. Arrival of the new object o .
2. Updating of the $max - S$ graph.
For each object o_i in the existing clusters, its similarity with o is calculated and the set of objects connected with o_i and its maximum β_0 -similarity are updated. Calculate the maximum β_0 -similarity of o and the set of objects connected with it. The sets *ClustersToProcess*, *ObjectsToJoin* and *ClustersToJoin* are constructed. Every time an object is added to *ObjectsToJoin* it is removed from the cluster in which it was located before.
3. Reconstruction of the compact sets.
Let C be a set including o and all the objects included in the clusters in *ClustersToProcess*. Construct the existing compact sets in C and add them to the existing cluster list.
Add all the objects in *ObjectsToJoin* and all the objects included in the clusters in *ClustersToJoin* to the compact set including o . The clusters in *ClustersToProcess* and in *ClustersToJoin* are removed from the existing cluster list.

This algorithm is $O(n^2)$. Its main advantage over other incremental algorithms is that the generated set of clusters is unique, independently of the order of incoming objects. Besides, this algorithm makes no irrevocable cluster assignments. Thus, the mistakes made at the beginning, when little information is available, could be corrected. Another advantage of this algorithm is that it can deal with mixed incomplete object descriptions. On the other hand, the algorithm is not restricted to the use of metrics to compare the objects. It has been successfully applied to the online event detection in a collection of digital news [8].

3 Incremental Compact Parallel Algorithm

Our parallel algorithm uses a message-passing architecture and a master-slaves model, where one of the processors acts as the master during some phases of the algorithm. The data is distributed among the processors, so that processor i stores the description of object j if $i = j \bmod p$. This data partition tries to balance the load among the processors in order to improve the efficiency of the parallel algorithm. Each processor also stores the maximum β_0 -similarity of each of its objects and the indexes of the objects connected to it in the $max-S$ graph. Besides, each processor maintains a list of clusters (compact sets) and a list of its objects in each cluster. With this distribution of the data, a processor only stores information about the clusters containing any of its objects.

When a new object o arrives, each processor computes the similarity of all its objects with o and updates its part of the graph. This updating process requires the exchange of some information with other processors. The clusters that can lose their compactness are determined during this process. While updating the graph, each processor constructs the sets *ClustersToProcess*, *ObjectsToJoin* and *ClustersToJoin* in the same way as they are constructed in sequential algorithm. Besides, each processor constructs the set *LostEdges*, that contains the pairs of objects (o', o'') , where o' is an object of the processor, o'' is an object of another processor and the edge between them in the $max-S$ graph is broken.

Then the compact sets are reconstructed starting from the new object and the objects included in the clusters of *ClustersToProcess*. Each processor, using its objects in the clusters of *ClustersToProcess*, constructs the compact subsets that can be formed. For each one of these subsets, a new set called *Arrival* is constructed. This set contains all the objects that do not belong to the compact subset but are connected to an object in this subset. Observe that, the set *Arrival* will only contain objects of other processors. Afterwards, processor 0 constructs the new compact sets using the compact subsets in all the processors and their associated *Arrival* sets. If a compact subset contains at least one object included in the *Arrival* set of another compact subset, then both subsets are merged into the same compact set and its associated *Arrival* set is formed. The main steps of the parallel algorithm are the following:

1. Processor 0 broadcasts the description of the new object o .
2. Updating of the maximum β_0 -similarity graph.

- On each processor do:

For each object in the processor, its similarity with o is calculated and its maximum β_0 -similarity and the set of objects connected to it in the $max-S$ graph are updated. The sets *ClustersToProcess*, *ObjectsToJoin* and *LostEdges* are constructed. Every time an object is added to *ObjectsToJoin*, it is removed from the cluster in which it was located before. Construct the set *ClustersToJoin* by including the clusters with objects in the processor for which o is added to its set of maximum β_0 -similarity objects. Construct the set *To* which contains the objects in the processor for which o is its most β_0 -similar, and construct the set *From*

- which contains the objects in the processor that are the most β_0 -similar to o . Also compute $MaxSem$, the maximum β_0 -similarity to o of any of the objects in the processor.
- Join the sets *ClustersToProcess* and *LostEdges* of each processor in order to form *GlobalClustersToProcess* and *GlobalLostEdges*.
 - The processor in which o is stored gathers the sets *From* and *To* and the values $MaxSem_i$, calculates $MaxSem_o = \max \{MaxSem_i\}$, $i = 1, \dots, p$, stores the edges of o and broadcasts $MaxSem_o$.
 - Each processor updates *ClustersToJoin* and its edges with o using *From*, $MaxSem_o$, *GlobalClustersToProcess* and *GlobalLostEdges*.
3. Reconstruction of the compact sets.
- On each processor do:
 C = set of objects in the clusters included in *GlobalClustersToProcess*. If the processor owns o , add o to the set C . Form the existing compact subsets in C and their associated *Arrival* sets.
 - Join the sets *ClustersToJoin* of each processor in order to form *GlobalClustersToJoin*.
 - Processor 0 gathers all the compact subsets and their associated *Arrival* sets. It constructs the compact sets and broadcasts them.
 - Each processor updates its clusters and adds the objects in *GlobalClustersToJoin* or in *ObjectsToJoin* to the cluster including o . The clusters in *GlobalClustersToJoin* or in *GlobalClustersToProcess* are removed.

Some of the main features of our parallel algorithm are the following:

1. The load is balanced because each processor works with approximately $\frac{n}{p}$ objects and it computes approximately the same quantity of similarities. Also the $max - S$ graph is distributed and it is updated in parallel.
2. The compact sets that change due to the arrival of the new object are constructed in parallel. The construction of the sets *GlobalClustersToProcess* and *GlobalLostEdges* are carried out in parallel. The construction of the set *GlobalClustersToJoin* and the compact sets are carried out in parallel too.

4 Performance Evaluation

The target platform for our experimental study is a cluster of personal computer connected through a Myrinet network. The cluster consists of 32 Intel Pentium II-300MHz processors, with 128 Mbytes of SDRAM each one. The communication switch has a latency of 16.1 milliseconds and the bandwidth is 1 Gb/sec. The algorithms have been implemented on a Linux operating system, and we have used an used a specific implementation of the message-passing library MPI that offers small latencies and high bandwidths on the Myrinet network. We have executed the parallel algorithm varying the number of processors from 2 to 20.

First, we have performed an experimental analysis of the parallel algorithm using synthetic data generated randomly in $\mathbb{R}^m$. The number of objects varies

in 10000, 50000 and 200000. The number of features of the objects varies in 2, 8, 16, 64 and 128 and $\beta_0 = 0.5$. The values of the features have been generated so that the average density of the synthetic sets is the same for all data sets. The similarity measure used was $Sim(o_i, o_j) = \frac{1}{1+D(o_i, o_j)}$, where $D(o_i, o_j)$ is the euclidean distance between o_i and o_j .

Another experimental analysis was carried out using two document collections of articles published in the Spanish newspaper “El País” during June 1999. The first collection consist of 554 international articles. The second contains 2463 articles belonging to the sections Opinion, Health, National, International, Culture and Sports. The documents are represented using the traditional vectorial model. The terms of documents represent the lemmas of the words appearing in the texts. Stop words, such as articles, prepositions and adverbs are disregarded from the document vectors. Terms are statistically weighted using the normalized term frequency (TF). These data sets are of very high dimensionality. In our experiments the β_0 threshold used was 0.25 and to compare the documents we use the cosine measure.

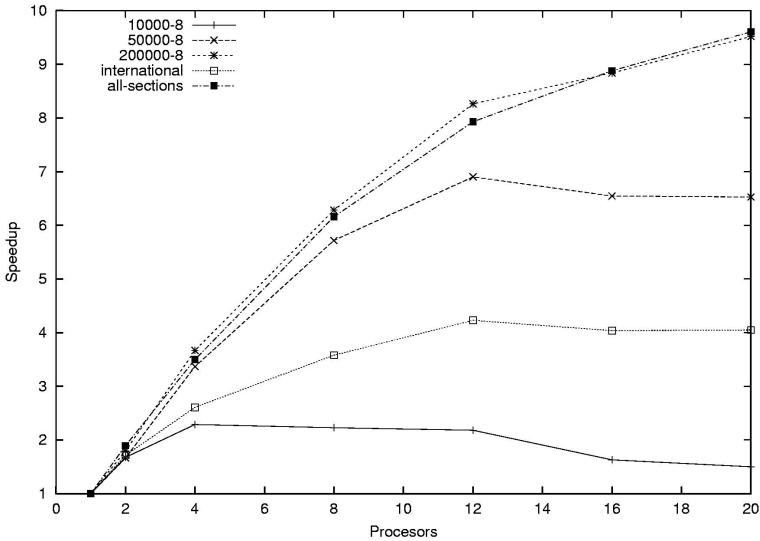


Fig. 1. Speedups of the synthetic and document data sets.

Figure 1 shows the speedup behavior of the document data sets and the synthetic data when its size is varied in a 8 dimensional data space. From the plot it can be seen that we have achieved near linear speedups for up to a certain number of processors depending on the data size. This is due to the algorithm being heavily data parallel and the computation time decreasing linearly with the increase in the number of processors. Parallelism reduces computation time for up to a certain number of processors depending on the data size, after which

communication overhead overshadows computational gain. The higher the data size, the greater the speedup for the same number of processors. Although the document data size is small compared to the our synthetic data sets, the obtained speedups are larger. We think that this behaviour is due to the computation time of the similarity measure between documents being much greater than the computation time of the similarity measure between objects.

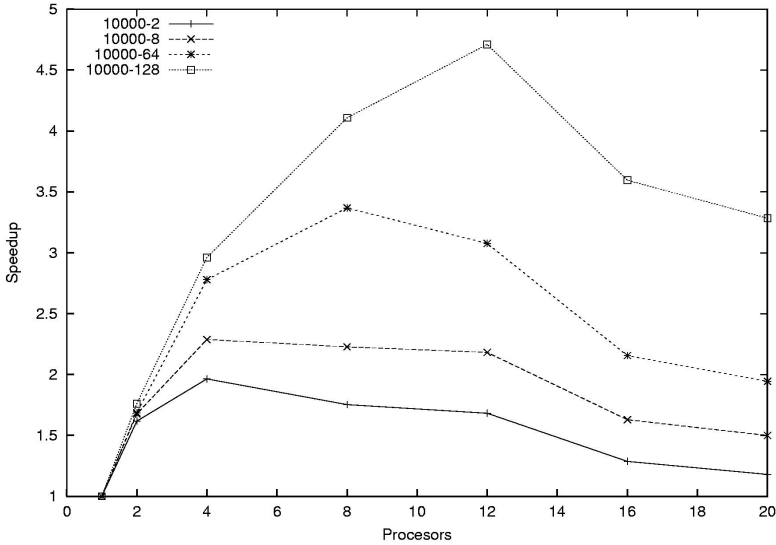


Fig. 2. Speedups of the synthetic data with a fixed data size.

Further, we set the data size and vary the dimensionality of the data. As it can be noticed in Figure 2, the speedup increases with the growth of the data dimensionality for the same number of processors. The data size taken in this experiment was small. If a similar experiment with larger data size is carried out, greater speedups are obtained and the maximum speedup for a fixed data is reached in a bigger number of processors.

5 Conclusions

In this paper we presented an efficient parallel algorithm that implements an incremental method to determine the compact sets in a collection of objects. The generated set of clusters is unique, independently of the objects arrival order of the objects. Another advantage of this algorithm is that it can deal with mixed incomplete object descriptions. On the other hand, the algorithm is not restricted to the use of metrics to compare the objects. The proposed parallel algorithm can be used in many problems such as the event detection

tasks, in the knowledge discovery problems and others. Besides, the resulting parallel algorithm is portable, because it is based on standard tools, including the message-passing library MPI.

We have implemented and tested the parallel code in a cluster of personal computers. The experimental evaluations on a variety of synthetics and real data sets with varying dimensionality and data sizes show the gains in performance. The obtained results show a good behaviour of the parallel algorithm that clearly reduces the sequential time. Moreover, we have achieved near linear speedups when the collection of objects and the number of features are large. The main reason for this behaviour is that we have tried to minimize the communications and to balance the load on the processors by carefully distributing the objects and the tasks that each processor performs during each step of the algorithm.

References

1. Dhillon, I. and Modha, B. A.: Data Clustering Algorithm on Distributed Memory Multiprocessor. Workshop on Large-scale Parallel KDD Systems (2000) 245–260.
2. Nagesh, H., Goil, S. and Choudhary, A.: A Scalable Parallel Subspace Clustering Algorithm for Massive Data Sets. International Conference on Parallel Processing (2000) 447–454.
3. Gil-García, R. and Badía-Contelles, J.M.: GLC Parallel Clustering Algorithm. In Pattern Recognition. Advances and Perspectives. Research on Computing Science CIARP'2002 (In Spanish), México D.F., November (2002) 383–394.
4. Hill, D. R.: A vector clustering technique. Samuelson (ed.), Mechanized Information Storage, Retrieval and Dissemination, North-Holland, Amsterdam (1968).
5. Walls, F., Jin, H., Sista, S. and Schwartz, R.: Topic Detection in Broadcast news. Proceedings of the DARPA Broadcast News Workshop (1999) 193–198.
6. Aslam, J., Pelekhev, K. and Rus, D.: Static and Dynamic Information Organization with Star Clusters. Proceedings of the 1998 Conference on Information Knowledge Management CIKM 98, Baltimore, MD (1998).
7. Pons-Porrata, A., Ruiz-Shulcloper, J., Berlanga-Llavori, R. and Santiesteban-Alganza, Y.: An Incremental Clustering Algorithm to find partitions with mixed data. In Pattern Recognition. Advances and Perspectives. Research on Computing Science, CIARP'2002 (In Spanish). México D.F., November (2002) 265–276.
8. Pons-Porrata, A., Berlanga-Llavori, R. and Ruiz-Shulcloper, J.: Detecting events and topics by using temporal references. Lecture Notes in Artificial Intelligence 2527, Springer Verlag (2002) 11–20.

Preventive Multi-master Replication in a Cluster of Autonomous Databases*

Esther Pacitti¹, M. Tamer Özsu², and Cédric Coulon¹

¹Atlas Group, INRIA and IRIN, Nantes – France
{Esther.Pacitti, Cedric.Coulon}@irin.univ-nantes.fr

²University of Waterloo – Canada
tozsu@uwaterloo.ca

Abstract. We consider the use of a cluster of PC servers for Application Service Providers where applications and databases must remain autonomous. We use data replication to improve data availability and query load balancing (and thus performance). However, replicating databases at several nodes can create consistency problems, which need to be managed through special protocols. In this paper, we present a lazy preventive data replication solution that assures strong consistency without the constraints of *eager* replication. We first present a peer-to-peer cluster architecture in which we identify the replication manager. Cluster nodes can support autonomous, heterogeneous databases that are considered as black boxes. Then we present the multi-master refresher algorithm and show all system components necessary for implementation. Next we describe our prototype on a cluster of 8 nodes and experimental results that show that our algorithm *scales-up* and introduces a negligible loss of *data freshness* (almost equal to mutual consistency).

1 Introduction

Clusters of PC servers provide a cost-effective alternative to tightly-coupled multiprocessors. They make new businesses such as Application Service Providers (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need to be available, typically through the Internet, as efficiently as if they were local to the customer site. Thus, the challenge is to fully exploit the cluster's query parallelism and load balancing capabilities to improve performance. The typical solution is to replicate applications and data at different nodes so that users can be served by any of the nodes depending on the current load. This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. This solution has been successfully used by, for example, Web search engines using high-volume server farms (e.g., Google). However, Web sites are typically read-intensive which makes it easier to exploit parallelism. In the ASP context, the problem is far more difficult [1] since applications can be update-intensive. Replicating databases at several nodes, so they can be accessed by different users through the same or different applications in parallel, can create consistency problems [4].

* Work partially funded by the Leg@net RNTL project.

Nevertheless, replication is the best solution to efficiently manage high transaction loads by exploiting parallelism.

In this paper, we present a data replication solution that assures strong consistency for parallel query processing in a cluster of autonomous databases. There are two basic approaches to manage data replication: *eager* and *lazy*. With *Eager replication* (e.g. Read-One-Write All – ROWA [9]) whenever a transaction updates one replica, all other replicas are updated inside the same transaction. Therefore mutual consistency of replicas and *strong consistency*¹ are enforced; however it violates system autonomy.

With *lazy replication* [4], a transaction can commit after updating one replica copy at some node. After the transaction commits, the updates are propagated to the other replicas, and these replicas are updated in separate transactions. Hence, the property of mutual consistency is relaxed and strong consistency is eventually assured. A major virtue of lazy replication is its easy deployment. In addition, lazy replication has gained considerable pragmatic interest because it is the most widely used mechanism to refresh data in several emerging distributed application environments [7]. In this paper, we focus on a particular lazy replication scheme, called *multi-master*, adapted for PC clusters.

In lazy replication, a *primary copy* is stored at a *master* node and *secondary copies* are stored in *slave* nodes. A primary copy that may be stored at and updated by different master nodes is called a *multi-owner copy*. These are stored in *multi-owner* nodes and a *multi-master* configuration² (or *lazy group*) consists of a set of multi-owner nodes on a common set of multi-owner copies.

This paper makes several contributions. First, we introduce architecture for processing user requests to applications into the cluster system and discuss our general solutions for submitting transactions and managing replicas. Second, we propose a multi-master refresher algorithm that *prevents* conflicts, by exploiting the cluster's high speed network, thus providing strong consistency, without the constraints of eager replication. We also show the architectural components necessary to implement the algorithm. Third, we describe the implementation of our algorithm over a cluster of 8 nodes and present experimental results that show that it scales-up and introduces a negligible loss of data freshness.

The rest of the paper is structured as follows. Section 2 presents the ASP cluster architecture in which we identify the role of the replication manager. Section 3 presents the refreshment management issues including the principles of the refresher algorithm and the architectural components necessary for its implementation. Section 4 describes our prototype over a cluster of 8 nodes and show our performance model and experiments results. Section 5 compares our work with the most relevant related work. Finally, section 6 concludes the paper.

¹ For any two nodes, the same sequence of transactions is executed in the same order.

² In addition to the multi-master that we consider in this paper, there are other lazy cluster configurations such as lazy master and hybrid. Our research addresses these as well, but space limitations force us to focus only on multi-master.

2 Multi-master Cluster Architecture

In this section, we discuss the system issues and solutions when using the multi-master refreshment management to refresh multi-owner copies in a cluster of replicated databases. We introduce the architecture for processing user requests against applications into the cluster system and discuss our general solutions for placing applications, submitting transactions and managing replicas. Therefore, the replication layer is identified together with all other general components.

Shared-nothing is the only architecture that supports sufficient node autonomy without the additional cost of special interconnects. Thus, we exploit a shared-nothing architecture. Each cluster node is composed of five layers (see Figure 1): Request Router, Application Manager, Transaction Load Balancer and Replication Manager. We discuss these in the following. A request may be a query or update transaction on a specific application.

The general processing of a user request is as follows. When a user request arrives at the cluster, traditionally through an access node, it is sent randomly to a cluster node. There is no significant data processing at the access node to avoid bottlenecks. Within that cluster node, the user is authenticated and authorized through the Request Router, available at each node, using a multi-threaded *global user directory* service. Notice that user requests are managed completely asynchronously. Next, if a request is accepted, then the Request Router chooses a node j , to submit the request. The choice of j involves selecting all nodes in which the required application is available, and, among these nodes, the node with the lightest load. Therefore, eventually i may be equal to j . The *Request Router* then routes the user request to an application node using a traditional load balancing algorithm.

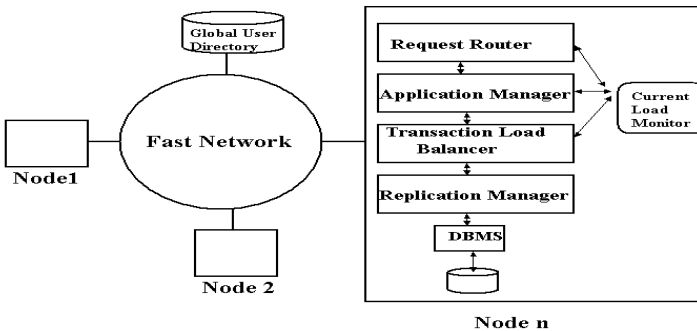


Fig. 1. Cluster Architecture

Notice, however, that the database accessed by the user request may be placed at another node k since applications and databases are both replicated and not every node hosts a database system. In this case, the choice regarding node k will depend on the cluster configuration and the database load at each node.

A node load is specified by a *current load monitor* available at each node. For each node, the load monitor periodically computes application and transaction loads using

traditional load balancing strategies. For each type of load, it establishes a load grade and multicasts the grades to all the other nodes. A high grade corresponds to a high load. Therefore, the Request Router chooses the best node for a specific request using the node grades.

The *Application Manager* is the layer that manages application instantiation and execution using an application service provider. Within an application, each time a transaction is to be executed, the *Transaction Load Balancer* layer is invoked which triggers transaction execution at the best node, using the load grades available at each node. The “best” node is defined as the one with lightest transaction load.

The *Replication Manager layer* manages access to replicated data and assures strong consistency in such a way that transactions that update replicated data are executed in the same serial order at each node. Transaction execution is managed by the local transaction of each DBMS. Therefore the ACID (atomicity, consistency, isolation, durability) properties [9] are ensured. The Replication Manager signals to the Transaction load balancer whenever a transaction commits or abort. Afterwards the transaction load balancer signals the Application Manager the same event. We employ data replication because it provides database access parallelism for applications. Our preventive replication approach, implemented by the multi-master refreshment algorithm, avoids conflicts at the expense of a forced waiting time for transactions, which is negligible due to the fast cluster network system.

3 Multi-master Refreshment

In this section, we present, in more detail, multi-master refreshment with its algorithm and system architecture. In a first step we consider fully replicated multi-master configurations³. We assume that the network interface provides a global FIFO reliable multicast: messages multicast by one node are received at the multicast group nodes in the order they have been sent. We denote by Max , the upper bound of the time needed to *multicast* a message from a node i to any other node j [3]. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a drift and to be ϵ -synchronized [3]. This means that the difference between any two correct clocks is not higher than the ϵ (known as the *precision*). We now describe the multi-master refresher algorithm.

To define the algorithm, we need a formal correctness criterion to define strong consistency. In *Multi-Master* configurations, inconsistencies may arise whenever the serial orders of two multi-owner transactions at two nodes are not equal. Therefore, multi-owner transactions must be executed in the same serial order at any two nodes. Thus, Global FIFO Ordering is not sufficient to guarantee the correctness of the refreshment algorithm. Hence the following correctness criterion is necessary:

Definition 3.1 (*Total Order*). Two multi-owner transactions MOT_1 and MOT_2 are said to be executed in *Total Order* if all multi-owner nodes that commit both MOT_1 and MOT_2 commit them in the same order.

³ N multi-master nodes. Each node stores the same set of multi-owner copies.

Proposition 3.1. *For any cluster configuration C that meets a multi-master configuration requirement, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

We now present the refresher algorithm. Similar to [8], each multi-owner transaction is associated with a chronological time stamp value. The principle of the multi-master refresher algorithm is to submit a sequence of multi-owner transactions in chronological order at each node. As a consequence, total order is enforced since multi-owner transactions are executed in same serial order at each node. To assure chronological ordering, before submitting a multi-owner transaction at node i , the algorithm has to check whether there is any older committed multi-owner transaction enroute to node i . To accomplish this, the submission time of a new multi-owner transaction MOT_i is delayed by $Max + \varepsilon$ (recall that Max is the upper bound of the time needed to multicast a message from a node to any other node). After this delay period, all older transactions are guaranteed to be received at node i . Thus chronological and total orderings are assured.

However, different from [8], for the multi-master configuration, whenever a multi-owner transaction MOT_i is to be triggered at some node i , the same node multicasts MOT_i to all nodes $1, 2, \dots, n$, of the multi-owner configuration, including itself. Once MOT_i is received at some other node j (notice that i may be equal to j), it is placed in the *pending queue* for the multi-owner triggering node i , called multi-owner pending queue (noted moq_i). Therefore, at each multi-owner node i , there is a set of multi-owner queues, $moq_1, moq_2, \dots, moq_n$. Each pending queue corresponds to a node of the multi-owner configuration and is used by the refresher to perform chronological ordering.

To implement the multi-master refresher algorithm in a multi-owner node we assume that six components are added to a regular DBMS in order to support lazy replication. Our goal is to maintain node autonomy since the implementation solution does not require the knowledge of system internals. We also assume that it is known whether users write on multi-owner copies. Figure 2 shows all architectural components necessary to implement our algorithm. The *Replica Interface* manages the incoming multi-owner transaction submission. The *Receiver* and *Propagator* implement reception and propagation of messages, respectively. The *Refresher* implements the multi-master refreshment algorithm. Finally, the *Deliverer* manages the submission of multi-owner transactions to the local transaction manager in FIFO order using a *running queue*. In the following we focus on the Replica Interface and Refresher components.

Multi-owner transactions are submitted through the Replica Interface. The application program calls the Replica Interface passing as parameter the multi-owner transaction MOT . The Replica Interface then establishes a timestamp value C for MOT , that corresponds to MOT start submission time. Afterwards, the sequence of operations of MOT is written in the *Owner Log* followed by C . Whenever the multi-owner transaction commits, the *Deliverer* notifies the Replica Interface. Meanwhile, the Replica Interface waits for MOT commitment.

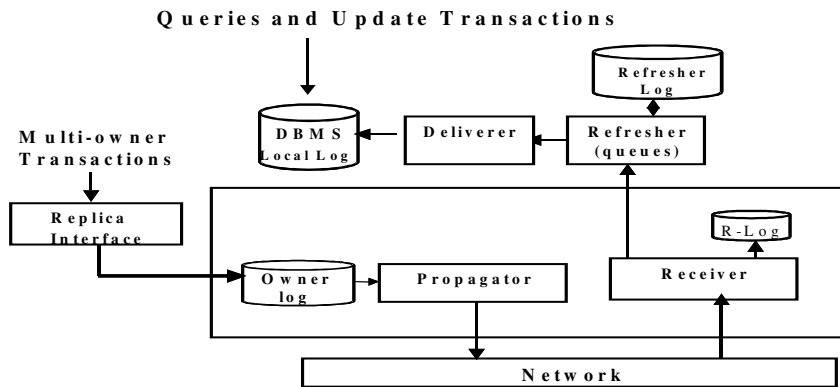


Fig. 2. Multi-owner node Architecture

The Refresher continuously reads the contents of the multi-owner pending queues. Each pending queue i stores a sequence of multi-owner transactions in multicast order. Therefore the contents of these pending queues form the input for the refresher algorithm. Whenever a *MOT* is selected for submission the refresher inserts it into the *running queue*, which contains all ordered transactions that are not yet entirely executed. The Deliverer then manages the submission of multi-owner transactions to the local transaction manager in the order in which they were inserted into the running queue.

4 Validation

To validate our solution to multi-master refreshment, we implemented the refresher algorithm and architecture in a cluster system. We use this implementation to study the refresher's behavior in terms of *scalability* [10] and *data freshness*. In this section, we briefly describe our implementation and present our performance model and experimentation results.

We did our implementation on a cluster of 5 nodes, where one node is used exclusively as the access point to the cluster. Each node is configured with a 2GHz Pentium 4 processor, 512 MB of memory and 40GB of disk. The nodes are linked by a 1 Gb network. We use Linux Mandrake 8.0/Java and Spread toolkit that provides a reliable FIFO message bus and high performance message service among the cluster nodes that is resilient to faults. We use Oracle8i as the underlying database system at each node. In our implementation we consider four modules: *Client*, *Replicator*, *Network* and *Database Server*. The Client module simulates the clients of the ASP. It submits multi-owner transactions randomly to any cluster node, via RMI-JDBC, which implement the Replica Interface. Each cluster node hosts a Database Server and at least one instance of the Replicator module. The Replicator module implements all system components necessary for a multi-owner node: Replica Interface, Propagator, Receiver, Refresher and Deliverer. Notice that each time a transaction is to be executed it is first sent to the Replica Interface which checks whether the incoming transaction writes on replicated data. Whenever a transaction does not write

on replicated data it is sent directly to the local transaction manager. Even though we do not consider node failures in our performance evaluation, we implement all the necessary logs for recovery to understand the complete behaviour of the algorithm. The Network module interconnects all cluster nodes through the Spread toolkit.

Our performance model takes into account multi-owner transaction size, arrival rates and the number of multi-owner nodes. We vary multi-owner transactions' arrival rates (noted λ). We define two types of multi-owner transactions (denoted by $|MOT|$), defined by the number of write operations. Small transactions have size 5, while long transactions have size 50. To understand the behaviour of our algorithm in the presence of short and long transactions, we define four scenarios. Each scenario is defined in terms of a parameter called *long transaction ratio* (denoted by ltr). We set ltr as follows: $ltr = 0$ (all update transactions are short), $ltr = 30$ (30% of update transactions are long), $ltr = 60$ (60% of update transactions are long) and $ltr = 100$ (all update transactions are long). Updates are done on the same attribute of a different tuple. $Nb-Inst$ defines the number of multi-owner instances i (Replicator instances) on the same node. That is, since our cluster has 4 nodes, we let the same cluster node execute more than one Replicator instance to be able to simulate up to 8 multi-owner nodes. Hence, $Nb-Multi-Owner$ defines the number of multi-owner nodes. To simulate 8 multi-owners, each cluster node carries two Replicator instances. Each node may execute at most 2 multi-owner node instances because the delay introduced by memory management may compromise our results. For fairness, we always consider an equal number of multi-owner instances at each node. Finally our single table schema has two attributes and carry 100000 tuples and no indexes. The parameters of the performance model are described in Table 1.

Table 1. Performance parameters

<i>Param.</i>	Definition	Values
$ MOT $	Transaction size	5; 50
ltr	Long transaction ratio	0; 30; 60; 100%
$Nb-Multi-Owner$	Number of Replicators	2; 3; 4; 8
$Nb-Inst$	Number of Replicators on 1 node	1, 2
λ	Mean time between transactions (workload)	bursty: 200ms; low: 20s
M	Time of the test	60000ms; 600000ms
$Max + \epsilon$	Submission time of a multi-owner transaction is delayed	100ms

We describe two experiments to study scalability and freshness. The first experiment studies the algorithm's scalability. That is, for a same set of incoming multi-owner transactions, scalability is achieved whenever in augmenting the number of nodes the response times remain the same. We consider *bursty* and *low* workloads ($\lambda = 200ms$ and $\lambda = 20s$), varying the number of multi-master nodes (2, 4 and 8). In addition, we vary the ratio of long transactions ($ltr = 0/30/60/100$). Finally, we perform this experiment during a time interval of 60000ms.

This experiment results (see Figure 3.a and 3.b) clearly shows that for all ltr values scalability is achieved. In fact, for each ltr value the response times⁴ correspond to the average transaction response time. We can clearly see in Figure 3 that the difference in response times is a function of the *long transaction ratio* (ltr). The higher the ltr

⁴ Time spent to execute a transaction submitted by a client (involves the Multi-Master Refresher Algorithm).

value, the higher are the response times, because, due to the size of transactions and the *bursty* (Figure 3.a) workload, the running queue (recall that the running queue is used by the Deliverer to submit multi-owner transactions), becomes a bottleneck for transaction processing, since transaction submission is done sequentially. As a consequence the average response times are much higher compared to the low workload scenario (Figure 3.b). However, the algorithm still scales-up since the network delay introduced to multicast a message is very small, avoiding network contention. Furthermore, in contrast with synchronous algorithms, our algorithm has linear response time behaviour since it requires only the multicast of n messages for each incoming multi-owner transaction, where n corresponds to the number of nodes.

The second experiment analyzes the degree of freshness. Suppose $\Pi = \{i, j, k, \dots\}$ is a set of multi-owner nodes of R^5 . Intuitively, the degree of freshness of R on node i over Π is defined as the difference between the number of committed multi-owner transactions on R at node i and the average of number of committed multi-owner transactions on others multi-owner nodes ($\Pi - i$).

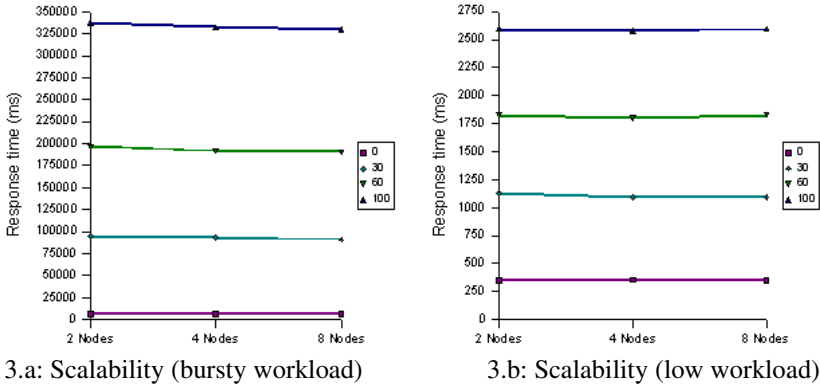


Fig. 3. Performance Results

Thus, the best degree of freshness is 1, which means that multi-owner copies are *mutually consistent*. Recall that the mutual consistency property is assured by eager algorithms. On the other hand, degrees of freshness close to zero express the fact that there are a significant number of updates committed at some multi-owner copy that were still not performed in others copies.

We consider *bursty* workloads ($\lambda = 200\text{ms}$) varying the number of multi-owner nodes ($Nb\text{-}Multi\text{-}Owner = 2/4/8$) and the long transactions ratio ($ltr = 0/30/60/100$). Finally, we perform this experiment during a time interval of 60000ms. The freshness degrees for the 4 scenarios are almost equal (between 0.98 and 0.99). In all cases very good degrees of freshness are attained (very close to 1). These results clearly demonstrate that the time delay introduced to submit transactions at each node is the same, because all nodes executed the same sequence of multi-owner transactions. As a result, the transaction processing speed at each node is uniform, independent of the number of nodes and the ratio of long transactions. Notice however that we do not

⁵ Each node holds a multi-owner copy R.

consider query loads. Nevertheless, we can safely assume that query loads do not impact freshness degrees whenever the underlying database server does not use strict two-phase-locking concurrency protocol [9]. That is, queries do not block multi-owner transactions. This experiment shows that our algorithm, in a cluster environment, does not introduce any significant loss of data freshness. In fact, it almost provides mutual consistency.

5 Related Work

There are several interesting projects that deal with replicated data management in cluster architectures: PowerDB [2], GMS [11], and TACT [13]. However, none of them employ asynchronous multi-master replication as we do. A general framework for optimistic and preventive data replication for the same ASP cluster system is presented in [1]. As an evolution of [1], we propose, in this paper, a cluster architecture that enables user requests to be managed completely asynchronously in order to avoid any type of bottleneck. In addition, we detail the preventive refreshment management, showing the implementation environment and some performance results. In the following, we compare our replication solution with other existing solutions.

With synchronous (or eager) replication, the property of mutual consistency is assured. A solution proposed in [5] reduces the number of messages exchanged to commit transactions compared to 2PC, but the protocol is still blocking and it is not clear if it scales up. It uses, as we do, communication services [6] to guarantee that messages are delivered at each node according to some ordering criteria.

Recently, [8] proposed a refreshment algorithm that assures strong consistency for lazy master-based configurations. They do not consider the multi-master configuration as we do. Our solution uses the same principle of their refreshment algorithm. However, in our work, we show how it can be employed in a multi-master configuration.

6 Conclusion

In this paper, we proposed a preventive lazy multi-master replication database solution in a cluster system for ASP management. In this context, data replication is used to improve data availability and query load balancing (and thus performance). In this paper we first proposed a cluster architecture that enables users' requests to be managed completely asynchronously in order to avoid any type of bottleneck. Second, we proposed a multi-master refresher algorithm that *prevents* conflicts, by exploiting the cluster's high speed network, thus providing strong consistency, without the constraints of eager replication. Cluster nodes can support autonomous, heterogeneous databases that are considered as black boxes. Thus our solution achieves full node autonomy. We also showed the system architecture components necessary to implement the refresher algorithm. Finally, we described our experiments over a cluster of 8 nodes. In our experimental results, we showed that the

multi-master algorithm scales-up and introduces a negligible loss of data freshness (almost equal to mutual consistency).

For future work, we will propose some optimizations for the refresher algorithm in managing multi-owner transactions on the running queue. For instance, for *bursty* workloads one can check whether ordered transactions in the running queue do not conflict. If there is no conflict, these transactions can be triggered concurrently. In addition, we plan to consider other types of multi-master configurations.

References

1. S. Gançarski, H. Naacke, E. Pacitti, P. Valduriez: Load Balancing of Autonomous Applications and Databases in a Cluster System, Parallel Processing with Autonomous Databases in a Cluster System, *Int. Conf. of Cooperative Information Systems (CoopIS)*, 2002.
2. T. Grabs, K. Böhm, H. Scheck: Scalable Distributed Query and Update Service Implementation for XML Documents Elements, *IEEE RIDE Int. Workshop on Document Management of Data Intensive Business and Scientific Applications*, 2001.
3. L. George and P. Minet: A FIFO worst analysis for a hard real time distributed problem with consistency constraints, *Int. Conf. On Distributed Computing Systems, Baltimore (ICDCS97)*, 1997.
4. J. Gray and P. Helland and P. O'Neil and D. Shasha: The Danger of Replication and a Solution, *ACM SIGMOD Int. Conf. on Management of Data, Montreal*, 1996.
5. B. Kemme, G. Alonso: Don't be lazy be consistent: Postgres-R, a new way to implement Database Replication, *Int. Conf. on Very Large Databases (VLDB)*, 2000.
6. V. Hadzilacos and S. Toueg: A Modular Approach to Fault-Tolerant Broadcasts and Related Problems, *Technical Report TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY 14853*, 1994.
7. E. Pacitti, P. Valduriez: Replicated Databases: concepts, architectures and techniques, *Network and Information Systems Journal, Hermès*, 1(3), 1998.
8. E. Pacitti, P. Minet, E. Simon: "Replica Consistency in Lazy Master Replicated Databases". *Distributed and Parallel Databases, Kluwer Academic*, 9(3), May 2001.
9. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1999.
10. P. Valduriez: Parallel Database Systems: open problems and new issues. *Int. Journal on Distributed and Parallel Databases*, 1(2), 1993.
11. G. M. Voelker et. al: Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System, *In ACM Sigmetrics Conf. on Performance Measurement, Modeling, and Evaluation*, 1998.
12. U. Röhm, K. Böhm, H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. *Int. Conf. on Data Engineering (ICDE)*, 2001.
13. H. Yu, A. Vahdat: Efficient Numerical Error Bounding for Replicated Network Service, *Int. Conf. on Very Large Databases (VLDB)*, 2000.

Pushing Down Bit Filters in the Pipelined Execution of Large Queries^{*}

Josep Aguilar-Saborit, Victor Muntés-Mulero, and Josep-L. Larriba-Pey^{**}

Departament d'Arquitectura de Computadors
CEPBA-IBM Research Institute
Universitat Politècnica de Catalunya - Campus Nord UPC
C/Jordi Girona, 1-3 08034 Barcelona
Mòdul D6 Despatx 005
{jaguilar, vmuntes, larri}@ac.upc.es

Abstract. We propose a new strategy to use Bit Filters for complex pipelined queries on large databases that we call Pushed Down Bit Filters. The objective of the strategy is to make use of the Bit Filters already created for upper nodes of the execution plan, in the leaves of the plan. The aim of this strategy is to reduce the traffic between the nodes of the execution plan. When traffic is reduced, the amount of CPU work is reduced and, in most of the cases, I/O is also reduced. In addition, this technique shows no degradation in cases with little effectiveness.

1 Introduction

The execution time of queries to large relational databases is greatly influenced by the I/O. However, the pipelined nature of execution engines makes the task of minimizing the I/O very complex, especially for large multi-join queries.

There are many strategies to reduce the I/O of multi-join queries like *hash teams* [9], the use of bitmaps to implement joins [10] or the use of Bloom Filters, also called Bit Filters [1].

In this paper we deal with Bit Filters. They are an efficient tool to reduce the amount of I/O in the execution of pipelined queries. In particular, they are used in commercial DBMSs like DB2 [6], Oracle [5] and SQLServer [6] to reduce the I/O of Hash Join operations.

In those commercial DBMSs, Bit Filters are created during the build phase of the Hash Join operation. Then, during the probe phase, they are used to filter out records of the probe Relation. Quite interestingly, they may reduce considerably the amount of I/O of the Hash Join nodes [12] at the cost of keeping a bit map structure in memory during a part of the query execution. In addition, Bit Filters adapt naturally to the execution of Hash Joins; they are created during

^{*} This work was supported by the Ministry of Education and Science of Spain under contract TIC2001-0995-C02-01, CEPBA, CIRI and an IBM CAS Fellowship grant.

^{**} The authors want to thank Calisto Zuzarte and Adriana Zubiri for their helpful comments in preliminary versions of this paper.

the build phase, which is sequential by nature, and are used during the probe phase, which is pipelined by nature.

In this paper we want to take a step further in the use of Bit Filters. We explain and analyze how, being created for their local use in one Join node, they can be used by the Scan nodes of pipelined execution plans. We call these Pushed Down Bit Filters (PDBF) because, with our strategy, Bit Filters are Pushed Down to the leaves of execution plans.

We evaluate Pushed Down Bit Filters with queries implemented in PostgreSQL [11] on a 1 GB TPC-H populated database. We measure the amount of data traffic among nodes. We also measure the I/O and execution time for different join heap sizes. While the data traffic among nodes is independent of the DBMS, the I/O depends on the amount of memory available to perform the query and the execution time depends on the platform used and the DBMS.

We compare the Pushed Down Bit Filters to the basic PostgreSQL execution and the classic use of Bit Filters. The results show that our strategy reduces the execution time of some queries in more than a 50% both compared to the basic implementation and to the classic use of Bit Filters.

This paper is organized as follows. Section 2 describes our strategy, Pushed Down Bit Filters, explains in detail when the strategy can be used and the implementation details of the strategy in the execution engine. In Section 3 we describe and analyze the results obtained. Section 4 describes the previous work in the area. Finally, in Section 5 we conclude.

2 Pushed Down Bit Filters

The objective of Pushed Down Bit Filters is to obtain the maximum benefit from the Bit Filters created in the Join nodes of a query. In this paper we concentrate on Hash Joins and the Bit Filters created by them although our strategy could be generalized to other join operations with little changes.

PDBF are as simple as using the Bit Filters generated in the Hash Join nodes of the query plan in the lower leaves of the plan. This will be possible in those leaf nodes scanning a Relation with attributes on which the already created Bit Filters can be applied. Even the attributes that have non-functional dependencies with the attribute used to create the Bit Filter can be filtered.

The aim of this early elimination of tuples is to reduce the amount of data to be processed by the plan, and consequently, to decrease the work of its nodes.

Figure 1 shows a simple example of the strategy. The upper Hash Join operation starts with the build phase creating a Bit Filter on attribute a of relation $R3$. Once the build phase is finished, the join node invokes the probe subtree to obtain tuples. In the example, the subtree is formed by a join on attribute b , and two Scan nodes. When the Scan node of relation $R2$ is invoked, it will look up the Bit Filter given that Relation $R2$ contains attribute a . With this, we reduce the output cardinality of the Scan node, thus, the volume of data to be processed by join node " $R1.b = R2.b$ ". Also, the output cardinality of join node " $R1.b = R2.b$ " is reduced and, consequently, the amount of data to be processed

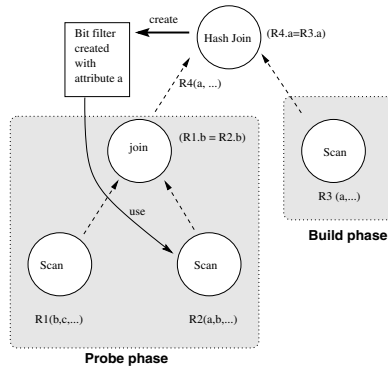


Fig. 1. An example of the Pushed Down Bit Filters.

by the Hash Join node. If R1 contained attribute a , the Scan of R1 could also look up the Bit Filter. This would happen in the case of composed primary keys.

Our strategy does not require the creation of any special large data structure. The only information necessary in the Scan nodes is a link between each attribute and its corresponding Bit Filter.

Applicability. The special characteristics of Bit Filters and query plans impose, at least, the following restrictions to our strategy:

- A Scan node can only look up the Bit Filters created by Hash Join operations within the same **select** statement. Two different **select** statements have data flow streams that do not interfere.
- A Bit Filter can only be pushed down to the Scan nodes of the subtree that belongs to the probe Relation of that Hash Join.

Implementation details. Once the target Scan nodes are chosen, it is necessary that those Scan nodes maintain some information about the Bit Filters they have to look up and the attribute affected by each Bit Filter. We define the set S^* of pairs <Bit Filter, attribute> as the data used by a Scan node to apply the Pushed Down Bit Filters. With this information, for every tuple processed by a Scan node, each Bit Filter of the set S^* will be applied to the corresponding attribute. If any of the Bit Filters returns a zero for a tuple, that tuple can be immediately discarded.¹

It is important to note the low cost of our strategy:

- Memory space: It is only necessary to keep the set S^* for each Scan node. Thus, the memory space necessary is not significant at run time.
- CPU time: If the Bit Filters are looked up during the Scan operations, it will not be necessary to look them up during the probe phase of any of the upper Hash Joins. Thus, our strategy does not imply any replication of work.

¹ All the Bit Filters are accessed with one only hash function, h

3 Evaluation

In this section we present different results. First, we evaluate a query with different parameter sets to show the features of our strategy. The evaluation results measure traffic volume and execution time. Then, we give measures for the I/O volume using one of the parameter sets. Before starting with the detailed evaluation, we explain the query that we analyze and the software and hardware setups we used.

Query description. We have executed a synthetic query with four different parameter sets on a TPC-H populated database. Figure 2.a shows the query used and Figure 2.b shows its execution plan. The query counts the number of units of each product of brand z (“p_brand = z”) sold starting on date x (“o_orderdate > x”) for available quantities of stock larger than y (“ps_availqty > y”). We have varied the query restrictions to obtain different behaviors of the same query. This leads to the four different parameter sets that we call query executions, shown in Table 1.

It is frequent to have queries with several joins on a central table (called Star Joins) as it happens here on *Lineitem*. This does not mean that our strategy can only be applied to this type of queries (see Section 2 for the applicability of our strategy). However, this is a good example for the evaluation of our technique.

Figure 2.b also shows (in bold) the application of the PDBF in this query. There, we can see how the Scan on *Lineitem* looks up the Bit Filters of HJ1, HJ2 and HJ3, while the Scan on *Part* only looks up the Bit Filter of HJ2.

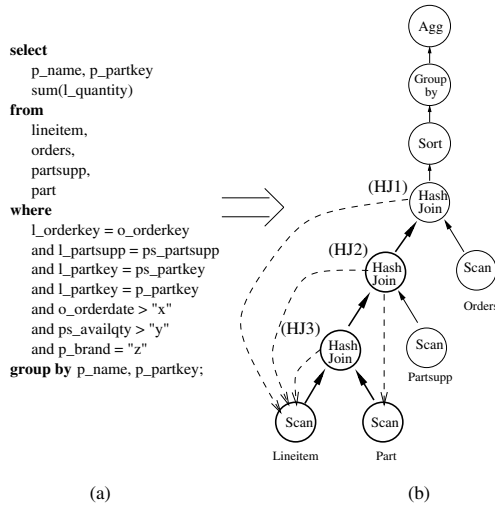


Fig. 2. Query used (a), execution plan and use of Bit Filters (b).

Table 1. Percentage of tuples eliminated by each combination of parameters and efficiency of the Bit Filters. (P) for Part, (PS) for Partsupp and (O) for Orders.

set #	Fraction of Tuples eliminated			Efficiency of Bit Filter		
	P."z"	PS."y"	O."x"	HJ3	HJ2	HJ1
1	0.0	0.7	0.8	0.0	0.16	0.81
2	0.0	0.7	0.15	0.0	0.16	0.18
3	0.8	0.15	0.15	0.81	0.0	0.18
4	0.0	0.15	0.15	0.0	0.0	0.18

We have used restrictions "x", "y" and "z", to control the amount of tuples projected from the build Relation of each Hash Join. Table 1 shows the fraction of tuples eliminated with each restriction for each different query execution. The restriction is one of the factors determining the effectiveness of a Bit Filter. The efficiency of the Bit Filters (fraction of tuples filtered out from the scan where the Bit Filter is applied) is also shown in Table 1 for each Hash Join. We have computed these figures using the data obtained after the execution of the query.

Software Setup. We have generated a TPC-H database with scale factor 1 (1 GB) and executed the 4 queries on it.

We have used PostgreSQL to implement the use of Bit Filters and PDBF. Therefore, we have used three versions of the PostgreSQL query engine; the original engine without Bit Filters, the engine with Bit Filters and the engine with PDBF. The size of the Bit Filters we use is larger than the number of distinct values of the attributes used to build the Filters.

The implementations do not include any modification of the Optimization engine. Instead, they only implement the query plan and the modifications of PostgreSQL necessary to execute the query that we designed for the tests.

PostgreSQL decides the number of partitions in the Hybrid Hash Join at optimization time. With the application of our strategy, some Hybrid Hash Join nodes end up having a very small number of tuples in the build Relation. This means that the number of partitions could be smaller than planned by the optimizer of PostgreSQL. However, our implementations keep the number of partitions planned by the optimizer at the cost of making our strategy less efficient.

The memory space assigned to each Hash Join node is called *Sortmem* in PostgreSQL. For the following analysis, we use different *Sortmem* sizes ranging from very restrictive to a size that allows the execution of Hash Joins in memory with one only data read per Relation. The *Sortmem* sizes range from 5 MB (very restrictive) to 60 MB (one data read). The Bit Filters that we have implemented are always used unless we know its effectiveness is zero, in which case, they are not created. The Bit Filters used are stored in a *Bit Filter Heap* that we created.

Hardware Setup

We run all the experiments on a 550M Hz Pentium-III based computer, with 512 MB of main memory and a 20 GB IDE Hard Disk. The Operating System used was Linux Red Hat 6.2.

Traffic and Execution Time

Figure 3 shows results of execution time, and tuple traffic. The plots show results for the basic implementation of PostgreSQL, for the use of Bit Filters in the nodes where they are created, and PDBF. The execution time plots show results for different *Sortmem* sizes while the traffic plots show the total traffic per node in the execution plan (the traffic is independent from the *Sortmem* size). The traffic of a node is equivalent to the output cardinality of that node. It is important to note that only the traffic of nodes HJ3, HJ2, Scan Lineitem and Scan Part are affected by PDBF.

While the execution time is specific to the PostgreSQL implementation, the traffic avoided depends on the data and can be regarded as a general result of applicability to any DBMS.

We divide the analysis into two groups of queries.

Query executions #1 and #2: These are the query executions that obtain better results both in execution time and tuple traffic. As shown in the plots of those query executions, the major improvements in the traffic are obtained by nodes HJ2, HJ3 and the Scan of Lineitem. This is because the Bit Filter created in HJ1 with Relation *Orders* has significant efficiencies in query execution #1 (see Table 1). When pushing down that Bit Filter, the Lineitem table is filtered out significantly, reducing the traffic in all its upper nodes. The plots show that the smaller the traffic, the larger the reduction in execution time.

Another aspect to understand is the fact that the size of the *Sortmem* has an insignificant influence on query execution #1 for our strategy while it has a somewhat more significant influence on the execution time of executions #2. The reason is that the Bit Filter of HJ1 has a large efficiency (81%) in query execution #1 which reduces the Lineitem table to an extent that minimizes the I/O to the compulsory reads from disk even for small *Sortmem* sizes.

One final aspect to note is that the executions on *Sortmem* sizes of 60 MB show the reduction in CPU effort of the total execution time. In the three PostgreSQL implementations, the size of the *Sortmem* reduces the amount of disk reads to the compulsory ones. It is important to note that, even in those cases, the amount of CPU time saved is significant which gives strength to our strategy.

Query executions #3 and #4: These executions achieve good results but are not as significant as those in the previous query executions. They also show that PDBF are not a burden even when there is little benefit to obtain. Query execution #3 has a significant restriction on Part. This means that HJ3 only has the compulsory reads from disk because the build Relation fits in memory. Therefore, the output cardinality of HJ3 is very small in any of the three engines tested. Also, almost all the tuples filtered out by our strategy on Lineitem are caused by the Bit Filter created in HJ3 and the only work saved is that of projecting the tuples from the Scan of Lineitem up to HJ3, which implies a very little amount of execution time.

Query execution #4 does not have a restriction on Part and it has very weak restrictions on Partsupp and Orders. Most probably, Bit Filters would not

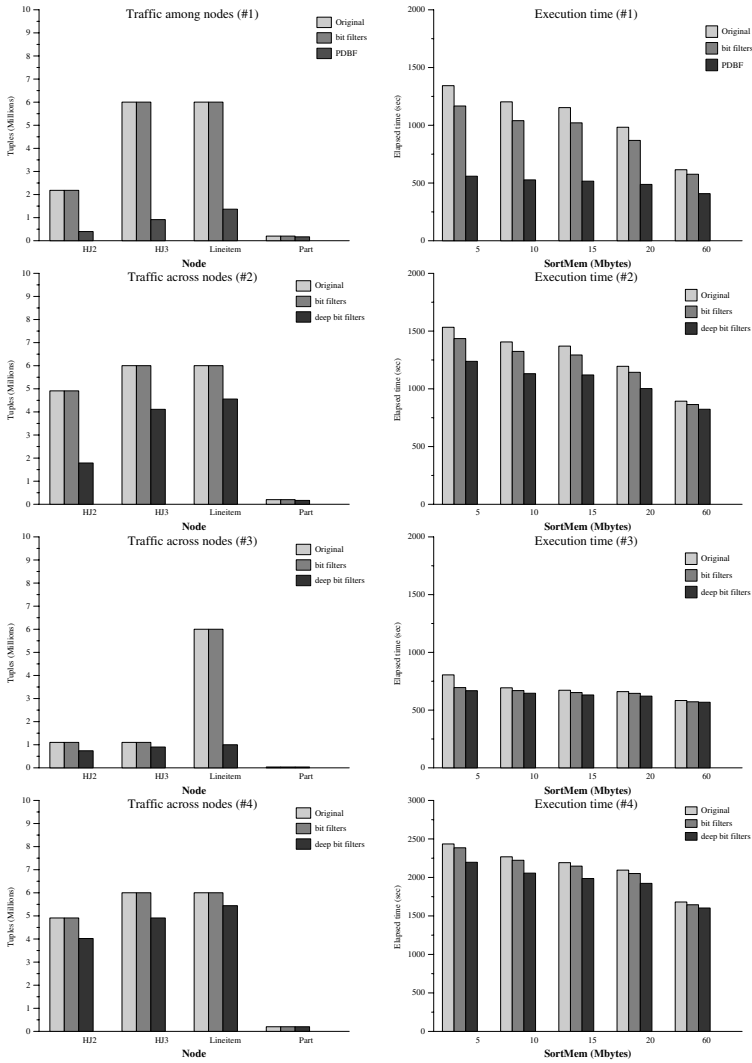


Fig. 3. Execution time and traffic across nodes for executions #1, #2, #3 and #4. PDBF stands for Pushed Down Bit Filters.

be used in this case (the optimizer would not trade memory for an uncertain execution time benefit).

I/O for Query Execution #1

Figure 3 shows that PDBF on query execution #1 improves the execution time of the basic and Bit Filter implementations of PostgreSQL by more than a 50%. Now we analyze the implications of the I/O on that query execution.

Figure 4 shows the I/O performed by each of the three Hash Joins involved in query execution #1. We do not include the I/O for the Scan operations because

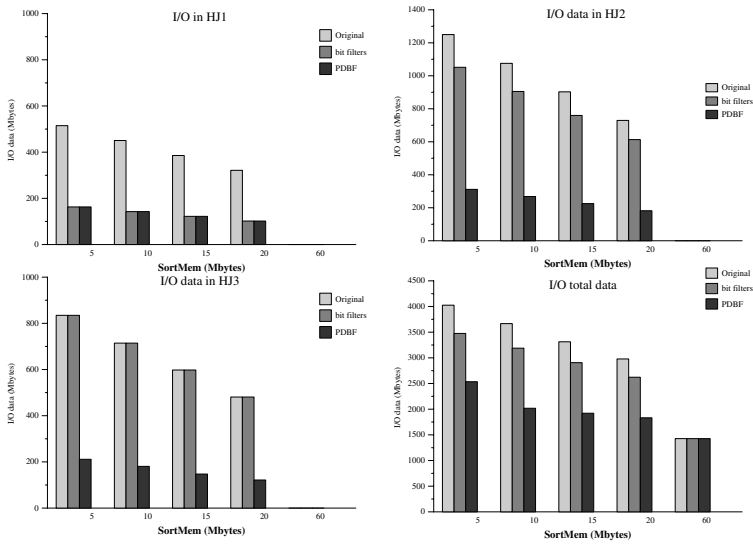


Fig. 4. I/O for query execution #1. PDBF stands for Pushed Down Bit Filters.

it does not change. The amount of I/O in Hash Join operations varies depending on the amount of data in the build Relations and the amount of memory available for the Hash Joins. If the amount of data in the build Relations fits in memory, the amount of I/O is zero in those operations, as shown in Figure 4.

The plots show a significant improvement of more than a 50% of PDBF compared to the other implementations. There is no improvement of our strategy in HJ1 compared to the only use of Bit Filters. This is because the Bit Filter is in the topmost Join and the Bit Filter already reduces the probe table before partitioning it. The amount of I/O on HJ2 and HJ3 is reduced due to the fact that the Bit Filters are applied in the bottom leaves of the plan. Notice that as the *SortMem* size is incremented, the I/O of the basic and Bit Filters executions is reduced but is still very far from that of the I/O of our strategy.

As shown in Figure 4, the execution on 60 MB *Sortmem* just uses the amount of IO necessary for compulsory reads, even for the original implementation in PostgreSQL. However, PDBF show a significant reduction of execution time (see Figure 3) because the amount of CPU work incurred by the tuple traffic saved is significant.

4 Related Work and Discussion

The use of Bit Filters is a popular strategy to support decision support queries. They have been used both in parallel Database management systems [7,12,13] and in sequential DBMSs [2] in order to speed-up the execution of joins. The original idea of the Bit Filter comes from the work by Bloom [1] where he designed them to filter out tuples without join partners.

The most relevant related work is [4]. Its aim is also to reduce the traffic between query nodes in order to reduce the I/O and the CPU costs. However, this proposal has a couple of limitations that we overcome with our Pushed Down Bit Filters. First, the algorithm proposed in [4] creates Bit Filters whenever an attribute belonging to one Relation or to the outcome of an operation can filter out tuples from another Relation. This leads to additional work and an extra need for memory space. Our technique relies on the Bit Filters already created, as we have shown in the paper. This saves time and memory space.

Second, as a consequence of the exhaustive creation of Bit Filters, the algorithm in [4] requires a non-pipelined execution. This is so because Bit Filters have to be created in full in order to be used. Thus, operations that create Bit Filters have to materialize their results. Materialization of nodes always incurs additional I/O. Our technique does not change the pipelined model of query execution. The pipelined execution of Hash Joins yields a better performance for multi-join queries as shown in [14], which also concludes that pipelined algorithms should be used in dataflow systems. In addition, most of the commercial DBMSs such as DB2 or Oracle use the pipelined execution philosophy to implement their DBMS engines [8]. Our Pushed Down Bit Filters require a small effort to be included in modern pipelined execution engines.

Generalized hash teams also use Bit Filters with a different purpose than we do here [9]. The idea comes from the original idea of *hash teams* in [6].

The technique uses a cascaded partitioning of the input tables using bitmaps. The partitioning is aimed at reducing the amount of I/O during the joining process and on the final aggregation. The bitmaps are used as a partitioning device. Our approach is not limited to multi-joins with aggregation but to a more general class of joins and allows one table being filtered by as many Bit Filters as different join attributes are projected from that table.

Generalized hash teams also have the limitation of false drops. When two different values map into the same Bit Filter position there may be false drops. False drops may cause a considerable amount of I/O in skewed environments. Our approach, given that it is not based on partitioning, does not have the false drops problem.

Other lines of related work include the use of Bit Filters as bitmap indexes [3, 10, 15]. In those cases, Bit Filters are used to perform joins in data warehousing contexts with the only use of bitmaps that reduce the I/O cost.

5 Conclusions

In this paper we propose PDBF, a strategy to use the Bit Filters created in the upper leaves of pipelined query execution plans, in the Scan nodes of the plan.

We can outline the following conclusions for our strategy. First, our strategy reduces the amount of tuple traffic of the query plan significantly. This has a direct influence on the amount of work to be performed by the nodes. Second, as a consequence of the reduction of traffic, it is possible to reduce significantly the amount of I/O of some queries. Third, the direct consequence of the reduction

of traffic and I/O is the reduction of execution time whenever the strategy can be used. The amount of benefit varies depending on whether it comes only from the traffic or both the traffic and the amount of I/O. Fourth, our strategy is simple to use. It just uses the Bit Filters already created by the Join nodes of the plan. PDBF require very simple run time data structures that only have to be managed by the Scan operations of the plan.

As a consequence of the benefits that can be obtained and the low implementation cost, it is most advisable to use PDBF in modern pipelined DBMSs.

Topics of future research related to PDBFs are optimization and scheduling issues and a further understanding of their applicability.

References

1. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. K. Bratbergseugen. Hashing methods and relational algebra operations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 323–333, 1984.
3. C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proc. of the SIGMOD Conf. on the Management of Data*, pages 355–366, 1998.
4. M.-S. Chen, H.-I. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *VLDB Journal: Very Large Data Bases*, 6(2):121–131, 1997.
5. P. Gongloor and S. Patkar. Hash joins: Implementation and tuning, release 7.3. Technical report, Oracle Technical Report, March 1997.
6. G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *Proceedings of the 25th VLDB Conference*, pages 86–97, August 1998.
7. H.-I. Hsiao, M.-S. Chen, and P. S. Yu. Parallel execution of hash joins in parallel databases. *IEEE-Trans. on Parallel and Distributed Systems*, 8(8):872–883, August 1997.
8. Roberto J. Bayardo Jr. and Daniel P. Miranker. Processing queries for first few answers. In *CIKM*, pages 45–52, 1996.
9. A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 30–41, September 1999.
10. P. O’Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, September 1995.
11. PostgreSQL. <http://www.postgresql.org/>.
12. Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD*, pages 110–121, 1989.
13. Patrick Valduriez and Georges Gardarin. Join and semi-join algorithms for a multiprocessor database machine. *TODS*, 9(1):133–161, 1984.
14. A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
15. M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Intl. Conference on Data Engineering*, pages 220–230, 1998.

Suffix Arrays in Parallel*

Mauricio Marín¹ and Gonzalo Navarro²

¹ University of Magallanes, mmarin@ona.fi.umag.cl

² University of Chile, Center for Web Research (www.cwr.cl)
gnavarro@dcc.uchile.cl.

Abstract. Suffix arrays are powerful data structures for text indexing. In this paper we present parallel algorithms devised to increase throughput of suffix arrays on a multiple-query setting. Experimental results show that efficient performance is indeed feasible in this strongly sequential and very poor locality data structure.

1 Introduction

In the last decade, the design of efficient data structures and algorithms for textual databases and related applications has received a great deal of attention due to the rapid growth of the Web [1]. To reduce the cost of searching a full large text collection, specialized indexing structures are adopted. *Suffix arrays* or *PAT arrays* [1] are examples of such index structures. They are more suitable than the popular inverted lists for searching phrases or complex queries composed of regular expressions [1]. A fact to consider in parallel processing natural language texts is that words are not uniformly distributed both in the text itself and the queries provided by the users of the system. This produces load imbalance.

The efficient construction in parallel of suffix arrays has been investigated in [3,2]. In this paper we focus on query processing. We propose efficient parallel algorithms for (1) processing queries grouped in batches of Q queries, and (2) load balancing properly this process when dealing with biased collections of words such as in natural language.

2 Global Suffix Arrays in Parallel

The suffix array is a binary search based strategy. The array contains pointers to the document terms, where pointers identify both documents and positions of terms within them. The array is sorted in lexicographical order by terms. A search is conducted by direct comparison of the terms pointed to by the array elements. A typical query is finding all text positions where a given substring appears in and it can be solved by performing two searches that locate the delimiting positions of the array for the substring.

We assume a broker (server) operating upon a set of P machines running the BSP model [4]. The broker services clients' requests by distributing queries onto

* Funded by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

the P machines implementing the BSP server. We assume that under a situation of heavy traffic the server processes batches of $Q = qP$ queries.

A suffix array can be distributed onto the processors using a *global index* approach in which a single array of N elements is built from the whole text collection and mapped evenly on the processors. In this case, each processor stands for an interval or range of suffixes (lexicographic partition). The broker machine maintains information of the values limiting the intervals in each machine and route queries to the processors accordingly. This fact can be the source of load imbalance in the processors when queries tend to be dynamically biased to particular intervals. We call this strategy G0.

In the *local index* strategy, on the other hand, a suffix array is constructed in each processor by considering only the subset of text stored in its respective processor. Unlike the global approach, no references to text positions stored in other processors are made. However, for every query it is necessary to search in all of the processors in order to find the pieces of local arrays that form the solution for a given interval defined by the query. It is also necessary to broadcast every query to every processor. We have found both theoretically and experimentally that the global index approach offers the potential of better performance.

A drawback of the global index approach is related to the possibility of load imbalance coming from large and sustained sequences of queries being routed to the same processor. The best way to avoid particular preferences for a given processor is to send queries uniformly at random among the processors. We propose to achieve this effect by multiplexing each interval defined by the original global array so that if the array element i is stored in processor p , then the elements $i+1, i+2, \dots$ are stored in processors $p+1, p+2, \dots$ respectively in a circular manner. We call this strategy G1.

In this case, any binary search can start at any processor. Once a search has determined that the given term must be located between two consecutive entries k and $k+1$ of the array in a processor, the search is continued in the next processor and so on, where at each processor it is only necessary to look at entry k of the local arrays. In general, for large P , the inter-processors search can be done in at most $\log P$ additional BSP supersteps by performing a binary search across processors.

The binary search on the global index approach can lead to a certain number of accesses to remote memory. A very effective way to reduce this is to associate with every array entry the first t characters of the terms respectively. The value of t depends of the average length of terms. This reduced remote accesses to less than 5% in [2], and less than 1% in our experiments, for relatively small t .

In G0 we keep in each processor an array of P strings of size ℓ marking the delimiting points of each interval of G0. The broker machine routes queries uniformly at random to the P real processors, and in every processor a $\log P$ time binary search is performed to determine to which processor send a given query (we do so to avoid the broker becoming a bottleneck). Once a query has been sent to its target processor it cannot migrate to other processors as in the case of G1. That is, this strategy avoids the inter-processors $\log P$ binary search.

3 Experimental Results and Conclusions

We now compare the multiplexed strategy (G1) with the plain global suffix array (G0). For each element of the array we keep t characters which are the t -sized prefix of the suffix pointed to by the array element. We found $t = 4$ to be a good value for our text collection. We also implemented the local index strategy, but it was always at least 3 times slower than G0 or G1.

The text collection is formed by a 1GB sample of the Chilean Web retrieved by the search engine www.todocl.cl. We treated it as a single string of characters, and queries were formed by selecting positions at random within this string. For each position a substring of size 16 is used as a query. In the first set of experiments these positions were selected uniformly at random. Thus load balance is expected to be near optimal. In the second set of experiments we selected at random only the positions whose starting word character were one of the four most popular letters of the Spanish language. This produces large imbalance as searches tend to end up in a subset of the global array. Experiments with an actual set of queries from users of “todocl” produced results between the uniform and biased cases.

The experiments were performed on a PC cluster of 16 machines. Runs with more than 16 processors were performed by simulating virtual processors. At each superstep we introduced $1024/P$ new queries in each processor. Most speed-ups obtained against a sequential realization of suffix arrays were super-linear. This was not a surprise since due to hardware limitations we had to keep large pieces of the suffix array in secondary memory whilst communication among machines was composed by a comparatively small number of strings.

The whole text was kept on disk so that once the first t chars of a query were found to be equal to the t chars kept in the respective array element, a disk access was necessary to verify that the string forming the query was effectively found at that position. With high probability this required an access to a disk file located in other processor, case in which the whole query is sent to that processor to be compared with the text retrieved from the remote disk.

Though we present running time comparisons below, what we considered more relevant to the scope of this paper is an implementation and hardware independent comparison between G0 and G1. This came in the form of two performance metrics devised to evaluate load balance in computation and communication. They are the average maximum across supersteps. During the processing of a query each strategy performs the same kind of operations, so for the case of computation the number of these ones executed in each processor per superstep suffices as an indicator of load balance for computation. For communication we measured the amount of data sent to and received from at each processor in every superstep. We also measured balance of disk accesses.

Table 1.1 shows results for queries biased to the 4 popular letters. Columns 2, 3, and 4 show the ratio G1/G0 for each of the above defined performance metrics (average maximum for computation, communication and disk access). These results confirm intuition, that is G0 can degenerate into a very poor performance strategy whereas G1 is a much better alternative. G1 is independent

of the application but, though well-balanced, it tends to generate more message traffic due to the inter-processors binary searches. The differences among G1 and G0 are not significant for the case of queries selected uniformly at random. G1 tends to have a slightly better load balance.

Table 1. G1/G0 ratios.

P	comp	comm	disk	P	comp	comm	disk	P	Biased	Uniform
2	0.95	0.90	0.89	16	0.39	0.35	0.36	4	0.68	0.78
4	0.49	0.61	0.69	32	0.38	0.29	0.24	8	0.55	0.78
8	0.43	0.45	0.53	64	0.35	0.27	0.17	16	0.61	0.86

(1) Performance metrics

(2) Running times

As speed-ups were superlinear due to disk activity, we performed experiments with a reduced text database. We used a sample of 1MB per processor which reduces very significantly the computation costs and thereby it makes much more relevant the communication and synchronization costs in the overall running time. We observed an average speed-up efficiency of 0.65.

Table 1.2 shows running time ratios obtained with our 16 machines cluster. The biased workload increased running times by a factor of 1.7 approximately. In both cases G1 outperformed G0, but G1 loses efficiency as the number of processors increases. This is because, as P grows, the effect of the inter-processor binary searches becomes more significant in this very low-cost computation scenario.

Note that the above G0 strategy can be extended to approximate G1 by partitioning the array in $V = kP$ virtual processors and mapping the V pieces of the array in a circular manner on the P actual processors. Preliminary experiments show that this strategy tends to significantly reduce the imbalance of G0 at a small k value.

Yet another method which solves both load imbalance and remote references is to re-order the original global array so that every element of it contains only pointers to local text (or most of them). This becomes similar to the local index strategy whilst it still keep global information which avoids the P parallel binary searches and broadcast per query. Unfortunately we now lose the capability of performing the inter-processors $\log P$ -cost binary search. We are currently investigating ways of performing this search efficiently using little extra space.

References

1. R. Baeza and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley., 1999.
2. J. Kitajima and G. Navarro. A fast distributed suffix array generation algorithm. In *6th Symp. String Processing and Information Retrieval*, pages 97–104, 1999.
3. G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In *8th Symp. Combinatorial Pattern Matching*, pages 102–115, 1997. LNCS 1264.
4. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

Revisiting Join Site Selection in Distributed Database Systems

Haiwei Ye¹, Brigitte Kerhervé², and Gregor v. Bochmann³

¹ Département d' IRO, Université de Montréal, CP 6128 succ Centre-Ville,
Montréal Québec, Canada H3C 3J7
ye@iro.umontreal.ca

² Département d'informatique, Université du Québec à Montréal, CP 8888,
succ Centre-ville, Montréal Québec, Canada H3C 3P8
Kerherve.Brigitte@uqam.ca

³ School of Information Technology & Engineering, University of Ottawa
P.O. Box 450, Stn A, Ottawa Ontario, Canada K1N 6N5
bochmann@site.uottawa.ca

Abstract. New characteristics for e-commerce applications, such as highly distributed data and unpredictable system nature, require us to revisit query processing for distributed database systems. As join operations involve relations over multiple sites, the site to carry out the join operation can have a significant impact on the performance. In this paper, we propose a new join site selection method. Our method is different from traditional methods in the following ways. First, the candidate sites are not restricted to the operand sites or query site as adopted by most database systems. Second, it considers dynamic system properties such as available bandwidth and system loads. We also conducted some experiments to offer a comparison between our method and the traditional one. The results show the advantage of our method under various system statuses.

1 Introduction

Given the explosive growth of data over the Internet and prevalence of web-based applications such as e-commerce, how to manage large volumes of data and provide fast and timely answers to queries become more challenging in today's information systems. Data are usually distributed across multiple locations in a distributed database system for the purpose of scalability and availability. As a result, a database query issued against a distributed database system generally requires retrieval of data from several locations to compute the final answer. Typical operations to combine the data from different locations are binary operations such as *join*. Even if techniques such as semi-joins have been introduced to reduce the cost of queries [1], where to perform the join is still of interest since it does have a great impact on the overall system performance, especially in today's Internet environment. The research dedicated to join site selection has not received adequate attention. Existing approaches of

join site selection do not consider the dynamic nature of the Internet environment. We propose an approach to integrate user-defined QoS requirements into a distributed query processing environment [2]. Join site selection is regarded as one of the essential steps in our QoS-aware query processing.

In a recent survey [3], Kossmann summarized three site selection strategies for client-server architectures. Depending on whether to move the query to the data (execution at servers) or to move the data to the query (execution at clients), the strategy is called *query-shipping* or *data-shipping*. A *hybrid* approach of these two solutions is also possible. Traditionally [4, 5], two types of strategies were proposed: *Move-Small* and *Query-Site*. In the query-site strategy, all joins are performed at the site where the query was submitted. In the move-small strategy, for each join operation, the smaller (perhaps temporary) relation participating in the join is always sent to the site of the larger relation. Selinger and Adiba suggested another option in [5]. They mentioned that for a join of two given tables at different sites, they could move both tables to a “third” site yet to be specified. However, this “third” site strategy has not been completely studied and not been adopted by commercial database systems.

The above strategies for join site selection in a distributed environment are inadequate in the sense that they are only suitable in a static environment where network performance and load of the database server are fixed and predictable. This is obviously not the case for today’s highly distributed and dynamic systems. In this paper, we address the issue of join site selection in distributed multidatabase systems deployed for e-commerce applications. We propose an approach where dynamic system properties such as the performance of the Internet and the load of the server are taken into account in cost-based decisions for join site selection. In the next section we describe our approach and we provide a performance analysis in Section 3.

2 Considering a Third-Site in Join Site Selection

In order to select the site where the join operation will be processed, we first build the set of candidate sites where we consider both operand sites as well as possible third sites. In this set, we then select the optimal site. This decision is based on the cost model we propose, where dynamic system properties are considered. A QoS monitoring tool is used to periodically collect system status information and to provide dynamic system properties to the query optimizer.

2.1 Building the Set of Candidate Sites

The key issue in site selection is to decide which site is the best (optimal) for each binary operator. The site selection process becomes complicated when several candidate sites are capable of handling the operator. In fact, the crucial question is how many candidates should be considered for the third site. There are three possible approaches to determine candidate sites:

- 1) Consider all the available sites in the system. This is simple but this will usually incur too much overhead for the optimizer.
- 2) We can shrink the above set to all the sites involved in this particular query. By considering these sites, we may benefit from the situation where the result of this join needs to be shipped to the selected third site for further calculation. However, if the number of locations involved in the query is large, we may have the same problem as above: too much optimization overhead.
- 3) We can apply some heuristics to further decrease the size of the candidate sites set. For example, we can restrict the third site for a particular join operator to its “close relatives”, such as niece/nephew sites in the join tree.

In our approach, we combine options 2 and 3. A *threshold* for the number of sites is therefore used to describe the situation where option 3 should be used. That is,

$$\text{Candidate sites for a join} = \begin{cases} \text{All the sites in the tree,} & \text{if } N \leq \text{threshold} \\ \text{Close relative (children and niece),} & \text{otherwise} \end{cases}$$

where N is the total number of sites involved in the query. The value of threshold should be derived from the experiment. In the following algorithm, we use a procedure *CandidateSitesSelection()* to represent this procedure.

2.2 Algorithm

The procedure of join site selection can be regarded as marking the site for each join node in the tree, usually done in a bottom-up fashion. Thus we can employ one of the standard bottom-up tree traversal algorithms for this purpose. In our algorithm, we use *post order* tree traversal to visit the internal nodes of the tree. We ignore the post order algorithm and only give the algorithm (*SiteSelection()*) to visit each node.

Algorithm: *SiteSelection* (*treenode*)

```

1.  {
2.  if (hasChild(treenode) == true ) {
3.    candidate_set[] = CandidateSitesSelection (treenode);
4.    for each site_s in candidate_set
5.      cost [s] = Cost-node (treenode, site_s); //Use the cost
        model to compute the cost if the join is performed on this site
6.    min_site = select-min (cost[]);
7.    treenode.join_site = min_site;
8.    if (site_s is also marked as join site for another node m
        in treenode)
9.      Cost-node (m, site_s); //recalculate cost for node m under the new
        added load introduced by choosing site_s
10.  } //end if
11. }
```

The procedure of *SiteSelection()* picks up the join site based on the cost model and records the join site in the root node of the input tree. The procedure of *SiteSelection()* picks up the join site based on the cost model and records the join site in the root node of the input tree. In the algorithm (Line 8 and 9), we consider the impact of this site selection on other nodes of the tree. We then recalculate the cost for node m , taking

into account the additional load introduced by choosing *site_s*. Due to the optimization overhead that would increase if we recursively consider the impact, in the current algorithm, we do not check whether the site selection is still optimal. This is part of our future work.

2.3 Cost Model and QoS Information

The *Cost-node()* implements the cost model which provides an evaluation of the cost for each node in the query execution plan and enables the adaptation to the changing conditions in distributed systems. The cost function includes two major parts: the *local part* and the *network part*. The following formulas define our cost models to execute a join at a candidate site *s*:

$$\begin{aligned} \text{Cost-node}(\text{treenode}, \text{site}_s) &= \text{Local\_cost}_{\text{site}_s} + \text{Network\_cost}; \\ \text{Local\_cost}_{\text{site}_s} &= \text{join-cost}(\text{node.left}, \text{node.right}, \text{load}_{\text{site}_s}); \\ \text{Network\_cost} &= \max\{\text{ship-cost}(\text{node.left}, \text{site}_s), \text{ship-cost}(\text{node.right}, \text{site}_s)\}; \\ \text{ship-cost}(\text{node}, \text{site}_s) &= \text{node.data} / \text{bwd}_{\text{node.site}_i, \text{site}_s} + \text{delay}_{\text{site}_i, \text{site}_s}; \end{aligned}$$

where $\text{bwd}_{\text{node.site}_i, \text{site}_s}$ and $\text{delay}_{\text{node.site}_i, \text{site}_s}$ represent the available TCP bandwidth and delay, respectively, from the site of the node to site *i*.

The accuracy of our cost models relies on the up-to-date information of the current system status. Therefore, to keep track of the current dynamic performance information about the underlying network and database server, we use the QoS monitoring tools developed in our work [6].

3 Experimentation

In this section, we evaluate the performance of our join site selection strategy according to the framework proposed in the previous section. The objective of our experiment is to show that our strategy can adapt to workload changes (both server load and network load) and always provides a better performance as compared to traditional strategy. For the purpose of comparison, we also implement the traditional join site selection strategy where the chosen site is always the site where the larger table resides; we call this strategy “larger-site” strategy.

Two types of system loads are used for our measurement: network load and server load. For network load, we mainly focus on the available bandwidth as the indication of network congestion level. For server load, we concentrate on the CPU utilization as the indication of server load. We observed the TCP traffic using IPERF [7]. As a result, the network congestion levels are classified into 6 levels: level 0 (no congestion) to level 5 (highest congestion). Concerning the server load, we degrade the performance of one server by loading it with additional processes. Each process simply eats up CPU and competes with the database system for CPU utilization. Additional load is

quantified by the number of these processes spawned on a server. We categorize this load into 4 levels: *no load*, *low load*, *medium load*, and *high load*.

It should be noted that as an experimental prototype, our execution engine was designed for ease of implementation and was not tuned for performance. The main purpose is to demonstrate the feasibility of our ideas. We conducted a number of experiments on two-way join and collected performance data for the two sets of experiments identified previously. Figure 1 provides a comparison of various loads.

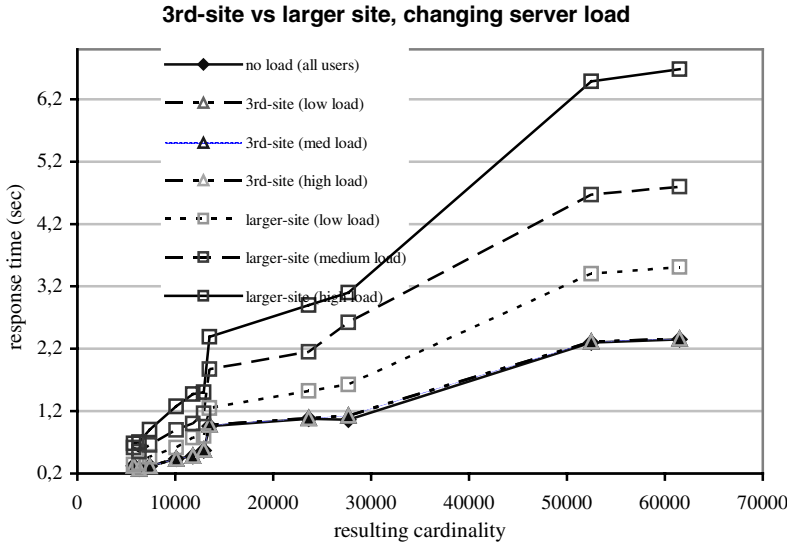


Fig. 1. Third-site vs. Larger-site with various server loads

The number of records in the resulting table (noted as resulting cardinality) varied from 180 to 3075. In the experiment, the network condition is the same for the entire load test, and the result in Figure 1 is collected while the network bandwidth is 5Mbps for all links. We only load the servers taking part in the joins and the rest remains unloaded. This opens up the possibility to ship joined tables to other nodes for the purpose of performance gain.

We also conducted some experiments to compare the performance under different network congestion levels. In these tests, we assume that the links among the nodes involved in the join are congested while other links have the normal throughput (5Mbps). In addition, there is no load of the server during the experimental periods. The results show that, under the no load circumstance, the two strategies get the same performance. With the increasing load, the “third-site” selection strategy almost provides the same performance, while the “larger-site” strategy will experience higher response time. The advantage of third-site strategy increases with the increasing of server load. Traditional site selection fixes the join site to the larger site. As the network congestion level increases, the data transfer time will increase too, which will in turn affect the response time of the whole query. However, for the third site selection,

when the network congestion level reaches a certain point (usually at level 3, called shifting point), the algorithm will suggest to avoid the congested link and ship the two tables to a third site. This also explains why the response times (for the third-site algorithm) after the shifting point remain the same. In the performance study of the site selection problem, our third-site strategy not only reduces the response time but also achieves good load balancing among different database servers. According to our algorithm, if a server is heavily loaded, then the cost to perform a join operation on that server might be higher. This leads to the optimizer to avoid using that database server.

Both sets of experiments show the superiority of our “third-site” selection algorithm: it can pick up the fast response plan under different system conditions. In addition, it achieves good load balancing among different database servers. According to our algorithm, if a server is heavily loaded, then the cost to perform a join operation might be higher leading the optimizer to avoid using that server.

4 Conclusion

In this paper, we have proposed a new join site selection strategy for our QoS-aware distributed query processing environment. The strategy is based on cost models which integrate dynamic system properties. The candidate sites set considered in our approach is not restricted to the two operand sites as used in the traditional way. For the moment, we focus on the performance aspect of the QoS parameters provided by the QoS monitor. In the future, we will consider other QoS parameters, such as money cost or data quality.

References

1. Stocker, K., Kossmann, D., Braumandl, R., Kemper, A.: Integrating Semi-Join-Reducers into State of the Art Query Processors. ICDE 2001, pp. 575–584.
2. Ye, H., Kerhervé, B., Bochmann, G.v., Oria, V.: Pushing Quality of Service Information and Requirements into Global Query Optimization, the Seventh International Database Engineering and Applications Symposium (IDEAS 2003), Hong Kong, China, July 16–18
3. Kossmann, D.: The state of the art in distributed query processing, ACM Computing Surveys (CSUR), Volume 32, Issue 4, December 2000, pp 422–469.
4. Cornell, D. W., Yu, P. S.: Site Assignment for Relations and Join Operations in the Distributed Transaction Processing Environment. ICDE 1988, pp. 100–108.
5. Selinger P. G., Adiba, M.: Access path selection in distributed data base management systems. In Proc. of the International Conference on Data Bases, 1980, pp. 204–215.
6. Ye, H.: Integrating Quality of Service Requirements in a Distributed Query Processing Environment, Ph.D thesis, University of Montreal, May 2003.
7. National Laboratory for Applied Network Research, <http://www.nlanr.net/>

SCINTRA: A Model for Quantifying Inconsistencies in Grid-Organized Sensor Database Systems

Lutz Schlesinger¹ and Wolfgang Lehner²

¹ University of Erlangen-Nuremberg Department of Database Systems Martensstr. 3, 91058 Erlangen, Germany, schlesinger@informatik.uni-erlangen.de

² Dresden University of Technology Database Technology Group 01062 Dresden, Germany, lehner@inf.tu-dresden.de

Abstract. Sensor data sets are usually collected in a centralized sensor database system or replicated cached in a distributed system to speed up query evaluation. However, a high data refresh rate disallows the usage of traditional replicated approaches with its strong consistency property. Instead we propose a combination of grid computing technology with sensor database systems. Each node holds cached data of other grid members. Since cached information may become stale fast, the access to outdated data may sometimes be acceptable if the user has knowledge about the degree of inconsistency if unsynchronized data are combined. The contribution of this paper is the presentation and discussion of a model for describing inconsistencies in grid organized sensor database systems.

1 Introduction

Data sets of sensor data are typically collected in sensor database systems ([BoGS01]). The concept of a federated ([ÖzVa91]) or data warehouse system ([Inmo96]) is the natural way to combine data of different sensor databases. However, the approaches suffer on the strong consistency property which cannot be maintained due to the high data refresh rate. Instead, we propose a combination of sensor database systems with the concept of *grid organized computing structures* ([FoKe99]), which is well known in the ‘number crunching’ area (the Globus Project; <http://www.globus.org>) but is not widely used for ‘data crunching’ where local nodes are represented by database systems.

A grid provides an alliance of autonomous computing systems (figure 1). Each node locally stores sensor data. Since sensor data permanently change their values or add new values to already monitored data sets, we introduce the concept of recording periods. After a time interval or a maximum amount of recorded data the recording period is finished, the existing data are frozen and a new period starts. In comparison to the federated approach, we avoid the permanent data synchronization. In the opposite, the packaged updates do not guarantee the strong consistency property: An update of a local snapshot during

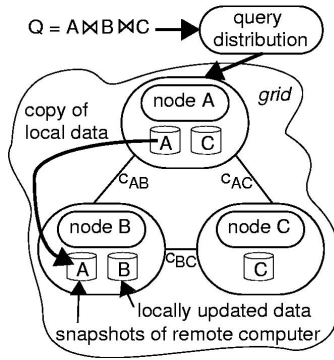


Fig. 1. Architecture of the discussed grid scenario

query processing is time and cost consuming regarding the communication costs. If a fallback to a local existing but stale snapshot is possible, highest consistency is given up: The snapshots are not the most current ones and the locally cached snapshots need not to have the same valid time. The time difference between the current time and the time point of the selected snapshots should be quantified by a metric. A user may integrate this metric in the query specification to control the maximum inconsistency. Additionally, the metric may be used by the system to distribute queries.

Contribution and Structure of the Paper

In this paper we quantify the metric and propose a general model for dealing with inconsistencies in our grid organized sensor database system. Our *SCINTRA* (*semi-consistent integrated time replicated approach*) data model is presented in section 2. Section 3 discusses the usage of the model for distributing queries to a single node of the grid. An overview of related work is given in section 4. The paper closes with a conclusion.

2 The SCINTRA Data Model

In this section, we introduce the SCINTRA data model as a container model with disjoint fragments consisting of three parts.

2.1 System Model

As illustrated in figure 1 a grid database system consists of a set of nodes N_i , connections and communication costs c_{ij} between two neighboring nodes. The total communication costs between two arbitrary grid nodes are the sum of the communication costs of those nodes, which lie on the path between the communicating nodes:

2.2 Data Structure Concept

The data structure concept at a single node of the grid is divided into three layers: Atomic layer: Sensors measure (physical) signals like temperature, sales or bank account and send the data sets to a sensor database, where they are collected and made available for applications to analyze the data sets. Each such a (sensor) object monitors atomic data sets consisting of a timestamp and a value. An *atomic object* is notated as (x, t, d) or shortly $x(t) = d$ with x as unique sensor identifier, which measures exactly one object of the real world, t as the timestamp and d as measured data.

Micro Layer: At the micro layer a set of atomic objects, which are events in the real world, is grouped together. A *micro object (MiO)* is formed after a number of monitored atomic objects, a time interval or on demand. A MiO consists of a set of atomic objects and gets timestamps representing the time interval of the data set. A function may be defined to describe the value patterns of the real object, because the monitored atomic objects are only discrete, although the type of the data source may be discrete, partial steady or continuous ([TCG+93]). At a single node the monitored atomic objects are exactly assigned to a single MiO, why two different MiOs do not overlap.

Macro Layer: On the next layer, the *macro layer*, several MiOs are grouped to *macro objects (MaOs)*. The schema of a MaO is built by schema integration of the schemas of the underlying MiOs. Missing instance values of the MiOs are set to NULL. A MaO forms a container for a set of MiOs. For each logical object (MiO or MaO), which exists exactly once in the grid, several physical representations may exist in the grid.

Grid Layer: Now the focus is turned away from a local view to a global view: the grid layer. At a node, denoted as the *data source or collecting database D_s* (figure 2), the monitored atomic objects are grouped to MaOs which forms

the local data D_L . A copy of the MaO is sent as a snapshot to other nodes of the grid (*consuming database D_t*). The new MaO is stored in addition to already existing MaOs from the same data source. All MaOs from foreign nodes form the foreign data D_F . The time of storing the data locally is the transaction time. The valid time is either the upper time bound of the MaO or the transaction time if no other time is known. The object-time-diagram (figure 3) illustrates the situation.

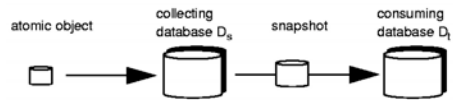


Fig. 2. Illustration of objects in the grid

2.3 Data Manipulation Concept

The data manipulation concept covers all operations based on the data structures presented in the last section. In this section the discussion is made in reverse order.

Operations at the Grid Layer: At the grid layer general operations executed on a single node regard operations on the collecting and consuming database. The standard operations are insertion for adding a new data source including schema integration and the reverse operation *deletion* implying the removal of all MaOs of the data source existing in D_F and the reduction of the integrated schema by the schema of the removed data source. Other operations are *distribution* for sending MaOs to other nodes and *request* for requesting the newest MaO from another node.

Operations at the Macro Layer: At the macro layer algorithms for a parameterized selection of a set of snapshots (MaOs) and operations between the snapshots are defined. Usually the latest MaOs are selected and joined, which are the closest MaOs to the horizontal line t_{NOW} reflecting the current time in the object-time-diagram (figure 3). In our approach, where an arbitrary number of MaOs of the same data source exists, a selection at any time $t_c (t_c \leq t_{NOW})$, called the *historic cut*, is possible. The time distance between the MaOs, t_c and t_{NOW} and the data change rate is quantified as the *inconsistency metric* and defined in the following. Algorithms for selecting the MaOs under consideration of the metric and the age of the MaOs are discussed in [ScLe02].

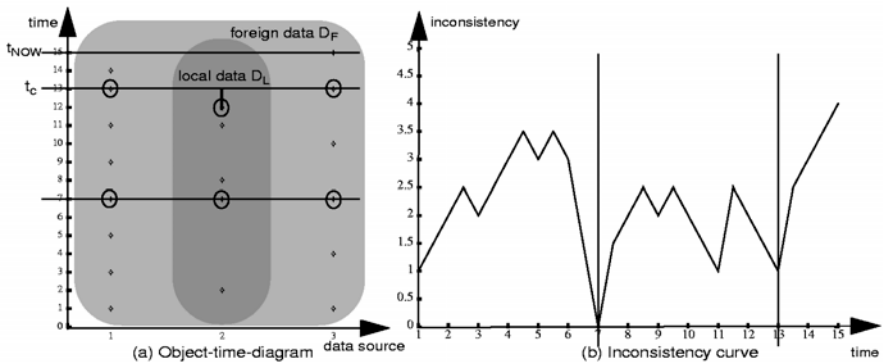


Fig. 3. Example for the selection of historic cuts and the resulting inconsistency curve

Disregarding the data change rate for a moment, the inconsistency for a single data source is defined as the distance between the valid time of the snapshot *time*

(S_k^k) and t_c . For k selected snapshots the inconsistency I is defined on the basis of the L_p -metric:

$$I =_p \sqrt[p]{\sum_{i=l}^k |(\text{time}(S_i^j) - t_c)|^p}$$

The example of figure 3 left shows snapshots of three data sources with $p = 1$. Figure 3 right illustrates the conflict between age and inconsistency: At point 7 the inconsistency is lower than the inconsistency at point 13, but the age is much higher.

The formula is now extended with the data change rate Δd_i , which reflects the data changes of the MaO and the underlying MiOs for each data source i . This signature regards existential and value based changes (values: 0–100%). Δd_i is a parameter of the reuse rate ρ at a time point t with t_s as the time point of the global oldest snapshot:

$$\begin{aligned} 0 < \Delta d_i \leq 0,5 : \rho(t) &= \frac{-1}{|t - (t_{NOW} - t_s) - 1|^{\frac{1}{2\Delta d_i} - 1}} \\ &\cdot \left(\frac{1}{t_{NOW} - t_s} \cdot (t - (t_{NOW} - t_s)) + 1 \right) + 1 \\ 0,5 \leq \Delta d_i < 1 : \rho(t) &= \frac{-1}{(t+1)^{\frac{1}{2(\Delta d_i - 1)} - 1}} \cdot \left(\frac{-1}{t_{NOW} - t_s} \cdot t + 1 \right) \end{aligned}$$

The reuse rate of old snapshots depends on the data change rate (figure 4). At t_{NOW} ρ has a value of 100% independently of Δd_i and at t_s the value is 0. The combination of the distance in time and ρ leads to an extended inconsistency formula, where α_k denotes weights for each data source and $S_k^{j_k}$ are the selected snapshots:

$$I(\text{time}(S_l^{j_l}), \dots, \text{time}(S_k^{j_k}))$$

$$= \sum_{k=l}^n (\alpha_k \cdot |\text{time}(S_k^{j_k}) - t_c| \cdot (1 - \rho_k(\text{time}(S_k^{j_k}))))$$

After selecting a single snapshot for each data source standard operations like selection or join may be defined in a nature manner.

Operations at the Micro Layer: They are mostly realized by operations on atomic objects. The operations are standard operations focusing on sensor data sets under the consideration of the classification introduced by [TCG+93]. The operations *selection*, *projection*, *aggregation* and the *unary* operation are defined in a nature manner by applying the operation

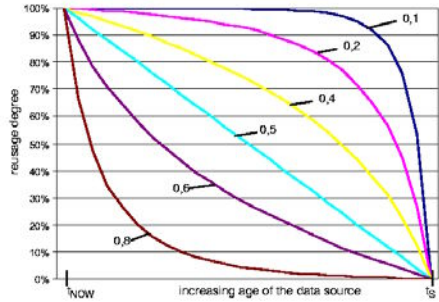


Fig. 4. Dependencies of reuse degree and age

to each atomic object of the MiO. A usual definition of the *join* operation would result in an atomic object with 6 elements. Since an atomic object consists of three elements, the join operation is defined in combination with a binary operation to get the 3-tuple. This special *SCINTRA-join* of two MiOs implies the problem of selecting the atomic objects, because the time points and the type of the sensor may be different. As illustrated in figure 5 the bullets mark atomic objects of a continuous and a discrete sensor. The selection of two atomic objects depends on the join semantic according to the SEQ-model ([SeLR95]), while the resulting continuity type depends on the application. The time interval of the resulting MiO is determined by the atomic objects of the new MiO. Finally, the *set* operation on two MiOs is only defined, if the MiOs contain data from the same sensor.

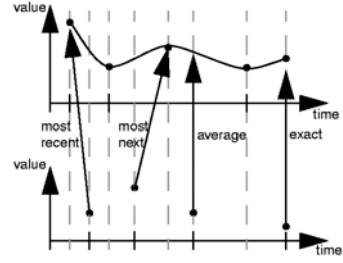


Fig. 5. Example for join semantics

Operations at the Atomic Layer: At the atomic layer the operations introduced above are now realized in a nature manner by using atomic objects of one or two MiOs. Regarding the *join* it is mentioned that the time point of the resulting atomic object is a parameter of the operation and defined by the join semantic. The result of an *aggregation* operation is an atomic object whereby the aggregation operation is either applied to the time (operators *min*, *max*, ...) or on the data (operators *sum*, *avg*, ...). While these operations result in a single atomic object the *set* operations result in a set, which forms a logical micro object. The set operations are also defined in a nature manner.

3 Query Distribution in the Grid

Figure 6 shows a simple example of a grid. A user sends a query to join snapshots of two data sources D_1 and D_2 . If the query is routed to N_1 , then the query can be evaluated at t_{c11} resulting in a computed inconsistency of 5. Another way is to transfer the snapshot S_2^2 from N_2 to N_1 with transfer costs $c_{21} = 50$ resulting in an inconsistency of 1 at t_{c12} . If the query is routed to N_2 , then a query evaluation is

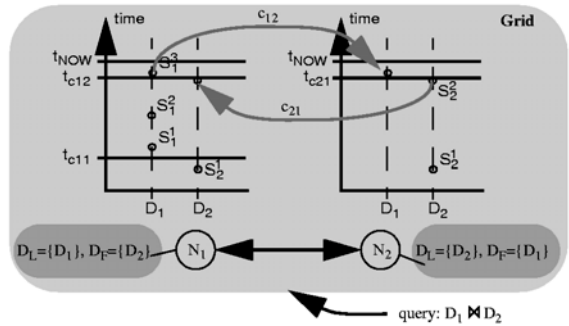


Fig. 6. Example for operations on the grid

If the query is routed to N_2 , then a query evaluation is

possible, if S_1^3 is copied from N_1 to N_2 with $c_{12} = 20$ and a resulting inconsistency of 1. Table 1 summarizes the facts. If the user specifies an upper bound of the allowed age of the historic cut, the transfer costs and the inconsistency, the system is able to decide on which node the query has to be evaluated.

With this decision the snapshots are selected and operations between the elements of the snapshots are executed corresponding to a specification in the query. At first, the MiOs are selected, which are determined by a sensor identifier and a time interval and which are parts of the query declaration. In the same way the operations on the atomic objects are specified. For example, for a join operation the join semantic, the binary operator and the resulting continuity type may be specified. Thus the introduced operators and its parameters allow the user to control the query execution to receive the expected result.

Table 1. Overview of computed data in the example

historic cut	transfer costs	inconsistency
t_{11}	0	5
t_{12}	20	1
t_{21}	50	1

4 Related Work

Since our approach is influenced by many different research areas, we outline only two areas. At first, database middleware systems like Garlic ([IBM01]) or mediator systems ([Wied92]) provide a query engine with sophisticated wrapper and query optimization techniques ([RoSc97]). Secondly, introducing replicas in distributed database systems ([ÖzVa91]) is a way to speed up access to data. Since replicas are synchronized in the context of an update operation, our approach is much more related to the concept of database snapshots ([AdLi80]). The selection of the “best” set of snapshots from a consistency point of view however, has not been discussed prior to our work.

5 Conclusion

The concept of sensor grid database systems allows a grid member to cache data from other grid members. If from a user point of view a fallback to outdated data is acceptable during query evaluation, the query may be faster evaluated but the user gives up highest consistency. In this paper we introduce a model for a sensor grid database system and a consistency framework to quantify the inconsistencies. Our approach is a solid base to decide on which node a query should be evaluated. The model is further helpful for the user to specify how operations on the selected snapshots should be executed.

References

- [AdLi80] Adiba, M.E.; Lindsay, B.G.: Database Snapshots. In: *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80, Montreal, Canada, October 1–3)*, 1980, pp. 86–91
- [BoGS01] Bonnet, P.; Gehrke, J.; Seshadri, P: Towards Sensor Database Systems. In: *Proceedings of the 2nd International Conference on Mobile Data Management (MDM'01, Hong Kong, China, January 8–10)*, 2001, pp. 3–14
- [FoKe99] Foster, T.; Kesselman, C. (Hrsg.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Verlag, San Francisco (CA) et al., 1999
- [IBM01] N.N.: *The Garlic Project*, IBM Corp., 2001
- [Inmo96] Inmon, W.H.: *Building the Data Warehouse*, 2nd edition. New York, Chichester, Brisbane, Toronto, Singapur: John Wiley & Sons, Inc., 1996
- [ÖzVa91] Özsu, M.; Valduriez, P.: *Principles of Distributed Database Systems*. Prentice-Hall, Englewoods Cliffs (New Jersey, USA), 1991
- [RoSc97] Roth, M.T.; Schwarz, P.M.: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In: *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97, Athens, Greece, August 25–29)*, 1997, pp. 266–275
- [ScLe02] Schlesinger, L.; Lehner, W.: Extending Data Warehouses by Semi-Consistent Database Views. In: *Proceedings of the 4th International Workshop on Design and Management of Data Warehouses (DMDW'02, Toronto, Canada, May 27)*, 2002
- [SeLR95] Seshadri, P.; Livny, M.; Ramakrishnan, R.: SEQ: A Model for Sequence Databases. In: *Proceedings of the 11th International Conference on Data Engineering (ICDE'95, Taipei, Taiwan, March 6–10)*, 1995, pp. 232–239
- [TCG+93] Tansel, A.; Clifford, J.; Gadia, S.; Jajodia, S.; Segev, A.; Snodgrass, R.: *Temporal Databases*. Benjamin/Cummings Publishing, Redwood City, 1993
- [Wied92] Wiederhold, G.: Mediators in the architecture of future information systems. In: *IEEE Computer* 25(3)1992, pp. 38–49

Topic 6

Grid Computing and Middleware Systems

Henri Bal, Domenico LaForenza, Thierry Priol, and Peter Kacsuk
Topic Chairs

Grid computing is originated from supercomputing in the mid nineties. The grand challenge problems were not solvable even the largest supercomputers. A natural idea was to run these high-end applications on several supercomputer resources in parallel. This concept led to the new direction of supercomputing, called metacomputing. The success of the first experiments in metacomputing initiated a more general view of metacomputing whereby not only supercomputers but any kind of computational, storage and other types of resources should be connected and exploited on demand. Even the objectives of metacomputing were widened and goals like supporting high-throughput computing, collaborative computing, tele-immersion, etc. became important aspects of this kind of generic view of metacomputing that was called later Grid computing.

Grid computing, streamlined by the successful Global Grid Forum (GGF), has become a major new research area over the past few years, with strong involvement from both academia and the computing industry. Although much progress has been made in the deployment of grid infrastructures, many challenges still lie ahead of us before the ultimate goal of the grid can be realized. Recognizing the importance and potential impact of Grid computing on the whole society we decided to organize a workshop in the framework of Euro-Par where researchers could report their recent advances in Grid middleware design.

The workshop received 28 submitted papers dealing with all possible aspects of Grid middleware research. Finally, after a careful review procedure we selected 6 regular and 4 short papers.

The regular papers cover the following main issues: An RPC system called GrAD-Solve that supports execution of parallel applications over Grid resources; A Grid enabled computational toolkit that provides transparent and stable access to Grid compute resources from Matlab; An opportunistic job migration scheme to decide if job migration is feasible and worthwhile when a new Grid resource appears; Introducing semantic access control to improve security in medical applications over the Grid; An automated negotiation engine that identifies mutually acceptable terms and could be used in the Grid Notification Service; A Grid monitoring system for grid job and re-source monitoring.

The short papers deal with the following problems: The design and implementation of a database toolkit for engineers, which has been incorporated into the Matlab environment; a resource accounting and charging system for a Condor-based Grid environment; a plug-in for the gSOAP Toolkit that allows development of Web Services exploiting the Globus Security Infrastructure; Optimisations of Java RMI for Grid applications.

Implementation of a Grid Computation Toolkit for Design Optimisation with Matlab and Condor

Gang Xue, Matthew J. Fairman, Graeme E. Pound, and Simon J. Cox

Southampton Regional e-Science Centre,
School of Engineering Sciences,
University of Southampton,
Highfield, Southampton, SO17 1BJ, UK
{gx, mjf, gep, sjc}@soton.ac.uk

Abstract. The process of design search and optimisation is characterised by its computationally intensive operations, which produce a problem well suited to Grid computing. Here we present a Grid enabled computation toolkit that provides transparent and stable access to Grid compute resources from Matlab, which offers comprehensive support for the design optimisation processes. In particular, the access and integration of the Condor resource management system has been achieved by using the toolkit components that are enabled by Web service and service enhancement technologies. The use of the computation toolkit for a four-dimensional CFD parameter study with Matlab and Condor is considered as an exemplar problem.

1 Introduction

Engineering design search and optimisation [1] is the process whereby engineering problems are modelled and analysed to yield improved designs. This process involves identifying design parameters that the engineer wishes to optimise, computing a measure of the quality of a particular design (the objective function) using an appropriate model, and using a number of design search algorithms to generate additional information about the behaviour of a model over the parameter space, as well as to optimise the objective function to improve the design quality. It is potentially computationally intensive, as lengthy and repetitive calculations of the objective function with regard to the design variables may be required.

The demand for compute resources by the design optimisation process can be well satisfied by the adoption of Grid computing technologies. Grid computing provides the infrastructure and technologies to enable large-scale resource sharing and the construction of Virtual Organisations (VOs) that address various science and engineering problems [2]. When applied in design optimisation, it allows the process to discover and access resources from heterogeneous environments in a transparent and consistent manner for the required compute tasks. An important aspect of the use of Grid technology for this purpose is to establish links between the design optimisation process and various compute resource management systems, which organise and manage most of the compute resources on the Grid. Notable examples of such systems include Globus [3], Condor [4], Maui [5] and UNICORE [6]. In our previous work [7], access to the Globus system has been provided to Matlab users by

building on the Java CoG toolkit [8]. Here we focus on the integration into Matlab of another important target, the Condor system, using an approach more strongly grounded in Web service technologies.

Condor is a resource management system that *creates a High-Throughput Computing (HTC) environment by harnessing the power of UNIX and NT clusters and workstations* [9]. Whilst it can manage dedicated clusters, it can also exploit pre-existing resources under distributed ownership, such as computers sitting on people's desks. For the design optimisation process, Condor provides a robust computational environment, which is able to find resources with specific requirements, carry out the compute operations, and manage the computation jobs for the process. The HTC style of Condor also suits the need of design optimisation, which can yield better results with further calculations.

The integration of Grid resource management systems with the design optimisation process is mainly based within the Matlab environment [10], which is selected as the interface to the Geodise PSE [7]. The Matlab package provides a fourth generation language for numerical computation, built-in math and graphics functions and numerous specialised toolboxes for advanced mathematics, signal processing and control design. It is widely used in academia and industry for algorithm prototyping, and for data visualisation and analysis. From version 6.5 Matlab also contains a number of Just-In-Time (JIT) acceleration technologies to improve the performance of native Matlab code.

In order to facilitate access to compute resources on the Grid, especially the Condor system, for users in Matlab and potentially other problem solving environments, a computation toolkit has been developed in the form of a set of client utilities and replaceable middleware components. The toolkit adopts a layered, highly flexible and easily configurable structure, which allows using a consistent user interface to access different compute resource systems with the help of corresponding middleware components. In particular, a middleware component based on Web service technology has been constructed to expose Condor as a Grid service, and hence make it accessible to the toolkit users.

In the rest of the paper, we first focus on the design and implementation of the computation toolkit, including the Web service interface to the Condor system. We then demonstrate the application of the toolkit in the Matlab environment for engineering design optimisation problems.

2 The Computation Toolkit

The computation toolkit provides users with a set of basic functions to perform Grid based computation tasks, which are described below together with the technologies behind and the design of the toolkit.

2.1 Functions for Grid Based Computation

In our computation toolkit, a number of functions are implemented both in the form of a low-level Java [11] class library, and a set of high-level commands for the Matlab environment, which enable users to query the resources, submit and manage the

compute jobs, perform data transmission, and configure security settings. The Java class library provides mappings between the APIs and the compute functions. It is therefore possible to perform a Grid based compute operation programmatically. The commands for Matlab users listed in Table 1 are written in the interpretive Matlab language, so that the compute functions are presented in a manner consistent with the behaviour and syntax of the Matlab environment. The commands can be used to start compute operations based on interactive user inputs, or directly on script files.

The `grid_createcredential` command needs to be called before any other commands in the toolkit can be used. It sets up the user credentials according to the security requirement policy of the target compute resource. The user credential can be as simple as a username/password pair, or as complicated as a PKI (Public Key Infrastructure) [12] based user certificate. When all compute operations are finished, the `grid_destroycredential` command is used to disable the user credential, so that it won't be misused or stolen.

The `grid_jobrequest`, `grid_sendfiles` and `grid_startjob` commands represent the major steps in starting a compute job submission process. Users start the compute tasks with requests to the target resources for job submission. Once accepted, the job files are delivered to the resources using the data transfer command. Users then send out instructions to start the computation. In addition, a few job management commands are also provided to enable users to monitor and manage the running of jobs. Once the jobs are finished, job results can be easily retrieved using the `grid_retrievefiles` command.

Table 1. Compute Commands

Function Name	Description
<code>grid_createcredential</code>	Loads the user credential to the toolkit for compute operations
<code>grid_destroycredential</code>	Destroys the user credential stored by the toolkit
<code>grid_jobrequest</code>	Requests the submission of a compute job with specification for the compute resources and description of the job. If accepted, a job handle is returned.
<code>grid_startjob</code>	Starts the submitted job identified by a job handle.
<code>grid_killjob</code>	Terminates the job identified by the job handle.
<code>grid_getstatus</code>	Queries the status of submitted jobs identified by the job handles.
<code>grid_listjobs</code>	Returns job handles for all jobs submitted to the target resource that belong to the user.
<code>grid_sendfiles</code>	Uploads the job files required for job execution to the target resource.
<code>grid_retrievefiles</code>	Retrieves all result data/files generated by the finished job identified by a job handler.
<code>grid_jobsubmit</code>	Automates the process of job request, job files transfer, and job start.

2.2 Design of the Computation Toolkit

One of the most important features of Grid technologies is to provide transparency in accessing computational resources spread across the Grid. It is important because of the heterogeneity of the Grid environment. For instance, compute resource systems that can be targeted by the design optimisation processes include Globus, Condor, and

UNICORE. Each of them has a different user interface, incompatible security policy, and diverse workflow model. Traditionally, separate client packages would be needed to access these different resources, which is clearly unscalable and not ideal for system integration. Moreover, these tightly bound client packages are less adaptable to potential changes made by the target systems, and would force frequent modification to design optimisation applications.

To provide the essential transparency, the computation toolkit has been implemented in a distinctive structure, which separates the user interface from the underlying message processor that is responsible for interactions with different remote resources based on various protocols. The user interface is an API that represents the basic semantics of compute operations and is designed to be simple but stable. In contrast, the message processor is much more dynamic and powerful. It is implemented as two chains of filters for input and output message processing. Each filter is responsible for processing a specific message part, or even the entire message based on the protocol it is responsible for. At the end of the chains, there is a communication handler, which is responsible for direct interactions with the compute resource or compute service middleware. By using Java reflection, the filter chains are dynamically constructed at the runtime based on the information loaded from a configuration file, which is detached from the client applications and can be easily modified. It is therefore possible to access different resource systems or adapt to changes by loading different set of filters.

Figure 1 shows the implementation of the file upload functionality when accessing the Condor service. The interaction is primarily based on the SOAP protocol [13], and needs to conform to the security regulation set by the service. For data transmission, the service uses DIME [14]. HTTP is used as the underlying communication protocol. Accordingly, a SOAP output filter, a Security handler and a DIME builder are loaded in turn to the output chain. Since the response is expected in plain SOAP format, only a SOAP input filter is loaded to the input chain. And at the end of the chains, an HTTP handler is loaded to handle the actual message exchanges.

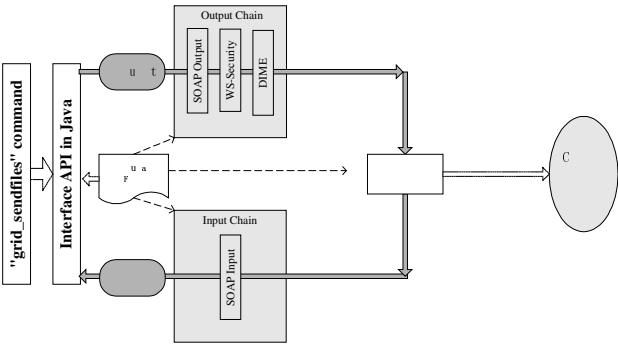


Fig. 1. Implementation of Data Upload with the Message Filter Chains

The flexibility brought by the structure of the computation toolkit allows different compute resource systems accessible in a client-server mode to be integrated into the toolkit. Apart from the middleware component for the Condor system provided in the

current implementation, we will also construct message filters based on our previous work on Globus client tools, so that the Globus system can also be accessed using the toolkit.

3 The Web Service Enabled Interface to Condor

The computation toolkit consists of a set of Web service based middleware components that expose the Condor system. Several enhancements have been applied in addition to the standard Web service technologies to provide features required for Grid based operations. We have also exploited features of Condor and Web service inspection technology to support compute tasks with specific requirements on the resources.

3.1 The Web Service Interface

The Web service enabled interface to the Condor system has been constructed in order to achieve programmatic, platform and language neutral access to the resources managed by Condor. It provides mappings between the computation toolkit functions and the resource management mechanisms of Condor. The interface is primarily an interpreter between XML messages representing the compute operations and the ClassAd language of Condor. It also hides from the users details of resource management that are proprietary to Condor, so as to achieve the desired transparency.

In addition to interactions with Condor, the service also implements general Grid service functions that are not provided by the Condor system, such as management of security and process status. Furthermore, the service has also adopted several enhancements from future Web service technologies, which are described in detail in the following section.

3.2 Enhancements to the Service for Compute Operations

The standard Web services technology collection, which includes XML, SOAP, WSDL and UDDI, has not supplied a solution to the management of service security. To address this problem and make the service compatible with common Grid security management practise such as GSI, we have employed the WS-Security specification in the Condor service implementation. WS-Security [15] is a maturing technology for Web service security management. It extends the simple structure of SOAP to establish a standard security mechanism on the message level, which is independent of the underlying transportation methods. Since WS-Security focuses mainly on the infrastructure, rather than detailed security techniques such as authentication and encryption, it allows established security solutions, including Kerberos [16], Public Key Infrastructure (PKI), and GSI [17] to be integrated so that they can be applied to Web services in a standard and consistent manner.

In the Condor service, we have deployed PKI based asymmetric encryption in order to keep message confidentiality and perform user authentication. The exchanges of credential information, i.e. the X.509 certificates that contain public keys, are carried out following the WS-Security definition. In addition, we will also apply

XML digital signature [18] to the service messages, as proposed by WS-Security, so as to achieve message integrity.

Another problem about interfacing Condor with Web service is the degraded performance in data transfer. Traditionally, data transfer with SOAP is based on the Base64 encoding, which imposes heavy costs due to the serialisation/deserialisation of the Base64 strings. We address this problem in the Condor service by exploiting a recently proposed data transfer format named Direct Internet Message Encapsulation (DIME). DIME defines a MIME-like message format in which variously typed data that does not fit expediently or efficiently into XML can be directly contained and transmitted along with the standard SOAP messages. Significant improvement to the performance on data transfer can be achieved by using DIME, as overheads for data conversion are avoided. Test results for this have been demonstrated in [19].

The enhancements to the Condor service have been implemented with the help of Microsoft's recently released WSE1.0 [20], which provides sound support for several emerging Web service technologies based on the .NET framework. Corresponding to the service enhancements, message filters for the client tools have also been constructed using related Java technologies [21] [22].

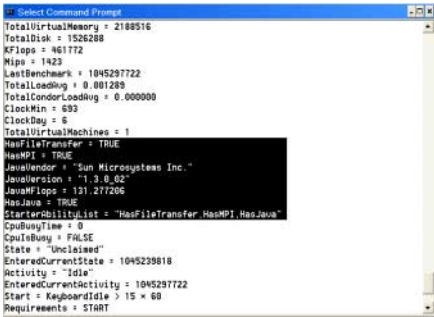
3.3 Discoveries and Access of Resources with Special Capabilities

We define resources to encompass compute facilities with certain features, e.g. memory, processor, disk capacities, database/archive facilities, and also specialist software environment or licensed applications. Design optimisation requires access to all of these resource types at various stages of the process. It is therefore necessary for the users to be able to discover all resources on demand, and manage access to these resources in a consistent way to familiar requests for particular compute capabilities.

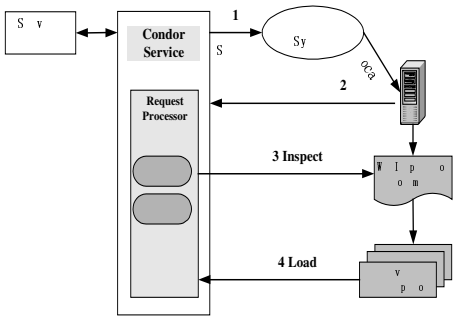
Our service interface to Condor provides the solution to this problem by combining features of Condor and inspection technology for Web services. The Condor system provides a convenient method for the discovery of resources with special capabilities in the Condor pool. When a computer joins the Condor pool, it declares its capabilities by adding corresponding attributes to its Condor system configuration file. These attributes will then be reflected in the information generated by the resource status query performed by Condor, which is passed on to the Web service. The service will therefore be able to make the discovery by identifying the target attributes. Figure 2(a) shows the Condor status query results for a machine that supports Java and MPI.

Additional regulations that need to be enforced for computing on special resources can be presented in XML Schema and WSDL file that extends the original ones of the Condor service. Once the service makes the discovery, it follows the convention defined by the WS-Inspection specification [23] to locate the WS-I documents on the resources, which point to the extended schema and WSDL files. The service will then use the extended service definition to process the incoming requests. The entire process is illustrated in Figure 2(b).

This model of resource discovery and inspection can be extended to suit the situation where resources are presented to the Web service as Web service themselves. In that case the sub-services are registered directly with the interface service, which becomes a service broker in addition to its original functions.



(a)



(b)

Fig. 2. (a) Discovering Resources of Special Capabilities in Condor. (b) Process of Special Capability Discovery through the Service

4 The Computation Toolkit Application Exemplar

To demonstrate the possible use of our Grid-enabled computation toolkit in Matlab, we choose a simple problem of fluid dynamics, which is a parameter study of the fluid dynamics of a 2D parameterised geometry of the nacelle for an aircraft engine. A suitable objective function was calculated across four design variables which describe curves on the upper and lower surface of the nacelle. In this example, each simulation is set up as a compute task, which is submitted to a Condor managed resource from Matlab using the toolkit command, as shown in Figure 3(a).

Figure 3(b) shows the results of 648 simulations that were performed. The jobs were distributed by the Condor system over a cluster of 12 NT nodes. A parameter study such as this allows the engineer to easily evaluate the impact of different design variables upon the quality of a design.

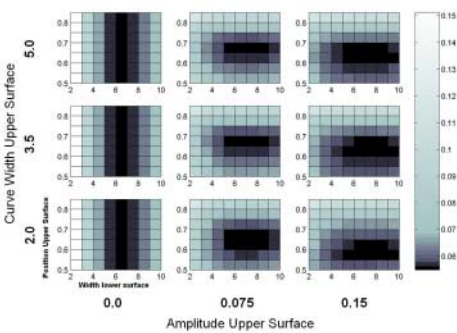
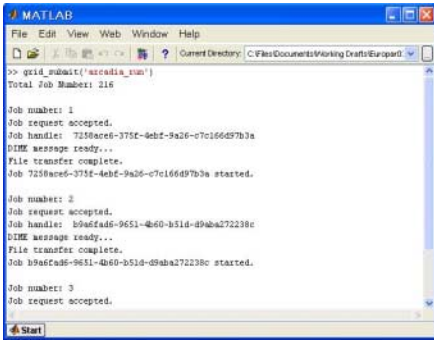


Fig. 3. (a) Starting Grid Computation Tasks Using the Computation Toolkit. (b) Visualised Result of a Four Dimensional CFD Parameter Studies

Once an engineer has developed a Matlab script based on our toolkit for design studies like this, improvements to the robustness and reliability of the underlying Condor service, or additional Condor-enabled resources and capabilities can be exploited in a seamless and transparent way without further modification to the script.

5 Conclusion and Future Work

The significant demand for computation resources in the design optimisation process can be satisfied in a transparent way using Grid computing. The computation toolkit presented in this paper facilitates access to computation resources, in particular the Condor system, from an engineering PSE such as Matlab. The implementation of the toolkit is based on a highly flexible structure and a few future Web service technologies. As an exemplar, we have demonstrated the use of the toolkit for a simple CFD design optimisation problem. Future work on the toolkit will mainly focus on the implementation of filter and middleware components for the access of different computation resources.

Acknowledgements. This work is supported by the Geodise e-Science pilot project (UK EPSRC GR/ R67705/01). We thank the authors of the Condor system at the University of Wisconsin, Madison.

References

- [1] The Geodise Project. <http://www.geodose.org>
- [2] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organisations, *International Journal of Supercomputer Applications*, 15(3):200-222, 2001
- [3] The Globus Project. <http://www.globus.org>
- [4] The Condor Project. <http://www.cs.wisc.edu/condor/>
- [5] The Maui Scheduler. <http://supercluster.org/maui/>
- [6] UNiform Interface to COmputing Resources. <http://www.unicore.de/>
- [7] G. E. Pound, M. H. Eres, et al. A Grid-Enabled Problem Solving Environment (PSE) for Design Optimisation within Matlab. *Proceeding of IPDPS 2003, Nice, France*.
- [8] Commodity Grid Kits. <http://www.globus.org/cog/>
- [9] Miron Livny, and the Condor Team. Condor User Manual. <http://www.cs.wisc.edu/condor>
- [10] Matlab 6.5. <http://www.mathworks.com>
- [11] Java 2. Sun Microsystems Inc., <http://java.sun.com>
- [12] Public Key Infrastructure. <http://www.ietf.org/html.charters/pkix-charter.html>
- [13] Simple Object Access Protocol. <http://www.w3.org/2000/xp/Group/>
- [14] DIME. <http://www.ietf.org/internet-drafts/draft-nielsen-dime-02.txt>
- [15] The WS-Security Specification. <http://www.ibm.com/developerworks/library/ws-secure/>
- [16] Kerberos: The Network Authentication Protocol. <http://web.mit.edu/kerberos/www/>
- [17] Grid Security Infrastructure. <http://www.globus.org/security/>
- [18] The IETF/W3C XML Signature Work Group. <http://www.w3.org/Signature/>

- [19] G. Xue, G. E. Pound, S. J. Cox. Performing Grid Computation with Enhanced Web Service and Service Invocation Technologies. Proceedings of ICCS 2003, Melbourne, Australia.
- [20] Web Services Enhancements 1.0 for Microsoft .NET. <http://msdn.microsoft.com/>
- [21] Java DIME Library v1.0.2. <http://onionnetworks.com/dime/javadoc/>
- [22] Java Cryptography Extension (JCE). <http://java.sun.com/products/jce/>
- [23] Web Service Inspection Language 1.0.
<http://www.ibm.com/developerworks/Webservices /library/ws-wsilspec.html>

Grid Resource Selection for Opportunistic Job Migration^{*}

Rubén S. Montero¹, Eduardo Huedo², and Ignacio M. Llorente^{1,2}

¹ Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain.

² Centro de Astrobiología (Associated to NASA Astrobiology Institute), CSIC-INTA, 28850 Torrejón de Ardoz, Spain.

Abstract. The ability to migrate running applications among different grid resources is generally accepted as the solution to adapt to dynamic resource load, availability and cost. In this paper we focus on opportunistic migration when a new resource becomes available in the Grid. In this situation the performance of the new host, the remaining execution time of the application, and also the proximity of the new resource to the needed data, become critical factors to decide if job migration is feasible and worthwhile. We discuss the extension of the GridWay framework to consider all the previous factors in the resource selection and migration stages in order to improve response times of individual applications. The benefits of the new resource selector will be demonstrated for the execution of a computational fluid dynamics (CFD) code.

1 Introduction

Computational Grids are inherently dynamic environments, being characterized by unpredictable changing conditions, namely: high fault rate, dynamic resource availability, dynamic resource load, and dynamic resource cost. Consequently, in order to obtain a reasonable degree of both application performance and fault tolerance, a job must be able to migrate among the Grid resources adapting itself according to their characteristics, availability, load, and cost.

Probably, the most sensitive step to the above conditions in job scheduling is resource selection, which in turn relies completely in the dynamic information gathered from the Grid. Resource selection usually takes into account the performance offered by the available resources, but it should also consider the proximity between them [1,2]. The size of the files involved in some application domains, like Particle Physics or Bioinformatics, is very large. Hence, the quality of the interconnection between resources, in terms of bandwidth and latency, is a key factor to be considered in resource selection [3]. This fact is specially relevant in the case of adaptive job execution, since job migration requires the transfer of large restart files between the compute hosts.

^{*} This research was supported by Ministerio de Ciencia y Tecnología through the research grant TIC 2002-00334 and Instituto Nacional de Técnica Aeroespacial (INTA).

In this paper, we focus on the opportunistic migration of jobs when a “better” resource is discovered, because either a new resource is added to the Grid, or because the completion of an application frees a Grid resource. Opportunistic migration has been widely studied in the literature [4,5,6,7], previous works have clearly demonstrated the relevance of considering the amount of the computational work already performed by the application, the need of a metric to measure the performance gain due to migration, and the critical factor of dynamic load information of Grid resources. However, previous migration frameworks do not consider the proximity of the computational resources to the needed data, and therefore the potential performance gain can be substantially decremented by the overhead induced by job migration.

The migration and brokering strategies presented in this work have been implemented on top of the GridWay framework [8], whose architecture and main functionalities are briefly described in Section 2. In Section 3 we discuss the extension of the GridWay framework to also consider resource proximity in the resource selection stage. This selection process is then incorporated to the GridWay migration system in Section 4. The benefits of the new resource selector will be demonstrated in Section 5 for the adaptive execution of a CFD code on a research testbed. Finally, Section 6 includes some conclusions and outlines our future work.

2 The GridWay Framework

GridWay is a new Globus-based experimental framework that allows an easier and more efficient execution of jobs on a dynamic Grid environment in a “submit and forget” fashion. The core of the GridWay framework is a personal *submission agent* that automatically performs the steps involved in job submission: system selection, system preparation, submission, monitoring, migration and termination. The user interacts with the framework through a *request manager*, which handles client requests (submit, kill, stop, resume...) and forwards them to the *dispatch manager*. The *dispatch manager* periodically wakes up at each scheduling interval, and tries to submit pending and rescheduled jobs to Grid resources (obtained through the *resource selector* module). Once a job is allocated to a resource, a *submission manager* and a *performance monitor* are started to watch over its correct and efficient execution (see [8] for a detailed description of the architecture of the GridWay framework).

3 The Resource Selector

Due to the heterogeneous and dynamic nature of the Grid, the end-user must establish the requirements that must be met by the target resources (discovery process) and criteria to rank the matched resources (selection process). The attributes needed for resource discovery and selection must be collected from the information services in the Grid testbed, typically the Globus Monitoring and Discovery Service (MDS). Usually, resource discovery is only based on static

attributes (O.S., architecture...) collected from the Grid Information Index Service (GIIS), while resource selection is based on dynamic attributes (disk space, processor load...) obtained from the Grid Resource Information Service (GRIS).

The dynamic network bandwidth and latency between resources will be also considered in the resource brokering scheme. Different strategies to obtain these network performance attributes can be adopted depending on the services available in the testbed. For example, MDS could be configured to provide such information by accessing the Network Weather Service (NWS) [9] or by activating the reporting of GridFTP statistics [10]. Alternatively, the end-user could provide its own network probe scripts or static tables.

The brokering process of the GridWay framework is shown in figure 1. Initially, available compute resources are discovered by accessing the GIIS server and, those resources that do not meet the user-provided requirements are filtered out. At this step, an authorization test (via GRAM ping request) is also performed on each discovered host to guarantee user access to the remote resource. Then, the dynamic attributes of each host are gathered from its local GRIS server. This information is used by an user-provided rank expression to assign a rank to each candidate resource. Finally, the resultant prioritized list of candidate resources is used to dispatch the job.

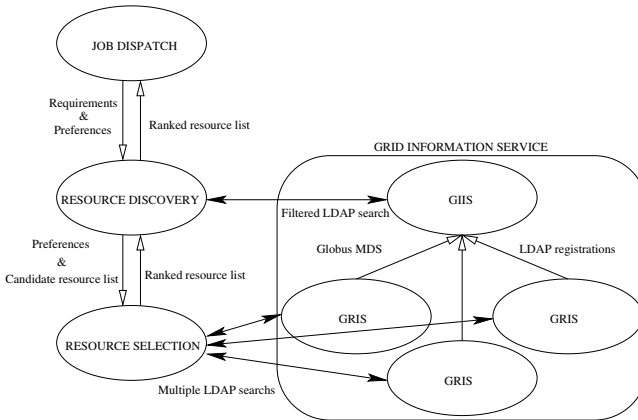


Fig. 1. The brokering process scheme of the GridWay framework

The new selection process presented in this paper considers both dynamic performance and proximity to data of the computational resources. In particular, the following circumstances will be considered in the resource selection stage:

- The estimated computational time on the candidate host being evaluated when the job is submitted from the *client* or migrated from the current *execution host*.
- The proximity between the candidate host being evaluated and the *client* will be considered to reduce the cost of job submission and monitoring, and file staging.

- The proximity between the candidate host being evaluated and a remote *file server* will be also considered to reduce the transfer costs when some input or output files, specified as a GridFTP URL, are stored in such server.
- The proximity between the candidate host being evaluated and the current *execution host* will be also considered to reduce the migration overhead.

3.1 Performance Model

In order to reflect all the circumstances described previously, each candidate host (h_n) will be ranked using the *total* execution time (lowest is best) when the job is submitted or migrated to that host at a given time (t_n). In this case, we can assume that the total execution time can be split into:

$$T_{exe}(h_n, t_n) = T_{cpu}(h_n, t_n) + T_{xfer}(h_n, t_n), \quad (1)$$

where $T_{cpu}(h_n, t_n)$ is the estimated computational time and $T_{xfer}(h_n, t_n)$ is the estimated file transfer time.

Let us first consider a single-host execution, the computational time of the application on host h at time t can be estimated by:

$$T_{cpu}^s(h, t) = \begin{cases} \frac{Op}{FLOPS_{Op}} & \text{if } CPU(t) \geq 1; \\ \frac{Op}{FLOPS \cdot CPU(t)} & \text{if } CPU(t) < 1. \end{cases} \quad (2)$$

where $FLOPS$ is the peak performance achievable by the host CPU, Op is the number of floating point operations of the application, and $CPU(t)$ is the total free CPU at time t , as provided by the MDS default scheme.

However, the above expression is not accurate when the job has been executing on multiple hosts and then is migrated to a new one. In this situation the amount of computational work that have already been performed must be considered [5]. Let us suppose an application that has been executing on hosts $h_0 \dots h_{n-1}$ at times $t_0 \dots t_{n-1}$ and then migrates to host h_n at time t_n , the overall computational time can be estimated by:

$$T_{cpu}(h_n, t_n) = \sum_{i=0}^{n-1} t_{cpu}^i + \left(1 - \sum_{i=0}^{n-1} \frac{t_{cpu}^i}{T_{cpu}^s(h_i, t_i)} \right) T_{cpu}^s(h_n, t_n), \quad (3)$$

where $T_{cpu}^s(h, t)$ is calculated using (2), and t_{cpu}^i is the time the job has been executing on host h_i , as measured by the framework. Note that, expressions 2 and 3 become equivalent when $n = 0$.

Similarly, the following expression estimates the total file transfer time:

$$T_{xfer}(h_n, t_n) = \sum_{i=0}^{n-1} t_{xfer}^i + \sum_j \frac{Data_{h_n, j}}{bw(h_n, j, t_n)} \quad j = client, file\ server, exec\ host, \quad (4)$$

where $bw(h_1, h_2, t)$ is the bandwidth between hosts h_1 and h_2 at time t , $Data_{h_1, h_2}$ is the file size to be transferred between them, and t_{xfer}^i is the file transfer time on host h_i , as measured by the framework.

4 GridWay Support for Adaptive Job Execution

The GridWay framework supports job adaption to changing conditions by means of automatic job migration. Once the job is initially allocated, it is dynamically rescheduled when one of the following events occurs:

- a new “better” resource is discovered (opportunistic migration),
- the remote host or its network connection fails,
- the submitted job is canceled or suspended,
- a performance degradation is detected,
- the requirements or preferences of the application changes (self-migration).

In this work we will concentrate on opportunistic migration. The *dispatch manager* wakes up at each discovery interval, and it tries to find a better host for each job by activating the resource selection process described in Section 3. In order to evaluate the benefits of job migration from the current execution host (h_{n-1}) to each candidate host (h_n), we define the migration gain (G_m) as:

$$G_m = \frac{T_{exe}(h_{n-1}, t_{n-1}) - T_{exe}(h_n, t_n)}{T_{exe}(h_{n-1}, t_{n-1})}, \quad (5)$$

where $T_{exe}(h_{n-1}, t_{n-1})$ is the estimated execution time on current host when the job was submitted to that host, and $T_{exe}(h_n, t_n)$ is the estimated execution time when the application is migrated to the new candidate host. The migration is granted only if the migration gain is greater than an user-defined threshold, otherwise it is rejected. Note that although the migration threshold is fixed for a given job, the migration gain is dynamically computed to take into account the dynamic data transfer overhead, the dynamic host performance, and the application progress. In the experiments presented in Section 5 the migration gain has been fixed to 10%.

5 Experiments

The behavior of the resource selection strategy previously described is demonstrated in the execution of a CFD code, that solves the 3D incompressible Navier-Stokes equations using an iterative multigrid method. In the following experiments, the *client* host is **ursa**, which holds an input file with the simulation parameters, and the *file server* is **cepheus**, which holds the executable and the computational mesh. The output file with the velocity and pressure fields is transferred back to the *client*, **ursa**, to perform post-processing. Table 1 shows the available machines in the testbed, their corresponding CPU performance (MFLOPS), and the maximum bandwidth (MB/s) between them and the hosts involved in the experiment.

We will impose two requirements on the compute resources: a minimum main memory of 128MB, enough to accommodate the CFD simulation; and a total free CPU higher than 90% to prevent oscillating migrations. Initially, the application is submitted to **draco**, since it is the only resource that meet the previous

Table 1. Available machines in the testbed, their CPU performance, and bandwidth between them and the machines involved in the experiment (*client*=ursa, *file server*=cepheus and *exec host*=draco).

host	Model	CPU	OS	Memory	Bandwidth		
					<i>client</i>	<i>file server</i>	<i>exec host</i>
ursa	Sun Blade 100	330	Solaris 8	256MB	∞	0.4	0.4
draco	Sun Ultra 1	175	Solaris 8	128MB	0.4	0.4	∞
columba	Pentium MMX	225	Linux 2.4	160MB	0.4	0.4	0.4
cepheus	Pentium Pro	325	Linux 2.4	64MB	0.4	∞	0.4
solea	Sun Enterprise 250	350	Solaris 8	256MB	0.2	0.2	0.2

requirements. We will evaluate rescheduling strategies based on expressions 3 and 1, when the artificial workload running on *columba* and *solea* completes at different execution points (iterations) of the application running on *draco*.

Let us first suppose that the application is rescheduled using expression 3 (figure 2, left-hand chart). In this case, as the file transfer time is not considered, migration to both hosts always presents a performance gain. The *dispatch manager* will consider feasible the migration to the best ranked host, *solea*, until the eight iteration ($T_{exe} = 302$) is reached.

Figure 2 (right-hand chart) shows the dynamic ranks of *solea* and *columba* when the application is rescheduled using expression 1. In this situation, migration to *solea* only will be granted until the second iteration ($T_{exe} = 325$) is reached. Note that from fifth iteration, the performance gain offered by *solea* and *columba* is not high enough to compensate the file transfer overhead induced by job migration. Moreover, from sixth iteration the best ranked host is *columba* (nearest host) although it presents a CPU performance lower than *solea*, as proximity to data becomes more important as the application progresses.

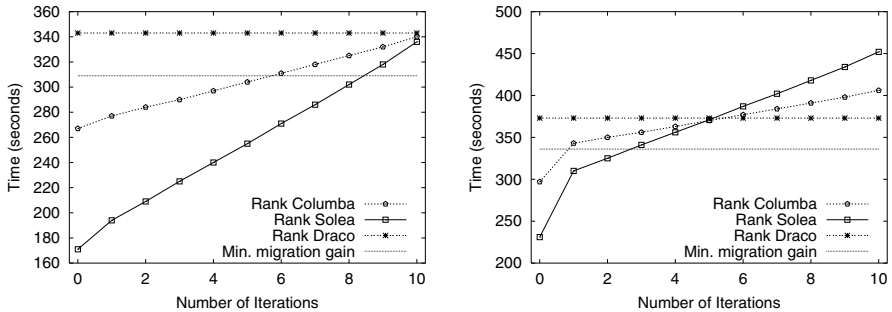


Fig. 2. Estimated execution times (ranks) of the application when it is migrated from draco to different machines at different execution points, using expressions 3 (left-hand chart) and 1 (right-hand chart)

Figure 3 shows the measured execution profile of the application when it is actually migrated to *solea* and *columba* at different iterations, and the execution profile on *draco* without migration. These experimental results clearly show that reschedules based only on host performance and application progress (expression 3) may not yield in performance benefits. In particular, rescheduling the job based on expression 1 results in a performance gain of 13% (12% predicted). This resource selection strategy will reject job migration from third iteration, and prevents performance losses up to 15%, that would occur with the rescheduling strategy based on expression 3. Note that this drop in performance can always be avoided with a pessimistic value of G_m . However using expression 1 allows a more aggressive value of the migration gain threshold, and therefore an improvement in the response time of the application.

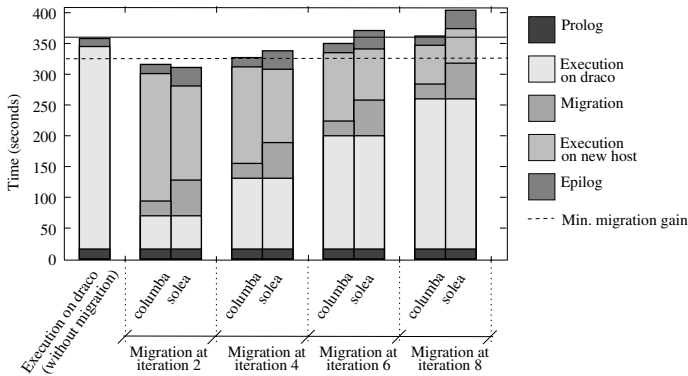


Fig. 3. Execution profile of the application when it is migrated from *draco* to *solea* or *columba* at different execution points

6 Conclusions and Future Work

In this work we have analyzed the relevance of resource proximity in the resource selection process in order to reduce the cost of file staging. In the case of opportunistic migration the quality of the interconnection network has also a decisive impact on the overhead induced by job migration. In this way, considering resource proximity to the needed data is, at least, as important as considering resource performance characteristics.

We would like to note that the decentralized and modular architecture of the GridWay framework guarantees the scalability of the brokering strategy, as well as the range of application, since it is not specialized for a specific application set.

We are currently applying the same ideas presented here to develop a *storage resource selector* program that considers the proximity to a set of replica files belonging to a logical collection. The storage resource selection process is

equivalent to the one presented in figure 1, although the discovery process is performed by accessing the Globus Replica Catalog.

References

1. Liu, C., Yang, L., Foster, I., Angulo, D.: Design and Evaluation of a Resource Selection Framework for Grid Applications. In: Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing. (2002)
2. Kennedy, K., et al.: Toward a Framework for Preparing and Execution Adaptive Grid Applications. In: Proceedings of NSF Next Generation Systems Program Workshop, International Parallel and Distributed Processing Symposium. (2002)
3. Allcock, W., Chervenak, A., Foster, I., Pearlman, L., Welch, V., Wilde, M.: Globus Toolkit Support for Distributed Data-Intensive Science. In: Proceedings of Computing in High Energy Physics (CHEP '01). (2001)
4. Evers, X., de Jongh, J.F.C.M., Boontje, R., Epema, D.H.J., van Dantzig, R.: Condor Flocking: Load Sharing Between Pools of Workstations. Technical Report DUT-TWI-93-104, Delft, The Netherlands (1993)
5. Vadhiyar, S., Dongarra, J.: A Performance Oriented Migration Framework for the Grid. In: Proceedings of the 3rd IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid). (2003)
6. Wolski, R., Shao, G., Berman, F.: Predicting the Cost of Redistribution in Scheduling. In: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Applications. (1997)
7. Allen, G., et al.: The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High-Performance Computing Applications* **15** (2001)
8. Huedo, E., Montero, R.S., Llorente, I.M.: An Experimental Framework for Executing Applications in Dynamic Grid Environments. Technical Report 2002-43, ICASE NASA Langley (2002) submitted to Intl. J. Software Practice & Experience.
9. Wolski, R., Spring, N., Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems* **15** (1999) 757–768
10. Vazhkudai, S., Schopf, J., Foster, I.: Predicting the Performance of Wide-Area Data Transfers. In: Proceedings of 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002). (2002)

Semantic Access Control for Medical Applications in Grid Environments^{*}

Ludwig Seitz, Jean-Marc Pierson, and Lionel Brunie

LIRIS, INSA de Lyon

7, av. Jean Capelle, 69621 Villeurbanne cedex, FRANCE

{ludwig.seitz,jean-marc.pierson, lionel.brunie}@liris.insa-lyon.fr

Abstract. Access control is the field of security which deals with permissions to access resources, where resources may be computing power, storage capacity and data. On the other hand computational grids are systems, where users share those resources in a mostly transparent way. Grid access control poses novel challenges, since the distributed nature of grids make it difficult to manage access control by a central authority. Numerous overlapping domains with different access control policies exist and the sharing of storage resources makes it possible that data leaves the domain of its owner. To enable the owner to enforce his access control policy in such cases, access control solutions adapted to grid environments are needed. In this article we introduce Semantic Access Certificates as an extension to existing access control solutions for grids, to solve some problems that arise when grids are used to process medical data.

1 Introduction

Grid computing [1] is becoming a very popular solution for researchers looking for vast storage and computing capacity. A *computational grid* is a set of computing elements and data storage elements, heterogeneous in hard- and software at geographically distant sites which are connected by a network and use mechanisms to share *resources* as computing power, storage capacity and data. The usage of those computational grids is called *grid computing*.

Different interested parties like high-energy physics, terrestrial observation and genome decoding have recognized the potential value of such tools (see [2], [3] or [4] for application examples).

Another field of application is to use grid infrastructures for medical data processing (see [5], [6]). Goals of such infrastructures can be to provide researchers with a broad spectrum of data for analysis and to make it possible for patients to gain access to their data regardless where they are and where it is stored.

It is clear that in medical data processing privacy protection needs to be enforced much more rigorously than in other application areas. Moreover, the

^{*} This work is partly supported by the Région Rhône-Alpes and the French ministry for research ACI-GRID project (<http://www-sop.inria.fr/aci/grid/public/>)

users of such systems (patient, medical staff) will mostly not be trained in computer security and thus need easy to use and mostly transparent services. Thus we need tools for classical security tasks like *authentication*, *data integrity*, *private communications* and *access control* in grid environments. Authentication is the task of proving ones identity over a computer network. Data integrity means to ensure that data has not been fraudulently manipulated, which can be done through *electronic signatures*. In private communications, involved parties want to communicate confidential data over an insecure channel, here *encryption* is the technique that is most commonly used. These fields of security are well covered in common applications and have been adapted to grid environments without major changes (see [7] or [8] for examples).

This article focuses on access control for medical data in grid environments. We define access control as the domain of security, which deals with the permissions of *entities* to access *resources*, where entities may be users, groups of users, programs, daemons etc. and resources may be computing power, storage capacity and data. Unlike the other security aspects the structure of the grid poses novel problems for access control that can not be dealt with by classical solutions. These problems arise from the distributed nature of grid users and grid resources, where user activity may cross organizational boundaries, and data may be stored outside the user's domain of control. This also means that users can have to handle the access rights of data belonging to the same semantic context but physically distributed on different storage elements. As we consider the patient to be the owner of his data in our medical application, flexible mechanisms are needed, that permit the patient to manage the rights of his data, even when he is not connected to its storage site.

The structure of the article is as follows: In section 2 we describe the access control problems, arising specially in medical applications on grid environments. We then extract some desirable functionalities of an access control system for those applications. In section 3 we present classical approaches to access control while section 4 gives grid specific solutions. In section 5 we propose an extension to the existing grid mechanisms, designed to provide the desirable functionalities defined in section 2: access control through *Semantic Access Certificates*. Finally we summarize and give directions of our future work in section 6.

2 Medical Data on Grids

We present a somewhat constructed but still realistic use-case, to illustrate a class of access control problems we want to address in this article.

Imagine a patient who visits a new doctor (for instance a patient in vacation visiting a doctor far from home). He wants to give this doctor access to some of his medical data, for example radiographs made at a certain hospital and the medical files of his former doctor. He doesn't know the storage element, neither does he know the filenames or directories where his data is. The patient only wants to interact with the doctor and not with the storage site to grant the access. If a large number of medical files are concerned, he would prefer to grant

access to all those files at once through some kind of classification mechanism, and not separately for every single file.

On the other hand the doctor wants to use the rights given to him to locate and retrieve the data. If the treatment is done by a team of doctors, all team members should be able to use the access rights. The doctor also needs to be able to access future data of the patient, necessary for treatment and diagnosis. He doesn't want to have to take difficult security measures to protect the permissions that were given to him by his patients, since this is not his specialty.

From this example we extract the following list of desirable functionalities specific to medical data processing.

- *Group granting*: It should be possible to specify permissions for groups of doctors, thus a patient could give medical teams access to necessary data in one operation.
- *Class granting*: It should be possible to specify permissions to classes of data, to enable the patient to give access to all data concerning a certain field of medicine in one operation.
- *Storage location independence*: Patients should be able to specify permissions without exact knowledge of the resource storage sites, because a system that requires the patient to keep track of his medical data and remember how to index it in the system would neither be accepted nor would it be usable.
- *Offline delegation*: Patients should be able to assign permissions offline, without being connected to the storage sites. Since the patient is the owner of his medical data, and the storage site is mostly some kind of laboratory or clinic it would be cumbersome, if the patient had to contact the storage site, every time he wants to grant access to some of his data.
- *Personalized permissions*: The assigned permissions should be bound to a specific entity or group of entities, for a given amount of time and not be usable by unauthorized entities. This minimizes the impact of loss or theft of those permissions.

In the two following sections, we explain why classical access control schema do not address completely all these problems before giving a possible solution through enriched access certificates.

3 Classical Access Control

The UNIX style access control [9] with read, write and execute permissions and its user, group and other classification has the benefit of simplicity. However it does not provide any functionality to deal with distributed data of the same class. Access rights management has to be done online and there is no way to define specific access rights for a user that has no account on the system.

Using *access control lists (ACL)* [10] adds some flexibility, however it is a local access control system as well and does neither provide offline access rights granting and distributed data classes management.

Role based access control (RBAC) [11] groups entities and resources by so called *roles*. Those are collections of entities and access rights grouped together based on different tasks they perform in the system environment. Entities are given the right to take certain roles and thus use their permissions, to be able to use the resources that are necessary for their tasks. The RBAC approach permits to group access rights by semantic meanings, but to deal with distributed data, extensions of the classical RBAC approaches are needed. Also pure RBAC is somewhat non flexible for the granting of specific rights, since it only permits to grant rights by defining an appropriate role and assigning users the right to use it.

One such extension, that aims specifically at distributed systems is *OASIS* [12],[13]. OASIS differs from other RBAC schemes in three major ways. First the role management is decentralized, which means that every service in the distributed system may define their own roles and control the conditions how those can be used. The services interoperate through Service Level Agreements (SLA). Second OASIS permits to parameterize roles, which allows to define generic roles (e.g. A is doctor of B) where the parameters can be chosen for specific applications. Third it replaces privilege delegation by a mechanism called *appointment*. Certain roles have the function to permit the issuing of appointment certificates, that permit to activate roles together with an authentication mechanism. The OASIS approach permits to manage access control for distributed data of the same semantic class, however it does not permit offline granting of access rights, since one has to activate an appointment role to grant rights.

4 Grid Access Control Solutions

This section deals with today grid solutions for access control. We present three common grid computing environments, Condor [14], Legion [15] and Globus [16] and we argue why their solutions have to be extended with further functionality for medical grid applications.

Condor [14] is a specialized workload management system for compute-intensive jobs. Its resource management mechanisms are similar to UNIX style access control. Main differences are some additional modes of access besides traditional read and write permissions. Also the classical meanings of the read and write permissions are slightly changed, since all permissions are not relative to files, but to the Condor system itself. Therefore read access means the permission to access information from Condor, write to change settings etc.

The Legion project [15] uses an object oriented approach to grid computing. Thus resources, as files, services and devices are considered as objects and access to those are through functions of these objects. The Legion access control approach is that each object is responsible for enforcing own access control policy. Thus each object has a *MayI* function, invoked before any other functions of the object may be called. This function allows resource owners to define their own access control mechanisms. A default MayI implementation exists that is based on ACL's and credential checking (authentication).

The Globus grid toolkit [16] proposes mechanisms for translating users grid identities into local identities. This would allow users to sign onto the grid and then use resources without further authentication (single sign-on). The benefit for local resource providers would be that through the translations mechanism, local access control policies can be enforced.

More recently Pearlman et al.[17] have also been working on an access control system that relies on a Community Authorization Service (CAS). The idea is that users shall be able to request access right at a CAS server, which will assign them capabilities, based on the request and the user's role within the community. The user can present those capabilities at a resource server to gain access on behalf of the community. Thus this approach integrates the benefits of RBAC into a grid access control system.

All approaches described above lack some mechanisms for managing access to distributed data belonging to a common semantical class and they do not provide mechanisms for offline granting of rights. Therefore they have to be extended for use in our medical grid applications.

5 Semantic Access Certificates

A number of classical approaches exist that grant access to data through the use of electronic documents, like the Kerberos V5 protocol [18] access tickets, however these approaches do not permit the management of transparent access to distributed data, since the storage site has to be known, to issue a valid ticket. We have therefore decided to enrich the concept of access tickets to get rid of those limitations.

5.1 The Structure of a Semantic Access Certificate

We will use a doctor/patient scenario as in our motivation to describe the structure and use of access certificates, this does of course not mean, that we limit ourselves to use in this context, but serves to clarify the relations between the involved entities.

We first emphasize the medical data we want to give (or refuse !) access to. Upon creation of a medical data file, it will be assigned a certain set of meta-data. This set necessarily contains an unique identifier of the patient that owns the data (since one of our assumption is that one data belong to one patient), the date of creation, and a hash of the data signed by the patient to guaranty integrity.

Additionally, data can be associated to classes, to allow for more coarse grained access control decisions. In medical applications, classifying data by medical specialty allows a patient to give access rights to a doctor for the specific class of data that is needed for his treatment or diagnosis. A patient visiting a dentist should give him access to all dental data and medicine allergy, while not giving access to other classes of information such as psychological analysis or lung radiographs. Some ontology of the medical domain should allow for a fine

classification of the data produced at different medical services (but note here that such an ontology does not currently exist globally, but this research field is very active today, see the US Unified Medical Language System for instance [19]).

We define *Semantic Access Certificates (SAC)* as electronic documents that contain the following information (see figure 1):

- an unique identifier of the doctors that are granted access,
- an unique identifier of the patient granting access,
- an unique identifier of the data or class of data to which access is granted,
- the allowed modes of access,
- a validity period,
- and the electronic signature of the entity granting the rights.

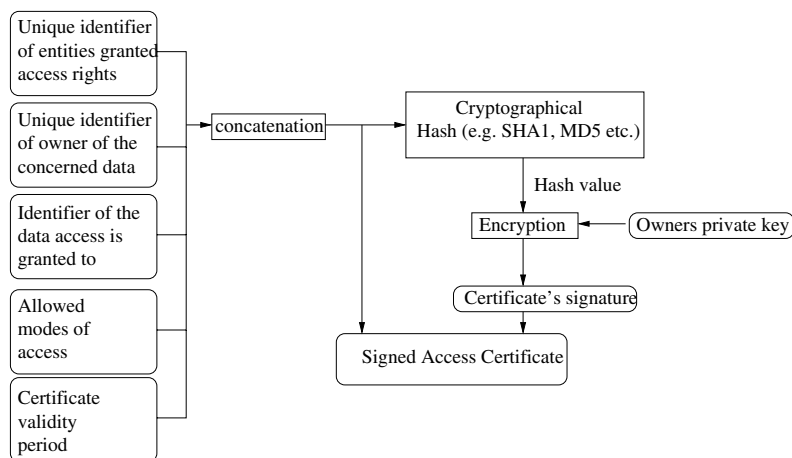


Fig. 1. Components of a signed access certificate

Let us now detail and argue each of the fields present in this certificate.

First appears the doctor (or the group) for whom the certificate has been issued : For a single doctor its public key may be used as unique identifier, management of groups of medical staff is done through trusted authorities that issue group membership certificates.

Similarly, the unique identifier of the patient may be its public key.

Creating unique identifiers for files can be done by concatenating the patient identifier with a hash (MD5, SHA1, ...) of the file content (or the semantic class name). The semantic value transported with the certificate when classes of access are given that way allows for a much more flexible and easy to use system. The users can thus effectively give access to entire part of his medical data. The classification of data allows to give access rights based on semantical criterions,

as it is independent of storage sites, even access to distributed data of the same semantical context may be managed this way.

Fourth, the allowed access mode should be as simple as “read”, “write” or “read/write” or more complicated such as “read anonymised” (for instance when an user wants to give access to some of its medical information to medical researchers for epidemiologic studies) or “append information” (when an user wants the doctor to add new information to its medical file). The handling of these different access mode lies beyond the scope of this article.

The validity period in access certificates is generally set so that a hacker stealing the certificate won’t be able to decode it and use it before it expires. We see another interest for us : a patient should want a doctor to access his medical data during a given period of time (for instance while in the doctor office or for the treatment duration, but not longer).

Finally, the electronic signature allows classically for the integrity of the issued certificate.

To revoke SACs the patient will be able to issue a SAC revocation request, that uses the identifiers of the patient and the doctor holding the SAC with the signature to form an unique identifier of the SAC. This revocation request will be broadcasted to the resource server sites, in a similar way as certificate revocation lists.

As the patient owns his data, he is the only one that can issue access certificates or change the data. A copy of the unique file identifiers of all his data need to be stored with him, so he can issue access certificates to specific data files. If the data is associated to a class, the classification also needs to be stored with him. For the revocation mechanism, the unique identifiers of the certificates need to be stored with the patients too. Such information will be stored on a smart-card or a similar cryptographic token along with the patients public/private key pair. Those tokens will be kept by the patients like today medical insurance cards.

5.2 Usage of SAC

Let’s imagine a scenario where a doctor has successfully acquired a valid SAC and wants to access some data. The process would have the following actors: The doctor requesting some data, the doctors grid interface, a resource broker mechanism provided by the grid infrastructure to locate resources and resource server sites that offer access to resources.

The request formulated by the doctor would be translated by his grid interface into a grid readable format, like for example a request to the Globus GASS system [20]. The exact format of the request lies beyond the scope of this article and shall thus not be treated any further.

The whole process that follows the submission of a request should remain transparent to the doctor, until the request results in either success or definitive failure.

Figure 2 illustrates the use of access certificates and the protocol is specified here:

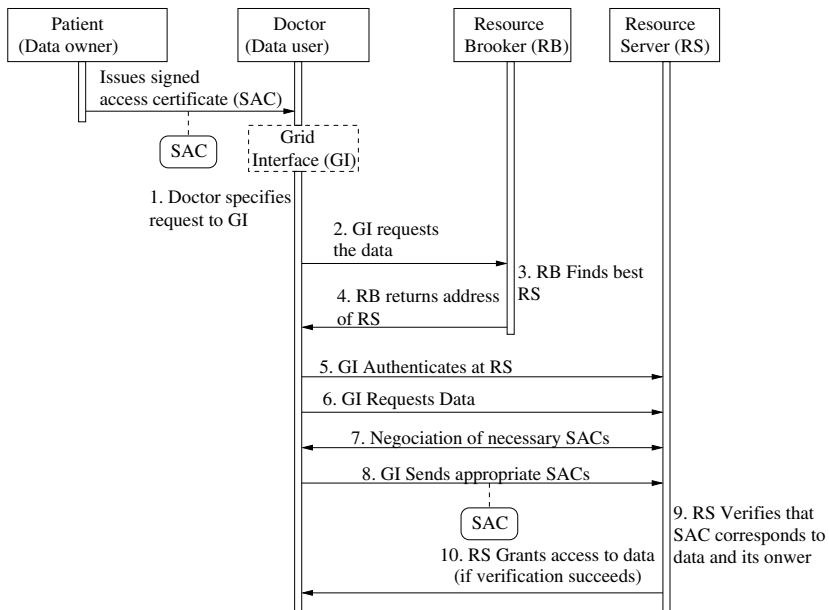


Fig. 2. Usage of access certificates

1. The doctor specifies what data it wants to his grid user interface.
2. The doctor's grid interface contacts the resource broker and submits the data request.
3. The resource broker locates the best resource server¹ that has the requested data available.
4. The resource broker transmits the address to the doctor's grid interface.
5. The doctor's grid interface authenticates at the resource server.
6. The doctor's grid interface transmits the data request.
7. The doctor's grid interface and the resource server negotiate which access certificates are needed for the requested access. The negotiation is carried out in the following steps:
 - a) The doctor's grid interface submits the request to the resource server
 - b) The resource server locates the requested data's meta-data section and extracts the patient's unique identifier and the data classification if one exists. It then returns those to the grid interface.
 - c) The grid interface uses the provided meta-data to scan the doctor's access certificate repository for appropriate access certificates and returns those to the resource broker.

¹ The determination of the best server could be done by various criterions like: the one with the fastest connection, one in the same country for legal reasons, a trusted site etc.

8. The doctor's grid interface sends the appropriate access certificates to the resource server.
9. The resource server checks that the access certificates bears a valid signature by the patient owning the data.
10. If access can be granted the resource server gives access to the grid interface, otherwise it returns an appropriate request failure message.
11. If the access is granted the grid interface proposes the access to the doctor, if the request failed due to insufficient permission the doctor is notified, if the request failed for other reasons, the grid interface resubmits the request to the resource broker, excluding the addresses of the previously contacted resource servers in this new request.

In section 2, we have exhibited five desirable functionalities of access control in medical area. We now address their adequation with the SAC definition and protocol. The SACs allow directly, from their definition, for *Group granting* and *Class granting*, a certificate being issued for a doctor or group of doctors and for a data or a given class of data). *Storage location independence* is assured since the identifier depends only on the patient and the content of the data. Thus any replica of the data stored on the grid can be accessed with the same SAC. Access certificates can be granted offline (*Offline delegation*) i.e. without being connected to the resource server site. The patient simply has to prepare the electronic document in the right format, to assure he has the right identifiers and to sign it digitally. From this point on, the certificate is bound to the doctor or group of doctors (*Personalized permissions*) it was issued for and can be transmitted even through an insecure channel. The electronic signature of the patient ensures its integrity and thus no unauthorized person can use the SAC.

6 Conclusion

We have proposed in this article a novel approach to access right management on grids, in the medical data field. Our contribution is the definition of a Semantic Access Certificate (SAC), holding the access control policy not only based on roles (patient, doctor, nurse, etc.) but also on the semantic of the stored data (the specialty it belongs to). While being illustrated here in the medical field, the proposed schema can be used in any application where privacy protection have to be enforced.

However this article only covers a part of the access control problems arising in medical data processing on the grid. Special modes of access like anonymized read for medical research, problems of access logging and data persistence are not treated in this article. Therefore it will need to be expanded and should not be used alone. We are currently working on the design of reactive documents that will carry their special access mode programs, access logging and perenity enforcement mechanisms with them. These will allow the users to specify their own access modes, logging and perenity conditions and to enforce them, even if the data is stored at untrusted sites.

References

1. Foster, I., Kesselman, C., eds.: The Grid Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Inc. (1999)
2. European Data Grid: The datagrid project. <http://eu-datagrid.web.cern.ch/eu-datagrid/> (2001)
3. Cactus Community: The cactus code. <http://www.cactuscode.org/> (2002)
4. National Institute of Advanced Industrial Science and Technology: Grid technology research center. <http://www.aist.go.jp/> (2002)
5. MEDIGRID: Medical data storage and processing on the GRID. <http://creatis-www.insa-lyon.fr/MEDIGRID> (2002)
6. University of Manchester: Manchester Visualization Centre: ix-grid. <http://www.sve.man.ac.uk/mvc/Research/iX-Grid/> (2002)
7. Foster, I., Kesselman, C., Tsudik, G., Tuecke, S.: A security architecture for computational grids. In: Proceedings of the 5th ACM Conference on Computer and Communications Security Conference. (1998) 83–92
8. Ferrari, A., Knabe, F., Humphrey, M., Chapin, S., A.Grimshaw: A flexible security system for metacomputing environments. In: Proceedings of the High Performance Computing and Networking conference 99 (HPCN'99). (1999)
9. Nemeth, E., Snyder, G., Seebass, S., Hein, T.: UNIX System Administration Handbook (3rd Edition). Prentice Hall PTR (2000)
10. Lampson, B.: Protection. In: Proc. of the 5th Princeton Conf. on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev. Volume 8, 1. (1974) 18–24
11. Sandhu, R.: Role-based access control. In: Advances in Computers. Volume 46. Academic Press (1998)
12. Yao, W., Moody, K., Bacon, J.: A Model of OASIS Role-Based Access Control and its Support for Active Security. In: Proceedings of Sixth ACM Symposium on Access Control Models and Technologies, SACMAT. (2001) 171–181
13. Bacon, J., Moody, K., Yao, W.: Access Control and Trust in the use of Widely Distributed Services. In: Lecture Notes in Computer Science. Volume 2218. Springer (2001) 295–310
14. Condor Team of the University of Wisconsin: Condor, high throughput computing. <http://www.cs.wisc.edu/condor/> (1988)
15. Legion Research Group of the University of Virginia: Legion, a worldwide virtual computer. <http://legion.virginia.edu/> (1993)
16. Globus Project: Globus toolkit. <http://www.globus.org/> (1998)
17. Pearlman, L., Welch, V., Foster, I., Kesselman, C., Tuecke, S.: A community authorization service for group collaboration. In: Proceedings of the 2002 IEEE Workshop on Policies for Distributed Systems and Networks. (2002)
18. Kohl, J., Neuman, C.: The Kerberos Network Authentication Service (V5). Technical report, The Internet Engineering Task Force IETF (1993) <http://www.ietf.org/rfc/rfc1510.txt>.
19. UMLS: Unified medical language system. (<http://www.nlm.nih.gov/research/umls/>)
20. Bester, J., Foster, I., Kesselman, C., Tedesco, J., Tuecke, S.: Gass: A data movement and access service for wide area computing systems. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems. (1999)

Automated Negotiation for Grid Notification Services

Richard Lawley¹, Keith Decker², Michael Luck¹, Terry Payne¹, and Luc Moreau¹

¹ Department of Electronics and Computer Science, University of Southampton, UK
{ral01r,mml,trp,L.Moreau}@ecs.soton.ac.uk

² Computer and Information Sciences Department, University of Delaware, Newark, DE 19716
decker@cis.udel.edu

Abstract. Notification Services mediate between information publishers and consumers that wish to subscribe to periodic updates. In many cases, however, there is a mismatch between the dissemination of these updates and the delivery preferences of the consumer, often in terms of frequency of delivery, quality, etc. In this paper, we present an automated negotiation engine that identifies mutually acceptable terms; we study its performance, and discuss its application to a Grid Notification Service. We also demonstrate how the negotiation engine enables users to control the Quality of Service levels they require.

1 Introduction

Notification services play an important role within distributed systems, by acting as intermediaries responsible for the asynchronous delivery of messages between publishers and consumers. Publishers (such as information services) provide information which is then filtered and delivered to subscribed consumers [1,2,3] based on a specification of topic and delivery parameters. Notification service features include persistent, reliable delivery and prioritisation of messages. Notifications may include announcements of changes in the content of databases [4], new releases of tools or services, and the termination of workflow execution. Notification services can also be used for replicating service directory contents [5]. As such, the Grid community has recognised the essential nature of notification services such as the Grid Monitoring Architecture [6], the Grid Notification Framework [7], the logging interface of the Open Grid Services Architecture [8] and peer-to-peer high performance messaging systems like NaradaBrokering [9]. They are also core architectural elements within the MyGrid [10] project.

While the mechanisms for asynchronous notifications are well understood and robust implementations can be found, some issues still remain open. For example, providers hosting databases in the bioinformatics domain prefer to control the frequency at which notifications are published (such as daily digests), and discourage clients from continually polling for changes. However, clients have their own preferences about the frequency, format, quality or accuracy of the information being propagated. Similarly, many services within this domain are hosted by public institutions and are free to the community, but there are also paying customers expecting a certain quality of service from providers. The prices charged for notifications will affect the type (e.g. quality and frequency) of messages sent. As these examples suggest, both providers and consumers have preferences about the way notifications should be published, yet current notification

service technologies provide no support for determining a set of parameters that would be acceptable to both parties.

Automatically finding mutually acceptable notification parameters for a set of terms (e.g. price, bandwidth, etc) can be viewed as a search for an optimum in a multidimensional space. A *cooperative* approach requires both parties to make valuation functions and preferences available to a search component. Here, preferences and utility functions are shared, enabling the optimal values to be found. However, it is not always possible to share preferences and valuation functions freely — businesses may view preferences as private, and valuations may be linked to a locally sensed environment. In these situations, an automatic search is not possible, as an optimum cannot be calculated. These situations require *competitive* approaches using mechanisms such as *negotiation*. Various approaches exist; in this paper we present a *bilateral negotiation framework* [11] that is applicable to the context of notification service negotiation.

Our contributions are a practical implementation of an automatic negotiation engine, a study in terms of performance of bilateral negotiation and a study of negotiation in the specific context of notification services. This paper is organised as follows. Section 2 discusses negotiation in general. Section 3 describes the design of our system. In Section 4 we study the negotiation engine in general and more specifically in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Negotiation

Negotiation is the process by which two or more parties exchange proposals in order to reach a mutually acceptable agreement on a particular matter. Parties in negotiation exchange *proposals* [12] that are either accepted or rejected. Rejection involves either turning down the proposal and allowing another to be sent or submitting a *counter-proposal*, so that both parties converge towards an agreement. *Utility functions* (used to evaluate a proposal) and *preferences* (defining an acceptable range of values for each term) remain private and, because they are stored locally, can be linked to external conditions such as resource levels. For this reason, it is not practical to use cooperative searching in a Grid environment, where limited system resources need to be allocated. One solution is Faratin's *negotiation decision functions* [11] algorithm, which is a bilateral negotiation model that allows external resource functions to be used to evaluate proposals and generate counter-proposals.

In Faratin's algorithm, the methods for generating proposals and counter-proposals are based on *tactics* and *strategies*. Tactics are functions that generate the value for a single negotiation term for inclusion in a proposal, and come in different flavours: *time-dependent* tactics use the amount of time remaining in the negotiation thread to concede, whereas *resource-dependent* tactics use a resource function to determine how much of a particular resource is consumed. This resource may be the number of negotiations currently taking place or the load on the system, and may involve callbacks to monitor external resources. To combine different tactics during counter-proposal generation, strategies are used. These modify the weightings given to each tactic, which can be changed during a negotiation, for example to honour a resource-dependent tactic at the start of a negotiation and a time-dependent one nearer the deadline. Utility functions

evaluate the utility of a single negotiation term in a proposal. These can be simple linear functions or more complex callback functions to link the utility to resource conditions. The utility of a proposal is a weighted summation of the utility of the elements within the proposal. Proposals become acceptable when the counter-proposal generated has lower utility than the incoming proposal. Faratin's algorithm is the basis for the rest of this paper.

3 Negotiation Engine Description

Our system is intended to allow applications to be programmed without specific knowledge of negotiation protocols. The actual negotiation mechanism is based on the work of Faratin [11], which allows a host to easily supply external conditions, such as resource levels, as inputs into the negotiation process without knowing *how* to negotiate.

Negotiations take place between a *requester* and a *requestee*, and we assume that there is always an item that is the subject of the negotiation. The conditions being negotiated over are called *negotiation terms*, and can be such things as cost or duration. A conversation between a requester and requestee, where proposals are exchanged, is called a *negotiation thread*. Each party has a set of *preferences*, which consist of two values: an *ideal value* and a *reservation value*. The ideal value represents the initial negotiating value (i.e. the value the party would like to get in an ideal world) while the reservation value is the limiting point of concession. Values beyond the reservation value are unacceptable, and the negotiation will fail if it is not possible to find a value that satisfies all preferences. The negotiations in this system work to *deadlines* that are measured in terms of the number of messages exchanged between the two parties — the *negotiation thread length*. We refer to the application containing the negotiation component as a *host* and to the complete system as the *negotiation engine*.

Although we conceptually regard the negotiation engine as an entity shared by the requester and requestee, it has been implemented as negotiation components distributed between each party. This approach maintains privacy of preferences and utility functions.

A description of the negotiation process is depicted in Figure 1. Before a negotiation starts, both parties initialise their negotiation components with their preferences. Then, negotiation process proper is initiated with the requester sending a proposal to the requestee. The communication mechanism is left for the host to implement.

As discussed in the previous section, when a proposal is received by party *p*, a counter-proposal is generated. Using *p*'s utility functions, *p*'s negotiation component determines if the utility of the counter-proposal is higher than the incoming proposal. If so, the counter-proposal is sent. This cycle continues until the incoming proposal has a higher utility, at which point an acceptance message is sent. Both negotiation components then give their hosts a successful proposal. Note that acceptance of a proposal is not a *commitment* — it is left to the host to commit, allowing negotiations with many parties.

If the deadline passes before a successful proposal is found, a failure message is sent to the other party, and the hosts are notified. Negotiations can also be terminated by means of a failure message for reasons such as a system shutting down or a deal being made with another party.

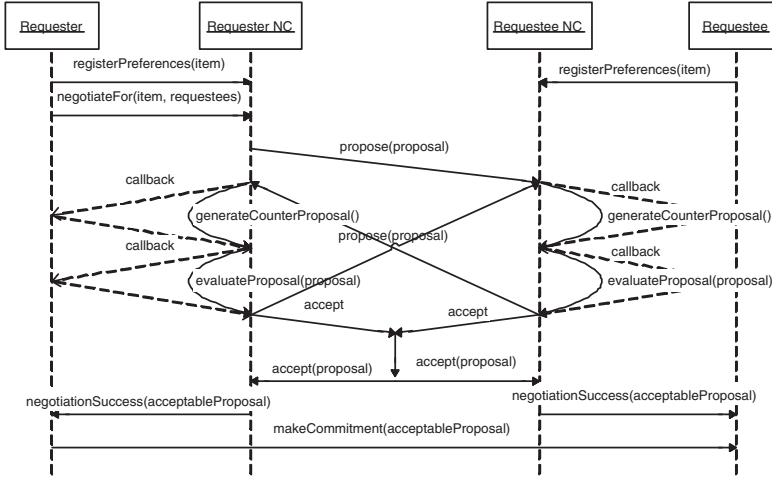


Fig. 1. Sequence diagram showing message flow during a negotiation

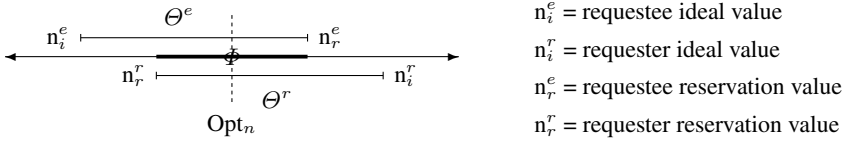


Fig. 2. Effect of environment parameters on preferences, also showing optimal value

4 Experimental Evaluation of Negotiation Engine

To verify the suitability of our negotiation engine for a Grid notification service, we must check that it scales up predictably to handle more negotiation terms, and longer negotiations, without any adverse performance. To determine that this component is suitable for our purposes, we performed a number of experiments.

4.1 Experiment Setup

The set of varying factors such as acceptable ranges and deadlines are grouped into an *environment*. Running two negotiations in the same environment produces identical results. As there is an infinitely large space of environments we generated a range of random environments using the methods from Faratin [11].

For each term, a fixed value min^r was chosen, representing the requester's reservation value. Random values within predefined ranges were assigned to the parameters Θ^r , Θ^e and Φ , where Θ^r represents the size of the acceptable region for the requester r , Θ^e represents the size of the acceptable region for requestee e , and Φ represents the degree of overlap between the acceptable ranges, with 0.99 indicating almost no overlap and 0 indicating complete overlap. These parameters are illustrated in Figure 2.

We used six tactics — three from the time-dependent and three from the resource-dependent families as in [11]. Utility functions are linear functions based on the preferences.

The experiments all had the same basic structure — each tactic was played against each of the tactics (including itself) in each of the generated environments. This allows us to build up some average values demonstrating the sort of results we should expect from a real-world implementation.

As the design of the system is independent of any communication mechanism we coupled the negotiation components together directly using method calls. Experiments measuring time only examined execution time.

4.2 Hypotheses and Results

The experiments described in this section were intended to determine the effects of varying both the deadline for negotiations and the number of negotiation terms. We consider the number of messages exchanged in a negotiation to be the primary component of time, as the dominant factor for message exchange in a real system is the transmission time. By contrast, transmission time in our experiments is very low, since the components are coupled by method calls, but we also measured *execution* time at the end of this section.

Variable Deadline. Sometimes, negotiations should be completed within short deadlines, but this yields worse results than long deadlines [11]. To examine how the utility varies with the deadline we varied the deadline between 1 and 100 messages, using a single negotiation term.

Hypothesis: *With short deadlines the utility to both parties is poor. As deadlines increase, utility also increases, but at a decreasing rate, since a utility of 1 would indicate that no concessions were made, and is unlikely. The percentage of successful deals made increases as the deadline increases.*

Figure 3A shows that the utility (U_r and U_e) for both parties is low for short deadlines. As the deadline increases, the utility also increases. The average optimal utilities (Opt_r and Opt_e) are plotted on the graph — these are time-independent and therefore constant. They appear to be asymptotes to the utility curve. Figure 3B shows that the percentage of successful negotiations has a similar curve, fitting our assumption that there is a predictable curve that can be used to determine the effect of limiting the deadlines of negotiations. This can be used to determine what sort of values should be used to limit the length of a negotiation thread without trading-off too much utility.

To determine how much of the available time is used, we examined a subset of the data using negotiations between time-dependent linear tactics. We plotted the utilities of the outcomes and the time taken for the negotiations for each environment, sorted by increasing client utility and provider utility. Figure 3C shows that the amount of time used in a negotiation ranges between 30% and 100% of the available time. There appears to be an interesting trend where the utility increased as the number of messages decreased. Our explanation for this is that the negotiations taking less time and giving greater utility have a better environment to negotiate in. The parameter of the environment that has the

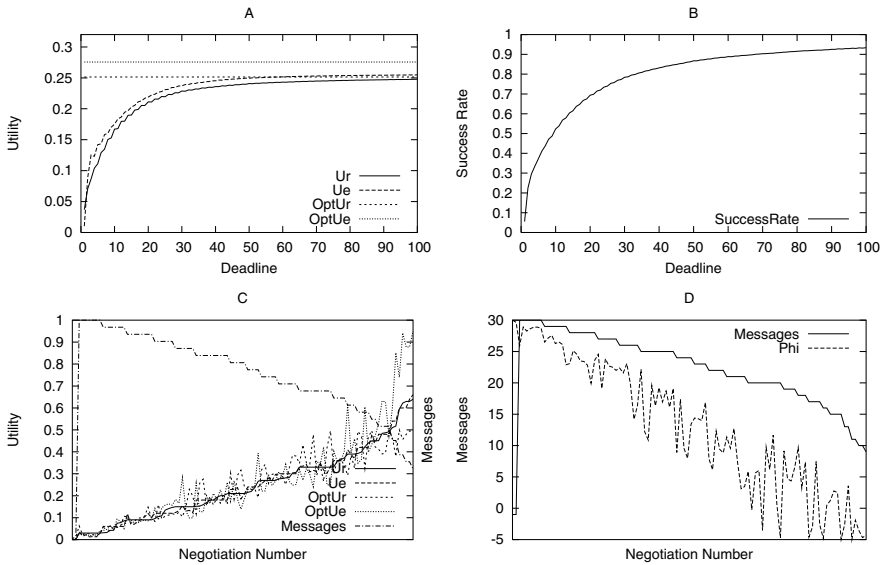


Fig. 3. A) Utilities, B) Success Rate, C) Utility vs. time used, D) Time vs. Φ

most significance is Φ , controlling the amount of overlap between the acceptable regions. This is plotted against the number of messages exchanged in Figure 3D, confirming this theory — negotiation finishes quicker with better negotiation environments and gives a greater overall utility.

Multiple Negotiation Terms. If this negotiation engine were to be deployed in the notification service, it would not be negotiating over a single negotiation term. There would be many terms, so we must ensure that the system scales up as the number of terms increases. In consequence, the negotiations were evaluated with the number of negotiation terms set at every value between 1 and 25. The experiments used deadlines of between 30 and 60 messages.

Hypothesis: *As the number of negotiation terms increases, the number of messages exchanged during a negotiation remains constant, as each negotiation term is independent and the component concedes on all terms at the same time. Thus the length of the negotiation is constrained by the most limiting negotiation term. The utility of the outcome remains constant since the utility is limited by the most constraining negotiation term. As the number of terms increases, the time taken to perform the negotiations increases linearly, assuming that all the terms are evaluated using the same linear utility functions and tactics.*

Figure 4A shows that the average utility achieved with a varying number of negotiation terms remains fairly constant, and does not begin to drop with respect to the number of terms. Similarly, Figure 4B shows that the time appears to be increasing linearly. While there are a few deviations upwards of the linear trend, these can be explained by garbage collection being triggered in the execution environment.

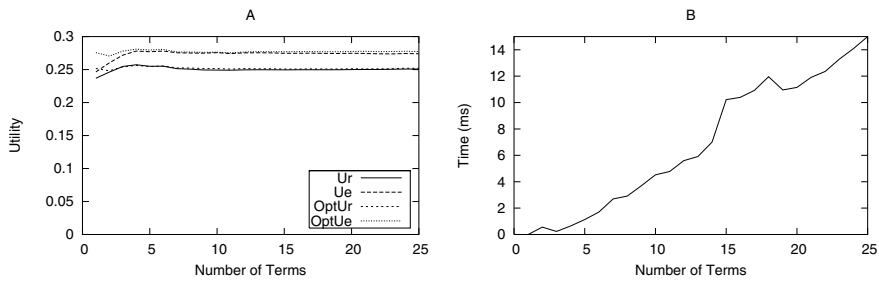


Fig. 4. Variable number of negotiation terms: A) Utilities of outcome, B) Time Taken

Execution Time. To confirm that the time taken for negotiation increases linearly with the number of messages exchanged, we measured the execution time for the negotiation to complete, averaged over 100 times to reduce inaccuracies. For a given number of messages exchanged, we recorded the average, minimum and maximum times taken for negotiations exchanging that quantity of messages. The deadlines in this experiment were between 30 and 60 messages.

Hypothesis: *As the number of messages exchanged between negotiations takes place, the corresponding increase in real time taken will be linear.*

As shown in Figure 5 there is a wide range of results for each number of messages exchanged. However, the average line is close to the minimum, indicating that few results are significantly higher than average. (We have explained the higher results as garbage collection in Java.) Failed negotiations are plotted as 0 messages, although they always take up to their deadline to complete, because reaching the deadline implies failure. Since the time taken for failed negotiations is approximately the same as with high numbers of messages, we conclude that the time taken is linearly related to the number of messages exchanged.

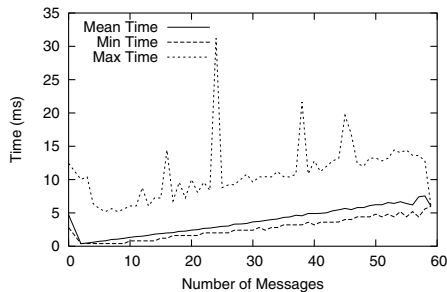


Fig. 5. Graph of time taken for negotiations

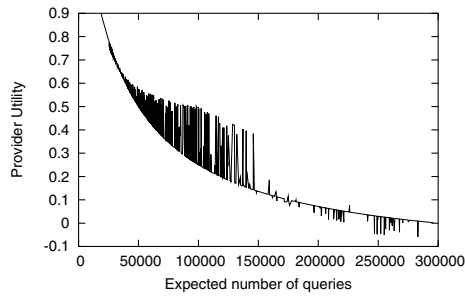


Fig. 6. Provider utility against expected number of calculations

5 Evaluation for the Notification Service

We chose a specific case from the bioinformatics field as an example use of the notification service. The SWISS-PROT Protein Knowledgebase is a curated protein sequence database providing a high level of annotation, minimal redundancy and high integration with other databases [13]. Over the past 14 months it has grown by 20% with approximately 200 changes per day. As an example, we assume 1000 subscribers are interested in anything that matches 100 different sequences, and a particular similarity search takes 1 second and can be run iteratively to refine results. To provide the notifications, the provider runs the similarity search after data has been added or annotated. Although subscribers are happy to receive updates every few hours, this would place an unacceptable load on the provider. For example, daily searches iterated 5 times require five million searches per day.

We use two negotiation terms for this experiment. Frequency represents the maximum number of hours between notifications: for the provider, this is between 24 and 168 hours, whereas for the consumer it is between 5 and 120 hours. The second term is the number of iterations of the search. The provider prefers this to be between 1 and 3, the client between 1 and 5. The preferences for the provider are kept constant and a random variation is introduced into the client preferences to simulate different clients. Negotiation deadlines are between 30 and 60 messages. After running the negotiations, the average value for the frequency was 67.6 hours and the number of iterations was 2.31 iterations. This works out as 651,000 searches per day, a reduction of 87%. The average utilities over the experiments were 0.39 for the consumer and 0.30 for the provider. Figure 6 shows that the provider utility decreases as the number of computations increases. The curve would have been smoother if we had weighted the utility of each term differently, as they influence the number of calculations in different ways.

While these figures are based on a hypothetical situation, it demonstrates that the consumer's requirements can be satisfied while reducing the number of searches the provider has to carry out to do so, indicating that a better Quality of Service can be achieved by using negotiation to establish QoS terms.

6 Related Work

There has been much research in automated negotiation. Bartolini et al. [14] produced a framework for negotiation allowing different types of negotiations to be specified using rules. We chose not to use this, as we wanted the two parties to be able to communicate directly rather than to use a third party to carry out the negotiation. Jennings et al. [15] give some details on a framework for automated negotiation, which focuses on rules of negotiation, and allowing many different types of negotiation to be carried out within the same framework. The RFC1782 [16] describes a simple extension to Trivial File Transfer Protocol (TFTP) to allow option negotiation prior to the file transfer. Although the idea is similar in principle, there is no economically sound model of negotiation used.

A number of examples of subscription services have been identified in the Grid community, allowing the information to be used for monitoring or rescheduling of allocations [17]. The Grid monitoring architecture [6] is a distributed architecture allowing monitoring data to be collected by distributed components. The Grid Notification Framework [7] allows information about the existence of a grid entity, as well as properties about its state, to be propagated to other grid entities. NaradaBrokering [9] is a peer-to-peer *event brokering system* supporting asynchronous delivery of events to sensors, high end performance computers and handheld devices. The Open Grid Services architecture [8] has also identified a logging service as an essential component: it also relies on producer and consumer interfaces. None of these notification services support negotiation and might benefit from our work.

7 Conclusion and Future Work

This paper has presented our design for a negotiation engine for inclusion in a Notification Service. We have presented our reasons for choosing a competitive negotiation method and shown how our negotiation engine works. We have also shown that the performance of the system is predictable and does not have any adverse effects when used with many negotiation terms.

Further development of this work is ongoing, and we have identified several areas we would like to proceed with. In our current system all negotiation terms are independent and negotiations concede on all of them. We would like to investigate introducing dependencies between negotiation terms and the possibilities of trading off one term against another. Negotiations currently take place between a requester and a notification service. We envisage a system where negotiations are chained between consumer, notification service and provider, and will examine negotiation in this situation.

Finally, it is worth noting that this negotiation component will be deployed in the Notification Service in a real Grid environment using MyGrid as a testbed.

Acknowledgements. This research is funded in part by EPSRC myGrid project (ref. GR/R67743/01).

References

1. Java Message Service API. <http://java.sun.com/products/jms/> (1999)
2. Object Management Group: Event service specification. www.omg.org (2001)
3. Object Management Group: Notification service specification. www.omg.org (2002)
4. Oinn, T.: Change events and propagation in mygrid. Technical report, European Bioinformatics Institute (2002)
5. UDDI version 3 features list. www.uddi.org (2001)
6. Tierney, B., Aydt, R., Gunter, D., Smith, W., Swamy, M., Taylor, V., Wolski, R.: A grid monitoring architecture. Technical report, GGF Performance Working Group (2002)
7. Gullapalli, S., Czajkowski, K., Kesselman, C.: Grid notification framework. Technical Report GWD-GIS-019-01, Global Grid Forum (2001)
8. Foster, I., Gannon, D., Nick, J.: Open grid services architecture: A roadmap. Technical report, Open Grid Services Architecture Working Group (2002)
9. Fox, G., Pallickara, S.: The narada event brokering system: Overview and extensions. In Arabnia, H., ed.: 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02). Volume 1., Las Vegas, Nevada, CSREA Press (2002) 353–359
10. myGrid Project: mygrid website. <http://www.mygrid.org.uk/> (2003)
11. Faratin, P., Sierra, C., Jennings, N.R.: Negotiation decision functions for autonomous agents. *International Journal of Robotics and Autonomous Systems* **24** (1998) 159–182
12. Sierra, C., Jennings, N.R., Noriega, P., Parsons, S.: A framework for argumentation-based negotiation. In: *Intelligent Agents IV: 4th International Workshop on Agent Theories Architectures and Languages*, (Springer)
13. EBI: Swiss-prot website. <http://www.ebi.ac.uk/swissprot/> (2003)
14. Bartolini, C., Preist, C., Jennings, N.R.: Architecting for reuse: A software framework for automated negotiation. In: *3rd International Workshop on Agent-Oriented Software Engineering*, Bologna, Italy (2002) 87–98
15. Jennings, N.R., Parsons, S., Sierra, C., Faratin, P.: Automated negotiation. In: *5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Systems*, Manchester, UK (2000) 23–30
16. Malkin, G., Harkin, A.: TFTP option extension (rfc 1782). Technical report, Network Working Group (1995)
17. Schwiegelshohn, Yahyapour: Attributes for communication between scheduling instances. Technical report, GGF, Scheduling Attributes Working Group (2001)

GrADSolve – RPC for High Performance Computing on the Grid*

Sathish Vadhiyar, Jack Dongarra, and Asim YarKhan

Computer Science Department
University of Tennessee, Knoxville
{vss, dongarra, yarkhan}@cs.utk.edu

Abstract. Although high performance computing has been achieved over computational Grids using various techniques, the support for high performance computing on the Grids using Remote Procedure Call (RPC) mechanisms is fairly limited. In this paper, we discuss a RPC system called GrADSolve that supports execution of parallel applications over Grid resources. GrADSolve employs powerful scheduling techniques for dynamically choosing the resources used for the execution of parallel applications and also uses robust data staging mechanisms based on the data distribution used by the end application. Experiments and results are presented to prove that GrADSolve's data staging mechanisms can significantly reduce the overhead associated with data movement in current RPC systems.

1 Introduction

The role of Remote Procedure Call (RPC) [2,15,18,4,9,3,12] mechanisms in Computational Grids [7] has been the subject of several recent studies [14,6,13]. Although traditionally RPCs have been viewed as communication mechanisms, recent RPC systems [3,12] perform a wide range of services for problem solving on remote resources. Computational Grids consist of large number of machines ranging from workstations to supercomputers and strive to provide transparency to the end users and high performance for end applications. While high performance is achieved by the parallel execution of applications on large number of Grid resources, user transparency can be achieving by employing RPC mechanisms. Though there are a large number of RPC systems that adequately support the remote invocation of sequential software from sequential environments, the number of RPC systems for supporting invocation of parallel software are relatively few [3,12,13,10,5]. Some of these parallel RPC systems [13,10] require invocation of remote parallel services from only parallel clients. Some of the RPC systems [3,12] support only master-slave or task farming models of parallelism. A few RPC systems [12,3] fix the amount of parallelism at the time when the services are uploaded into the RPC system and hence are not adaptive to

* This work is supported in part by the National Science Foundation contract GRANT #EIA-9975020, SC #R36505-29200099 and GRANT #EIA-9975015

the load dynamics of the Grid resources. A few RPC systems [10,5] supporting invocation of parallel software are implemented on top of object oriented frameworks like CORBA and JavaRMI and may not be suitable for high performance computing according to a previous study [16].

In this paper, we propose a Grid-based RPC system called GrADSolve¹ that enables the users to invoke MPI applications on remote Grid resources from a sequential environment. In addition to providing easy-to-use interfaces for the service providers to upload the parallel applications into the system and for the end users to remotely invoke the parallel applications, GrADSolve performs application-level scheduling and dynamically chooses the resources for the execution of the parallel applications based on the load dynamics of the Grid resources. GrADSolve also uses data distribution information provided by the library writers to partition the users' data and stage to the different resources used for the application execution. Our experiments show that the data staging mechanisms in GrADSolve helps reduce the data staging times in RPC systems by 20-50%. In addition to the above features, GrADSolve also enables users to store execution traces for a problem run and use the execution traces for subsequent problem runs.

Thus, the contributions of our research are:

1. design and development of an RPC system that utilizes standard Grid Computing mechanisms including Globus [8] for invocation of remote parallel applications from a sequential environment.
2. selection of resources for parallel application execution based on load conditions of the resources and application characteristics.
3. communication of data between the user's address space and the Grid resources based on the data distribution used in the application and
4. maintenance of execution traces for problem runs.

The current implementation of GrADSolve is only suitable for invoking remote MPI-based parallel applications from sequential applications and not from parallel applications.

Section 2 presents a detailed description of the framework of the GrADSolve system. The support in the GrADSolve system for maintaining execution traces is explained in Section 3. In Section 4, the experiments conducted in GrADSolve are explained and results are presented to demonstrate the usefulness of the data staging mechanisms and execution traces in GrADSolve. Section 5 looks at related efforts in the development of RPC systems. Section 6 presents conclusions and future work.

2 Overview of GrADSolve

Figure 1 illustrates the overview of the GrADSolve system.

¹ The system is called GrADSolve since it is derived from the experiences of the GrADS [1] and NetSolve [3] projects.

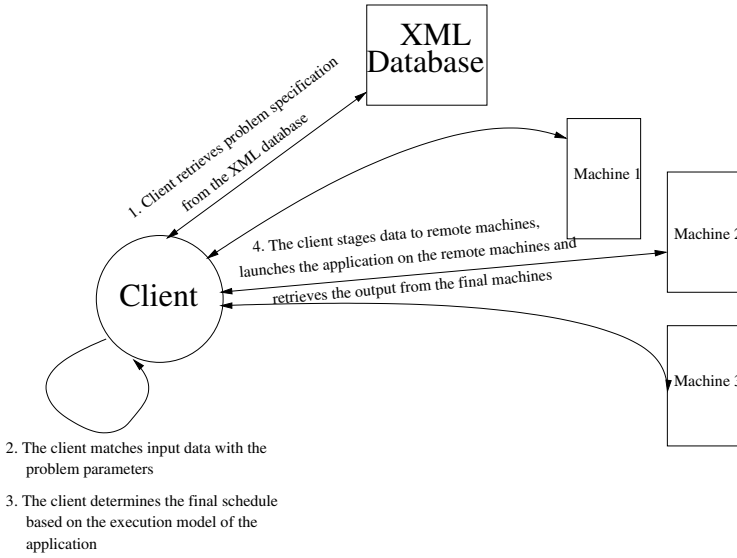


Fig. 1. Overview of GrADSolve system

At the core of the GrADSolve system is a XML database. This database maintains four kinds of tables - *users*, *resources*, *applications* and *problems*. The *users*, *resources* and the *applications* tables contain information about the different users, machines and applications on the Grid system respectively. The *problems* table maintains information about the individual problem runs.

There are three human entities involved in GrADSolve - *administrators*, *library writers* and *end users*. The role of these entities in GrADSolve and the functions performed by the GrADSolve system for these entities are explained below.

Administrators. The GrADSolve administrator is responsible for managing the users and resources of the GrADSolve system. The administrator creates entities for different users and resources in the XML database by specifying configuration files that contains information for different users and resources, namely the user account names for different resources, the location of the home directories on different resources in the GrADSolve system, the names of the different machines, their computational capacities, the number of processors in the machines and other machine specifications.

Library Writers. The library writer uploads his application into the GrADSolve system by specifying an Interface Definition Language (IDL) file for the application. In the IDL file, the library writer specifies the programming language in which the function is written, the name of the function, the set of input and output arguments, the description of the function, the names of the object files and libraries needed for linking the function with other functions, if the function is sequential or parallel, etc. For each input and output argument, the library writer specifies the name of the argument, if the argument is an input

or output argument, the datatype of the argument, the number of elements if the argument is a vector, the number of rows and columns if the argument is a matrix etc. An example of a IDL file written for a ScaLAPACK QR factorization routine is given in Figure 2.

```
PROBLEM qrwrapper
C FUNCTION qrwrapper(IN int N, IN int NB, INOUT double A[N][N],
                     INOUT double B[N][1])
  'a version of qr factorization that works with
  square matrices.'
LIBS = '/home/grads23/GrADSolve/ScaLAPACK/pdgeqrf_instr.o \
        /home/grads23/GrADSolve/ScaLAPACK/pdscaex_instrQR.o \
        ...'
TYPE = parallel
```

Fig. 2. An example GrADSolve IDL for a ScaLAPACK QR problem

After the library writer submits the IDL file to the GrADSolve system, GrADSolve translates the IDL file to a XML document. The GrADSolve translation system also generates a wrapper program and compiles the wrapper program with the appropriate libraries and stages the executable file to the remote machines in the GrADSolve system. Also, stored in the XML database for the application is the information regarding the location of the executable files on the remote resources.

If the library writer wants to add an performance model for his application, he executes the *getperfmodel.template* utility specifying the name of the application. The utility retrieves the problem description of the application from the XML database and generates a performance model template file. The template file contains the definitions of the performance model routines. The library writer fills in the performance model routines with the appropriate code for specifying if the given set of resources have adequate capacity to solve the problem, the predicted execution cost of the application and the data distribution used in the application. The library writer uploads his performance model by executing the *add.perfmodel* utility which stores the location of the wrapper program and the performance model to the XML database corresponding to the entry for the application.

End Users. The end users solve problems over remote GrADSolve resources by writing a client program in C or Fortran. The client program includes an invocation of a routine called *gradsolve()* passing to the function, the name of the end application and the input and output parameters needed by the end application. The invocation of the *gradsolve()* routine triggers the execution of the GrADSolve Application Manager. GrADSolve uses Globus Grid Security Infrastructure (GSI) for the authentication and authorization of users. The Application Manager then retrieves the problem description from the XML database

and matches the user's data with the input and output parameters required by the end application.

If an performance model exists for the end application, the Application Manager downloads the performance model from the remote location where the library writer had previously stored it. The Application Manager then retrieves the list of machines in the GrADSolve system from the *resources* table in the XML database, and retrieves resource characteristics of the machines from the Network Weather Service (NWS) [17]. The Application Manager uses the list of resources with resource characteristics, the performance model and scheduling heuristics [19] to determine a final schedule for application execution and stores the status of the problem run and the final schedule in the *problems* table of the XML database corresponding to the entry for the problem run.

The Application Manager then creates working directories on the scheduled remote machines for end application execution and enters the *Application Launching* phase. The Application Launcher stores the input data to files and stages these files to the corresponding remote machines chosen for application execution. An input data may be associated with data distribution information that was previously uploaded by the library writer. The data distribution information contains the kind of data distribution (e.g., block, block-cyclic, cyclic, user-defined etc.) used for the data. If data distribution information for an input data does not exist, the Application Launcher stages the entire input data to all the machines involved in end application execution. If the information regarding data distribution exists, the Application Launcher stages only the appropriate portions of the input data to the corresponding machines. For example, for data with block distribution, only the 2nd block has to be staged to the 2nd machine used for problem solving. This kind of selective data staging significantly reduces the time needed for the staging of entire data especially if large amount of data is involved. After the staging of input data, the Application Launcher launches the end application on the remote machines chosen for the final schedule using the Globus MPICH-G mechanisms. The end application reads the input data that were previously staged by the Application Launcher, solves the problem and stores the output data to the corresponding files on the machines in the final schedule. When the end application finishes execution, the Application Launcher copies the output data from the remote machines to the user's address space. The staging in of the output data from the remote locations is a reverse operation of the staging out of the input data to the remote locations. The GrADSolve Application Manager finally returns success state to the user client program.

3 Execution Traces in GrADSolve – Storage, Management, and Usage

One of the unique features in the GrADSolve system is the ability provided to the users to store and use execution traces of problem runs. There are many applications in which the outputs of the problem depend on the exact number and configuration of the machines used for problem solving. Ill-conditioned problems

or unstable algorithms can give rise to vast changes in output results due to small changes in input conditions. For these kinds of applications, the user may desire to use the same initial environment for all problem runs.

To guarantee reproducibility of numerical results in the above situations, GrADSolve provides capability to the users to store *execution traces* of problem runs and use the execution traces during subsequent executions of the same problem with the same input data. For storing the execution trace of the current problem run, the user executes his GrADSolve program with a configuration file called *input.config* that contains a TRACE_FLAG variable that is either 0 or 1. During the registration of the problem run with the XML database, the value of the TRACE_FLAG variable is stored. After the end application completes its execution and the output data are copied from the remote machines to the user's address space, the Application Manager removes the remote files containing the input data for the end application if the TRACE_FLAG is 0. But if the TRACE_FLAG is set to 1, the Application Manager retains the input data in the remote machines. At the end of the problem run, the Application Manager generates an output configuration file that contains a TRACE_KEY corresponding to the execution trace.

When the user wants to execute the problem with a previously stored execution trace, he executes his client program specifying the TRACE_KEY variable in the *input.config* file. The TRACE_KEY variable is set with the key that corresponds to the execution trace. During the Schedule Generation phase, the Application Manager, instead of generating a schedule for the execution of the end application, retrieves the schedule used for the previous problem run corresponding to the TRACE_KEY, from the *problems* table in the XML database. The Application Manager then checks if the capacities of the resources in the schedule at the time of trace generation are comparable to the current capacities of the resources. If the capacities are comparable, the Application Manager proceeds to the rest of the phases of its execution. During the Application Launching phase, the Application Manager, instead of staging the input data to remote working directories, copies the input data and the data distribution information, used in the previous problem run corresponding to the TRACE_KEY, to the remote working directories. Thus GrADSolve guarantees the use of the same execution environment used in the previous problem run for the current problem run, and hence guarantees reproducibility of numerical results.

4 Experiments and Results

The GrADS testbed consists of about 40 machines from University of Tennessee (UT), University of Illinois, Urbana-Champaign (UIUC) and University of California, San Diego (UCSD). For the sake of clarity, our experimental testbed consists of 4 machines:

- a 933 MHz Pentium III machine with 512 MBytes of memory located in UT,
 - a 450 MHz Pentium II machine with 256 MBytes of memory located in UIUC
- and

- 2 450 MHz Pentium III machines with 256 MBytes of memory located in UCSD connected to each other by 100 Mb switched Ethernet.

Machines from different locations are connected by Internet. In the experiments, GrADSolve was used to remotely invoke ScaLAPACK QR factorization. Since some of the unique features of GrADSolve include the data distribution mechanisms and the usage of execution traces, the experiments focus only on the times for staging data and not on the communication and total execution times. Block cyclic distribution was used for the matrix A. GrADSolve was operated in 3 modes. In the first mode, the performance model did not contain information about the data distribution used in the ScaLAPACK driver. In this case, GrADSolve transported the entire data to each of the locations used for the execution of the end application. This mode of operation is practiced in RPC systems that do not support the information regarding data distribution. In the second mode, the performance model contained information about the data distribution used in the end application. In this case, GrADSolve transported only the appropriate portions of the data to the locations used for the execution of end application. In the third mode, GrADSolve was used with an execution trace corresponding to a previous run of the same problem. In this case, data is not staged from the user's address space to the remote machines, but temporary copies of the input data used in the previous run are made for the current problem run.

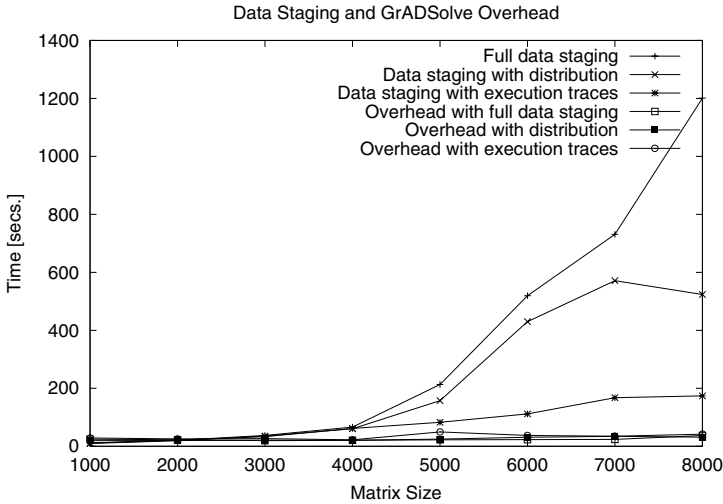


Fig. 3. Data staging and other GrADSolve overhead

Figure 3 shows the times taken for data staging and other GrADSolve overhead for different matrix sizes and for the three modes of GrADSolve operation. The machines that were chosen by the GrADSolve application-level scheduler for the execution of end application for different matrix sizes are shown in Table

1. The decisions regarding the selection of machines for problem execution were automatically made by the GrADSolve system taking into account the size of the problems and the resource characteristics at the time of the experiments.

Table 1. Machines chosen for application execution

<i>Matrix size</i>	<i>Machines</i>
1000	1 UT machine
2000	1 UT machine
3000	1 UT machine
4000	1 UT machine
5000	1 UT, 1 UIUC machines
6000	1 UIUC, 1 UCSD machines
7000	1 UIUC, 1 UCSD machines
8000	1 UT, 1 UIUC, 2 UCSD machines

Comparing the first two modes in Figure 3, we find that for smaller problem sizes, the times taken for data staging in both the modes are the same. This is because only one machine was used for problem execution and the same amount of data are staged in both the modes when only one machine is involved for problem execution. For larger problem sizes, the times for data staging using the data distribution is less than 20-55% of the times taken for staging the entire data to remote resources. Thus the use of data distribution information in GrADSolve can give significant performance benefits when compared to staging the entire data that is practiced in some of the RPC systems. Data staging in the third mode is basically the time taken for creating temporary copies of data used in the previous problem runs in remote resources. We find this time to be negligible when compared to the first two modes. Thus execution traces can be used as caching mechanisms to use the previously staged data for problem solving. The GrADSolve overheads for all the three modes are found to be the same. This is because of the small number of machines used in the experiments. For experiments when large number of machines are used, we predict that the overheads will be higher in the first two modes than in the third mode. This is because in the first two modes, the application-level scheduling will explore large number of candidate schedules to determine the machines used for the end application while in the third mode, a previous application-level schedule will be retrieved from the database and used.

5 Related Work

Few RPC systems contain mechanisms for the parallel execution of remote software. The work by Maassen et. al [10] extends Java RMI for efficient communications in solving high performance computing problems. The framework requires the end user's programs to be parallel programs. NetSolve [3] and Ninf [12] support task parallelism by the asynchronous execution of number of remote

sequential applications. OmniRPC [13] is an extension of Ninf and supports asynchronous RPC calls made from OpenMP programs. But similar to the approaches in NetSolve and Ninf, OmniRPC supports only master-worker models of parallelism. NetSolve, and Ninf also supports remote invocation of MPI applications, but the amount of parallelism and the locations of the resources to be used for the execution are fixed at the time when the applications are uploaded to the systems and hence are not adaptive to dynamic loads in the Grid environments.

The efforts that are very closely related to GrADSolve are PaCO [11] and PaCO++ [6,5] from the PARIS project in France. The PaCO systems are implemented within the CORBA [4] framework to encapsulate MPI applications in RPC systems. The data distribution and redistribution mechanisms in PaCO are much more robust than in GrADSolve and support invocation of remote parallel applications either from sequential or parallel client programs. The PaCO projects do not support dynamic selection of resources for application execution as in GrADSolve. Also, GrADSolve supports Grid related security models by employing Globus mechanisms. And finally, GrADSolve is unique in maintaining execution traces that can help bypass the resource selection and data staging phases.

6 Conclusions and Future Work

In this paper, an RPC system for efficient execution of remote parallel software was discussed. The efficiency is achieved by dynamically choosing the machines used for parallel execution and staging the data to remote machines based on data distribution information. The GrADSolve RPC system also supports maintaining and utilizing execution traces for problem solving. Our experiments showed that the GrADSolve system is able to adapt to various problem sizes and the resource characteristics and yielded significant performance benefits with its data staging and execution trace mechanisms.

Interfaces for the library writers for expressing more capabilities of the end application are currently being designed. These capabilities include the ability of the application to be preempted and continued later with different processor configuration. These capabilities will allow GrADSolve to adapt to changing Grid scenarios. Remote execution of non-MPI parallel programs, applications with different modes of parallelism and irregular applications are also being considered.

References

1. F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, 15(4):327–344, Winter 2001.

2. A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
3. H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
4. CORBA <http://www.corba.org>.
5. A. Denis, C. Pérez, and T. Priol. Achieving Portable and Efficient Parallel CORBA Objects. *Concurrency and Computation: Practice and Experience*, 2002.
6. A. Denis, C. Pérez, and T. Priol. Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing. In *Proc. of the 7th International Euro-Par'01 Conference (EuroPar'01)*, pages 835–844. Springer, August 2001.
7. I. Foster and C. Kesselman eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.
8. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.
9. Java Remote Method Invocation (Java RMI) java.sun.com/products/jdk/rmi.
10. J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.
11. C. René and T. Priol. MPI Code Encapsulating using Parallel CORBA Object. *Cluster Computing*, 3(4):255–263, 2000.
12. H. Nakada M. Sato and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metascomputing Issue*, 15(5-6):649–658, 1999.
13. M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In *In Workshop on OpenMP Applications and Tools (WOMPAT2001)*, July 2001.
14. K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *Lecture notes in computer science 2536 Grid Computing - GRID 2002*, volume Third International Workshop, pages 274–278, Baltimore, MD, USA, November 2002. Springer Verlag.
15. Simple Object Access Protocol (SOAP) <http://www.w3.org/TR/SOAP>.
16. T. Suzumura, T. Nakagawa, S. Matsuoka, H. Nakada, and S. Sekiguchi. Are Global Computing Systems Useful? - Comparison of Client-Server Global Computing Systems Ninf, Netsolve versus CORBA. In *In Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS '00*, pages 547–559, May 2000.
17. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.
18. XML-RPC <http://www.xmlrpc.com>.
19. A. YarKhan and J. Dongarra. Experiments with Scheduling Using Simulated Annealing in a Grid Environment. In M. Parashar, editor, *Lecture notes in computer science 2536 Grid Computing - GRID 2002, Third International Workshop*, pages 232–242, Baltimore, MD, USA, November 2002. Springer Verlag.

Resource and Job Monitoring in the Grid

Zoltán Balaton and Gábor Gombás

MTA SZTAKI Computer and Automation Research Institute
P.O. Box 63., H-1518 Hungary
{balaton, gombasg}@sztaki.hu

Abstract. In a complex system like the grid monitoring is essential for understanding its operation, debugging, failure detection and for performance optimisation. In this paper a flexible monitoring architecture is introduced that provides advanced functions like actuators and guaranteed data delivery. The application of the monitoring system for grid job and resource monitoring, and the integration of the monitoring system with other grid services is also described.

1 Introduction

The Grid [8] is a vast array of distributed resources. Monitoring is essential for understanding the operation of the system, debugging, failure detection and for performance optimisation. To achieve this, data about the grid must be gathered and processed to reveal important information. Then, according to the results, the system may need to be controlled. It is the task of the monitoring system to provide facilities for this.

Traditionally monitoring is considered as part of grid information systems such as, Globus MDS [5] and R-GMA [7]. This approach however has difficulties in supporting all monitoring scenarios, e.g. application monitoring. To overcome these limitations a number of application monitoring solutions have been proposed for grids. Systems such as NetLogger [10], OCM-G [4] and Autopilot [12] all focus on application monitoring independent of the information system. We proposed a division of the information and monitoring systems in [2] based on the properties of information they must handle.

The monitoring system must be able to provide information about the current state of various grid entities, such as grid resources and running jobs, as well as to provide notifications when certain events (e.g. system failures, performance problems) occur.

In the following sections a flexible and efficient monitoring system designed and implemented as part of the GridLab project [9] is introduced and its application for resource and job monitoring and connection to other grid services is presented.

2 The Monitoring System Architecture

Grids are very large-scale and diverse systems thus, gathering all information in a central location is not feasible. Therefore, centralised solutions are not adequate, only distributed systems can be considered. Monitoring also has to be specific, i.e. only data that is actually needed has to be collected. It is also important that data should only be sent where it is needed and only when requested. The Grid Monitoring Architecture (GMA, [13]) proposed by the Global Grid Forum (GGF) satisfies these requirements. The architecture of

the monitoring system is therefore based on the GMA. The input of the monitoring system consists of measurements generated by *sensors*. Sensors are controlled by *producers* that can transfer measurements to *consumers* when requested.

Sensors implement the measurement of one or more measurable quantities that are represented in the monitoring system as *metrics* defined by a unique name, a list of formal parameters and a data type. The metric name identifies the metric definition (e.g. `host.cpu.user`). Many metrics can be measured at different places simultaneously. E.g., CPU utilisation can be measured on several hosts or grid resources. The parameters in the metric definition help to distinguish between these different *metric instances*. The entity to be monitored is determined by providing actual values for the formal parameters. E.g., the metric “CPU time used by a process” must have process ID as a formal parameter. A metric instance is then formed by supplying a PID. A measurement corresponding to a metric instance is called *metric value*. Metric values contain a timestamp and the measured data according to the data type of the metric definition.

The data type is the definition of the structure used for representing measurement data. The monitoring system supports basic types (such as, 32 and 64 bit, signed and unsigned integers, double precision floating point numbers), strings and arbitrary binary data, which is uninterpreted by the monitoring system. The monitoring system also supports two complex types, arrays of identical types with either fixed or variable length, and a record type, which is a composite of other types with a predefined layout. These types allow the definition of metrics with arbitrarily complex data types however, monitoring data can typically be represented with simple types.

Metrics can either be event-like, i.e. an external event is needed to produce a measurement (a typical example is a notification of a job state change), or continuously measurable, i.e. a measurement is possible whenever a consumer requests it (such as the CPU temperature in a host). Our monitoring system supports both measurement types. Continuously measurable metrics can be made event-like by requesting automatic periodic measurements.

In addition to the components in the GMA, *actuators* (similar to actuators in Autopilot [12] or manipulation services in OMIS [11]) are also part of the system. Actuators are analogous to sensors but instead of taking measurements of metrics they implement *controls* that represent interactions with either the monitored entities or the monitoring system itself. Just like metrics a control is defined by a unique name, a list of formal parameters and a data type. In contrast to metrics that can generate multiple results however, controls are always single-shot, i.e. every invocation of a control produces exactly one result. The functional difference between metrics and controls is that metrics only provide data while controls do not provide data except for a status report but they influence the state or behaviour of the monitoring system or the monitored entity.

The use of metrics and controls makes the monitoring system very general and flexible. Producers are able to handle diverse sensors and actuators without detailed knowledge of their inner workings. At the same time, sensors and actuators are free to exploit all the implementation possibilities. Sensors do not have to take all measurements themselves, they could also contact external processes to get data. For example, a sensor can query the local accounting system if available or it is also possible to implement persistent sensors (such as network monitors) that are running independently of the

monitoring system and gather statistics over time. The producer can manage (start, stop and control) sensors, initiate measurements and invoke controls on a user's request.

The flexibility of metrics and controls is also exploited within the monitoring system. There are some metrics built in the producer that provide information about the monitoring system, such as producer capabilities and metric definitions. Influencing the monitoring system is also possible via controls that act on the monitoring system itself. Apart from very simple built-in sensors and actuators most sensors and actuators are loadable modules that are dynamically linked in the producer depending on configuration. This modularity makes the monitoring system very flexible and easily extensible.

The GMA proposal defines two methods of information retrieval: query and subscribe. Our monitoring system supports further producer–consumer interaction functions. Consumers can request buffering of measurements in the producer and multiple logical connections between a producer and a consumer (called *channels*) are also available. Channels can either be consumer-initiated or producer-initiated, but producer-initiated channels must be explicitly requested previously by a consumer. Consumer-initiated channels are used mainly for interactive monitoring or control execution while producer-initiated channels can be used for event reporting and data archiving in a storage service. The two possible channel directions can also be used for getting through firewalls which block communication in one direction.

Permanently storing data is outside of the responsibilities of the monitoring system thus, if a consumer needs to preserve monitoring data it must either save the data itself or supply a contact point for a storage service and appropriate credentials for authentication. Generally if the consumer fails to process some data due to a software or hardware failure the information may be lost. There are cases however (e.g. critical event notifications) when this behaviour is not acceptable. For these cases the monitoring system supports *guaranteed delivery* of measurements. Consumers can enable guaranteed delivery for a metric by executing a control. After guaranteed delivery is enabled, the producer awaits an acknowledgement of receiving each metric value from the consumer before deleting it from its buffer. Handling of guaranteed delivery is expensive as it consumes resources of the producer. Therefore, the producer might impose resource limits on the number of transactions being active at any given time.

3 Resource Monitoring

The monitoring system components used for resource monitoring is shown in Figure 1. drawn with solid lines. The figure depicts a grid resource consisting of three nodes. A Local Monitor (LM) service is running on each node and collects information from processes (P) running on the node as well as the node itself. The collected information is sent to a Main Monitor (MM) service. The MM is a central access point for local users (i.e. site administrators and non-grid users). Grid users can access information via the Monitoring Service (MS) which is also a client of the MM. In large sized grid resources there may be more than one MM to balance network load. The modularity of the monitoring system also allows that on grid resources where an MM is not needed (e.g. on a supercomputer) it can be omitted and the MS can talk directly to LMs.

As the Grid itself consists of several layers of services the monitoring system also follows this layout. The LM – MM – MS triplet demonstrates how a multi-level moni-

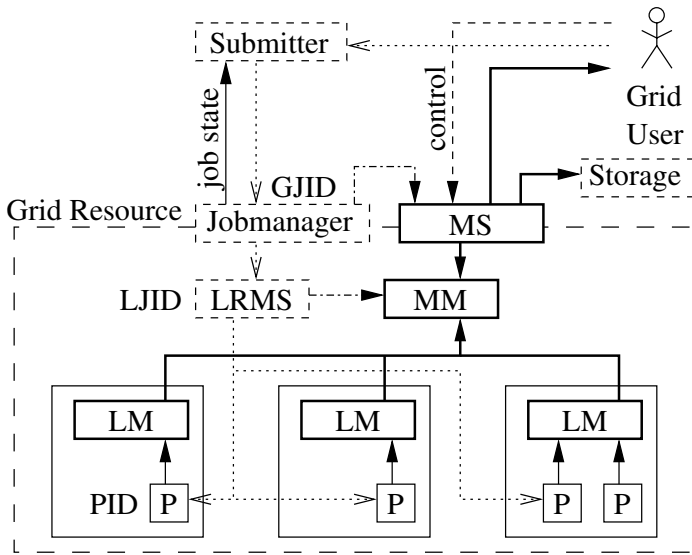


Fig. 1. Resource and job monitoring

toring system is built. The compound producer–consumer entities described in the GMA proposal is exploited here: each level acts as a consumer for the lower level and as a producer for the higher level. This setup has several advantages. Complex compound metrics are easily introduced by sensors at intermediate levels which get raw data from lower levels and provide processed data for higher levels. Transparent access to preprocessed data or data from multiple sources can be supported in the same way. In cases when there is no direct network connection (e.g., because of a firewall) between the consumer and a target host, an intermediary producer can be installed on a host that has connectivity to both sides. This producer can act as a proxy between the consumer and the target host. With proper authentication and authorization policies at this proxy, this setup is more secure and more manageable than opening the firewall.

The above described possibilities are exploited in the MS. The MS acts as a proxy between the grid resource and grid users. This is the component where grid security rules are enforced and mapped to local rules. As the Grid is a heterogenous environment it is very likely that different grid resources (or even different nodes of the same resource) provide different kinds of information. In order to make it possible for data from different sources to be interpreted and analysed in a uniform way, the raw data must be converted to a well-defined form. Such conversion is also done in the MS which takes resource specific metrics from the MM and converts them to *grid metrics* that are independent of the physical characteristics of the resource.

4 Job Monitoring

Monitoring of grid jobs is a complicated task that requires interoperation between the monitoring system and other grid services. Figure 1. depicts the situation where an application started as a grid job is running on a grid resource. The running application consists of processes (labelled P in the figure) running on hosts constituting the grid resource. Processes are identified locally by the operating system by process identifiers (PIDs). The local resource management system (LRMS) controls jobs running on hosts belonging to a grid resource. It allocates hosts to jobs, starts and stops jobs on user request and possibly restarts jobs in case of an error. It may also checkpoint and migrate jobs between hosts of the resource which can be considered as a special case of job startup. The LRMS identifies the job it manages by a local job identifier (LJID).

To monitor a job the monitoring system has to know the relation between LJIDs and PIDs. This is required because some metrics (e.g. CPU usage and low level performance metrics provided by hardware performance counters) are only available for processes, while other metrics (such as high level job status) can only be get from the LRMS by the LJID. The LRMS is the only component that has all the information about which processes belong to a job, thus it should have an interface for the monitoring system to provide the mapping between PIDs and LJIDs to support monitoring. In practice however, current LRMS implementations usually do not provide such an interface thus another way to get the mapping between PIDs and LJIDs is necessary.

The most simple solution would be to require processes to register and identify themselves to the monitoring system by calling a function at startup. This would however require modification to the application and assumes that the application processes have access to their LJID which may not be the case for some LRMS implementations. This solution would also rely on the application to honestly register all its processes thus, ill behaving applications could prevent correct monitoring. Problems may arise for cooperating applications as well. For example, if the LRMS transparently checkpoints and migrates application processes they may be lost for the monitoring system until they re-register themselves.

Another possible solution is to require the LRMS to set the name of the processes it starts in such a way that the LJID of the job the process belongs to can be inferred from it. This way, the monitoring system could identify processes being part of a grid job by looking at their process name. However, processes are able to change their process name thus, it is possible for applications to hinder monitoring. This solution also requires modification to the LRMS which may not be possible for some implementations.

Moreover, both solutions described above are insecure because they rely on an untrusted component (the application) to correctly identify itself to the monitoring system and not to interfere with the mechanisms used for identification. Thus, in situations where precision of monitoring is important (e.g. when monitoring data may be used for accounting purposes) these solutions are inadequate.

The jobmanager in Figure 1. represents the grid service, which allows grid users to start jobs on a grid resource (an example for the jobmanager is the Globus GRAM [6]). The user hands to the jobmanager a *job manifest*, a document that contains both the description of the job and the specification of the requested local resources. After successful authentication and authorization, the jobmanager translates the job manifest

into a form understood by the local resource management system and starts the job under a local user account (identified by a user ID).

The easiest and most secure way to identify processes belonging to a job is to start each job under a different user account. In this way, processes of a job can be identified by the user ID. For this, the jobmanager must ensure that each job is started under a user account that is distinct from accounts used by other currently running grid jobs. A free user account can be taken from a pool of accounts that are reserved for running grid jobs. This solution does not have the problems of the other two mechanisms described above. Moreover, this method has other advantages as well:

- It does not require any modifications to the job.
- The only support required from the LRMS is to start the processes of the job with the user account under which the job was submitted, that is supported by all common implementations.
- Operating system mechanisms ensure that grid jobs cannot interfere with each other either accidentally or intentionally. File system permissions guarantee that different jobs cannot overwrite each others' files. Different process ownership prevents sending of unexpected signals to other jobs.
- Operating system facilities can be used for authorisation and enforcing local policies. Resource usage limits can be set and enforced for each job. File system permissions can be used to implement shared use of cached resources (e.g. if multiple jobs process the same read-only data file, it can be shared between the jobs without having to be downloaded multiple times but still preventing accidental overwrites by any of the jobs).
- Highly accurate accounting is possible if the operating system supports process accounting. If the same grid user runs several jobs on the same (multiprocessor) machine under a single user ID, separating accounting records for different jobs can be difficult. The system-level accounting records are clearly separated however, if each job is mapped to a different local user ID.
- Users cannot interfere with or circumvent monitoring. A process created by a grid job (e.g. by forking) cannot hide from the monitoring system nor can it pretend to belong to a different job and/or user.
- Local account management of the resource can be simplified as there is no need to create local accounts for each grid user. Instead a pool of user accounts big enough for the maximum number of simultaneously running jobs is sufficient. On platforms having dynamic name services (NSS for Linux, Solaris and HP-UX, Loadable Security Modules for AIX, etc.) usernames can be generated on the fly.
- Left-over processes can be easily identified and killed even if the same grid user has several other jobs running. Left-over files created by a job can also be identified by owner and destroyed when the corresponding job has finished.

The jobmanager also allows the user to control the execution of the job. To reference jobs started by the jobmanager yet another identifier the grid job ID (GJID) is used. The jobmanager maintains the mapping between GJIDs and LJIDs and must provide an interface for the monitoring system to get this information together with the user ID that is assigned to this job. The multi-level setup of the monitoring system is useful for

making this mapping transparent. When a grid user connects to the MS and requests monitoring of a grid job by a GJID, the MS can convert the GJID to a LJID and pass it to the MM. The MM then converts the local job ID to a user ID and passes it to the appropriate LMs that take the actual measurements.

To support jobs running on or migrating between multiple grid resources (such as [1]) another grid service labelled Submitter in Figure 1. is required that keeps track on which sites processes belonging to the job are actually running. The Submitter represents a grid entity which starts a grid job and keeps track of the job status. The Submitter always gets notifications about changes in the state of the job and thus it always knows which grid resources the application is running on currently. The user can use this information to find the monitoring service contact points at the appropriate resources in order to control the monitoring of the job and get measurements. The Submitter can also act as a single concentrator element for jobs that run on multiple resources at the same time. This way information about the job can be requested at one well-known point even if the job is constantly moving from one resource to another. This solution provides a distributed registry for jobs running on the Grid.

5 Conclusion

In this paper we presented a flexible and efficient monitoring system designed and implemented as part of the GridLab project. The architecture of the monitoring system is based on the GMA and exploits its distributed design, compound producer–consumer entities and generality. In addition to the features described in the GMA document our monitoring system also supports actuators and additional producer–consumer interaction functions such as, buffering of measurements in the producer, multiple channels and guaranteed delivery of critical data.

The application of the monitoring system for resource and job monitoring was also presented, and the interoperation with and support necessary from other grid services was discussed. We proposed a simple and secure solution for keeping track of processes and different identifiers belonging to a job that simplifies both the monitoring system and local management of a grid resource.

An implementation of the monitoring system is available and being used in the GridLab project. An alpha version with the basic features implemented was released to the GridLab community in August 2002, and the first official prototype is released in January 2003 for the general public. The prototype contains the implementation of the producer and consumer APIs and implementation of the LM and the MM. It also contains sensors providing host monitoring, support for application monitoring using the GRM [3] instrumentation library, and several test consumers. Sensors, actuators and even the protocol encoding are implemented as loadable modules. This modular design allows flexibility and easy extensibility.

To test the performance of the monitoring system implementation we measured the time required to send GRM events from an instrumented application via the GRM sensor, the LM and the MM to a consumer. These tests showed that the monitoring system is capable to handle more than 13000 GRM events per second. The time needed to transfer data was a linear function of the number of events. However, during development so far

the top priority was correctness and completeness and not performance, thus there are several areas for further improvements. Development is continued, actuators, guaranteed delivery and security functions are being implemented during 2003.

Acknowledgements. The work described in this paper has been sponsored by the European Commission under contract number IST-2001-32133, the Hungarian Scientific Research Fund (OTKA) under grant number T032226 and the Research and Development Division of the Hungarian Ministry of Education under contract number OMFB-01549/2001.

References

1. G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, J. Shalf: The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications*, Volume 15, Number 4, 2001.
2. Z. Balaton, G. Gombás, Zs. Németh: Information System Architecture for Brokering in Large Scale Grids. *Proc. 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Linz, Austria, September/October 2002.
3. Z. Balaton, P. Kacsuk, N. Podhorszki, F. Vajda: From Cluster monitoring to Grid Monitoring Based on GRM. *Proc. Euro-Par 2001 International Conference on Parallel and Distributed Computing*, Manchester, United Kingdom, August 2001.
4. B. Baliś, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller: An Infrastructure for Grid Application Monitoring. *Proc. 9th European PVM/MPI Users' Group Meeting*, Linz, Austria, September/October 2002.
5. K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman: Grid Information Services for Distributed Resource Sharing. *Proc. 10th IEEE International Symposium on High Performance Distributed Computing*, San Francisco, California, August 2001.
6. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke: A Resource Management Architecture for Metacomputing Systems. *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.
7. S. Fisher et al. R-GMA: A Relational Grid Information and Monitoring System 2nd Cracow Grid Workshop, Cracow, Poland, 2003.
8. I. Foster and C. Kesselman, editors: *The Grid: Blueprint for a Future Computing Infrastructure*, 1999.
9. The GridLab Project <<http://www.gridlab.org>>
10. D. Gunter, B. Tierney, K. Jackson, J. Lee, M. Stoufer: Dynamic Monitoring of High-Performance Distributed Applications. *Proc. 11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.
11. T. Ludwig, R. Wismüller: OMIS 2.0 – A Universal Interface for Monitoring Systems. *Proc. 4th European PVM/MPI Users' Group Meeting*, Cracow, Poland, November 1997.
12. R. Ribler, J. Vetter, H. Simitci, D. Reed: Autopilot: Adaptive Control of Distributed Applications. *Proc. 7th IEEE Symposium on High Performance Distributed Computing*, Chicago, Illinois, July 1998.
13. B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, R. Wolski: A Grid Monitoring Architecture. GGF Technical Report GFD-I.7, January 2002.
<<http://www.gridforum.org/Documents/GFD/GFD-I.7.pdf>>

Delivering Data Management for Engineers on the Grid¹

Jasmin Wason, Marc Molinari, Zhuoan Jiao, and Simon J. Cox

School of Engineering Sciences, University of Southampton, UK
{j.l.wason, m.molinari, z.jiao, sjc}@soton.ac.uk

Abstract. We describe the design and implementation of a database toolkit for engineers, which has been incorporated into the Matlab environment, to help manage the large amount of data created in distributed applications. The toolkit is built using Grid and Web services technologies, and exchanges XML metadata between heterogeneous Web services, databases and clients using open standards. We show an application exemplar of how this toolkit may be used in a grid-enabled Computational Electromagnetics design search.

1 Introduction

Engineering design search and optimization (EDSO) is the process whereby engineering modelling and analysis are exploited to yield improved designs. This may involve lengthy and repetitive calculations requiring access to significant computational resources. This requirement makes the problem domain well-suited to the applications of Grid technology which enable large-scale resource sharing and coordinated problem solving within a virtual organisation (VO) [1]. Grid technology provides scalable, secure, high-performance mechanisms for utilizing remote resources, such as computing power, data and software applications over the Internet.

While compute power may be easily linked into applications using grid computing middleware, there has been less focus on database integration, and even less still on providing it in an environment familiar to engineers. Traditionally, data in many scientific and engineering disciplines have been organized in application-specific file structures, and a great deal of data accessed within current Grid environments still exists in this form [2]. When there are a large number of files it becomes difficult to find, compare and share the data. If database technologies are used to store additional information (metadata) describing the files, they can be located more easily using metadata queries. The Storage Resource Broker (SRB) [3] uses its Metadata Catalog (MCAT) for dataset access based on attributes rather than names or physical locations. However, MCAT has limited support for application specific metadata which is often essential in assisting engineers to locate data specific to their problems. In Geodise [4] our goal is to develop sophisticated but easy-to-use toolkits to help engineers make use of grid resources from environments they use daily, such as Matlab [5] and Jython [6]. These consist of the Geodise computational toolkit [7], XML toolbox [8] and a database toolkit, the focus of this paper, in which we adopt an open standards and service oriented approach to leverage existing database technologies and make them accessible to engineers and suitable for their problems.

¹ The full paper is available at [4]. This work is supported by the Geodise e-Science pilot project (UK EPSRC GR/R67705/01) and the DTI-funded GEM project.

2 Architecture

A major aim of the Geodise data management architecture is to bring together flexible, modular components for managing data on the Grid which can be utilized by higher level applications. Another objective is to provide a simple, transparent way for engineering users to archive files in a repository along with metadata. Various technical and application specific metadata about files, their locations and access rights are stored in databases. Files are stored in file systems and transported using the Globus Toolkit [9] which provides middleware for building computational grids and their applications. We use the platform independent Java CoG kit [10] to utilize the Grid Security Infrastructure (GSI) [11] for authentication, and GridFTP [12] for secure file transfer. As shown in Fig. 1, client side tools initiate file transfer and call Web services [13] for metadata storage, query, authorisation and file location.

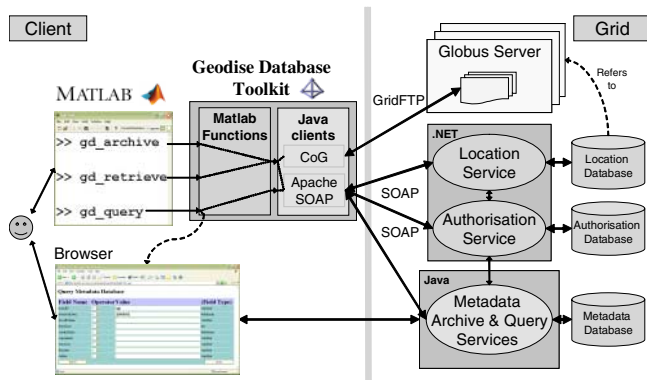


Fig. 1. A high level set of scripting functions sits on top of a client side Java API to provide an interface to data management Web service functionality and secure file transfer.

Access to databases is provided through Web services, which may be invoked using the Simple Object Access Protocol (SOAP) [14], to transfer data between programs on heterogeneous platforms. For example, our Java client code running on Linux or Windows can communicate with .NET Web services on a Windows server and Java Web services on a Linux server.

A unique handle is all that is required to locate and retrieve a file, as the *file location service* keeps a record of where it is stored. The *metadata archive service* allows the storage of additional descriptive information detailing a combination of technical characteristics (e.g. size, format) and user defined application domain specific metadata. The *metadata query service* provides a facility for engineers to find the data required without the need to remember the file names and handles.

We use relational tables for structured data and XML for more flexible storage of complex, deeply nested engineering specific metadata. We require a set of services that allow us to access and interrogate both types of data storage in a standard way. We currently provide APIs to specific databases with tailored Web services and will use these on top of implementations compliant with proposed standards from the Data Access and Integration Services Working Group [15] of the GGF [16].

The *authorisation service* uses a database of registered users, containing data access permissions mapping between VO user IDs and authenticated Globus

Distinguished Names, globally unique identifiers representing individuals. Query results are filtered and only metadata about files the user has access to are returned.

3 Problem Solving Environment and Application Example

The basic tasks (Fig. 2) for an engineer to undertake to manage and share their data are to (A) *generate the data* using standard engineering tools, (B) *store it* in the repository, (C) *search for data* of interest, and (D) *retrieve results* to the local file system. The wrapping of the core services enabling these tasks is straightforward as much of the logic of the client side components is written in Java, which can be directly exposed to Matlab or other high-level scripting environments, such as Jython [6]. Matlab contains a large number of toolboxes tailored to scientists and engineers, and makes it easy for them to quickly generate, analyse and visualize their data. We have implemented a range of Matlab functions on top of our core database services so that they can be incorporated programmatically into the user's scripts in a way consistent with the behaviour and syntax of the Matlab environment.

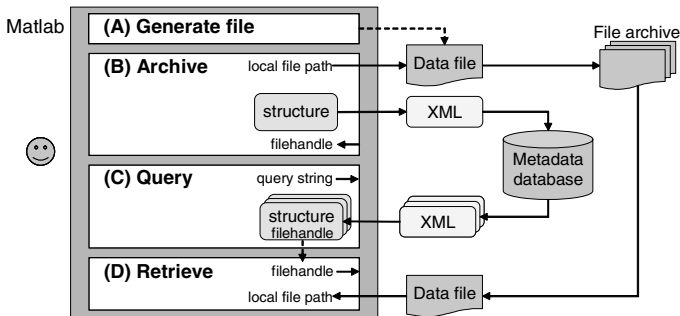


Fig. 2. Data flow of files and metadata: (A) file generation, (B) archive of file and user metadata, (C) querying of metadata, and (D) file retrieval.

We have applied the Geodise database toolkit in a real world example of design search in Computational Electromagnetics (CEM). The GEM project [16] is developing software for industrial use to improve the design of optical components for next generation integrated photonic devices. A specific device is the photonic crystal with certain light transmission and reflection properties which depend on the size and distribution of structures drilled into a slab of dielectric material. To investigate the characteristic photonic bandgap (PBG) of the crystal, an engineer samples a range of parameters (e.g. radius *radius* and spacing *d* of the holes) with each sample point giving rise to a different value of the objective function (the bandgap). Initially a large number of designs are explored which yield many solutions and large amounts of data. All of these solutions, whether good or poor, may yield valuable information for future simulations or devices and need to be preserved.

Fig. 3 shows two scripts, one to create data and archive it, the other for query and retrieval of data. The first stage (a) involves the user creating a certificate proxy so they can be authenticated and then generating data files and defining custom metadata about the geometry parameters and resulting bandgap as a Matlab structure, *m*. Then

the spectrum results file is stored using the `gd_archive` function (*b*), along with the metadata structure, which is converted into XML by our XML Toolbox for Matlab [8]. `gd_archive` then transports the file to a server and the metadata to the metadata service for storage in an XML database.

```

a  gd_createproxy;
    m.model = 'pgb_design'; m.param.d = [...]; m.param.radius = [...];
    ...
    compute_pgb(m.param, infile, outfile);
    m.result.bandgap = postprocs_pbg(outfile);
b  gd_archive(outfile, m);
    ...
c  Q = gd_query('model = pbg_design & result.bandgap < 99.7');
    Q: 4x1 struct array with fields standard, model, param, result
d  gd_retrieve({Q.standard.fileID}, '/home/Eng007/pbg_files/' );
    visualise_pbg_landscape ('/home/Eng007/pbg_files/*' );

```

Fig. 3. Example scripts to generate, archive, query, retrieve and post-process data.

This script is run a number of times with different parameters. After the computations have finished and the design results are available in the database, the Engineer can check the results with a simple query (*c*). The query can be formed using a combination of named metadata variables and comparison operators or alternatively through a graphical interface. *Q* is a vector of structures containing the metadata of all PBG designs with a bandgap less than 99.7. In this case, four designs match the query. For further investigation or visualization, the Engineer can retrieve files associated with the above four designs to the local file system (*d*).

Fig. 4 shows typical data we can obtain for the various design parameters. The simulation results form an objective function landscape of the photonic bandgap from which a full design search and optimization may be performed. The storage of the results in a database as well as the transfer of files to a file store on the Grid additionally allows data re-use by engineers.

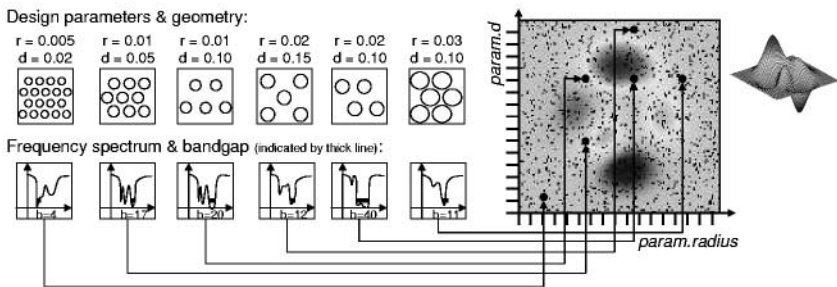


Fig. 4. Example of CEM design search using Geodise database technology. Shown are design geometries, the computed frequency spectrum with the bandgap, and representative data for the objective function landscape (dots indicate sample points in parameter space).

5 Conclusions and Future Work

We have described a framework which allows the use of databases on the Grid in an engineer-friendly environment. We have implemented a suite of services which combine a commercial PSE (Matlab) with a core framework of open standards and service oriented technologies. We have shown how design search in electromagnetics can be supported by the Geodise database toolkit. The transparent integration of database tools into the engineering software environment constitutes a starting point for database applications in EDSO, and is only one of many potential applications in the engineering domain (CFD, CEM, etc.). The functions we have implemented to extend the Matlab environment allow engineers to share and re-use data conveniently. The automatic generation of standard metadata and support for user-defined metadata allows queries to be formed that represent the Engineer's view of the data.

Future work will involve providing tools to generate, compare and merge XML Schemas describing users' custom metadata. We will also evolve our Web service based components to GGF DAIS-WG compliant Grid services.

References

- [1] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.
- [2] M.P. Atkinson, V. Dialani, L. Guy, I. Narang, N.W. Paton, D. Pearson, T. Storey and P. Watson. Grid Database Access and Integration: Requirements and Functionalities. UK e-Science Programme Technical Report
<http://www.cs.man.ac.uk/grid-db/papers/DAIS:RF.pdf>
- [3] The Storage Resource Broker, <http://www.npaci.edu/DICE/SRB/>
- [4] Geodise Project <http://www.geodise.org/>
- [5] Matlab 6.5. <http://www.mathworks.com>
- [6] J. Hugunin. Python and Java: The Best of Both Worlds. 6th International Python Conference. San Jose, California, USA, 1997.
- [7] G. Pound, H. Eres, J. Wason, Z. Jiao, A. J. Keane, and S.J. Cox, A Grid-enabled Problem Solving Environment (PSE) For Design Optimization Within Matlab. IPDPS-2003, April 22–26, 2003, Nice, France.
- [8] M. Molinari. XML Toolbox for Matlab. <http://www.soton.ac.uk/~gridem>
- [9] The Globus Project. <http://www.globus.org/>
- [10] Commodity Grid Kits. <http://www.globus.org/cog/>
- [11] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. National-Scale Authentication Infrastructure, *IEEE Computer*, 33(12):60–66, 2000.
- [12] GridFTP, <http://www.globus.org/datagrid/gridftp.html>
- [13] Web Services Activity, <http://www.w3.org/2002/ws>
- [14] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H.F. Nielsen. SOAP Version 1.2, W3C Candidate Recommendation, 2002
- [15] Global Grid Forum, <http://www.ggf.org/>
- [16] The GEM project, <http://www.soton.ac.uk/~gridem>

A Resource Accounting and Charging System in Condor Environment

Csongor Somogyi, Zoltán László, and Imre Szeberényi

BUTE Dept. of Control Engineering and Information Technology, Budapest, Hungary
{csongor, laszlo, szebi}@iit.bme.hu

Abstract. The authors' aim is to create a resource accounting and charging system for the Grid environment that can be set up as part of the supercomputing Grid prototype, which is developed in an ongoing domestic project. Stiller, Gerke, Reichl and Flury [1,2] has introduced a distributed accounting and charging concept, in which they point out that it could be applied in Grid context as well. This model has been adopted and improved by the authors in the given context; they had created the specification of a proposed solution at algorithm level. The necessary data structures and its query interfaces have been defined. The metering, accounting and pricing processes have been implemented. The system has been adapted to the Condor workload management system. The resulting application has been deployed at the departmental cluster. The authors funnel is to continue developing the remaining modules of the system and bring out the completed version as the part of the professional services for the Grid.

This paper introduces the applied model, the specification that was built upon it, the results of the implementation and the operating test environment.

Keywords: Grid, Resource accounting and charging, Condor

1 Introduction and Motivation

1.1 Professional Services for Commercial Grids

It is necessary to consider a kind of infrastructure, which can give access to exploit many types of resources in mass without either interfering with the legacy environment of resource providers or constraining the resource consumers in the method of access, and it has to match the security and interoperability requirements, which are recently becoming more important. System scalability and high grade of diversity of utilization are also very important factors. This is the *Grid* initiative. The final result of this achievement is a proposal of standard.

The challenging requirements against the Grid resource management have been identified by the researchers of the Globus meta-computing system [3]. The collected experiences led to draw the initial concept of the Grid architecture, a four-layer "hourglass" model [4]. They have started standardizing the architecture through the *Open Grid Services Architecture* model [5] using Web Services.

1.2 Research Context

Base services of the Grid may include the resource representation abstractions, resource management, informational services, data connectivity (data repositories, application repositories, file transfer, etc.) and the underlying communication protocols.

On the other hand, users or consumers can only exploit the benefits given by Grid providers if they have adequate access. We have to define what the most important accessibility features are:

- application development environment (like P-GRADE [7])
- refined user interface for the management of resource usage
- accounting and charging of resource usage and payment services
- user support (documentation, help, downloads, etc.)

The *Hungarian Supercomputing Grid* project [6] supported by the home Ministry of Education has been created to establish the first working prototype of an interior Grid. Their aim is to suit the above mentioned demands, set up against a professional Grid – except the user support, which is a very costly business.

The authors of this article have been developing the accounting and charging infrastructure to be applied in the Hungarian SuperGrid. On the other hand, their objective is to provide a flexible, sophisticated toolkit, which can be easily aligned with the requirements of the evolving *Grid architecture*.

The authors are also involved in the EASYCOMP¹ – IST Future and Emerging Technologies program – project. The goal of the project is to develop the foundation of such composition technology, which should enable users to compose (standard) components powerfully and easily.

2 System Specification

We have been applying in the field of Grid computing the charging and accounting model described by Stiller, Gerke, Reichl and Flury [1]. They started their work in order to elaborate the basis of the commerce of network (in the so-called CATI project — Charging and Accounting Technologies for the Internet) but they pointed out that this architecture is capable of being implemented in Grid context as well [2] because of its generality and modularity, which can be exploited in such a distributed environment like the *Grid*. The work of the authors of this article was to improve this model to adapt it to the Grid context.

The CATI researchers have identified and decomposed the terms related to accounting and charging. This decomposition contains the abstract definitions of the terms *metering*, *mediating*, *accounting*, *charging* and *billing* and the schema of their relations.

¹ “Work partially supported by the European Community under the IST programme – Future and Emerging Technologies, contract IST-1999-14191 – EASYCOMP. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.”

The authors of this article examined the mentioned model in the context of Grid and proposed an escalation of the original model.

The three main subsystems (*accounting* – involving metering and mediation processes –, *charging* and *billing*) have been decomposed into their atomic elements, which have been matched with software components. The Grid related processes, which pass information between the Grid and the accounting and charging process, have been identified. The necessary data abstractions have been defined. These include the representations of Grid related entities (e.g. *resource*, *resource user*), the parameters that are exchanged between the atomic processes of the system and the data constructs that are stored or used during the entire process (*like accounting record*, *pricing rule*, *charging record* and *receipt entries*). For each of these abstractions, the corresponding data structures have been declared.

The process flow has been summarized in a data-flow model, while the data abstractions are represented as an entity-relationship model. Finally, a textual description of the dynamic behavior of the accounting and charging processes has been created.

Both the *logical*, which was briefly introduced above, and the *physical* specification of the system have been completed. The most important requirement, distributiveness, is fulfilled. The metering and mediation modules should only be placed locally at the resource, while the rest of the components can be placed independently anywhere. Surely, one component can serve an arbitrary set of resources or resource users, it does not matter how the information at various levels (accounting, charging or billing) are grouped together. Actually, this approach provides us balanced information processing and storage.

3 Implementation and Results

The implementation of the fundamental components (metering, accounting and pricing) has been completed. CPU time consumption has been chosen as sample measured parameter. SQL and HTTP/XML are used for interfacing the different components. Later, these can be easily changed into Web Services. Development work is done in UNIX environment using mostly Perl and partially C languages, Perl DBI and GTop libraries.

Condor system [8] is the selected local job manager infrastructure. Condor can run a job wrapper each time a job is started on a given host. This wrapper was developed in order to establish binding between the job manager and the accounting subsystem.

The developed parts of the system are deployed at the departmental Condor cluster. Test runs are performed demonstrating the operation of the accounting and charging system.

While the structure and the behavior of the most accounting and charging components are clarified, metering still faces with different challenges. The various platforms treat resource consumption parameters, especially *process accounting* parameters, in different ways. On the other hand, there are no stan-

dard(ized) units defined. Since our measuring method takes resource usage snapshots at equal time-intervals, sampling error can be perceived. Since we need to know the resource consumption at a given moment as well as the total amount of consumption at the resource usage completion, we have to examine what accuracy is necessary and may have to combine the usage of different operating system provided ways of resource usage accounting in order to achieve accuracy precisely. We are investigating these questions in the near future.

4 Summary and Conclusions

The initial accounting and charging model has been extended in order to fit the requirements against the Grid context. The authors have completed the specification of the proposed solution. Partial development of the system, the accounting and the pricing modules have been completed. The application has been merged with the Condor workload management environment and the test system has been set up at the departmental cluster. Test runs have been performed in order to demonstrate the operation of the system. Our funnel is to continue the development of the remaining parts of the system, integrate the solution with the mentioned domestic supercomputing Grid and investigate the metering related issues.

References

1. B. Stiller, J. Gerke, P. Reichl and P. Flury: A Generic and Modular Internet Charging System for the Cumulus Pricing Scheme. *Journal of Network and Systems Management*, **3(9)** (September 2001) 293–325
2. B. Stiller, J. Gerke, P. Reichl, P. Flury and Hasan: Charging Distributed Services of a Computational Grid Architecture. *IEEE International Symposium on Cluster Computing (CCGrid 2001)*, Workshop on Internet QoS for the Global Computing (IQ 2001), (May 2001) 596–601
3. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke.: A Resource Management Architecture for Metacomputing Systems. *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing* (1998)
4. I. Foster, C. Kesselman and S. Tuecke.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, **15(3)** (2001)
5. I. Foster, C. Kesselman, J. Nick and S. Tuecke: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum* (June 22, 2002)
6. IKTA-00075/2001, Hungarian Supercomputing Grid. Tender for Information and Communication Technology Applications by the Hungarian Ministry of Education, Budapest (November 2002)
7. P. Kacsuk: Parallel Program Development and Execution in the Grid. *Proc. of PARELEC 2002, Warsaw* (2002)
8. Douglas Thain and Miron Livny: Condor and the Grid. in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley (2003)

Secure Web Services with Globus GSI and gSOAP

Giovanni Aloisio¹, Massimo Cafaro¹, Daniele Lezzi¹, and Robert van Engelen²

¹ High Performance Computing Center
University of Lecce/ISUFI, Italy
`giovanni.aloisio@unile.it`
`massimo.cafaro@unile.it`
`daniele.lezzi@unile.it`

² Computer Science Department
Florida State University, USA
`engelen@cs.fsu.edu`

Abstract. In this paper we describe a plug-in for the gSOAP Toolkit that allows development of Web Services exploiting the Globus Security Infrastructure (GSI). Our plug-in allows the development of GSI enabled Web Services and clients, with full support for mutual authentication/authorization, delegation of credentials and connection caching. The software provides automatic, transparent transport-level security for Web Services and is freely available.

1 Introduction

Recently, the Web Services framework [1] has gained considerable attention. Based on XML (Extensible Markup Language) [2] technologies like SOAP (Simple Object Access Protocol) [3], WSDL (Web Services Description Language) [4], WSFL (Web Services Flow Language) [5] and UDDI (Universal Description, Discovery and Integration) [6], the Web Services approach to distributed computing represents the latest evolution supporting the creation, deployment and dynamic discovery of distributed applications. As a matter of fact, the Internet and the Web allow to publish and retrieve documents easily, and to access a number of commercial, public and e-government services. However, the focus on the usage of such services is now been shifted from people to software applications.

The Web Services framework makes this shift possible, due to the convergence of two key technologies: the Web, with its well known and universally accepted set of standard protocols for communication, and Service-Oriented computing where both data and business logic is exposed through a programmable interface e.g. CORBA (Common Object Request Broker Architecture), Java RMI (Remote Method Invocation), DCE RPC (Remote Procedure Call). Web Services can be accessed through the HTTP (Hyper Text Transfer Protocol) and HTTPS (Secure Hyper Text Transfer Protocol) protocols and utilize XML to exchange data. This implies that Web Services are independent of platform, programming language and network infrastructure.

Even in the Grid community, the focus is now shifted from protocols [7] to Grid Services [8], as envisioned by the Open Grid Services Architecture (OGSA) [9]. Grid Services extend the Web Services framework, and the Grid itself becomes an extensible set of Grid Services that may be aggregated to provide new capabilities. However, a Grid Service is "a (potentially transient) stateful service instance supporting reliable and secure invocation (when required), lifetime management, notification, policy management, credential management, and virtualization" [8]. So, Grid Services leverage both WSDL and SOAP but additional interfaces able to manage service lifetime, policies and credentials, and to provide support for notification are mandated by the OGSA specification.

Since the OGSA specification is not yet completed and the Globus Toolkit v3 is not yet available to develop production software (only an alpha version has been released as of this writing), we decided to adopt the Web Services framework jointly with the Globus Toolkit v2 as our middleware/computing infrastructure in the GridLab project [10].

The adoption of Globus GSI [11] as the security infrastructure and of the gSOAP Toolkit [12] for the development of Web Services led us to write the GSI plug-in for gSOAP, needed to secure the Web Services developed in the context of the GridLab project. The plug-in provides automatic, transparent transport-level security for Web Services and is freely available [13].

The paper is organized as follows. Section 2 describes the gSOAP Toolkit. We present our GSI plug-in in Section 3, recall related work in section 4 and conclude the paper in Section 5.

2 The gSOAP Toolkit

The gSOAP toolkit is a platform-independent development environment for C and C++ Web services. The toolkit provides an easy-to-use RPC compiler that produces the stub and skeleton routines to integrate (existing) C or C++ applications into SOAP/XML Web services. A unique aspect of the gSOAP toolkit is that it automatically maps native C/C++ application data types to semantically equivalent XML types and vice versa. This enables direct SOAP/XML messaging by C/C++ applications on the Web. As a result, full SOAP interoperability can be achieved with a simple API relieving the user from the burden of SOAP details, thus enabling him or her to concentrate on the application-essential logic. The toolkit uses the industry-standard SOAP 1.1/1.2 and WSDL 1.1 protocols and offers an extensive set of features that are competitive to commercial implementations, including stand-alone HTTP server capabilities, Zlib compression, SSL encryption, and streaming direct internet message encapsulation (DIME) attachments. For many companies, gSOAP has proven an excellent strategy for developing Web services based on C and C++ applications. For example, gSOAP is integrated in the IBM alphaWorks Web Services Tool Kit for Mobile Devices (WSTKMD) [14]

The gSOAP toolkit was designed with ease-of-use in mind. It exploits a novel schema-driven XML parsing technique to deserialize C/C++ application data

from SOAP/XML messages in one sweep, thereby eliminating the overhead that is incurred by SOAP/XML software with multi-tier communication stacks.

gSOAP is available for download from SourceForge [15] and is licensed under the open source Mozilla Public License 1.1 (MPL1.1).

3 The GSI Plug-in for gSOAP

Our plug-in exploits the modular architecture of the gSOAP Toolkit that enables a simple extension mechanism of gSOAP capabilities. To take advantage of a plug-in, a developer must register it with gSOAP, so that full access to run-time settings and function callbacks is granted. The registration associates the plug-in's local data with gSOAP run-time and is done using the gSOAP *soap_register_plugin* function, supplying as one of the arguments the plug-in initialization function. In our case, we perform the necessary initialization steps inside the *globus_gsi* function : we activate the Globus Toolkit I/O module, set-up local data and extend gSOAP capabilities overriding gSOAP function callbacks.

The last initialization step is to provide two callbacks that will be used by the gSOAP environment respectively to copy and delete the plug-in's local data (when de-registering the plug-in). The plug-in's local data can be accessed through the *soap_lookup_plugin* function. Currently, as local data we have Globus I/O related variables (connection attribute, handle, etc), the distinguished names that identify a client or a server when mutual authentication is performed, and the pathname where on the local file system the proxy that a client sends when performing delegation of credentials is written by our plug-in. Finally, we have a boolean variable that distinguishes a client from a server. This mechanism is exploited for instance in the Globus I/O authorization callback that a developer must provide to perform authorization upon authentication: if the software acts as a client, then authorization is based on the server's identity (distinguished name as found in the X509v3 certificate) and vice-versa.

We now briefly describe how the plug-in functions utilize the Globus Toolkit I/O API to provide gSOAP with automatic, transparent transport-level security.

The *gsi_connect* function sets the client mode and calls *globus_io_tcp_connect* to establish a connection to the remote Web Service; the *gsi_disconnect* function checks the state of the connected handle and closes the connection calling *globus_io_close*; the variables related to the server's or client's identity that have been dynamically allocated, are then freed.

The *gsi_send* function is in charge of actually sending data on the Globus I/O connected handle; this is done calling in a loop until needed *globus_io_write*. Symmetrically, the *gsi_recv* function reads data from a Globus I/O connected handle calling *globus_io_read*.

The *gsi_listener*, *gsi_listen* and *gsi_accept* functions are all needed to develop a server; *gsi_listener* sets server mode and calls *globus_io_tcp_create_listener* to create a Globus I/O listening handle, while *gsi_listen* calls *globus_io_tcp_listen* and blocks waiting for incoming connections on the listening handle. Thus, the

Globus function behaves differently from the traditional TCP sockets API *listen* call. Finally, the *gsi_accept* function calls *globus_io_tcp_accept* and in case of success creates the connection handle and calls *globus_io_tcp_get_remote_address* to retrieve the peer's IP address and port.

The *gsi_connection_caching* and *gsi_reset_connection_caching* are used respectively to setup and reset connection caching. This is achieved using the gSOAP internal mechanism for keep-alive connections and calling the Globus function *globus_io_attr_set_socket_keepalive* as needed.

The other functions we provide are needed to setup properly the GSI channel: the developer is allowed to setup the TCP socket reuse addr option, which is useful server side, and to setup authentication, channel mode, protection mode, authorization mode and delegation mode.

Both clients and servers developed using our plug-in can use the Globus I/O authorization callback mechanism; this entails writing a related Globus function called *globus_io_secure_authorization_callback* to enforce the developer's policy.

The plug-in software requires GNU autotools (autoconf v2.57, automake v1.7.2), the Globus Toolkit v2.x and the gSOAP Toolkit v2.2.3d. We provide in our distribution example servers and their related clients, that show how to write a simple threaded server, a pre-threaded server, a simple fork server and a pre-forked server. The threaded servers provide a good example of how to do mutual authentication/authorization and how to setup and use connection caching; the fork servers in addition show how to do delegation of credentials: the server receives from the client the delegated credentials and uses them to submit a job to a remote Globus gatekeeper. Our software is licensed under the open source GNU General Public License.

4 Related Work

A similar GSI plug-in for gSOAP has been developed in the context of the Globus project for a C based preliminary version of OGSA. This software provides basically the same functionalities described here, however, our plug-in has been made available to the Grid community before and it is widely used in several academic institutions and software companies, while the Globus plug-in will not be considered production-level software until the final official release of the Globus Toolkit v3. Moreover, the new version of the Globus gSOAP plug-in, to be included in the C based alpha version of OGSA, will support in addition to transport-level security a new GSI mechanism for securing grid services, using message-level security.

The issues that arise using transport-level security are the high computational cost (the entire message must be encrypted and/or signed), the problem related to firewalls (non privileged ports must be opened), and the problem of intermediate hops. A transport-level security mechanism provides two-way encryption and one-way or two-way authentication between a predefined pair of SOAP message endpoints, but SOAP request and response messages may be required to traverse multiple hops between the Web service and the consuming

application. In this case, message security and user identity can be compromised at the intermediate hops.

A message-level security mechanism is safer because there is no exposure at intermediaries (hop-to-hop vs end-to-end) and flexible because building only on proven Web infrastructure (HTTP etc). Another advantage is that firewalls are not an issue here (Web protocols are usually allowed through a firewall) and the associated computational cost may be relatively low, due to the possibility of selective encryption and/or signature of SOAP messages (only some XML elements actually need encryption because they contain sensitive information).

Web Services security is an active research field, and many specifications are now available. We are now developing a complementary version of our plug-in to support message-level security. Both our software and the Globus based plug-in will harness the following specifications:

- WS Secure Conversation [16];
- WS Security [17];
- XML Encryption [18];
- XML Signature [19].

SOAP messaging is enhanced using message integrity, confidentiality and authentication. These specifications allow exchanging a security context between a Web Service and a client, establishing and deriving session keys, and provide a general purpose mechanism for associating security tokens with messages. An important property of these specifications is that they are extensible, i.e., they support multiple security token formats.

5 Conclusions

In this paper we have reported about a GSI plug-in for the gSOAP Toolkit. Our plug-in allows the development of GSI enabled Web Services and clients, with full support for mutual authentication/authorization, delegation of credentials and connection caching. The software is being used in several academic institutions for grid projects and testbeds including the GridLab project, moreover, some software companies have demonstrated an interest for it. The plug-in provides automatic, transparent transport-level security for Web Services and is freely available.

We will continue to support the GSI plug-in to provide transport-level security for Web Services; in addition our focus is currently on the development of a complementary version of the software to provide message-level security.

Acknowledgements. We gratefully acknowledge support of the European Commission 5th Framework program, grant IST-2001-32133, which is the primary source of funding for the GridLab project.

References

1. Kreger, H.: Web Services Conceptual Architecture WSCA 1.0. IBM, 2001
2. XML specification. <http://www.w3.org/XML/Core/>
3. SOAP specification. <http://www.w3.org/TR/SOAP/>
4. WSDL specification. <http://www.w3.org/TR/wsdl>
5. WSFL specification.
<http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>
6. UDDI specification. <http://www.uddi.org/specification.html>
7. Foster, I., Kesselmann, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, Vol.15, 2001, No. 3, pp. 200–222
8. Foster, I., Kesselmann, C., Nick, J., Tuecke, S.: Grid Services for Distributed System Integration. *Computer*, Vol. 35, 2002, No. 6, pp. 37–46
9. Foster, I., Kesselmann, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed System Integration. Technical Report for the Globus project. <http://www.globus.org/research/papers/ogsa.pdf>
10. The GridLab project. <http://www.gridlab.org>
11. Foster, I., Kesselmann, C., Tsudik G., Tuecke, S.: A security Architecture for Computational Grids. *Proceedings of 5th ACM Conference on Computer and Communications Security Conference*, pp. 83–92, 1998.
12. Van Engelen, R.A., Gallivan, K.A.: The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. *Proceedings of IEEE CCGrid Conference*, May 2002, Berlin, pp. 128–135
13. Cafaro, M., Lezzi, D., Van Engelen, R.A.: The GSI plugin for gSOAP.
<http://sara.unile.it/~cafaro/gsi-plugin.html>
14. IBM alphaWorks, Web Services Tool Kit for Mobile Devices,
<http://www.alphaworks.ibm.com/tech/wstkMD>
15. The gSOAP Toolkit. <http://gsoap2.sourceforge.net>
16. WS Secure Conversation specification.
<http://www-106.ibm.com/developerworks/library/ws-secon/>
17. WS Security specification. <http://www.oasis-open.org/committees/wss/>
18. XML Encryption. <http://www.w3.org/Encryption/2001/>
19. XML Signature. <http://www.w3.org/Signature/>

Future-Based RMI: Optimizing Compositions of Remote Method Calls on the Grid

Martin Alt and Sergei Gorlatch

Technische Universität Berlin, Germany

Abstract. We argue that the traditional RMI (remote method invocation) mechanism causes much unnecessary communication overhead in Grid applications, which run on clients and outsource time-intensive method calls to high-performance servers. We propose *future-based RMI*, an optimization to speed up compositions of remote methods, where one method uses the result of another method as an argument. We report experimental results that confirm the performance improvement due to the future-based RMI on a prototypical Grid system on top of Java.

1 Introduction and Motivation

Grid computing aims to combine different kinds of computational resources connected by the Internet to make them easily available to a wide user community. One popular approach to developing applications for Grid-like environments is to provide libraries on high-performance servers, which can be accessed by clients using some remote invocation mechanism, e. g. RPC/RMI.

Our work is motivated by many important applications where remote method calls are composed with each other. For illustration, we use a very simple Java code fragment, where the result of `method1` is used as an argument by `method2`:

```
result1 = server1.method1();  
result2 = server2.method2(result1);
```

The execution of the methods can be distributed across different servers of the Grid, i. e. with different RMI references assigned to `server1` and `server2`.

The timing diagram for such a distributed composition of methods using standard RMI is shown in Fig. 1 (left): the result of `method1` is first sent to the client, and from there (as a parameter of `method2`) to the second server. In larger applications, this leads to much unnecessary communication between the client and servers, thus increasing execution time of Grid programs.

2 Optimizing RMI Using Future References

The aim of *future-based RMI* is to optimize the remote execution of compositions of methods. The idea is that a method invocation immediately returns a remote reference to the result (a *future reference*), which is sent to the client and can immediately be passed on to the next method. Attempts to retrieve the referenced data are blocked until the data becomes available. For details, see [1].

The timing diagram for our future-based RMI is shown in Fig. 1 (right): the reference to **result1** is returned immediately and passed on to **method2**. The second server requests **result1** from the first one, while the first method is still executing. When the result is available it is sent directly to the second server.

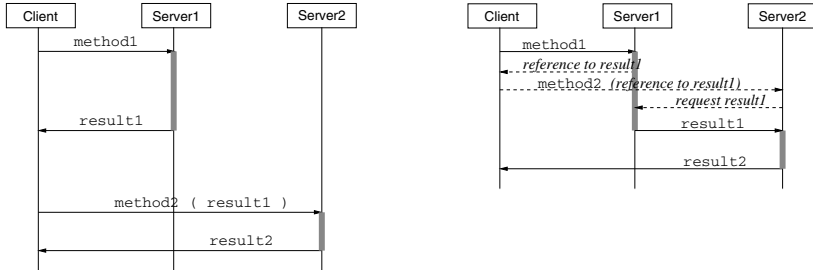


Fig. 1. Distributed composition using plain RMI (left) and future-based RMI (right)

The future-based RMI has two advantageous properties in the Grid context:

Laziness: the amount of data sent from the server to the client upon method completion is reduced substantially, because only a remote reference is returned. This reference is passed to the next server, and used to request the result. The result itself is sent directly from one server to the next one.

Asynchrony: communication between client and server overlaps with computations, because future references are returned immediately. Thus, network latencies can be hidden.

2.1 Implementation and Exception Handling

Under our optimized RMI, methods do not return their results directly. Instead, they return a remote reference to an object of the new class `RemoteReference` which is instantiated on the server side. It provides two methods:

```
public Object getValue() ...; //remotely accessible
public void setValue(Object o) ...;
```

The `setValue()` method is called by the application method's implementation when the result of the method is available, passing the result as a parameter. Method `getValue()` is used to retrieve this result and may be called remotely. Calls to `getValue()` block until a value has been assigned using `setValue()`. If `getValue()` is called remotely, the result is sent over the network to the caller.

To execute a method asynchronously after returning a future reference to the caller, remote methods spawn a new thread to carry out computations and then return immediately. To avoid thread-creation overhead, a pool of threads is created when the server starts; threads are taken from the pool on demand.

From the user's point of view, a distributed composition of methods under the future-based RMI is expressed in much the same way as with plain RMI.

The pseudocode for a method taking a `RemoteReference` as parameter and returning another one as result is as follows:

```
public RemoteReference method(RemoteReference ref) {
    Object parameter = ref.getValue();
    RemoteReference ret = new RemoteReference();
    /* execute method1 in a new thread, that will
       call ret.setValue() upon completion */
    Thread t = new Method1Thread(parameter, ret);
    return ret;}

```

An important mechanism for error handling in Java are exceptions, which should thus also be present in a distributed setting. With plain RMI, exception handling resembles local method invocation: an exception thrown in the remote method is caught on the server, sent to the client and rethrown there. In the future-based RMI, a remote method invocation returns to the calling client before the method on the server is actually completed; thus, an exception thrown on the server cannot be thrown immediately on the client. In our implementation, the exception is caught on the server and stored in the object, that the future reference returned to the client points to. When the `getValue()` method of this object is called, the exception is rethrown. If the caller of `getValue()` was another server-sided method, the exception is caught again and wrapped in the next object referenced by a future reference until it finally reaches the client.

3 Experimental Results

Our testbed environment consists of two university LANs, one at TU Berlin and the other at the University of Erlangen, at a distance of approx. 500 km. We used a SunFire6800 as server in Berlin and an UltraSparc Iii 360 MHz as the client in Erlangen, both with SUN JDK1.4.1 (HotSpot Client VM, mixed mode).

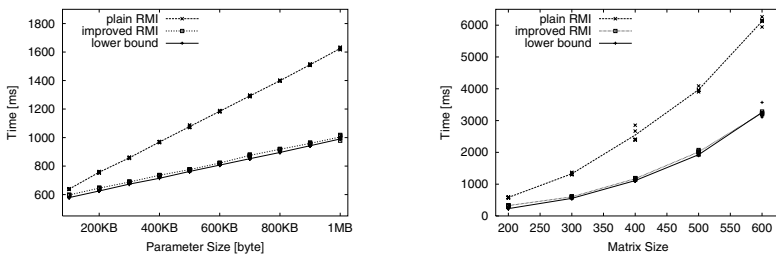


Fig. 2. Runtimes for simple method composition (left) and linear system solver (right).

We measured the performance of the small code from Sect. 1, with `method1` and `method2` both taking 500 ms, and the amount of data sent ranging between 100 KB and 1000 KB. Our second case study is a linear equations system solver using a server-side matrix library; it consists of composed RMI calls for solving a minimalization problem, matrix multiplication and subtraction.

Fig. 2 (left) shows the runtimes for three different versions of the small code: (1) two method calls with plain RMI, (2) two calls with future-based RMI, and (3) one method call that takes twice as much time as the original call. We regard the one-call version as providing ideal runtime (“lower bound”) for a composition of remote methods. The figure presents five measurements for each version of the program, with the average runtimes for each parameter size connected by lines.

For both the simple method composition and the linear system solver, the figure shows that the future-based RMI is between 100 ms and 350 ms faster than the plain RMI. The composition under future-based RMI is only 10-15 ms slower than the “lower-bound” version, which means that our optimizations eliminated between 85% and 97% of the original overhead.

4 Conclusions and Related Work

We have proposed future-based RMI, which greatly reduces communication overhead and network dataflow when executing compositions of remote methods. The concept of hiding latencies using futures was studied e. g. in [2] and implemented in Java (e. g. [3,4,5]). However, these approaches lack the ability to send futures to remote servers and can thus not realize server/server communication.

This feature can be found in [6], where RMI calls are optimized using call-aggregation and where a server can invoke methods on another server directly. While this approach optimizes RMI calls by reducing the amount of data, the method invocations are not asynchronous as in our implementation, instead they are delayed to find as much optimization possibilities as possible.

The prototypical implementation using Sun’s Java RMI confirmed the advantages of our future-based solution, which can be built on top of an arbitrary RMI mechanism, including optimized versions like KaRMI or Manta.

Acknowledgements. We wish to thank the anonymous referees and also Thilo Kielmann and Robert V. van Nieuwpoort for their very helpful comments on the preliminary version of this paper.

References

1. Alt, M., Gorlatch, S.: Optimizing the use of Java RMI for grid application programming. Technical Report 2003/08, TU Berlin (2003) ISSN 1436-9915.
2. Walker, E.F., Floyd, R., Neves, P.: Asynchronous remote operation execution in distributed systems. 10th Int. Conference on Distributed Computing Systems. (1990)
3. Raje, R., Williams, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism in Java. *Concurrency: Practice and Experience* **9** (1997)
4. Falkner, K.K., Coddington, P., Oudshoorn, M.: Implementing asynchronous remote method invocation in Java. In: *Proc. of Parallel and Real Time Systems* (1999)
5. Baduel, L., Baude, F., Caromel, D.: Efficient, flexible, and typed group communications in Java. In: *JGI’02, Seattle, ACM* (2002)
6. Yeung, K.C., Kelly, P.H.J.: Optimising Java RMI programs by communication restructuring. In: *Middleware 2003, Springer* (2003)

Topic 7

Applications on High-Performance Computers

Jacek Kitowski, Andrzej M. Goscinski, Boleslaw K. Szymanski, and
Peter Luksch
Topic Chairs

The emergence of low-cost PC clusters together with the standardization of programming models (MPI and OpenMP) have paved the way for parallel computing to come into production use. In all domains of high performance computing, parallel execution routinely is considered as one of the major sources of performance. In some domains, like computational fluid dynamics, commercial codes already offer parallel execution as an option. In new domains, like bioinformatics, parallel execution is considered early in the design of algorithms and software. Besides clusters, grid computing is receiving increasing attention.

This year, 24 papers were submitted, 13 of which have been accepted (acceptance rate: 54.2%), eight as regular papers, three as short papers. The topic is subdivided into three sections.

Numerical Analysis. Algorithms from linear algebra are considered in “*Parallel Linear System Solution and its Application to Railway Power Network Simulation*” and “*Improving Performance of Hypermatrix Cholesky Factorization*”. “*Implementation of Adaptive Control in Algorithms in Robot Manipulators using Parallel Computing*” deals with robot manipulators whose dynamic parameters are not completely known. The paper “*Cache Performance Optimizations for Parallel Lattice Boltzmann Codes*” shows that efficient utilization of the memory hierarchy is essential to application performance.

Algorithms from other domains. Two papers address image processing: “*Low Level Parallelization of Nonlinear Diffusion Filtering Algorithms for Cluster Computing Environments*” and “*Interactive Ray Tracing on Commodity PC*”. The paper “*Comparing two Long Biological Sequences Using a DSM System*” describes parallel implementations of the Smith-Waterman algorithm on a distributed shared memory system. “*Parallel Agent-Based Simulation on a Cluster of workstation*” presents a discrete time simulation implemented in a dynamically typed language. “*Effectiveness of Parallelizing the ILOG-CPLEX Mixed Integer Optimizer in the PUBB2 Framework*” describes the parallelization of branch and cut algorithms that solve mixed integer optimization problems on clusters of PCs.

Management of parameter studies and automatic optimization is addressed in three papers. “*CAD-Grid: Corporate Wide Resource Sharing for Parameter Studies*” describes a framework for corporate-wide shared use of simulation software and HPC resources that is being deployed in an industrial environment. A tool that manages execution of a large number of experiments,

including resubmission of failed jobs, is presented in “*Towards Automatic Management of Embarrassingly Parallel Applications*”. “*Two Dimensional Airfoil Optimization using CFD in a Grid Computing Environment*” describes a tool for automatic CFD optimization that assembles different industrial software components.

CAD Grid: Corporate-Wide Resource Sharing for Parameter Studies

Ed Wheelhouse¹, Carsten Trinitis², and Martin Schulz³

¹ Department of Computer Science, University of Newcastle, Newcastle, UK
`edwheelhouse@yahoo.co.uk`

² Technische Universität München (TUM), Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation,
Boltzmannstr. 3, 85748 Garching bei München, Germany
`Carsten.Trinitis@cs.tum.edu`

³ School of Electrical and Computer Engineering, Cornell University,
Ithaca, NY, 14853, USA
`schulz@csl.cornell.edu`

Abstract. The optimization process for most modern engineering problems involves a repeated modeling of the target system, simulating its properties, and refining the model based on the results. This process is both time and resource consuming and therefore needs to rely on a distributed resource sharing framework in order to optimally exploit the existing resources and minimize the response time for the design engineers.

We have implemented such a framework for the design process of high voltage components and have shown its applicability to a real industrial environment. First results are very encouraging and show a high acceptance rate with the end-users. In addition, experiments with various different models show the profound impact of the optimization on the design of high-voltage components.

1 Motivation

The geometric shape of transformers and other high voltage gear has a profound impact on their electrical properties and on their performance. Suboptimal designs can lead to overheating, flashovers between close parts, higher energy loss rates, etc. . It is therefore necessary to optimize the geometric shape already in the initial design phase using a detailed simulation of the electric properties of the designs. This enables the engineers to detect critical regions and to change the respective CAD geometry in order to obtain an optimal design. This can be automated using an iterative optimization approach. During each iteration, a new CAD model is generated and used as the input for the following simulation of the electric field. The result of the simulation is then used to refine the model. This process continues until an optimal design has been found.

Each iteration requires repeated access to the CAD modeling tool and to the compute server for the simulation. The requests for these resources are interleaved and the usage of each resource potentially requires a significant amount

of time since both model generation and simulation are non-trivial operations. In addition, the resources for each step are strictly limited by the number of available parallel systems for the simulation and by appropriate licenses for the CAD modeling. Therefore, a single optimization request should not block a compute or a CAD modeling resource for its entire runtime and neither should each engineer require his or her own, dedicated set of resources. Instead, a corporate-wide resource sharing infrastructure is needed, which dynamically assigns jobs to available resources.

We have implemented such a resource sharing environment in cooperation with ABB corporate research. Design engineers can access the necessary resources for the complete optimization process from their desktop. The overall process has been integrated seamlessly into the design workflow. In addition, the infrastructure enables an efficient resource sharing across all corporate-wide resources, which eliminates long response times for optimization requests, allows for a higher system utilization, and significantly reduces the number of required software licenses and hence cost.

2 Optimization Process

In the area of numerical optimization, several software packages are available that enable the user to apply different algorithms to a specific simulation problem, most notably Optimus [7], the DAKOTA iterator toolkit [6], and Nimrod/O [5]. These tools, however, are primarily designed as universal optimization tools requiring significant changes for the use in electric field specific simulation. In addition, these system are unable to cope with limited resources during the model generation phase, as it is given here due to the CAD modeling. We have therefore decided to design our own optimization environment, which we will describe below.

2.1 Optimization Loop

Our framework consists of three primary components: a parametric CAD modeling system based on a commercially available system, a model evaluation component using field simulation, and the numerical optimization algorithm. The latter is designed to restrict the search space to a small subset for an optimal set of parameters.

The workflow between these components is as follows: Starting with an initial set of parameters, the CAD program generates the first instance of the model that is to be calculated. The model is prepared for the field calculation, i.e. the boundary conditions and dielectrics are assigned and the model is discretized. Using simulation, the quality of the generated model is computed and passed to the optimizer. Combined with the result of all previous simulation runs within this optimization invocation, the optimization algorithm defines a new set of design parameters. These design parameters are again read by the parametric CAD modeler, which creates a new instance of the model to be optimized.

In [4] several minimization algorithms have been investigated for a two dimensional field optimization problem. Based on the results of this study, the Hooke-Jeeves Method [8], the Nelder-Mead Method [10], and the Fletcher-Reeves Algorithm [12] have been used for the optimization process in this work.

This optimization loop is repeated until a termination criterion is reached. The exact criterion depends on the optimization algorithm in use. In any case, it indicates that a local minimum has been found. Due the physical properties of this particular problem, this will be either equivalent to the global minimum or sufficiently close to it.

2.2 Electrical Simulation

In order to facilitate the evaluation of generated models, a simulation environment for three dimensional electric fields in real-world high voltage apparatus is required. From a mathematical point of view, the calculation of fields can be described as the calculation of the potential $\Phi(x, y, z)$ and the electric field strength $\mathbf{E}(x, y, z) = -\nabla\Phi$. This can be achieved by solving Laplace's differential equation $\Delta\Phi = 0$. For electrostatic fields this differential equation can be solved by

$$\Phi(\mathbf{r}_p) = \frac{1}{4\pi\epsilon} \iiint \frac{\rho(\mathbf{r}_q)}{|\mathbf{r}_p - \mathbf{r}_q|} dV \quad (1)$$

with $\mathbf{r}_p$ being the radius vector in the respective point of interest and $\mathbf{r}_q$ being the radius vector in the integration point.

To solve these problems, Boundary Element Methods [3] have proven to be very efficient. They reduce the problem to a system of algebraic equations by discretizing the model's surfaces with well selected, small curvilinear patches (boundary elements) on the interfaces between media with different material characteristics. Over a boundary element, the field is expressed as an analytical interpolation function between the field values at the nodes (element vertices).

Based on these principles, a parallel simulation environment named POLOPT [2] has been developed in cooperation with ABB corporate research and is in production for the design of high-voltage equipment. It uses a master/slave approach for its computation and is implemented using MPI [9] as the parallel programming model. Encouraging results showing a high efficiency have been achieved under production conditions using large-scale input sets [11].

2.3 Implications

The overall optimization process requires repeated access to two kind of resources: compute servers to execute the electric field simulation and CAD servers to perform the model generation. The requests for these two are always interleaved and require both substantial execution time. It is therefore inefficient to allocate both resources throughout the whole runtime of the optimization process; instead both resources should be allocated on demand in order to allow a more efficient execution of concurrent optimization requests.

In addition, in a corporate environment these resources should ideally be shared across the whole company. Each of them is associated with a non-trivial amount of money (the compute servers are high-end clusters and CAD servers are associated with expensive CAD licenses). Such resource sharing allows an optimal utilization of each resource and hence the reduction of the number of required resources. This cuts cost and reduces the number of sites hosting the resources, which allows more centralized system management with a lower total cost of ownership.

3 Software Architecture and Implementation

This kind of corporate-wide resource sharing can be achieved in the form of a grid-like infrastructure. This grid splits into two main components: a) the grid for the compute servers and b) a grid for the CAD servers. As a result, the final system, which we call CAD grid, has to deal with two orthogonal sets of resource restrictions: the availability of a set of specific machines required for the simulation and the availability of CAD licenses (the CAD programs itself run on standard workstations).

3.1 System Structure

To achieve such flexible resource management we have designed a modular system. Its structure is depicted in Figure 1 and consists of the following five major components:

1. Clients, from which jobs get submitted.
2. A set of compute clusters to execute the field simulation, forming the simulation grid.
3. A set of CAD workstations to generate the models depending on varying input parameters, forming the CAD grid.
4. The optimization module executing the optimization algorithm.
5. A central coordinator, which retrieves jobs from the clients and is responsible for the execution.

The latter two components will usually be combined within one system, as the optimization algorithm does not require major compute times. Hence it is not necessary to offload this work to remote server farms analog to the simulation or the CAD grid. Note, that this central component, which forms the cornerstone of our framework, does not impose a major bottleneck, since the execution times of the model generation and the electric field simulations are several orders of magnitude higher than the one consumed by the coordinator.

This coordinator is implemented as a server, which acts as the interface for communication involving the clients. An important consequence of this design is that the client needs no knowledge of the overall architecture of the system. In particular, it need not know the location of the CAD hosts or the simulation clusters. Therefore, changes may be made to the rest of the system without

altering the client's view of it. This ensures the easy-of-use of the overall system, as end-users generally should not be influenced by such configuration changes, and at the same time significantly increases the manageability and the ability to tolerate system faults.

3.2 Workflow

When a job is submitted by a client to the coordinator, all of the CAD files, which specify the model to be analyzed, are sent, along with any design constraints and the initial parameter values as specified by the user. The optimization program is then started and loops until it reaches an acceptable value. The loop consists of the following stages:

1. An available CAD server is selected depending on the availability of a (potentially floating) software license and the capabilities of the target machine. The CAD files and parameter values needed to generate the model are then sent to the chosen CAD server.
2. The CAD server generates the specified model. For this task it uses special API software allowing to connect to the CAD program. In our work, we used the CAD package Pro/Engineer together with the Pro/Toolkit API [1].
3. A file encapsulating the required details of the CAD model is returned to the coordinator.
4. The coordinator selects an available compute cluster within the simulation grid depending on the compute requirements for the chosen task and then sends the file containing the CAD to this system and initiates POLOPT.
5. POLOPT computes the electric field of the given model and computes the objective value used for the optimization (in our case the maximal field value).
6. The objective value is passed back to the coordinator.
7. The design parameters of interest and the objective value are handed to the optimization algorithm. It analyzes the value and, if appropriate, generates a new set of parameters and sends its result to the coordinator.
8. Depending on the result received from the optimization algorithm, the coordinator starts again with step 1) or aborts the loop and returns the objective value together with the final set of parameters and the last CAD model to the client.

After the completion of the iterative optimization process, the client receives the final result in the form of an optimized CAD model. This can then be displayed together with the final field distribution for a final investigation by the design engineer.

3.3 User Interface

In order to ease the use of the system, we have implemented a graphical user interface which provides a straightforward access to the optimization framework. Users can upload their initial files and specify the design parameters, which

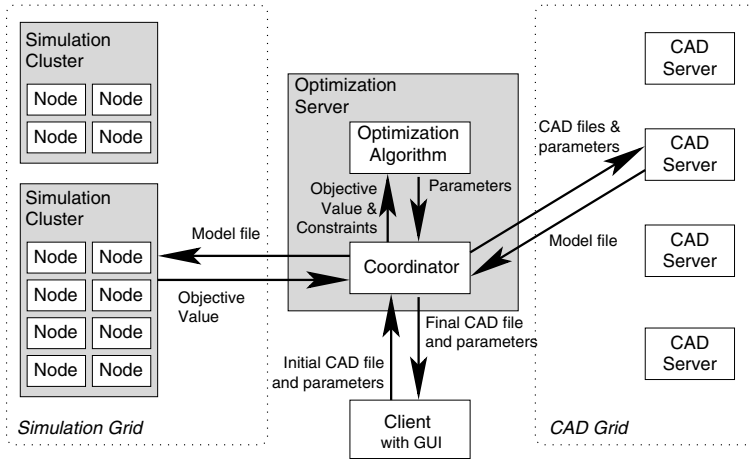


Fig. 1. System Structure

should be investigated by the optimization algorithm. After the job has been submitted, the user can monitor the progress of the optimization, and, after its completion, the final result is displayed. This integrates the optimization framework into the design workflow of the target users, the design engineers, and hence reduces the associated learning curve.

This GUI has been implemented in Java as a standalone client. This decision was made to enable the client to access local data and use this data for uploads, which would have not been possible in applets. This approach, however, maintains the platform independence of the client enabling an easy deployment across a large number of platforms without porting efforts.

4 Technical Evaluation

We have implemented this resource sharing infrastructure in cooperation with ABB corporate research and have deployed it at their site for production use. It currently operates with two PC clusters, each eight nodes, and up to three CAD workstations. Within the compute clusters, the POLOPT environment [11] is used to execute the simulations, while a set of CAD workstations with Pro/Engineer [1] is deployed to generate the model files. First experience shows a positive feedback and high acceptance rate among the design engineers. The unified GUI effectively hides the complexity of the optimization from the user and provides a clean interface, significantly lowering the learning curve. In addition, by its implementation in Java, it can be used from arbitrary workstations, further simplifying its application and deployment.

The availability of this unifying infrastructure provides the design engineers with a straightforward access to corporate-wide resources and enables the com-

putation of complex optimization problems. This is illustrated in the following based on a sample model. Figure 2 shows eight design parameters of a transformer output lead shielding electrodes for which an optimal set with regard to the maximum field strength had to be found. The goal is the minimization of the overall space requirements by this component, while staying within a safe range in terms of possible flashovers.

This example has been optimized using the optimization algorithms discussed above. The numerical changes of the eight design parameter during the optimization process can be seen in Table 1.

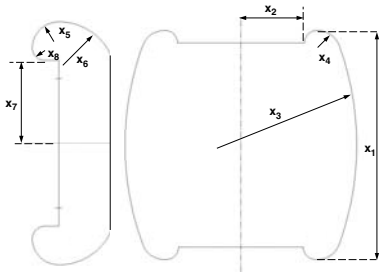


Fig. 2. Design parameters for the transformer

Table 1. Change of design parameters during optimization

Parameter	Initial Value	Optimal Value
x_1 mm	420.0	438.3
x_2 mm	113.0	17.3
x_3 mm	531.0	237.0
x_4 mm	42.0	135.1
x_5 mm	32.0	147.0
x_6 mm	113.0	236.5
x_7 mm	165.0	56.2
x_8 mm	25.0	60.7

5 Business Perspective

The ability to compute these optimization processes is invaluable to companies like ABB. They allow a predictable assessment of properties of their products and enable significant design improvements without having to build expensive and time-consuming prototypes. This leads to more competitive products with respect to both price, due to reduced engineering cost and development time, and quality, due to highly optimized systems. The exact impact can hardly be quantified, but can be assumed to be substantial considering that most components in this product area (e.g., large scale transformer and switching units) are custom designs without the ability to mass produce and can be in price ranges well beyond millions of dollars.

6 Conclusions

The optimization of the geometry is an integral part of the design process of high voltage components. This is done iteratively by repeated simulations of the electric field and an adjustment of the geometric model based on the simulation results. Combined with optimization algorithms, which are capable of minimizing a chosen objective value for a given set of parameters, this can be used to automatically compute an optimal set of design parameters. We have presented a framework, which implements such an optimization process. It is designed to

efficiently leverage corporate wide resource and to allow an interleaving of several concurrent optimization requests. It thereby distinguishes between two different sets of resources — compute servers to run the electric field simulations and CAD servers to perform the model generation based on a given set of parameters — and ensures the efficient utilization of both.

This framework has been implemented in a real industrial environment and is already in production use. It is highly accepted by the end users, the design engineers, due to its easy-to-learn interface and clean integration into their overall workflow. It has already successfully been deployed to optimize the design of high voltage components. In the example shown in this work, it has lead to significant optimizations in the geometric design and to a reduction of the electric field by over 25%.

References

1. Web Page – URL: <http://www.ptc.com>.
2. Z. Andjelic. *POLOPT 5.3 User's Guide*. Internal document, Asea Brown Boveri Corporate Research, Heidelberg, 1999.
3. R. Bausinger and G. Kuhn. *Die Boundary-Element Methode (In German)*. Expert Verlag, Ehingen, 1987.
4. C. Trinitis, H. Steinbigler, M. Spasojevic, P. Levin, and Z. Andjelic. *Accelerated 3-D optimization of High Voltage Apparatus*. Conference Proceedings of the 9th Int. Symposium on High Voltage Engineering, paper 8867, Graz, 1995.
5. L. Kotler D. Abramson, J. Giddy. High performace parametric modelling with nimrod/g:ller application for the global grid? *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000. pp 520–528, Cancun, Mexico.
6. M.S. Eldred and W.E. Hart. Design and implementation of multilevel parallel optimization on the intel teraflops. *Seventh AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, September 1998. pages 44-54, St. Louis, MO, AIAA-98-4707.
7. P. Guisset and N. Tzannetakis. Numerical methods for modeling and optimization of noise emission applications. *ASME Symposium in Acoustics and Noise Control Software*, November 1997. ASME International Mechanical Engineering Congress and Exposition, Dallas, TX.
8. R. Hooke and T.A. Jeeves. Direct Search Solution of numerical and statistical Problems. In *Journal of Ass. of Comp.*, vol. 8, pages 212–229, 1961.
9. Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. <http://www.mpi-forum.org/>.
10. J.A. Nelder and T. Mead. A simplex method for function minimization. In *Computer Journal* 7, pages 308–313, 1965.
11. C. Trinitis, M. Schulz, and W. Karl. *A Comprehensive Electric Field Simulation Environment on Top of SCI*. Lecture notes in Computer Science 2474, EuroPVM'2002, Springer Verlag, pp. 114–121, 2002.
12. G.N. Vanderplaaats. *Numerical Optimization Techniques for Engineering Design*. Mc-Gaw Hill Book Company, New York, 1994.

Cache Performance Optimizations for Parallel Lattice Boltzmann Codes

Jens Wilke, Thomas Pohl, Markus Kowarschik, and Ulrich Ruede

Lehrstuhl für Systemsimulation (Informatik 10)
Institut für Informatik

Friedrich–Alexander–Universität Erlangen–Nürnberg, Germany
{Jens.Wilke,Thomas.Pohl,Markus.Kowarschik,Ulrich.Ruede}@cs.fau.de

Abstract. When designing and implementing highly efficient scientific applications for parallel computers such as clusters of workstations, it is inevitable to consider and to optimize the single-CPU performance of the codes. For this purpose, it is particularly important that the codes respect the hierarchical memory designs that computer architects employ in order to hide the effects of the growing gap between CPU performance and main memory speed. In this paper, we present techniques to enhance the single-CPU efficiency of lattice Boltzmann methods which are commonly used in computational fluid dynamics. We show various performance results to emphasize the effectiveness of our optimization techniques.

1 Introduction

In order to enhance the performance of any parallel scientific application, it is important to focus on two related optimization issues. Firstly, it is necessary to minimize the parallelization overhead itself. These efforts commonly target the choice of appropriate load balancing strategies as well as the minimization of communication overhead by hiding network latency and bandwidth. Secondly, it is necessary to exploit the individual parallel resources as efficiently as possible; e.g., by achieving as much performance as possible on each CPU in the parallel environment. This is especially true for distributed memory systems found in computer clusters based on off-the-shelf workstations communicating via fast networks. Our current research focuses on this second optimization issue.

In order to mitigate the effects of the growing gap between theoretically available processor speed and main memory performance, today's computer architectures are typically based on *hierarchical memory designs*, involving CPU registers, several levels of cache memories (caches), and main memory [10]. Remote main memory and external memory (e.g., hard disk drives) can be considered as the slowest components in any memory hierarchy. Fig. 1 illustrates the memory architecture of a current high performance workstation [12].

Efficient execution in terms of work units per second can only be obtained if the codes exploit the underlying memory design. This is particularly true for numerically intensive codes. Unfortunately, current compilers cannot perform

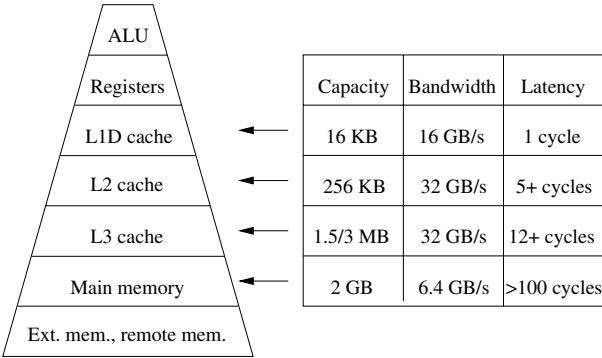


Fig. 1. Memory architecture of a workstation based on an Intel Itanium2 CPU with three levels of on-chip cache [12].

highly sophisticated code transformations automatically. Much of this optimization effort is therefore left to the programmer [8,14].

Generally speaking, efficient parallelization and cache performance tuning can both be interpreted as *data locality optimizations*. The underlying idea is to keep the data to be processed as close as possible to the corresponding ALU. From this viewpoint, cache optimizations form an extension of classical parallelization efforts.

Research has shown that the cache utilization of iterative algorithms for the numerical solution of linear systems can be improved significantly by applying suitable combinations of *data layout optimizations* and *data access optimizations* [3,6]. The idea behind these techniques is to enhance the *spatial locality* as well as the *temporal locality* of the code [11]. Similar work focuses on other algorithms of numerical linear algebra [16] and hardware-oriented FFT implementations [7]. An overview of cache optimization techniques for numerical algorithms can be found in [13]. Our current work concentrates on improving the cache utilization of a parallel implementation of the *lattice Boltzmann method (LBM)*, which represents a particle-based approach towards the numerical simulation of problems in computational fluid dynamics (CFD) [5,17].

This paper is structured as follows. Section 2 contains a brief introduction to the LBM. Section 3 presents code transformation techniques to enhance the single-CPU performance of the LBM and introduces a *compressed grid storage* technique, which almost halves its data set size. The performance results of LBM implementations on various platforms are shown in Section 4. We conclude in Section 5.

2 The Lattice Boltzmann Method

It is important to mention up front that we only give a brief description of the LBM in this section, since the actual physics behind this approach are not essential for the application of our optimization techniques.

The usual approach towards solving CFD problems is based on the numerical solution of the governing partial differential equations, particularly the Navier–Stokes equations. The idea behind this approach is to discretize the computational domain using finite differences, finite elements or finite volumes, to derive algebraic systems of equations and to solve these systems numerically.

In contrast, the LBM is a particle-oriented technique, which is based on a microscopic model of the moving fluid particles. Using $\mathbf{x}$ to denote the position in space, $\mathbf{u}$ to denote the particle velocity, and t to denote the time parameter, the so-called *particle distribution function* $f(\mathbf{x}, \mathbf{u}, t)$ is discretized in space, velocity, and time. This results in the computational domain being regularly divided into cells (so-called *lattice sites*), where the current state of each lattice site is defined by an array of floating-point numbers that represent the distribution functions w.r.t. to the discrete directions of velocity.

The LBM then works as follows. In each time step, the entire grid (lattice) is traversed, and the distribution function values at each site are updated according to the states of its neighboring sites in the grid at the previous discrete point in time. This update step consists of a *stream* operation, where the corresponding data from the neighboring sites are retrieved, and a *collide* operation, where the new distribution function values at the current site are computed according to a suitable model of the microscopic behavior of the fluid particles, preserving hydrodynamic quantities such as mass density, momentum density, and energy. Usually, this model is derived from the *Boltzmann equation*

$$\frac{\partial f}{\partial t} + \langle \mathbf{u}, \nabla f \rangle = \frac{1}{\lambda} \left(f - f^{(0)} \right) ,$$

which describes the time-dependent behavior of the particle distribution function f . In this equation, $f^{(0)}$ denotes the equilibrium distribution function, λ is the relaxation time, $\langle \cdot, \cdot \rangle$ denotes the standard inner product, and ∇f denotes the gradient of f w.r.t. the spatial dimensions [17].

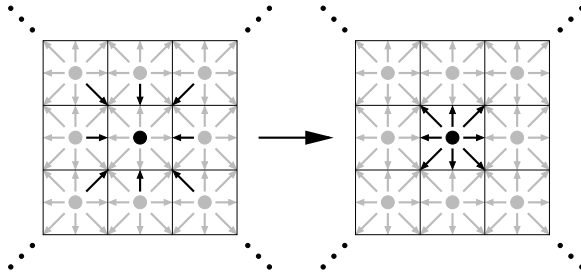


Fig. 2. Representation of the LBM updating a single lattice site by a stream operation (left) and a subsequent collide operation (right).

Fig. 2 shows the update of an individual lattice site in 2D. The grid on the left illustrates the source grid corresponding to the previous point in time, the

darker arrows represent the particle distribution function values being read. The grid on the right illustrates the destination grid corresponding to the current point in time, the dark arrows represent the new particle distribution function values; i.e., after the collide operation.

Note that, in every time step, the lattice may be traversed in any order since the LBM only accesses data corresponding to the previous point in time in order to compute new distribution function values. From an abstract point of view, the structure of the LBM thus parallels the structure of an implementation of Jacobi's method for the iterative solution of linear systems on structured meshes. The time loop in the LBM corresponds to the iteration loop in Jacobi's method.

In contrast to the lattice gas approach [17] where hexagonal grids are more common, we focus on orthogonal grids, since they are almost exclusively used for the LBM.

3 Optimization Techniques for Lattice Boltzmann Codes

3.1 Fusing the Stream and the Collide Operation

A naive implementation of the LBM would perform two entire sweeps over the whole data set in every time step: one sweep for the stream operation, copying the distribution function values from each lattice site into its neighboring sites, and a subsequent sweep for the collide operation, calculating the new distribution function values at each site.

A first step to improve performance is to combine the streaming and the collision step. The idea behind this so-called *loop fusion* technique is to enhance the temporal locality of the code and thus the utilization of the cache [1]. Instead of passing through the data set twice per time step, the fused version retrieves the required data from the neighboring cells and immediately calculates the new distribution function values at the current site, see again Fig. 2. Since this tuning step is both common and necessary for all subsequent transformations, we consider this fused version as our starting point for further cache optimizations.

3.2 Data Layout Optimizations

Accessing main memory is very costly compared to even the lowest cache level. Therefore, it is essential to choose a memory layout for the implementation of the LBM which allows the code to exploit the benefits of the hierarchical memory architecture.

Since we need to maintain the grid data for any two successive points in time, our initial storage scheme is based on two arrays of records. Each of these records stores the distribution function values of an individual lattice site. Clustering the distribution function values is a reasonable approach since the smallest unit of data to be moved between main memory and cache is a *cache block* which typically contains several data items that are located adjacent in memory.

In the following, we present two data layout transformations that aim at further enhancing the spatial locality of the LBM implementation; *grid merging* and *grid compression*.

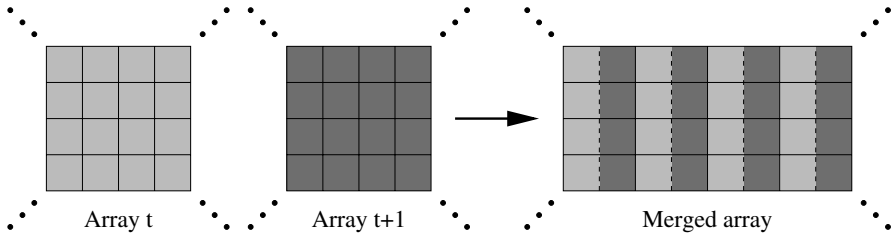


Fig. 3. The two arrays which store the grid data at time t and time $t + 1$, respectively, are merged into a single array.

Grid merging. During a fused stream-and-collide step it is necessary to load data from the grid (array) corresponding to time t , to calculate new distribution function values, and to store these results into the other grid corresponding to time $t + 1$.

Due to our initial data layout (see above) the first two steps access data which are tightly clustered in memory. Storing the results, however, involves the access of memory locations that can be arbitrarily far away. In order to improve spatial locality, we introduce an interleaved data layout where the distribution function values of each individual site for two successive points in time are kept next to each other in memory. This transformation is commonly called *array merging* [11,13]. Fig. 3 illustrates the application of this technique for the 2D case.

Grid compression. The idea behind this data layout is both to save memory and to increase spatial locality. We concentrate on the 2D case in this paper, while the extension of this technique to the 3D case is currently being implemented.

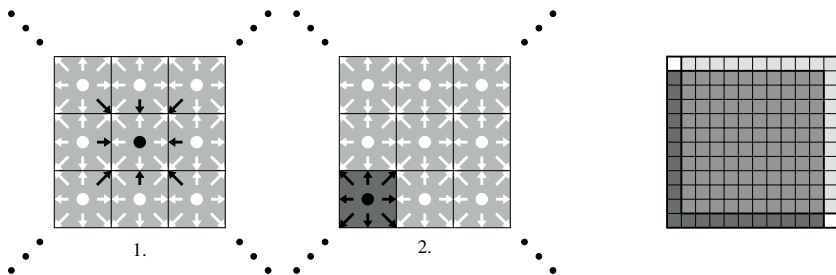


Fig. 4. Fused stream-and-collide step for the cell in the middle (left, 1 and 2), layout of the two overlaid grids after applying the grid compression technique (right).

In an implementation of the LBM in 2D, nearly half of the memory can be saved by exploiting the fact that only the data from the eight neighboring cells are required to calculate the new distribution function values at any regular site

of the grid¹. It is therefore possible to overlay the two grids for both points in time, introducing a diagonal shift of one row and one column of cells into the stream-and-collide operation. The direction of the shift then determines the update sequence in each time step since we may not yet overwrite those distribution function values that are still required henceforth.

In Fig. 4, the light gray area contains the values for the current time t . The two pictures on the left illustrate the stream-and-collide operation for a single cell: the values for time $t + 1$ will be stored with a diagonal shift to the lower left. Therefore, the stream-and-collide sweep must also start with the lower left cell. Consequently, after one complete sweep, the new values (time $t + 1$) are shifted compared to the previous ones (time t). For the subsequent time step, the sweep must start in the upper right corner. The data for time $t + 2$ must then be stored with a shift to the upper right. After two successive sweeps, the memory locations of the distribution functions are the same as before. This alternating scheme is shown in the right picture of Fig. 4.

3.3 Data Access Optimizations

Data access optimizations change the order in which the data are referenced in the course of the computation, while respecting all data dependencies. Our access transformations for implementations of the LBM are based on the *loop blocking* (*loop tiling*) technique. The idea behind this general approach is to divide the iteration space of a loop or a loop nest into blocks and to perform as much computational work as possible on each individual block before moving on to the next block. If the size of the blocks is chosen according to the cache size, loop blocking can significantly enhance cache utilization and, as a consequence, yield significant performance speedups [1,8,13].

In the following, we present two blocking approaches in order to increase the temporal locality of the 2D LBM implementation. Both blocking techniques take advantage of the stencil memory access exhibited by the LBM. Because of this local operation, a grid site can be updated to time $t + 1$ as soon as the sites it depends on have been updated to time t .

It is important to point out that each of these access transformations can be combined with either of the two layout transformations which we have introduced in Section 3.2.

1D blocking. Fig. 5 illustrates an example, where two successive time steps are blocked into a single pass through the grid. White cells have been updated to time t , light gray cells to time $t + 1$, and dark gray cells to time $t + 2$. It can be seen that in Grid 2, all data dependencies of the bottom row are fulfilled, and can therefore be updated to time $t + 2$, shown in Grid 3. This is performed repeatedly until the entire grid has been updated to time $t + 2$, see Grids 4 to 10. The downside to this method is that, even two rows may contain too much

¹ For the sake of simplicity, we focus on regular sites and omit the description of how to treat boundary sites and obstacle sites separately.

data to fit into cache if the grid is too large. In this case, no performance gain will be observed.

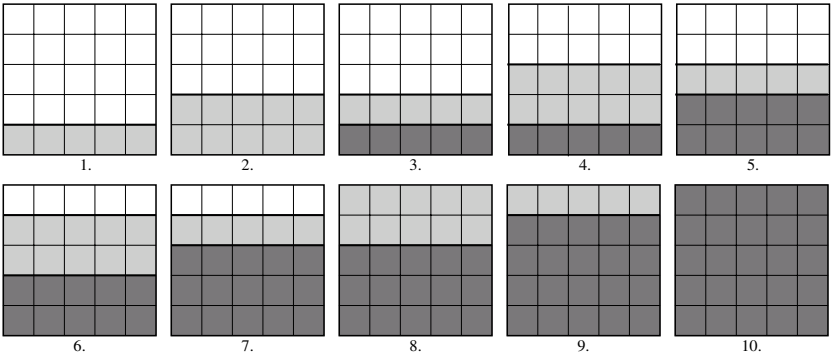


Fig. 5. 1D blocking technique to enhance temporal locality.

2D blocking. Fig. 6 illustrates an example, in which a 4×4 block of cells is employed. This means that four successive time steps are performed during a single pass through the grid. Since the data contained in the 2D block is independent of the grid size, it will always (if the size is chosen appropriately) fit into the highest possible cache level, regardless of the grid size. In Fig. 6, Grids 1 to 4 demonstrate the handling of one 4×4 block, and Grids 5 to 8 a second 4×4 block. The block which can be processed moves diagonally down and left in order to avoid violating data dependencies. Obviously, special handling is required for those sites near grid edges which cannot form a complete 4×4 block.

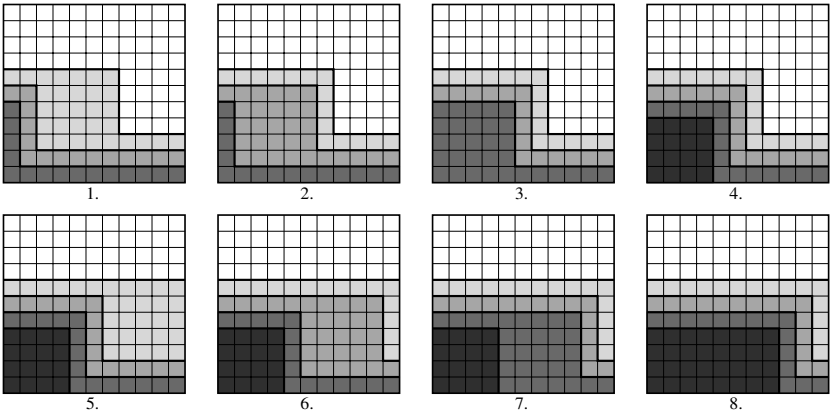


Fig. 6. 2D blocking technique to enhance temporal locality.

4 Performance Results

In order to test and benchmark the various implementations of the LBM, a well known problem in fluid dynamics known as the *lid-driven cavity* has been used. It consists of a closed box, where the top of the box, the lid, is continually dragged across the fluid in the same direction. The fluid eventually forms a circular flow around the center of the box [9].

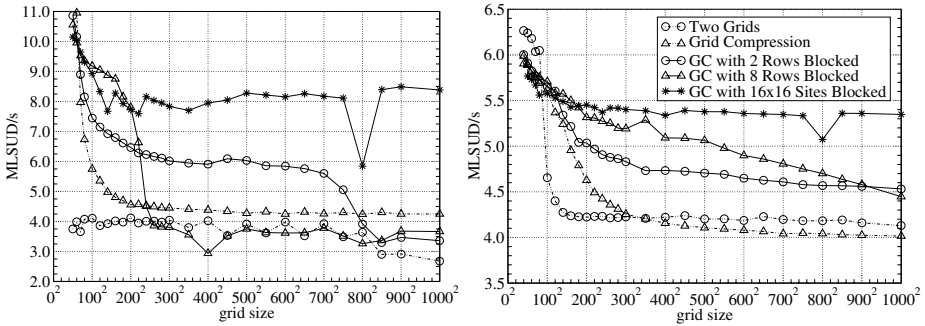


Fig. 7. Performance in millions of lattice site updates per second (MLSUD/s) on machines based on an AMD Athlon XP 2400+ (left) and on an Intel Itanium2 (right), respectively, for various grid sizes.

Performance. Fig. 7 demonstrates the performance of five different implementations of the LBM in ANSI C++ on two different machines² for various grid sizes. Note that problem size n means that the grid contains n^2 cells. The simple implementation involving a source and destination grid is the slowest, the implementation using the layout based on grid compression is slightly faster. Two implementations of 1D blocking are shown, one with two time steps blocked, and one with eight. Initially, they show a significant increase in performance. For large grids, however, performance worsens since the required data cannot fit into even the lowest level of cache. This effect is more dramatic on the AMD processor since it has only 256 kB of L2 and no L3 cache, whereas the Intel CPU even has 1.5 MB of L3 cache on-chip. Finally, the 2D blocked implementation shows a significant increase in performance for all grid sizes. It should be noted that each implementation based on the grid merging layout performs worse than its counterpart based on grid compression. Therefore, no performance results for the grid merging technique are shown.

We have obtained similar performance gains on several further platforms. For example, our results include speedup factors of 2–3 on machines based on DEC

² We use an AMD Athlon XP 2400+ based PC (2 GHz) [2], Linux, gcc 3.2.1, as well as an Intel Itanium2 based HP zx6000 [12], Linux, Intel ecc V7.0. Aggressive compiler optimizations have been enabled in all experiments.

Alpha 21164 and DEC Alpha 21264 CPUs. Both of them particularly benefit from large off-chip caches of 4 MB.

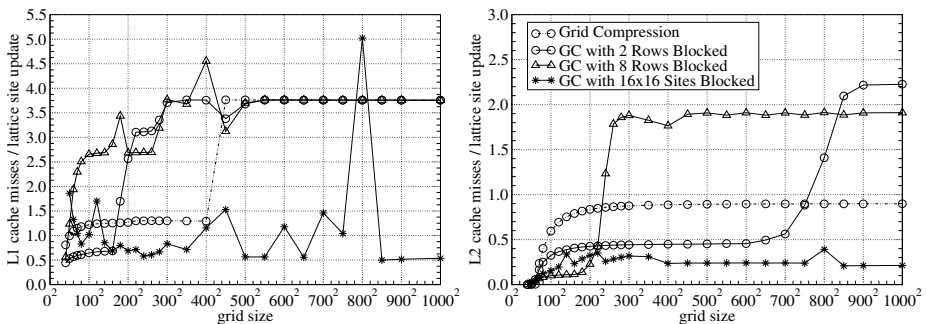


Fig. 8. Cache behavior of the AMD Athlon XP measured with PAPI.

Cache Behavior. Fig. 8 demonstrates the behavior of the L1 and L2 cache of the AMD Athlon XP for the different implementations of the LBM. These results have been obtained by using the profiling tool *PAPI* [4]. When compared to Fig. 7 it can be seen that there is a strong correlation between the number of cache misses and the performance of the code. The correlation between the performance drop of the two 1D blocked implementations and their respective rise in L2 cache misses is especially dramatic. Additionally, Fig. 8 reveals the cause of the severe performance drop exhibited by the 2D blocked implementation at a grid size of 800^2 . The high number of cache misses are *conflict misses* which are caused when large amounts data in memory are mapped to only a small number of cache lines [15].

5 Conclusions and Future Work

Due to the still widening gap between CPU and memory speed, hierarchical memory architectures will continue to be a promising optimization target. The CPU manufacturers have already announced new generations of CPUs with several megabytes of on-chip cache in order to hide the slow access to main memory.

We have demonstrated the importance of considering the single-CPU performance before using parallel computing methods in the framework of a CFD code based on the LBM. By exploiting the benefits of hierarchical memory architectures of current CPUs, we have obtained factors of 2–3 in performance on various machines. Unfortunately, the parallelization of cache-optimized codes is commonly tedious and error-prone due to their implementation complexity.

We are currently extending our techniques to the 3D case. From our experience in cache performance optimization of iterative linear solvers, we expect that appropriate data layout transformations and data access optimizations, in particular loop blocking, can be applied to improve the performance.

Acknowledgments. We wish to thank the members of the High Performance Computing Group at the Computing Center in Erlangen (*RRZE*) for their support.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2001.
2. AMD Corporation. *AMD Athlon XP Processor 8 Data Sheet*, 2002. Publication #25175 Rev. F.
3. F. Basseti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proc. of the Int. Conference on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, NV, USA, 1998.
4. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
5. S. Chen and G.D. Doolen. Lattice Boltzmann Method for Fluid Flow. *Annual Reviews of Fluid Mechanics*, 30:329–364, 1998.
6. C.C. Douglas, J. Hu, M. Kowarschik, U. Rde, and C. Wei. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.
7. M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. of the Int. Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, USA, 1998.
8. S. Goedecker and A. Hoesie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.
9. M. Griebel, T. Dornseifer, and T. Neunhoeffler. *Numerical Simulation in Fluid Dynamics*. SIAM, 1998.
10. J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.
11. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, second edition, 1996.
12. Intel Corporation. *Intel Itanium2 Processor Reference Manual*, 2002. Document Number: 251110–001.
13. M. Kowarschik and C. Wei. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.
14. D. Loshin. *Efficient Memory Programming*. McGraw–Hill, 1998.
15. G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.
16. R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. of the ACM/IEEE Supercomputing Conference*, Orlando, FL, USA, 1998.
17. D.A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.

Effectiveness of Parallelizing the ILOG-CPLEX Mixed Integer Optimizer in the PUBB2 Framework^{*}

Yuji Shinano¹, Tetsuya Fujie², and Yuusuke Kounoike¹

¹ Department of Computer, Information and Communication Sciences,
Tokyo University of Agriculture and Technology,
2-24-16, Naka-cho, Koganei-shi, Tokyo 184-8588, Japan
yshinano@cc.tuat.ac.jp, kounoike@al.cs.tuat.ac.jp

² Department of Management Science, Kobe University of Commerce,
8-2-1, Gakuen-nishimachi, Nishi-ku, Kobe 651-2197, Japan
fujie@kobeuc.ac.jp

Abstract. In this paper, we introduce a new method of parallelizing a MIP (Mixed Integer Programming) solver. This method is different from a standard implementation that constructs a parallel branch-and-cut algorithm from scratch (except using an LP solver). The MIP solver we use is ILOG-CPLEX MIP Optimizer (Version 8.0), which is one of the most efficient implementations of branch-and-cut algorithms. The parallelization of the solver is performed by using the software tool PUBB2 developed by the authors. We report a part of our computational experience using up to 24 processors. In addition, we point out some problems that should be resolved for a more efficient parallelization.

1 Introduction

A mixed integer programming problem (MIP) is a linear programming problem (LP) with a restriction that some or all of the decision variables must be integer. MIP has a wide variety of applications in scheduling, planning and other practical problems. MIP has been considered as one of the most difficult discrete optimization problems. However, recent remarkable advancements of branch-and-cut (B&C) algorithms, especially for traveling salesman problems[1] and other optimization problems, stimulate research interest on MIPs. The B&C algorithm is a branch-and-bound (B&B) algorithm in which each subproblem is evaluated by the cutting plane algorithm. We may point out several factors in the recent advancements of the B&C algorithm. The first is a development of fast and robust LP solvers. Fast LP solvers are important since most of computational time of the B&C algorithm is devoted to solving LP relaxation problems, and robust LP solvers enable us to implement involved strategies in the B&C algorithm. The second is that many classes of cutting planes are incorporated

^{*} This work was partially supported by MEXT in Japan through Grants-in-Aid (13680511).

into the B&C algorithms as a consequence of the fast and robust LP solvers. Until recently, Gomory cuts, proposed in the 1950's and the 1960's, have been considered less effective in practice. However, the recent advancement of the LP solvers make them work in the B&C algorithm. The last is that many effective strategies are incorporated in many parts of the B&C algorithm.

There are several parallel implementations of the B&C algorithm in the literature[2,3, etc.]. Recently, general frameworks of the parallel B&C algorithm have been developed, such as SYMPHONY[8], PICO[4] and COIN/BCP[9]. In this paper, we demonstrate a parallelization of a MIP solver itself, which is different from a construction of a parallel B&C algorithm from scratch (except using an LP solver). The MIP solver used in the study is ILOG-CPLEX MIP Optimizer Version 8.0[5], and is parallelized by using the software tool PUBB2 developed by the authors[11]. Motivation of this work comes from the following facts :

- (i) PUBB2 clearly separates the user specific parts from the common procedures of B&B algorithms, which means that a sequential implementation is enough for the users to execute both sequential and parallel B&B algorithms. PUBB2 further separates the computing platform, and it enables a parallelization, for example, via the computational GRID. The parallelization of a MIP solver is a part of our PUBB2 project, considering a MIP solver as the user specific implementation.
- (ii) More importantly, MIP solvers have been greatly improved so that middle and even large-scale problem instances become tractable. Hence, parallelizing MIP solvers can be very significant. In addition, the ILOG-CPLEX MIP Optimizer provides callback functions and classes, which enables us to parallelize such an efficient MIP solver in the PUBB2 framework.

The purpose of the paper is to examine how practical our parallelization is and to find problems that should be resolved in order to achieve a more efficient parallelization. The remainder of the paper is organized as follows. Section 2 presents a brief review of the B&C algorithm. Section 3 gives an outline of PUBB2 and describes the parallelization of the ILOG-CPLEX MIP Optimizer using PUBB2. Computational results are reported in Section 4. Finally, we describe concluding remarks in Section 5.

2 Branch-and-Cut Algorithm for MIPs

In this section, we briefly review the B&C algorithm. See [6,7,12] for more comprehensive treatment of the algorithm. Let us consider the MIP of the following form :

$$(\text{MIP}) \quad \text{Minimize } c^T x \text{ subject to } Ax \leq b, \ x_j : \text{integer } (j = 1, \dots, p),$$

where c is a n -dimensional vector, b is an m -dimensional vector, A is an $m \times n$ matrix and $p \leq n$. If $p = n$ then (MIP) is called a pure integer programming problem. A 0-1 MIP problem is referred to as (MIP) imposed on the additional

constraints $0 \leq x_j \leq 1$ ($j = 1, \dots, p$). An LP relaxation of (MIP) is obtained by dropping the integrality constraint, i.e.,

$$(\text{LP}) \quad \text{Minimize } c^T x \quad \text{subject to } Ax \leq b.$$

In a standard (LP-based) *B&B algorithm*, each subproblem is evaluated by solving its LP relaxation problem.

A *cutting plane algorithm* is another approach to the MIP. Let $\bar{x}$ be an optimal solution of (LP). Then, if an inequality $a^T x \leq b$ were found, which does not delete any feasible solutions of (MIP) but delete $\bar{x}$ (i.e., $a^T \bar{x} > b$), then the LP relaxation (LP) can be tightened by adding that inequality as an additional constraint. Such an inequality is called a *cut*, and the cutting plane algorithm iteratively add cuts to (LP) and then solve the updated LP relaxation problem.

A *B&C algorithm* is a B&B algorithm in which each subproblem is evaluated by the cutting plane algorithm. Cuts are classified into global and local. A cut that is generated by a subproblem evaluation is called *global* if it does not delete any feasible solutions of (MIP), and *local* otherwise. Global cuts are stored in a cut pool, and they are used in subsequent subproblem evaluations.

In summary, a prototype of the B&C algorithm is described as in Figure 1, which leaves some implementation flexibility. Corresponding to items (a), ..., (e) in Figure 1, the following strategies are particularly important in implementing an efficient B&C algorithm:

- (a) **Preprocessing.** Before performing the B&C or the B&B algorithm, preprocessing may be applied, in which redundant variables and constraints are deleted, lower and upper bounds of variables are tightened, coefficients of the constraints are modified to tighten the LP relaxation problem, and so on.
- (b) **Node selection.** Node selection is a common topic of the B&B algorithms. In PUBB2, the depth-first search, the best bound search, and the search based on a user-defined priority are available.
- (c) **Cutting planes.** There are several cuts that can be applied to MIPs such as the mixed integer Gomory cut for general MIPs, the fractional Gomory cut for pure IPs, the lift-and-project cut for 0-1 MIPs, etc. In addition, if the constraints of the MIP contain a structured constraint, then cuts valid for the constraint is available. For example, the (lifted) cover inequality for the knapsack constraints and the clique inequalities for the set packing constraints are known.
- (d) **Heuristics.** In a standard B&C or B&B algorithm for the MIP, an improved solution is found when an optimal solution of an LP relaxation problem is feasible to the MIP. In addition to this, recent solvers involve heuristics based on LPs.
- (e) **Variable selection.** Let $\bar{x}$ be an optimal solution of an LP relaxation problem of a subproblem S . Then branching is done by finding a variable with a fractional value $\bar{x}_j$ and creating two subproblems which are obtained from S by adding the new inequalities $x_j \leq \lfloor \bar{x}_j \rfloor$ and $x_j \geq \lceil \bar{x}_j \rceil$, respectively.

```

Branch-and-Cut Algorithm;
begin
  {Initialization Phase: See an explanation of the class InitData}
(a) Apply preprocessing to the problem (MIP);
    $z^* := \infty$ ;
   Put the root problem into the subproblem pool;
   Let the cut pool be empty;
   {Search Phase: See an explanation of the class SolverBase}
   while the subproblem pool is nonempty do begin
(b)  Choose a subproblem  $S$  from the subproblem pool;
      Delete  $S$  from the subproblem pool;
      Add (globally valid) inequalities in the cut pool to  $S$ ;
      {Evaluation Phase}
(c)  Execute a cutting plane algorithm;
(d)  if improved solution has been found then
      Update  $x^*$  and  $z^*$ ;
(e)  if new subproblems have been generated then
      Put them into the subproblem pool
   end;
   return  $x^*$ 
end.

```

Fig. 1. A prototype of the branch-and-cut algorithm

There are several selection rules including the maximum infeasibility rule, the pseudo-cost rule and the strong branching rule.

See [5] for the strategies incorporated in the ILOG-CPLEX MIP Optimizer Version 8.0.

3 Parallelization in the PUBB2 Framework

PUBB2[11] is a redesigned version of the software tool PUBB (Parallelization Utility for Branch-and-Bound algorithms, see [10]). In particular, the layer structure (see Figure 2) is improved so that the Provider layer for computing platforms is separated from the core layer. PUBB2 primarily consists of three layers: the User, the PUBB2 core, and the Provider layers. A B&B algorithm is completed by establishing connections between the User and the PUBB2 core layers and between the Provider and the PUBB2 core layers. PUBB2 is written in C++ and the connections are achieved by implementing derived classes of the abstract classes in the PUBB2 core layers. For a detailed description of PUBB2, see [11].

Our parallel implementation of the B&C algorithm is performed by embedding the ILOG-CPLEX MIP Optimizer into the User layer. For the interaction between the User and the PUBB2 core layers, four abstract classes are provided in the PUBB2 core building block:

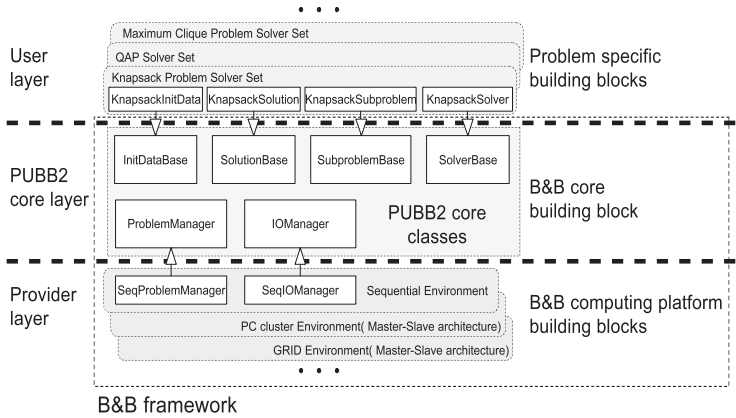


Fig. 2. Layers in the PUBB2 Architecture (quoted from [11])

- InitData:** An abstraction of data for an initialization of a B&B algorithm. The Initialization Phase in Figure 1 is described in a class derived from **InitData**.
- SubproblemBase:** An abstraction of a subproblem representation.
- SolutionBase:** An abstraction of a solution representation.
- SolverBase:** An abstraction of a search phase of the B&B algorithm. The Search Phase in Figure 1 is described in a class derived from **SolverBase**.

In this paper, an implementation of a new and quite simple parallelization strategy is used, named MPMS (Master-Pool Master-Slave), which was proposed by the authors [11] and was implemented in the PUBB2 framework. MPMS concerns with an implementation of the Provider layer. The following two abstract classes are provided in the PUBB2 core building block for interaction between the Provider and the PUBB2 core layers :

- ProblemManager:** An abstraction of the problem-solving manager. The B&B parallelization algorithm is encapsulated in this class, represented by the instances of the classes derived from this core class. These instances may be mapped over several tasks, processes or threads in parallelizations. Each task/process may have its own *pool*, which is a collection of unprocessed subproblems.
- IOManager:** An abstract of I/O and encapsulated I/O operations on a specific computing platform.

Figure 3 shows the construction of master and slave tasks, the pool allocation, the DerivedSolver linkage, the task mapping and the interaction between these tasks. The load-balancing strategy is basically the same as a hierarchical Master-Slave load balancing strategy. As long as a local pool in the MasterProblemManager is not empty and an idle DerivedSolver instance exists, a subproblem is assigned to that instance. However, in this paper, the parallelization strategy of MPMS

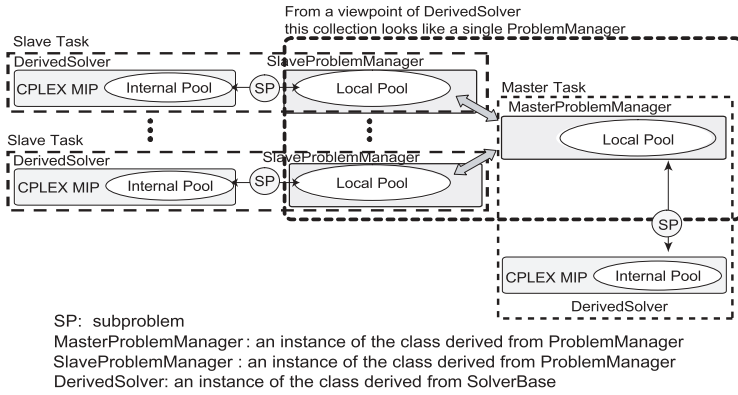


Fig. 3. Pool allocation, DerivedSolver linkage and task mapping

is not so important, because load balancing is controlled from the User layer. PUBB2 has such a mechanism.

In the instance of the DerivedSolver, a subproblem is received from the ProblemManager and is solved by the ILOG-CPLEX MIP Optimizer. The Optimizer itself performs the B&C algorithm and it stores subproblems in its internal pool. Since the internal pool is maintained and updated efficiently, we are better off performing the algorithm in the Optimizer as long as no idle DerivedSolver instance exists. Load balancing is considered only when an idle solver exists in the system (existence of idle solvers can be recognized through the state of the ProblemManager) and a subproblem is transferred between DerivedSolvers. This transfer is made possible by using public functions `void putSubproblem(SubproblemBase *)` and `void push()` of ProblemManager. The former stores a subproblem to a local pool of the ProblemManager and the latter forces the ProblemManager to perform the transfer. These procedures are written in the callbacks of the ILOG-CPLEX MIP Optimizer.

Finally, we describe some remarks concerning our specific implementation.

1. When tasks are loaded, (LP) is solved and its optimal basis is stored. The basis is used when a task receives a new “root” problem from another task, since it may help to reduce the time to solve an LP relaxation problem of the “root” problem.
2. Contrary to a standard Master-Slave, the master task executes the Search Phase too. Then, only when a local pool of the MasterProblemManager is empty, one of the two newly generated subproblems at the branching stage ((e) in Figure 1) is outputted to the local pool.
3. In order to avoid wasteful transfers, every instance of the DerivedSolver is forced not to output subproblems to the ProblemManager when $[\# \text{ of subproblems in the internal pool}] \times [\text{average computing time per subproblem}] < [\text{computing time for an LP relaxation problem of the received “new” subproblem}] + [\text{average computing time per subproblem}] \times [\text{run-time param-}]$

Table 1. Problem statistics

name	constraints	variables		
		total	0-1 integer	general integer
qiu	1192	840	48	0
swath2	884	6805	2406	0
seymour1	4944	1372	451	0
markshare2_1	7	74	54	0
mod011	4480	10958	96	0
fast0507	507	63009	63009	0

ter] is satisfied. For the results presented in the next section, this parameter was set equal to 10.

4 Computational Experience

A PC cluster of 16 PCs was used, each of which contains two Pentium III processors with 1GHz processing speed, 1Gbyte of memory and an Intel Pro/1000 MT LAN desktop adapter. All 16 PCs are connected to a NETGEAR GS 516T Gigabit Switch. PVM3.4.3 was used to implement Provider layer in Figure 2.

H. Mittelmann’s webpage¹ was referenced for problem instances including those from the MIPLIB library. At first, all the problem instances were tested by a default (sequential) MIP solver, and the moderately hard instances were selected for our parallel runs. A part of the results are reported here. In Table 1, statistics of the problem instances are shown. `qiu`, `swath2`, `seymour1`, `markshare2_1` and `mod011` are 0-1 MIP problems and `fast0507` is a pure 0-1 integer programming problem arising from the set cover problem.

Computational results are summarized in Table 2. In the table, **time** denotes the running time in seconds, **speedup** the ratio of the sequential run time to the corresponding parallel run time, **opt. time** the time it takes for an upper bound to reach an optimal solution value, **updates** the number that an incumbent solution (x^* in Figure 1) is updated and **nodes** the total number of subproblems generated during the run. Each value in the table is an average of 5 runs. We set the relative mipgap tolerance (`IloCplex::EpGap`) to 1.0^{-5} and the absolute mipgap tolerance (`IloCplex::EpAGap`) to 1.0^{-6} .

We first remark that preprocessing worked effectively in general. For most of the problem instances not listed in Table 1, preprocessing made the running time quite faster. Preprocessing also worked effectively for our parallel runs.

As the table shows, satisfactory speedups cannot be obtained except a few problem instances. One major reason may be explained by an observation that **speedup** depends on **opt. time**. To explore an another reason for that, upper and lower bounds were drawn as functions of time (Figure 4). The problem instance is `swath2` and the preprocessing is not applied. The running time for the parallel run of 4 tasks is almost the same as that of the sequential run : Comparing Figures 4(a) and 4(b), it may be seen that only one of the 4 tasks essentially

¹ <ftp://plato.asu.edu/pub/milpc.txt>

works. Vertical lines in Figure 4(b) indicate that the transferred “root” problem is solved with few branchings. Hence, many vertical lines reduce the effectiveness of parallelization. For the same reason, it can be seen that the parallel run of 8 tasks is executed effectively and, once again, the running time for the parallel run of 16 tasks is almost the same as that for the parallel run of 8 tasks. Therefore, it may be concluded that a variable selection rule ((e) in Figure 1) for parallel B&C algorithms is an important issue as well as a node selection rule which has been widely discussed for parallel B&B algorithms. Also, the frequency of the cut generation in subproblem evaluations is an issue to be reconsidered for parallel runs. Although reducing the frequency of the cutting plane phase will increase the number of subproblems, it may nevertheless enable efficient parallelizations.

It must also be noted that the global information that the MIP solver maintains is lost in the parallelization. For example, as already described, global cuts are unavailable and information on pseudo-costs is lost. Hence, some additional implementations of global cuts or pseudo-costs may be necessary in order to improve the performance of parallelizing MIP solvers.

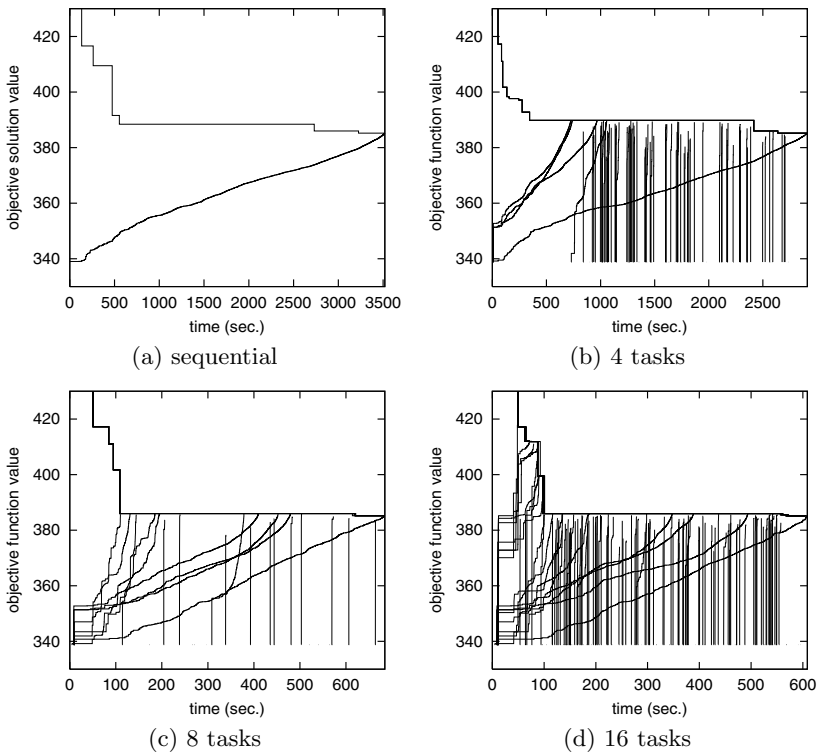


Fig. 4. Upper and lower bounds for the swath2 instance (executed without preprocessing)

Table 2. Computational Results ((p) means the preprocessing is applied.)

name		sequential run	parallel runs (# of tasks)				
			2	4	8	16	24
qiu (p)	time	1412.74	1521.86	916.83	633.12	510.93	1193.95
	speedup	—	0.93	1.54	2.23	2.77	1.18
	opt. time	833.61	400.54	73.35	50.49	34.88	42.36
	updates	9.00	11.00	12.60	17.40	28.00	46.80
	nodes	16004.00	25655.80	27282.20	31102.00	34430.40	64790.80
qiu	time	1211.84	1026.47	1044.98	824.36	57.69	31.33
	speedup	—	1.18	1.16	1.47	21.01	38.68
	opt. time	8.69	6.80	6.83	6.86	6.90	6.84
	updates	2.00	2.00	2.00	2.00	2.00	2.00
	nodes	12278.00	9101.00	9102.20	7281.80	292.20	48.00
swath2 (p)	time	5092.63	2905.16	1777.79	1444.39	1363.40	1397.26
	speedup	—	1.75	2.86	3.53	3.74	3.64
	opt. time	4885.59	1643.92	310.47	221.99	143.40	144.70
	updates	19.00	21.40	15.60	26.00	31.60	54.00
	nodes	258343.00	190872.40	111880.80	102903.60	91657.40	106970.40
swath2	time	2795.99	4781.72	2622.94	705.03	729.78	804.44
	speedup	—	0.58	1.07	3.97	3.83	3.48
	opt. time	2584.40	4304.44	2432.69	623.50	658.06	739.37
	updates	9.00	13.40	25.40	14.40	34.60	78.60
	nodes	67182.00	136040.40	92822.60	45171.80	52419.60	58211.00
seymour1 (p)	time	8605.53	7609.23	4778.17	5517.70	4998.62	3981.05
	speedup	—	1.13	1.80	1.56	1.72	2.16
	opt. time	2291.42	1430.03	1702.87	3066.39	361.95	1030.97
	updates	10.00	11.00	15.80	29.00	36.40	76.20
	nodes	5757.00	9324.80	9324.00	11976.80	15278.20	14306.40
seymour1	time	8501.69	6208.54	4188.94	3669.58	3213.46	3045.53
	speedup	—	1.37	2.03	2.32	2.65	2.79
	opt. time	7854.38	2726.51	2806.45	2650.63	880.57	994.43
	updates	10.00	13.00	15.20	29.60	34.40	74.40
	nodes	6127.00	6659.00	7963.80	8112.00	8670.20	9099.00
markshare2.1 (p)	time	1982.47	2188.45	634.29	229.93	86.76	167.60
	speedup	—	0.91	3.13	8.62	22.85	11.83
	opt. time	1982.20	2186.31	633.10	229.62	86.58	167.37
	updates	23.00	19.60	18.40	21.60	49.60	42.20
	nodes	5848476.00	11975374.20	7107435.60	4739941.60	3815132.80	9556764.60
markshare2.1	time	7954.91	1103.98	386.64	424.74	449.80	142.66
	speedup	—	7.21	20.57	18.73	17.69	55.76
	opt. time	7940.82	1101.67	386.06	423.55	449.07	142.26
	updates	20.00	17.00	18.80	24.60	33.80	49.20
	nodes	22370838.00	5792448.80	4058190.40	8719984.00	17794572.40	8639924.00
mod011 (p)	time	4886.01	3332.80	2143.70	1510.18	1279.63	1896.96
	speedup	—	1.47	2.28	3.24	3.82	2.58
	opt. time	4115.98	2696.17	1812.10	1171.62	209.87	238.16
	updates	13.00	12.00	13.00	22.60	26.40	49.60
	nodes	12806.00	15956.20	17960.60	18501.00	18842.00	22307.80
mod011	time	17314.76	10882.08	8076.57	5717.72	5571.91	5274.62
	speedup	—	1.59	2.14	3.03	3.11	3.28
	opt. time	12031.80	7071.45	2573.09	2648.01	2536.73	2689.66
	updates	9.00	8.00	8.00	16.20	31.20	41.60
	nodes	14398.00	15950.60	17183.80	18351.60	22268.00	22817.20
fast0507 (p)	time	23108.70	24590.12	8079.49	7520.61	4904.10	5098.77
	speedup	—	0.94	2.86	3.07	4.71	4.53
	opt. time	18488.30	22237.51	5242.34	4958.72	2331.63	2396.44
	updates	8.00	8.00	10.20	11.80	21.00	30.80
	nodes	10938.00	19669.00	9418.80	14165.40	10423.60	15301.00
fast0507	time	28592.92	17249.44	12339.97	11786.31	6174.30	5202.93
	speedup	—	1.66	2.32	2.43	4.63	5.50
	opt. time	25042.98	12740.87	8752.64	5814.15	2420.45	2360.70
	updates	9.00	10.00	10.00	12.60	19.60	32.60
	nodes	12685.00	12284.00	12775.80	17994.80	11224.20	13379.40

5 Concluding Remarks

In this paper, a parallelization of the ILOG-CPLEX MIP Optimizer (Version 8.0) in the PUBB2 framework was demonstrated. It was observed that, in order to achieve an efficient parallelization, additional information, which is maintained by the MIP solver and is unavailable via the callback functions and classes, might be necessary. On the other hand, it must be also pointed out that, due to the new parallelization scheme of the B&C algorithm, the load balancing strategy, as well as other strategies of the parallel B&C algorithm, must be studied in more detail. For example, as far as load balancing is concerned, it would be interesting to compare the strategy proposed in this paper to a strategy based on the status of each subproblem in the internal pool, e.g. lower bounds, etc. (see Figure 3), which is currently unavailable via the callback functions and classes.

References

1. Applegate, D., Bixby, R., Chvátal, V. and Cook, W.: On the Solution of Traveling Salesman Problems. *Doc. Math. J. DMV Extra Volume ICM III* (1998) 645–656
2. Bixby, R. E., Cook, W., Cox, A., Lee, E. K.: Computational Experience with Parallel Mixed Integer Programming in a Distributed Environment. *Ann. Oper. Res.* **90** (1999) 19–43
3. Eckstein, J.: Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5. *SIAM J. Optim.* **4** (1998) 794–814
4. Eckstein, J., Phillips, C.A., Hart, W.E.: PICO: An Object-Oriented Framework for Parallel Branch and Bound. *RUTCOR Research Report 40-2000*, Rutgers University (2000)
5. ILOG CPLEX 8.0 Reference Manual, ILOG (2002)
6. Martin, A.: General Mixed Integer Programming: Computational Issues for Branch-and-Cut Algorithms. In: Jünger, M., Naddef, D. (eds.) *Computational Combinatorial Optimization. Lecture Notes in Computer Science*, Vol. 2241. Springer-Verlag, Berlin Heidelberg New York (2001)
7. Nemhauser, G.L., Wolsey, L.A.: *Integer Programming and Combinatorial Optimization*: John Wiley & Sons, New York (1988)
8. Ralphs, T. K., Ladányi, L., Eső, M.: SYMPHONY 3.0 User's Manual. available at <http://www.branchandcut.org/SYMPHONY> (2001)
9. Ralphs, T.K., Ladányi, L.: COIN/BCP User's Manual. available at <http://www-124.ibm.com/developerworks/opensource/coin/documentation.html> (2001)
10. Shinano, Y., Higaki, M., Hirabayashi, R.: A Generalized Utility for Parallel Branch and Bound Algorithms. *Proceedings of the Seventh IEEE Symposium of Parallel and Distributed Processing* (1995) 392–401
11. Shinano, Y., Kounoike, Y., Fujie, T.: PUBB2: A Redesigned Object-Oriented Software Tool for Implementing Parallel and Distributed Branch-and-Bound Algorithms. Working Paper, Department of Computer, Information and Communication Sciences, Tokyo University of Agriculture and Technology (2001)
12. Wolsey, L.A.: *Integer Programming*: John Wiley & Sons, New York (1998)

Improving Performance of Hypermatrix Cholesky Factorization^{*}

José R. Herrero and Juan J. Navarro

Computer Architecture Department, Universitat Politècnica de Catalunya,
Jordi Girona 1–3, Mòdul D6, E-08034 Barcelona, (Spain)
{josepr,juanjo}@ac.upc.es

Abstract. This paper shows how a sparse hypermatrix Cholesky factorization can be improved. This is accomplished by means of efficient codes which operate on very small dense matrices. Different matrix sizes or target platforms may require different codes to obtain good performance. We write a set of codes for each matrix operation using different loop orders and unroll factors. Then, for each matrix size, we automatically compile each code fixing matrix leading dimensions and loop sizes, run the resulting executable and keep its Mflops. The best combination is then used to produce the object introduced in a library. Thus, a routine for each desired matrix size is available from the library. The large overhead incurred by the hypermatrix Cholesky factorization of sparse matrices can therefore be lessened by reducing the block size when those routines are used. Using the routines, e.g. matrix multiplication, in our small matrix library produced important speed-ups in our sparse Cholesky code.

1 Introduction

The Cholesky factorization of a sparse matrix is an important operation in the numerical algorithms field. This paper presents our work on the optimization of the sequential algorithm when a hypermatrix data structure is used.

1.1 Hypermatrix Representation of a Sparse Matrix

Sparse matrices are mostly composed by zeros but often have small dense blocks which have traditionally been exploited in order to improve performance [1]. Our application uses a data structure based on a hypermatrix (HM) scheme [2]. The matrix is partitioned recursively into blocks of different sizes. The HM structure consists of several (N) levels of submatrices. The top $N-1$ levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated as such. Null pointers in pointer matrices indicate that the

^{*} This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

corresponding subblock does not have any non-zero elements and is therefore unnecessary, both for storage and computation. Figure 1 shows a sparse matrix and its corresponding hypermatrix with 2 levels of pointers.

The main potential advantages of a HM structure w.r.t. 1D data structures, like the Compact Row Wise structure, are: the ease of use of multilevel blocks to adapt the computation to the underlying memory hierarchy; and the operation on dense matrices.

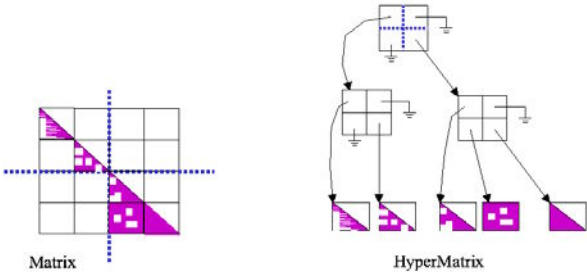


Fig. 1. A sparse matrix and its corresponding hypermatrix.

Choosing a block size for data submatrices is rather difficult. Large block sizes favour greater potential performance when operating on dense matrices. On the other hand, the larger the block is, the more likely it is to contain zeros. Since computation with zeros is useless, effective performance can therefore be low. Thus, a trade-off between performance on dense matrices and operation on non-zeros must be reached. The use of *windows* of non-zero elements within blocks allows for a larger default block size. When blocks are quite full operations performed on them can be rather efficient. However, in those cases where only a few non-zero elements are present in a block, only a subset of the total block is computed. Figure 2 shows a window of non-zero elements within a larger block. The window of non-zero elements is defined by its top-left and bottom right corners. Zeros stored outside those limits are not used in the computations. Null elements within the window are still computed. However, the overhead can be greatly reduced.

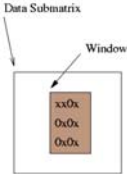


Fig. 2. A data submatrix and a window within it.

A commercial package known as PERMAS uses the hypermatrix structure [3]. It can solve very large systems out-of-core and can work in parallel.

However, the disadvantages of the hypermatrix structure mentioned above introduce a large overhead. Recently a variable size blocking was introduced to save storage and to speed the parallel execution [4]. In this way the HM was adapted to the sparse matrix being factored.

The work presented in this paper is focused on the optimization of a hypermatrix Cholesky factorization based on the data structure presented above. We have developed some rather efficient routines which work on small matrices. The purpose of these routines is to get high performance while keeping low the overhead due to unnecessary computations on null elements.

1.2 Goals

This work focuses on obtaining high performance from a hypermatrix Cholesky factorization. We want to reduce the overhead introduced by operations on zeros when large blocks are used. This can be done by reducing the block size. However, in order to keep high performance, we need specialized routines which operate very efficiently on small matrices.

1.3 Motivation

Figure 3 shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164 processor.

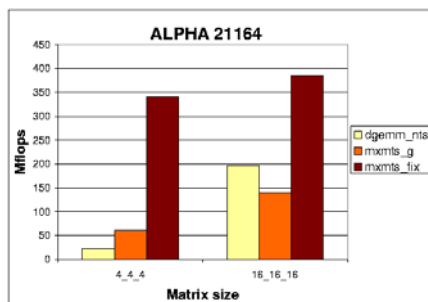


Fig. 3. Comparison of performance of different routines for matrix sizes 4x4 and 16x16.

The vendor's BLAS routine, labeled as *dgemm_nts*, fails to produce good performance for a very small matrix product (4x4) getting better results as matrix dimensions grow towards a size that fills the L1 cache (16x16). A simple matrix multiplication routine *mxmts_g* which avoids any parameter checking and scaling of matrices (*alpha* and *beta* parameters in *dgemm*) can outperform the BLAS for very small matrix sizes. The matrix multiplication code *mxmts_fix* in our library gets excellent performance for small matrices of sizes 4x4 and 16x16. There is actually one routine for each matrix size. In this paper we call all of them *mxmts_fix* for convenience. Each *mxmts_fix* routine is obtained by fixing

leading dimensions and loop limits at compilation time and trying a set of codes with different loop orders and unroll factors. The one producing the best result is then selected.

1.4 Related Work

The Cholesky factorization of a sparse matrix has been an active area of research for more than 30 years [1,5,6,7,8,9].

Iterative compilation [10] consists in a repetitive compilation of code using different parameters. Program transformations like loop tiling and loop unrolling are very effective techniques to exploit locality and expose instruction level parallelism. The authors claim that finding the optimal combination of tile size and unroll factor is difficult and machine dependent. Thus, they propose an optimization approach based on the creation of several versions of a program and decide upon the best by actually executing them and measuring their execution time. Our approach for obtaining high performance codes is similar with the difference that, while they apply a set of transformations, we use simple codes and let the compiler do its best.

Several projects were targeted at producing efficient BLAS routines through automatic tuning [11,12,13]. The difference with our work is that they are not focused on operations on small matrices.

A software for the parallel solution of sparse linear systems called Block-Solve95 [14] uses macros to put code inline for Level 2 BLAS routines GEMV and TRMV. They claim an improvement ratio between 1.2 and 2 for single processor codes working on small systems. Our approach however, is based on the improvement of Level 3 BLAS routines working on small matrices.

The remainder of the paper is organized as follows: first we present our optimization of routines operating on small matrices, namely the matrix multiplication operation. Then we show the impact of its application to the HM Cholesky factorization.

2 The Small Matrix Library (SML)

2.1 Generation of Efficient Code

Creation of efficient code has traditionally been done manually using assembly language and based on a great knowledge of the target architecture. Such an approach, however cannot be easily undertaken for many target architectures and algorithms.

Alternatively, creation of efficient codes specific for a target computer can be written in a high level language [15,16]. This approach avoids the use of the assembly language but keeps the difficulty of manually tuning the code. It still requires a deep knowledge of the target architecture and produces a code that, although portable, will rarely be efficient on a different platform.

A cheaper approach relies on the quality of code produced by current compilers. The resulting code is usually less efficient than that written manually by an expert. However, its performance can still be extremely good and some times it can even yield better code. We have taken this approach for creating a Small Matrix Library (SML).

For each desired operation, we have written a set of codes in Fortran. For instance, for a matrix multiplication we have codes with different loop orders (*kji*, *ijk*, etc.) and unroll factors. Using a *Benchmarking Tool* [17], we compile each of them using the native compiler trying several optimization options. For each resulting executable, we automatically execute it and register its highest performance. These results are kept in a database and finally employed to produce a library using the best combination of parameters.

By fixing the leading dimensions of matrices and the loop trip counts we have managed to obtain very efficient codes for matrix multiplication on small matrices. Since several parameters are fixed at compilation time the resulting object code is only useful for matrix operations using these fixed values. Actual parameters of these routines are limited to the initial addresses of the matrices involved in the operation performed. Thus, there is one routine for each matrix size. For convenience, we call all of them *m_xm_ts_s_fix* in this paper, but each one has its own name in the library.

We also tried feedback driven compilation using the Alpha native compiler but performance either remained the same or even decreased slightly. We conclude that, as long as a good compiler is available, fixing leading dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels.

2.2 Matrix Multiplication Performance

Figure 4a shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164. The matrix multiplication performed in all routines benchmarked uses the first matrix without transposition (*n*), the second matrix transposed (*t*), and subtracts the result from the destination matrix (*s*). This is the reason why we call the BLAS routine *dgemm_{nts}*.

The vendor BLAS routine *dgemm_{nts}* yields very poor performance for very small matrices getting better results as matrix dimensions grow towards a size that fills the L1 cache (8 Kbytes). This is due to the overhead of passing a large number of parameters, checking for their feasibility, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when operation is performed on large matrices. However, it is notable when small matrices are multiplied. Also, since its code is prepared to deal with large matrices, further overhead can appear in the inner code by the use of techniques like strip mining.

A simple matrix multiplication routine *m_xm_ts_s_g* which avoids any parameter checking and scaling of matrices can outperform the BLAS for very small matrix sizes.

Finally, our matrix multiplication code *m_xm_ts_s_fix* with leading dimensions and loop limits fixed at compilation time gets excellent performance for all block

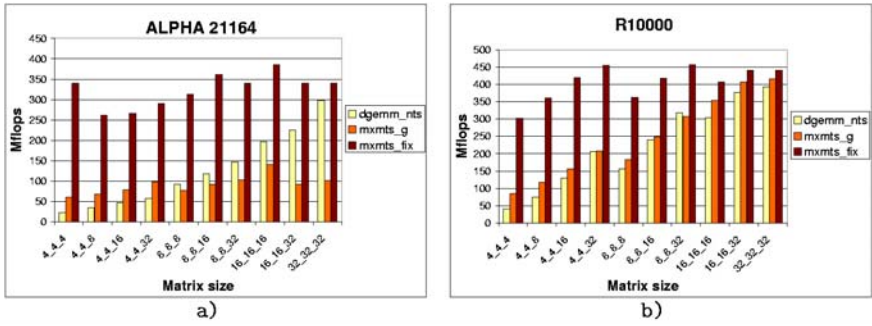


Fig. 4. Comparison of the performance of different routines for several matrix sizes: a) on an Alpha b) on an R10000.

sizes ranging from 4x4 to 16x16. The latter is the maximum value that allows for a good use of the L1 cache on the Alpha unless tiling techniques are used.

Figure 4b shows the performance of different routines for several matrix sizes on the R10000 processor. Results are similar to those of the Alpha with the only differences that the L1 cache is larger (32 Kbytes) and *mxmts_g* performs very well. This is due to the ability of the MIPSpro F77 compiler to produce software pipelined code, while the Alpha compiler hardly ever manages to do so.

Though our SML codes were adapted to the underlying architecture, they were written in a high level programming language (FORTRAN). The resulting codes were very good for both the Alpha-21164 and the R10000 processors. We believe these results can be generalized for other current superscalar architectures.

3 Using SML Routines as Computational Kernels for HM Cholesky

We have used SML routines to improve our sparse matrix application based on hypermatrices. Matrices were ordered with METIS [18] and renumbered by an elimination tree postorder. Performance varies substantially from matrix to matrix due to the different sparsity patterns and density. However, we see that using our matrix multiplication in SML improves performance substantially for all the benchmarks and block sizes.

Figure 5 shows results of the HM Cholesky factorization on an R10000¹ for matrix QAP15 from the Netlib set of sparse matrices corresponding to linear programming problems [19]; and problem pds40 from a Patient Distribution System (40 days) [20]. Ten submatrix sizes are shown: 4x4, 4x8, 4x16, ... 32x32. Effective Mflops are presented. They refer to the number of useful floating point operations performed per second. This metrics excludes useless operations on

¹ Results on the Alpha are similar.

zeros performed by the HM Cholesky algorithm when data submatrices contain zeros.

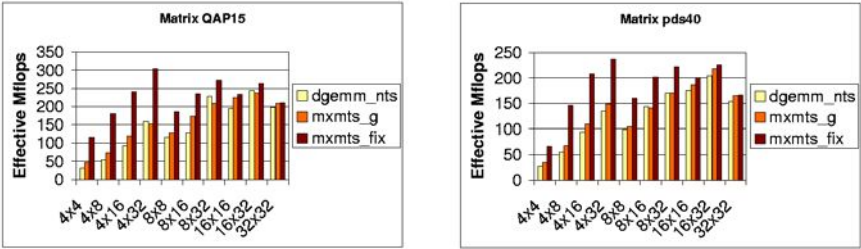


Fig. 5. Factorization of matrices QAP15 and pds40: Mflops obtained by different MxM codes in HM Cholesky on an R10000.

When *dgemm_nts* is used, the best performance is usually obtained with data submatrices of size 16×16 or 16×32 . Since the amount of zeros used can be large, the effective performance is quite low. Using *mxmts_fix* however, smaller submatrix sizes usually produce better results than larger submatrix sizes. Particularly effective in this application is the use of rectangular matrices due to the fill-in produced by the Cholesky factorization (skyline). For instance, using 4×16 or 4×32 submatrix sizes the routine used yields very good performance. Since the number of operations on zeros is considerably lower, the effective Mflops obtained are much higher than those of any other combination of size and routine.

The use of a fixed dimension matrix multiplication routine speeded up our Cholesky factorization between 20% and 100% depending on the input matrix.

Table 1. Characteristics and performance of HM Cholesky on several LP problems

Matrix	Dimension	Factor NZs	Density	Mflops
TRIPART1	4.2	1.1	0.127	223.0
TRIPART2	19.7	5.9	0.030	226.9
TRIPART3	38.8	17.8	0.023	237.4
TRIPART4	56.8	76.8	0.047	278.2
pds40	76.7	27.6	0.009	236.6
pds50	95.9	36.3	0.007	249.9
pds80	149.5	64.1	0.005	254.4
pds90	164.9	70.1	0.005	263.3
QAP15	6.3	8.7	0.436	303.1

Table 1 shows the characteristics of several matrices obtained from linear programming problems [19,20] and the performance obtained by our modified hypermatrix code. The factorization was performed on an R10000 processor with a theoretical peak performance of 500 Mflops. Dimensions are in thousands and

Factor non-zeros are in millions. Using SML routines our HM Cholesky often gets over half of the processor's peak performance for medium size matrices factored in-core.

4 Conclusions

We have shown that fixing dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels. Since no single algorithm was the best for all matrix sizes in any platform, we conclude that an exhaustive search is necessary to get the best one for each matrix size. For this reason we have implemented a *Benchmarking Tool* which automates this process.

We have generated a small matrix library (SML) on a couple of systems obtaining very efficient codes specialized on operations on small matrices. These routines outperform the vendor's BLAS routine for small matrix sizes.

Fast computations on small matrices are of great utility in sparse matrix computations. A direct application of our small matrix library can be found in the hypermatrix Cholesky factorization. High performance routines operating on small matrices allow for the election of small block sizes. This choice avoids operation of non-zeros while retaining good performance. In practice, we found that blocks of size 4x16 and 4x32 often produced the best results for the hypermatrix Cholesky factorization.

We obtained important speed-ups in our Cholesky factorization application by using the SML routines. Therefore, we believe it is worthwhile to develop this sort of library.

References

1. Duff, I.S.: Full matrix techniques in sparse Gaussian elimination. In: Numerical analysis (Dundee, 1981). Volume 912 of Lecture Notes in Math. Springer, Berlin (1982) 71–84
2. Fuchs, G., Roy, J., Schrem, E.: Hypermatrix solution of large sets of symmetric positive-definite linear equations. *Comp. Meth. Appl. Mech. Eng.* **1** (1972) 197–216
3. Ast, M., Fischer, R., Manz, H., Schulz, U.: PERMAS: User's reference manual, INTES publication no. 450, rev.d (1997)
4. Ast, M., Barrado, C., Cela, J., Fischer, R., Laborda, O., Manz, H., Schulz, U.: Sparse matrix structure for dynamic parallelisation efficiency. In: Euro-Par 2000, LNCS1900. (2000) 519–526
5. George, A., Liu, J.W.H.: Computer Solution of Large Sparse Positive-Definite Systems. Prentice-Hall, Englewood Cliffs, NJ (1981)
6. George, A., Gilbert, J.R., Liu, J.W., eds.: Graph Theory and Sparse Matrix Computation. Volume 56 of The IMA volumes in mathematics and its applications. Springer-Verlag, New York (1993)
7. Ng, E.G., Peyton, B.W.: Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.* **14** (1993) 1034–1056
8. Ashcraft, C., Grimes, R.G.: The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software* **15** (1989) 291–309

9. Rothberg, E., Gupta, A.: An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.* **15** (1994) 1413–1439
10. Kisuki, T., Knijnenburg, P., O’Boyle, M.: Combined selection of tile sizes and unroll factors using iterative compilation. In: *Parallel Architectures and Compilation Techniques*. (2000) 237–246
11. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: *11th ACM Int. Conf. on Supercomputing*, ACM Press (1997) 340–347
12. Cuenca, J., Gimenez, D., Gonzalez, J.: Towards the design of an automatically tuned linear algebra library. In: *Proceedings. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. (2002) 201–208
13. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: *Supercomputing ’98*, IEEE Computer Society (1998) 211–217
14. Jones, M.T., Plassmann, P.E.: *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*. Technical report, Argonne National Laboratory (1995)
15. Kamath, C., Ho, R., Manley, D.: DXML: A high-performance scientific subroutine library. *Digital Technical Journal* **6** (1994) 44–56
16. Navarro, J.J., García, E., Herrero, J.R.: Data prefetching and multilevel blocking for linear algebra operations. In: *Proceedings of the 10th international conference on Supercomputing*, ACM Press (1996) 109–116
17. Herrero, J.R., Navarro, J.J.: Automatic benchmarking and optimization of codes: an experience with numerical kernels. In: *Proceedings of the 2003 International Conference on Software Engineering Research and Practice*, CSREA Press (2003)
18. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR95-035, Department of Computer Science, University of Minnesota (1995)
19. NetLib: (Linear programming problems) <http://www.netlib.org/lp/>.
20. Frangioni, A.: (Multicommodity Min Cost Flow problems) <http://www.di.unipi.it/di/groups/optimize/Data/>.

Parallel Agent-Based Simulation on a Cluster of Workstations

Konstantin Popov¹, Vladimir Vlassov², Mahmoud Rafea¹, Fredrik Holmgren¹,
Per Brand¹, and Seif Haridi²

¹ SICS, Kista, Sweden. <http://www.sics.se>

² KTH/IMIT, Kista, Sweden. <http://www.imit.kth.se>

Abstract. We discuss a parallel implementation of an agent-based simulation. Our approach allows to adapt a sequential simulator for large-scale simulation on a cluster of workstations. We target discrete-time simulation models that capture the behavior of WWW. The real-world phenomena of emerged aggregated behavior of the Internet population is studied. The system distributes data among workstations, which allows large-scale simulations infeasible on a stand-alone computer. The model properties cause traffic between workstations proportional to partition sizes. Network latency is hidden by concurrent simulation of multiple users. The system is implemented in Mozart that provides multithreading, dataflow variables, component-based software development, and network-transparency. Currently we can simulate up to 10^6 Web users on 10^4 Web sites using a cluster of 16 computers, which takes few seconds per simulation step, and for a problem of the same size, parallel simulation offers speedups between 11 and 14.

1 Introduction

This paper discusses a *parallel implementation* of discrete-time *agent-based simulation* models. Such models are defined bottom-up, i.e. by defining individual behavior of components called agents. Our approach allows to adapt a sequential simulator for large-scale simulation on a cluster of workstations.

We target simulation models that capture the behavior of WWW. Web users are *active* and *stateful* agents connected with each other in a graph resembling the social network. Web sites are connected in a similar graph, forming a “landscape” where agents reside. At each time step, users exhibit certain behaviours such as visiting some of their bookmarked sites, exchanging information about Web sites in the “word-of-mouth” style, “web surfing” (i.e. visiting sites pointed by sites), and updating bookmarks. Once all users are simulated, the simulation proceeds to the next time step. In general, users access different sites at different time steps. Since users exchange information about sites, every user can acquire a reference and then access every site. The model parameters affect behavior of *individual* model components and their interaction. Running the simulation produces statistics which are used to study the *aggregated* behavior of the Internet population. In particular, the relationship between site popularity (i.e. number

of site visits) and the ratio of sites with that popularity, as well as the formation of clusters of popular sites are studied. Large-scale simulations with millions of model users may aid the understanding of the evolution of the Internet [9].

The simulator distributes data among workstations because of the memory consumption. Every workstation simulates its portion of model users. Simulation of 10^5 model users with 10^4 sites using in current model requires approximately 250Mb of main memory, while we target at least 10^6 users that would require *at least* 2.5Gb. More refined simulation models (for instance, reflecting the evolution of sites in terms of the quality of service, the type of content and the connectivity with other sites) require significantly more memory. On the other hand, we would like to speedup the execution of today's simulation runs – with 10^5 users for thousands of steps. The sequential simulator requires approximately 1 minute for 10^4 users for 10^2 steps on a 1GHz computer, so we could expect the parallel execution to finish in “hours” instead of “days”.

Model users can access arbitrary model sites, so the latter can be seen as *shared* data. This is different to simulation models such as computational fluid dynamics or neural networks, where model entities encapsulate their state entirely. In the worst case, distribution of shared data among computers causes network traffic proportional to the amount of that data. Moreover, users access *different* sites over time, as their bookmarks and bookmarks of their neighbours evolve, so that any distribution of sites aimed at minimizing the network traffic has to be dynamic. We opted for static partitioning because of the complexity and memory and run-time costs of a dynamic one.

The graphs that represent the social network of users and links between sites are “small-world” graphs, where every two nodes are connected through a rather short chain of links, while neighbour nodes are tightly connected with each other [23]. Intuitively, in order to meet these defining properties, such a graph must have a sufficient number of “long-distance” links. In a typical simulation run of our system, 30% of links point to remote nodes which are uniformly distributed over the entire set. Our heuristic for graph partitioning tries to keep tightly connected neighbours on the same workstation. We are unaware of an algorithm that optimizes partitioning with respect to links to remote nodes. Randomness of links to remote nodes causes the majority of them to cross the partition boundaries and thus cause network traffic. The amount of the traffic is proportional to the number of nodes in the graph. Very recently, however, we noticed a work on extraction of larger sub-graphs from a small-world graph [12]; we are studying its applicability for our system.

The system must be also maintainable, so that different web-simulation models could be captured within the same parallel simulation framework. Our colleagues who design and study models often change user behaviours and graph construction mechanisms. To meet this practical need, our system consists of a set of *modules* implementing behaviours of web users, constructing web sites, and also constructing users and sites graphs. The other part of the system is the *infrastructure* that holds collections of users and sites, and runs the simulation. Moreover, the interfaces between modules and the infrastructure part is

the *same* as in the sequential simulator, so our colleagues develop modules using the sequential system and we use the same modules in the parallel one.

Rapid prototyping of simulator modules is facilitated by our implementation platform, the Mozart system [13,21], available for both Un*x systems and Windows. Mozart implements Oz – a concurrent multi-paradigm programming language that offers symbolic computation with a variety of builtin data types, automatic memory management, and network-transparent distribution.

In the parallel system, the implementations of abstract data types (ADTs) that hold collections of users and sites *hide* the network from the simulation code. The data is partitioned between Mozart processes, but remote data can also be cached as long as the simulation model allows. The ADTs' implementations, in turn, are built atop of one single “distributed ADT” (**DistADT**) abstraction which completely *encapsulates* the operations over the network. When data is requested that does not belong to the local partition, a corresponding computer is contacted *inside* of the **DistADT** implementation. Since there are multiple simulation threads, network latency is automatically masked.

The parallel simulator can utilize different workstations. Our current model of data partitioning assigns partition sizes proportionally to the capacities of corresponding computers. Capacities are reversely proportional to the execution times of a certain fixed sequential simulation. The number of threads per Mozart process is a system-wide constant, which turned out to be good enough. The performance of the simulation has been improved by avoiding explicit and implicit synchronization between processes: barriers between time steps have been replaced by a more relaxed synchronization, and garbage collection in Mozart processes is initiated synchronously by the manager process.

We have already obtained promising results: on a cluster of 16 workstations simulating 10^6 users on 10^4 web sites, our simulator scales nearly perfectly if the shared data is stateless and replicated eagerly, nearly perfectly if it is replicated lazily, and deteriorates only by some 40% if data is stateful. If we interpolate the execution time to simulate 10^6 users on a single computer (the value we cannot directly obtain due to memory requirements), the simulation system offers the speedup of 14 with stateless replicated data, and 11 with stateful data.

2 Related Work

Our simulation models belong to the “discrete time system” class [25], which is a specialization of “discrete event system specification” (DEVS) class where time is continuous but only a finite number of events take place. Parallelization of discrete event models has received a lot of attention [25,8,6,16]. Our models are specified in terms of *pro-active behaviour* and *interaction* between entities rather than “events” and “response behaviour”.

It is acknowledged that most of the work on parallel discrete event simulation is done with the focus on parallel speedups, while the scalability, software engineering, and model reuse are the real issues for the modeler [18,24]. All these issues are our goals. Relatively little effort is devoted for large-scale simulation

and corresponding tools, with the notable exceptions of specially designed DEVS implementations [3], DaSSF (Dartmouth SSF, Scalable Simulation Framework)-related activities [4,17], as well as various attempts to build distributed simulators through *federation* of sequential ones [5]. We address the scalability issue by explicit distribution of data, and *encapsulating* the network operations within the implementations of collections of model users and sites.

Parallelizing a simulation involves at least the use of a specially designed simulation platform (e.g. [24]) that hides parallelization details, yet usually an adaptation of the model and some annotations for the simulation run-time system are necessary too [15,5]. Our approach is radically different: only the implementation of collections of agents (web users) and landscape locations (web sites) changes, while the implementations of the agent behaviours are kept unchanged.

Our work can be also assessed from the point of view of multi-agent simulation [10,20,1]. The difference to our work is twofold: while agents in multi-agent simulations appear to be more sophisticated than those in our case, we target orders of magnitude larger simulations. Another extreme are the so-called micro-simulations [2,14], where numerous agents are as simple as finite automata.

Our approach to distributed simulation is novel practically as it does not involve explicit message passing as in [19,11], but rather provides a distributed implementation of the same abstract data types as in the sequential simulator. The programmer's obligation is to spawn the number of threads sufficient for hiding communication latency. Our approach is similar to the one of Nexus [7], where special "global pointers" are network-transparent entities.

3 The Simulation Model

The "word of mouth" (WoM) simulation model is a stochastic socio-economical model that addresses the real-world phenomena of the behavior of the Internet population [9]. The model of time is discrete. The model is defined in terms of Web *users* and Web *sites*. Users are modeled as stateful entities: in particular, they have short-term ("loyalty") and long-term ("bookmarks") memory. Sites have ranks representing their "goodness". Users and sites are interconnected by two separate *small world* (SW) graphs [23]. The users graph represents a social network of interaction between users, and the sites graph represents the Web. The topology of graphs and the number of nodes are fixed. We use the Watts-Strogatz model for building SW-graphs, where a SW-graph is based on a 1-d lattice, which models interconnections between users/sites according to their physical proximity. A fraction of links of the lattice is then randomly rewired.

At each step, each user exhibits a number of *behaviours*: (a) it visits a random subset of its bookmarked sites, (b) contacts a random set of its neighbour users and asks them about sites worth visiting (the "word of mouth" (WoM) mechanism), (c) surfs some links on some of the sites visited during behaviours *a* and *b*, and (d) evaluates sites visited during behaviours *a,b* and *c*, and updates the loyalty/bookmark lists. Users that are contacted during WoM behaviour must complete the previous simulation step, and must not complete the next step.

Some of them *may* have completed the current step; thus, users can be simulated in *any* order within a time step. Note that information about “interesting” sites is propagated through the social network. Best visited sites are accumulated in the short-term memory, and can eventually be promoted to the bookmarks.

Model parameters define the number of users/sites, constants for SW-graph construction, sizes of loyalty/bookmarks lists, and various constants controlling user behaviours. The simulation output is a sequence of statistic vectors containing numbers of visits per each site taken at every N^{th} step. The simulation has shown that the distribution of users per sites follows a universal power law [9]. Note that the model is *stochastic*. The simulation results did not change once a more relaxed synchronization was introduced, where users contacted during the WoM behaviour were allowed to have completed the *next* simulation step.

4 Oz and Mozart

Oz is a dynamically typed language with a variety of basic data types, including atoms, numbers, records and arrays. Builtin dictionaries allow to store and retrieve (*key, value*) pairs. Oz is a lexically-scoped language. Procedures in Oz are *first-class*, and generally have *environment* which *encapsulates* values from the procedure’s lexical scope. Concurrent *threads* in Oz can synchronize by *dataflow variables*. A dataflow variable *represents* a value that can be defined only once. If a value is not defined yet, threads that depend on it are transparently *blocked* by the Mozart run-time system until a value is defined. Defining a variable’s value is referred to *binding a variable*. Threads can also communicate via asynchronous message passing through *ports*. A port is an abstract data type with the “send a value” operation that provides a *partially ordered mailbox* for messages: messages sent by the same thread are retrieved from the box in the same order as they are sent.

Mozart provides for a *network-transparent* distribution of Oz programs: an application runs on a *network of computers* as it were one computer. This is supported by the Mozart’s *distribution subsystem*. Oz entities shared by threads running on different Mozart processes are called *distributed*. Stateless distributed entities are *replicated*, and stateful ones are kept consistent with the help of *access structures* that connect involved processes and are used by entity-type specific *distribution protocols*. When an operation on a distributed stateful entity is performed, distribution subsystems of involved processes exchange message(s) according to the entity’s protocol. Mozart differentiates between *stationary* and *mobile* entities. A ports, for example, is stationary: once created, it stays at a process where created; messages to the port are delivered to its “home” process.

5 The Simulator

Our parallel simulator is based on the sequential simulator which organization follows the one of the *Worker* on Figure 1. Simulation is controlled by the *Manager* component. *Simulator* executes behaviours for each user using data

from **Users** and **Sites** modules, and records statistics into **Statistics**. Model users and sites are represented as records, and are indexed. **Users** and **Sites** abstractions store representations under indices. Users' neighbours, users' bookmarks, and outgoing links from sites are represented as lists of indices.

Parallel simulation is conducted by *workers*, each running several threads. **Users** and **Sites** modules hold now *partitions* of data, but provide transparent access to all data. Only locally stored users are simulated. **Users** and **Sites** are the *same abstractions* as in the sequential system but with different implementations, so the **Simulator** component is reused without changes. The **Manager** process constructs the WoM graphs for users and sites, and the representations of users and sites themselves. This data is initialized by the sequential **Users** and **Sites** components, so these components are reused as well. Once constructed and initialized, the data is divided into partitions and partitions are distributed to corresponding workers. Workers construct the distribution-aware **Sites** and **Users** modules containing the given partition data.

We have implemented both the strict and relaxed synchronization between workers, as described in Section 3. With the strict synchronization, there are barriers between simulation steps: the manager waits until all workers finish a step, then it initiates the next step. With the relaxed synchronisation, if a worker at a time step n contacts another worker at step m , then either $n = m$ or $n = m - 1$, i.e. a user that is contacted because of the WoM behaviour is allowed to have completed one step *ahead*. It is implemented as follows. At the beginning of a step n , each worker waits until all other workers report that they completed step $n - 2$, then it by itself tells other workers that it completed $n - 1$, and proceeds with the step n . It can be shown that the time in any two workers can differ by at most 1. Since synchronization messages and accesses to remote data go over the same channels, this guarantees that when a worker at step m receives a request from a worker at step n , then $n = m \pm 1$. The remaining case of $n = m + 1$ means the worker that received a request is lagging behind, so the request is queued until the next step.

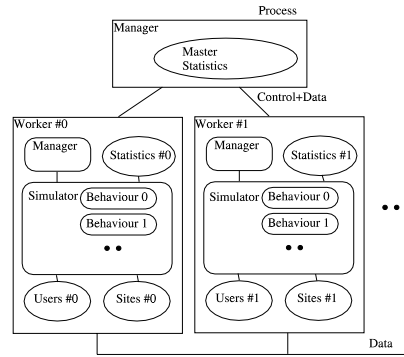


Fig. 1. The Simulator.

6 Distributed Abstract Data Types

Users and **Sites** data abstractions of the parallel system contain local data and **GetRemote** procedures which provide access to data from remote partitions. **GetRemote** abstractions are provided by the generic **DistADT** abstract data type, hence the generic “**GetRemote**” name. Let us consider the **Users**

abstraction; the **Sites** abstraction is implemented similarly. **Users** provides the **GiveVote** method that calculates a WoM recommendation from a specified user. **GiveVote** takes a user index and returns a user index. Assume that a requested user happens to be non-local (arrow 1 on Figure 2), in which case **GetRemote** is used. It sends a request to a server on the right partition (arrow 2). The server uses that partition's **GiveVoteLocalUser** procedure (arrow 3). The output of **GetVoteLocalUser** is delivered backwards (arrows 4, 5 and 6) in a *container* sent by **GetRemote** along with the request. A container is implemented as a dataflow variable: it is declared by **GetRemote**, becomes distributed between the two processes when a request arrives to the server, and is bound to the result by the server. Eventually the value is delivered back (arrow 5) by the Mozart distribution subsystem.

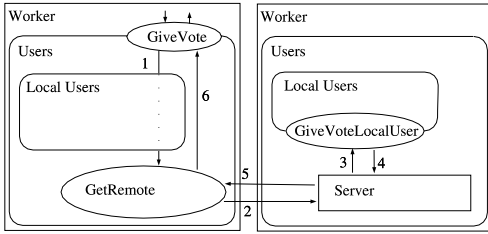


Fig. 2. Distributed Users ADT.

The **GetRemote** abstraction is provided for **Users** partition by the **DistADT** data abstraction (see Figure 3). **DistADT** modules are shared among all partition processes. In our example, the server Oz port and process logically belong to the **DistADT** module serving the **Users** module. Although not

shown on picture, ports and servers of other **Users** partitions belong to the same **DistADT** module too. There is exactly one port/server pair per partition. The **DistADT** module encapsulates a map from user indices to their partition indices, so for any user index **GetRemote** can choose the correct port and send a request to the correct partition.

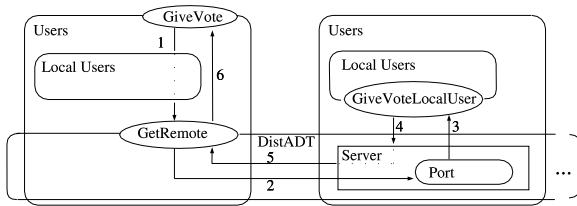


Fig. 3. Users ADT using DistADT.

In order to use a **DistADT** module, **Users** partitions have to *register*. Registration gives a **DistADT** module access to local partition data. In our example, the **GiveVoteLocalUser** procedure is the access abstraction. During registration a (stationary) Oz port is constructed and a local server (a thread that processes requests sent to the port) is started. **DistADT** modules are created by the manager and passed over to all partitions, so they become shared between partitions.

Note that invocation of `GetRemote` transparently returns a value of whatever type the server’s “`GetLocal`” (`GetVoteLocalUser` in our example) does. In this way, two (distinct) instances of the very same `DistADT` data abstraction are used for both `Users` and `Sites` ADTs.

Both the “*data to computation*” and “*computation to data*” approaches to parallel execution are supported by `DistADT`, and used in our system. In the `GiveVote` example, the vote is computed by the process holding the data. Computing a vote requires analysis of the bookmark list, while both the query and the response messages exchanged between processes contain only an index. Thus, it is beneficial to move the computation to data. On the other hand, the complex algorithm of user simulation requires representations of several sites visited by the user. Those sites can even belong to different partitions. Thus, site representations (data) are moved to the partition where simulation takes place.

7 Performance

We have evaluated our system on a cluster of 16 AMD Athlon 1900+ computers connected by the 100Mbit switched Ethernet. Scalability and speedup properties are presented on Figure 4. The “no caching/replication” case refers to the situation when model sites that reside on a remote worker are retrieved over the network every time they are needed. The “caching” version keeps retrieved sites for the duration of the model time step. In the version “with replication”, all workers have their own copies of *stateless* model sites. While in the WoM model the sites are stateless, our experiments show how the simulator would behave if that data *were* stateful. Since the *majority* of data is remote even with 16 workers, the system should scale well also beyond that size. The system also delivers respectful speedups when it counts *most* – for larger problem sizes.

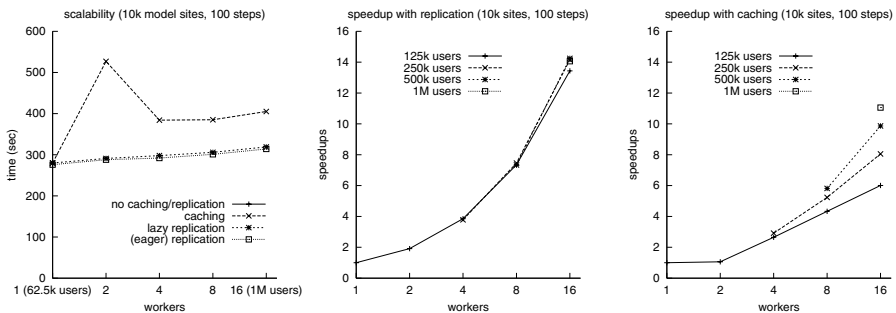


Fig. 4. Scalability and Speedups.

Figure 5 shows the impact of multithreading. Given the properties of the simulation model and the best-case communication latency in Mozart, we have also estimated the number of threads required to hide communication latency using the (analytical) deterministic model of processor utilization in a multithreaded

architecture (see e.g. [22])¹. The model predicts that approximately 100 threads is needed to hide the latency, which is in compliance with our experimental data.

The next plot shows that sporadical load imbalance due to e.g. activities of other processes in OSeS is a no big issue for our simulator. We emulated such kind of imbalance by deliberate over/underloading of one worker during initial partitioning. Note also that *underloading* a worker, which apparently causes its more frequent preemption by the OS, does *not* have a “cascading” effect on other workers. We are investigating this issue further.

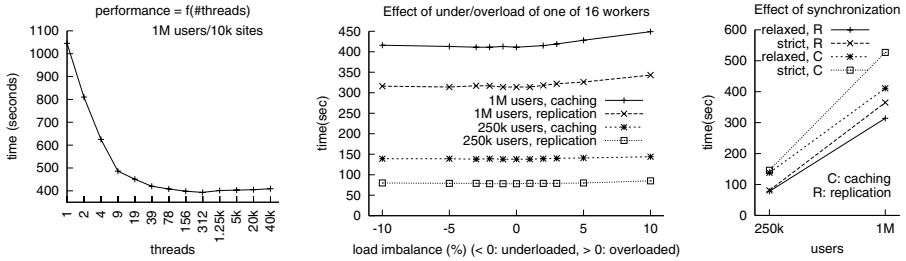


Fig. 5. Number of threads, load balance, and synchronization.

Finally, the effect of “relaxed” synchronization discussed in Sections 3 and 5 is presented: one can see quite a significant improvement, in particular for *larger* simulations. Given that a barrier in Mozart can require ca. 50msec, so on 100 steps ca. 5 seconds could be saved just by eliminating the barriers, the improvement is larger than we have expected for *large* simulations. Our hypothesis here is that relaxed synchronization allows to “smooth out” *temporal* imbalance of work *between* adjacent time steps.

8 Conclusions

We have shown a design of a parallel simulator for discrete-time agent-based simulation models targeted for studying the evolution of WWW. Distribution of data among computers is mandatory due to memory requirements, but efficient parallel simulation is not trivial because of the large amount of spatially and temporally irregular inter-process communication. At the same time, the simulator must be easily adapted for a set of different simulation models.

The parallel simulator runs on both a network of workstations and a multi-processor, and shows good scalability and speedup for up to 1M of model agents. The simulator is small and easy to maintain. It is implemented in Oz/Mozart that supports symbolic, concurrent and network-transparent programming, all necessary for this work. The evaluation of the simulation on a cluster of workstations is still going on, concurrently with integration of new simulation models.

¹ $N_{threads} = L/(R + C) + 1$, where L and R are the mean times for communication latency and thread run time between remote memory accesses, respectively, and C is the thread context switch time.

Acknowledgments. We are thankful to all members of our lab for the creative environment, ubiquitous support and constant encouragement. Special thanks go to Petros Kavassalis and Stelios Lelis from ICS-FORTH, Heraklion, Crete, Greece. The work is done within the EU project iCities (IST-1999-11337).

References

1. Anderson, J.: A Generic Distributed Simulation System for Intelligent Agent Design and Evaluation. In: Proc. of the 10th Int. Conf. on AI, Simulation, and Planning in High Autonomy Systems (2000) 36–44
2. Cetin, N., Nagel, K., Raney, B., Voellmy, A.: Large-Scale Multi-Agent Transportation Simulations. In: Proc. of the Workshop on Cellular Automata, Neural Networks and Multi-Agent Systems in Urban Planning, Milan, Italy (2001)
3. Chow, A.C.H., Zeigler, B.P.: Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism. Proc. of the 26th Winter Simulation Conf. (1994) 716–722
4. Cowie, J., Liu, H., Liu, J., Nicol, D., Ogielski, A.: Towards Realistic Million-Node Internet Simulations. In: Proc. of PDPTA'99, Las Vegas, NV, USA (1999)
5. Ferenci, S.L., Perumalla, K.S., Fujimoto, R.M.: An Approach for Federating Parallel Simulators. In: Proc. of PADS'00, Bologna, Italy. IEEE CS Press (2000) 63–70
6. Ferscha, A.: Parallel and Distributed Simulation of Discrete Event Systems. In: Parallel and Distributed Computing Handbook. McGraw-Hill (1996) 1003–1041
7. Foster, I., Kesselman, C., Tuecke, S.: The Nexus Approach to Integrating Multithreading and Communication. J. of Paral. and Dist. Computing **37** (1996) 70–82
8. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley (2000)
9. Lelis, S., Kavassalis, P., Sairamesh, J., Haridi, S., Holmgren, F., Rafea, M., Hatismatiou, A.: Regularities in the Formation and Evolution of Information Cities. In: Second Kyoto Workshop on Digital Cities. LNCS 2362, Springer (2001) 41–55
10. Logan, B., Theodoropoulos, G.: The Distributed Simulation of Multiagent Systems. Proceedings of the IEEE **89** (2001) 174–185
11. Lowenthal, D.K., Freeh, V.W.: Architecture-Independent Parallelism for Both Shared- and Distributed-Memory Machines Using the Filaments Package. Parallel Computing **26** (2000) 1297–1323
12. Matsuo, Y.: Clustering using Small World Structure. Proc. of the Int. Conf. Knowledge-Based Intelligent Inform. and Engineer. Systems, Crema, Italy (2002)
13. Mozart Consortium: The Mozart Programming System. <http://www.mozart-or.oz/>
14. Nagel, K., Rickert, M.: Parallel Implementation of the TRANSIMS Micro-Simulation. Parallel Computing **27** (2001) 1611–1639
15. Nicol, D.M., Heidelberg, P.: Parallel Execution for Serial Simulators. ACM Transactions on Modeling and Computer Simulation (TOMACS) **6** (1996) 210–242
16. Nicol, D.M.: Principles of Conservative Parallel Simulation. In: Proc. of the 28th Winter Simulation Conf., Coronado, CA, USA. ACM Press (1996) 128–135
17. Nicol, D.M., Liu, J.: Composite Synchronization in Parallel Discrete-Event Simulation. IEEE Transactions on Parallel and Distributed Systems **13** (2002) 433–446
18. Page, E.H.: Beyond Speedup: PADS, the HLA and Web-Based Simulation. In: Proc. of PADS'99, Atlanta, GA, USA. IEEE Society Press Press (1999) 2–9
19. Strumpen, V.: Software-Based Communication Latency Hiding for Commodity Workstation Networks. In: Proc. of ICPP'96. CRC Press (1996) 146–153
20. Uhrmacher, A.M., Gugler, K.: Distributed, Parallel Simulation of Multiple, Deliberative Agents. Proc. of PADS'00, Bologna, Italy. IEEE CS Press (2000) 101–108

21. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004) (to appear)
22. Vlassov, V., Ayani, R.: Analytical Modeling of Multithreaded Architectures. *J. of Systems Architecture* **46** (2000) 1205–1230
23. Watts, D.L.: *Small Worlds: the Dynamics of Networks Between Order and Randomness*. Princeton University Press, New Jersey, USA (1999)
24. Wieland, F., Blair, E., Zukas, T.: Parallel Discrete-Event Simulation (PDES): A Case Study in Design, Development, and Performance Using SPEEDES. In: *Proc. of PADS'95*, Lake Placid, NY, USA. IEEE CS Press (1995) 103–110
25. Zeigler, B.P., Praehofer, H., Kim, T.G.: *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press (2000)

Low Level Parallelization of Nonlinear Diffusion Filtering Algorithms for Cluster Computing Environments

David Slogsnat¹, Markus Fischer¹, Andrés Bruhn², Joachim Weickert², and Ulrich Brünig¹

¹ Department of Computer Science and Engineering
University of Mannheim, 68131 Mannheim, Germany
`slogsnat@uni-mannheim.de`

² Faculty of Mathematics and Computer Science
Saarland University, Building 27.1, 66041 Saarbrücken, Germany

Abstract. This paper describes different low level parallelization strategies of a nonlinear diffusion filtering algorithm for digital image denoising. The nonlinear diffusion method uses a so-called additive operator splitting (AOS) scheme. This algorithm is very efficient, but requires frequent data exchanges. Our focus was to provide different data decomposition techniques which allow for achieving high efficiency for different hardware platforms. Depending on the available communication performance, our parallelization schemes allow for high scalability when using fast System Area Networks (SAN), but also provide significant performance enhancements on slower interconnects by optimizing data structures and communication patterns. Performance results are presented for a variety of commodity hardware platforms. Our most important result is a speedup factor of 210 using 256 processors of a high end cluster equipped with Myrinet.

1 Introduction

Over the recent years, distributed processing has been a powerful method to reduce the execution time of computationally intensive applications. Using components off the shelf (COTS) in combination with a fast network has become a viable approach to achieve high performance known from supercomputers, however at a fraction of the cost. The platform independent message passing interface MPI [7] allows applications to run on a variety of systems. This reduces the overhead of porting applications significantly.

Image processing is becoming a more and more important application area. Variational segmentation and nonlinear diffusion approaches have been very active research fields in the area of image processing and computer vision as well. Motivating factors for this research on PDE (partial differential equation) based models are for example the continuous simplification of images or shapes which help to understand what is depicted in the image. Image enhancement including

denoising, edge enhancement, active contours and surfaces or closing interrupted lines are other points of interest. Within this context efficient numerical algorithms have been developed. They are the basis for this work, which focuses on reducing the execution time using distributed processing. AOS schemes for nonlinear diffusion filtering have been applied successfully on a parallel system with shared memory [2] and a moderate processor count. This motivated us to investigate their suitability on distributed memory systems with a large number of processors to enable close-to-real-time processing. One specific application area which we would like to address is the analysis of 2D and 3D medical images. For this scenario the approach of using clusters as computing resources gives a dynamic range from low- to high-end systems.

In the following chapter, we will describe the algorithm and how it was parallelized. Chapter 3 briefly describes the clusters that have been used as testbeds. In Chapter 4 we will present the performance results. Chapter 5 concludes our description.

2 Parallelization of the Algorithm

This section describes the algorithm and explains two possible solutions for data decomposition, a dimension segmentation and a mesh segmentation method. The algorithm is described for the three dimensional case, the two dimensional case is analogous. Figure 2 shows a filtered 2D image, figure 1 shows the original image.



Fig. 1. Image with Noise

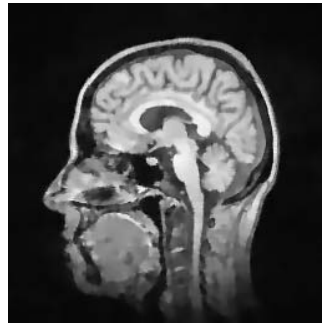


Fig. 2. Image after Filtering

2.1 Algorithm Components and Constraints

In the last decade, PDE based models have become very popular in the fields of image processing and computer vision. The nonlinear diffusion models as we know them today were first introduced by a work of Perona and Malik [3], who developed a model that allows denoising images while retaining and enhancing

edges. This model has been improved by Catté et al. [11] from both a theoretical and practical viewpoint. Anisotropic extensions with a diffusion tensor are described in [5].

In practice, nonlinear diffusion filters require numerical approximations. In [1] a finite difference scheme based on an additive operator splitting (AOS) technique [4] is used for this purpose. This AOS technique is the basis for our parallelization efforts. Typical AOS schemes are one order of magnitude more efficient than simple diffusion algorithms.

The AOS scheme is applied in an iterative way until the desired level of denoising is reached. The AOS iteration loop starts with a Gaussian convolution, which needs to be processed for each dimension. Based on this smoothed image, derivatives are computed that are of need to determine the diffusivity values. Finally the diffusion processes for all dimensions are calculated and the results are averaged in a final recombination step.

Gaussian Convolution. The Gaussian convolution is processed separately for each dimension $d \in \{1, 2, 3\}$. At iteration step n , the algorithm computes a matrix M^n using the matrix from the previous step $n-1$, thus $M_{gauss}^n = f_{gauss}(M^{n-1})$. The convolution takes places for one dimension after another. The computation itself is a stencil operation with a stencil size of $1 \times g$ for every dimension, with a typical value for $g = 3$. As a consequence, communication takes place before the computation for each dimension, but not during the computation for a single dimension.

Diffusivity. The diffusivity is calculated from the output matrix of the Gaussian convolution step: $M_{diff}^n = f_{diff}(M_{gauss}^n)$. In contrast to the Gaussian convolution, the diffusivity is not calculated separately for every dimension, but only once for all dimensions. The stencil has a fixed size of $3 \times 3 \times 3$ pixels. As a result, communication between nodes has to be performed only once before the diffusivity is calculated, but not during the calculation process.

Diffusion. The diffusion is computed from two matrices, those are the matrix from the previous iteration M^{n-1} and the output matrix of the diffusivity M_{diff}^n . To compute the diffusion, d tridiagonal linear systems have to be solved by a fast Gaussian algorithm, where every system describes diffusion in one of the dimensions. These systems can be solved independently of each other, although they are solved one after another for practical reasons. Also, they can again be decomposed into many small independent equation systems, where each system corresponds to a single line in diffusion direction. These lines are solved by an algorithm using first forward and then backward propagation. This means that if a line crosses process boundaries, these processes have to communicate with each other during the computational phase. This is different from the Gaussian convolution and diffusivity steps, where no communication is required.

Finally, all three resulting matrices have to be merged into the final matrix M^n by taking the average.

2.2 Data Decomposition

In the following we present two data decomposition methods for 2D and 3D scenarios that have both been applied to the algorithm. Figure 3 depicts schemes for the decomposition methods.

The first decomposition method is a *multidimensional slice decomposition* [14]. Given a number of NP processes and a discrete 3D image $f[x_1, x_2, x_3]$, $x_1 \in \{0, \dots, X_1 - 1\}$, $x_2 \in \{0, \dots, X_2 - 1\}$ and $x_3 \in \{0, \dots, X_3 - 1\}$ then the partial image $f[x_1, x_{2np}, x_3]$ with $x_{2np} \in \{X_2/NP * np, \dots, X_2/NP * (np + 1) - 1\}$, will be processed by processor np, $np \in \{0, \dots, NP - 1\}$. In a similar manner, the image can also be sliced up in one of the other two dimensions. Since the diffusion algorithm performs filters in all three dimensions, this means that all three decomposition types are used to reduce communication between processes. When the program switches between filters of different dimensions, the direction of decomposition is changed accordingly. However, this implies that an all-to-all communication has to take place with a total traffic of $\frac{NP-1}{NP} * Imagesize$ pixels, since every processor has to acquire the actual values for all pixels in the new slice.

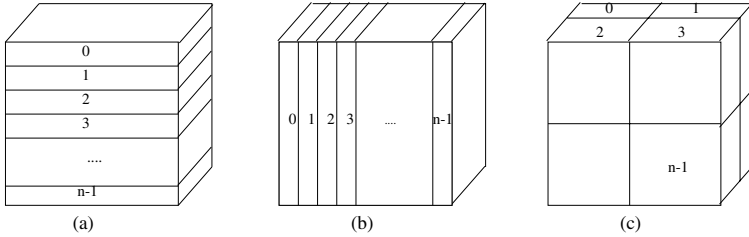


Fig. 3. Column (a) and Row- (b) versus Mesh Decomposition (c)

The second method is a *mesh partitioning* which divides the data as shown in the figure. Mesh partitioning has the advantage that it requires less interconnect performance than the slice decomposition method. In contrast to slice partitioning, the image does not have to be redistributed among the processors, but only the borders of the meshes have to be exchanged with the neighboring meshes. This leads to a lower amount of data that has to be communicated. Also, communication has only to be performed with a small number of neighbors in contrast to the all-to-all communication required by the slice decomposition method.

However, mesh partitioning has a downside too. Communication between processes takes places during computational steps, rather than in-between them. This increases the chance of processes waiting for results of neighboring processes. This problem can be reduced by means of pipeling. This has been implemented for the diffusion step of the algorithm: the processing of different lines is interleaved.

3 Available Cluster Environments

Ethernet type solutions for a network of workstations or a small cluster are currently still being considered as a solution for distributed computing. One of the reasons is that it is available as a built-in solution with no additional costs, and applications using sockets will work on any type of Ethernet ranging from Fast Ethernet to Gigabit Ethernet. In contrast to Ethernet network interfaces, System Area Networks (SANs) always provide direct user access. Also, SAN network adapters implement reliable transport services directly in hardware. They also deliver very high bandwidths (typically more than 1 GB/s) with very low latency. Currently, the most popular SAN is Myrinet from Myricom[8]. Other SAN implementations are the Scalable Coherent Interface (SCI) from Dolphin[9], Qsnet from Quadrics[10] and ATOLL from the University of Mannheim[13]. Our goal was to provide efficient implementations on clusters, using a variety of interconnection networks. Although SANs are clearly the choice of system for parallel processing, we analyzed our algorithm on Ethernet clusters too. The reason to do so is to provide an algorithm that performs best on any type of cluster, not only on dedicated high performance systems.

In the following, we will give a short overview of the platforms which were available for our performance evaluations.

Paderborn Center for Parallel Computing, Germany. The PC2 hosts a hpcLine system which includes 96 nodes, consisting of 2 PIII 850 Mhz CPU's each. A Linux 2.4 is running in SMP mode. Besides regular Ethernet, a faster interconnect is available with the SCI network plugged into a 32Bit/33Mhz PCI interface. The MPI library is a commercial MPI implementation from Scali, Norway, called Scampi.

Real World Computing Partnership, Japan. The ScoreIII cluster consists of 524 nodes, two PIII 933 Mhz processors each, running a modified Linux 2.4 SMP Kernel. The Cluster is fully interconnected to a CLOS network using a Myrinet2000 network interface. Although Myrinet offers a universal 64Bit/66Mhz interface, the motherboard only supports the 64Bit/33Mhz mode. The cluster makes use of the Score cluster system software[6], which provides fast access to both Myrinet and Ethernet devices. The cluster is ranked 90th in the November 2002 Top500 Supercomputer List.

FMI Passau, Germany. The FMI in Passau is also equipped with a hpcLine, offering latest SCI cards with 64Bit/66Mhz. The dual Intel PIII 1000 Mhz Processors are running a Linux 2.4 Kernel. Just as the PC2 cluster, the FMI uses the Scampi environment.

All clusters do also have Fast Ethernet network interfaces.

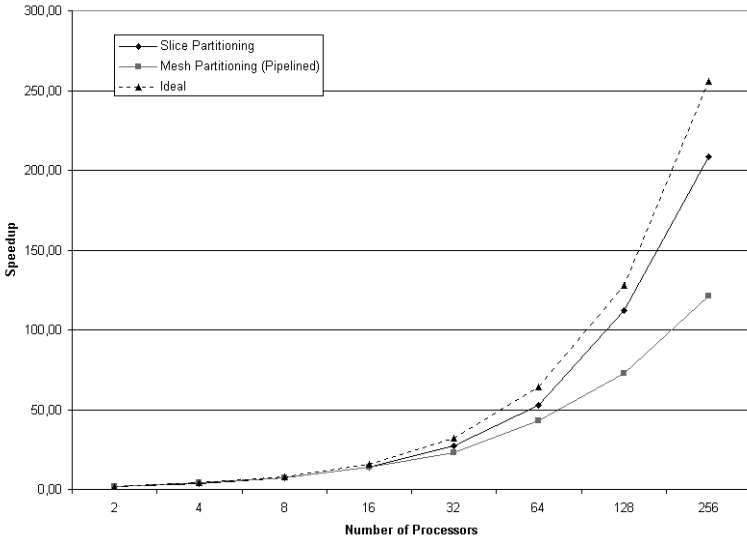


Fig. 4. Speedup on Myrinet

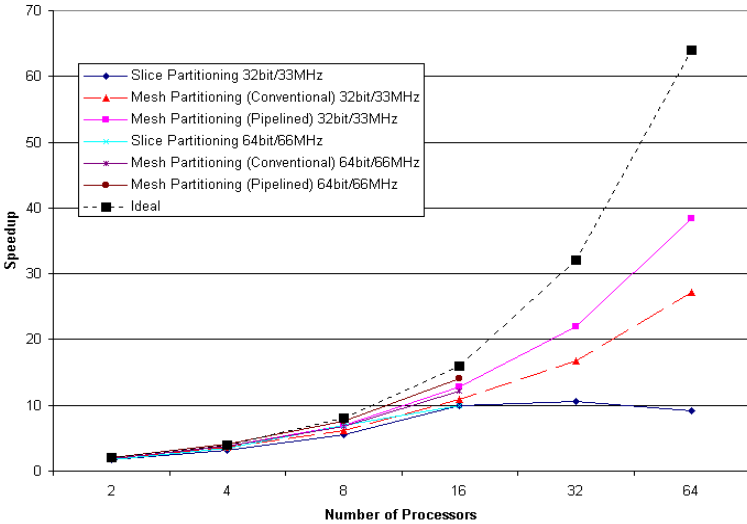


Fig. 5. Speedup on SCI with 32bit/33MHz PCI and 64bit/66MHz PCI

4 Performance

In the following, the results of a number of different versions of the algorithm will be presented. All are using non blocking communication calls. In contrast to collective scatter communication, non blocking communication allows for better

scalability with the system size. We expected a performance difference when switching from a SAN to Fast Ethernet and we were interested in breaking down the ratio of communication and computation.

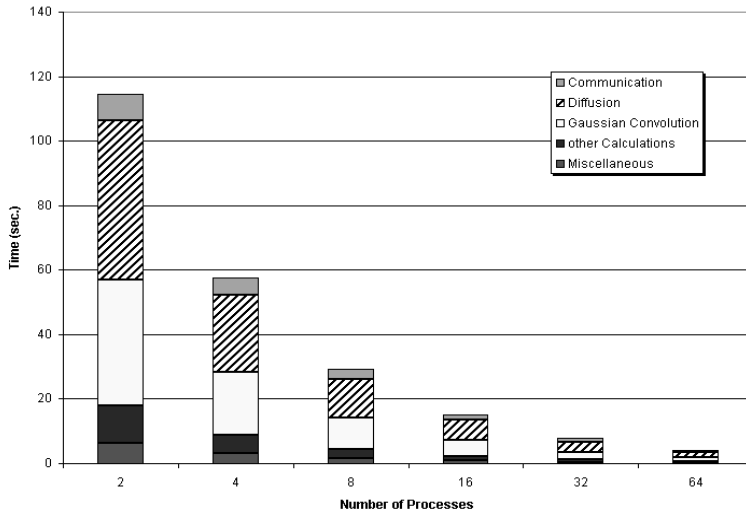


Fig. 6. Myrinet execution time breakdown using slice partitioning

As expected, the highest speedup can be achieved on Myrinet clusters. As shown in figure 4, a speedup factor of 210 on 256 processors using the slice decomposition method reveals an almost linear increase in performance. Mesh partitioning scales well too, but it is by far outperformed by the slice decomposition method, which is up to 50% faster than mesh partitioning. Much to our surprise, we observed a contrarian scaling behaviour on SCI, illustrated in figure 5. While mesh partitioning scales almost as good on SCI as on Myrinet, slice partitioning does not scale at all on SCI. The scaling behaviour using Ethernet is very similar to the behaviour using SCI. However, the total speedup achieved is noticeable smaller: the use of 64 processors leads to a speedup factor of 22.

A breakdown of the total execution time into computation and communication helps to reveal bottlenecks. For Myrinet, figure 6 shows that the time spent for communication is negligible. In contrast, the communication overhead for SCI increases significantly, as shown in figure 7. According to our results and as indicated by benchmarks like the Effective Bandwidth Benchmark [12], the bandwidth of the SCI interface itself is not the bottleneck. We measured a peak bandwidth of 85 MB/s and a one-way latency of $5.1\mu\text{s}$ for the slower 32bit/33MHz PCI cards, which is more than sufficient for our algorithm. It is more likely that the SCI ring structure can not deal very well with high data traffic. On the unidirectional SCI ring, applications may also be hampered by

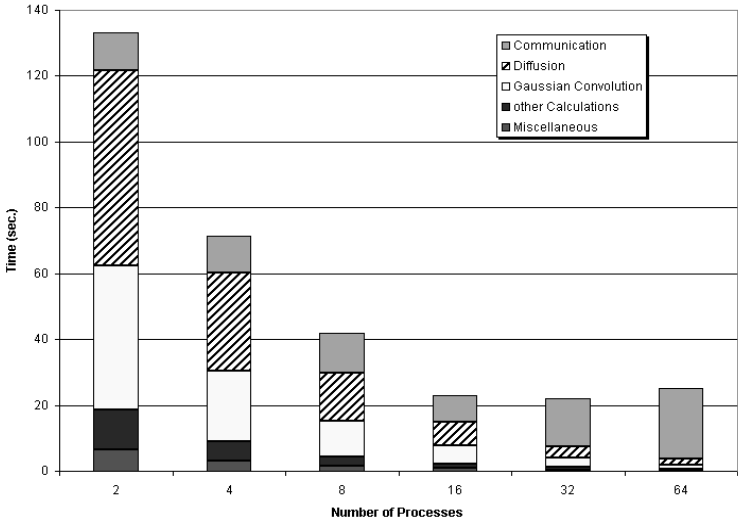


Fig. 7. SCI execution time breakdown using slice partitioning

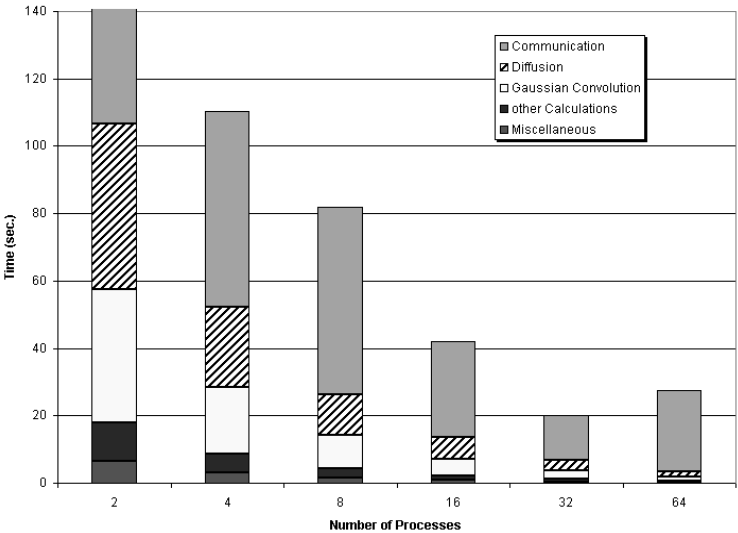


Fig. 8. Ethernet execution time breakdown using slice partitioning

other applications running on the cluster. Weak scalability is especially the case for low end networks such as Fast Ethernet. Figure 8 shows that the execution time of the algorithm is clearly dominated by the communication.

It can be concluded that using Myrinet, the communication overhead is negligible for our algorithms. On SCI and FastEthernet, communication is still the major bottleneck. Therefore mesh partitioning, solely optimized for a low interconnect usage, is the method of choice on the slower FastEthernet and SCI interconnects. Slice decomposition does not scale well on these interconnects.

5 Conclusion

We have shown the potential of cluster computing for parallelization of a nonlinear diffusion algorithm. Two major versions have been implemented and analyzed. With different variants, they offer solutions for both environments with highest network performance, as well as systems which are only loosely coupled. Our current focus is to generate an autonomous initial mapping based on cluster characteristics. This includes the processor performance as well as network features. Another investigation is to support heterogeneous computing resources in which processors obtain tasks based on their raw performance and current load. This will lead to an adaptive method which will enable the application to perform most efficient under several environments ranging from networks of workstations to tightly integrated systems.

Acknowledgements. We want to thank Andrés Bruhn and Timo Kohlberger for their valuable contributions, Joachim Weickert and Christoph Schnörr for the excellent cooperation in our joint project, and Tobias Jakob for his analysis of various data distribution schemes. Thanks go also to the operators of above clusters for granting us access to their systems. This work was sponsored by the DFG unter project number Schn 457/4-1.

References

- [1] J. Weickert, B.M. ter Haar Romeny, M.A. Viergever. Efficient and reliable schemes for nonlinear diffusion filtering, *IEEE Transactions on Image Processing*, Vol. 7, 398–410, 1998.
- [2] J. Weickert, J. Heers, C. Schnörr, K.J. Zuiderveld, O. Scherzer, H.S. Stiehl, Fast parallel algorithms for a broad class of nonlinear variational diffusion approaches, *Real-Time Imaging*, Vol. 7, 31–45, 2001.
- [3] P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion *IEEE Trans. Pattern Anal. Mach. Intell.* 12, 629–639, 1990.
- [4] T. Lu, P. Neittaanmaki and X.-C. Tai. A parallel splitting up method for partial differential equations and its application to Navier–Stokes equations, *RAIRO Mathematical Models and Numerical Analysis* 26(6), 673–708, 1992.
- [5] J. Weickert Anisotropic diffusion in image processing *Teubner*, 1998
- [6] Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O’Carroll, and H. Harada. RWC PC Cluster II and SCore Cluster System Software – High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pages 55–62, 1999.

- [7] Message Passing Interface *MIT Press*, 1994.
- [8] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. Su, Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro*, Vol. 15, 29–36, 1995
- [9] *IEEE Std for Scalable Coherent Interface (SCI)*. Inst. of Electrical and Electronical Eng., Inc., New York, NY 10017, IEEE std 1596-1992, 1993.
- [10] F. Petrini, A. Hoisie, W. Feng, and R. Graham. Performance Evaluation of the Quadrics Interconnection Network. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.
- [11] F. Catté, P. L. Lions, J. M. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Num. Anal.*, 29(1):182–193, 1992.
- [12] K. Solchenbach. EMP: Benchmarking the Balance of Parallel Computers. *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems*, Wuppertal, Germany, Sept. 13, 1999
- [13] Ulrich Brüning, Holger Fröning, Patrick R. Schulz, Lars Rzymianowicz. ATOLL: Performance and Cost Optimization of a SAN Interconnect. *IASTED Parallel and Distributed Computing and Systems (PDCS)*, Nov. 4–6, 2002, Cambridge, USA
- [14] A. Bruhn, T. Jakob, M. Fischer, T. Kohlberger, J. Weickert, U. Brüning, and C. Schnörr. Designing 3-d nonlinear diffusion filters for high performance cluster computing. *Pattern Recognition, Proc 24th DAGM Symposium*, volume 2449 of *Lect. Not. Comp. Sci.*, pages 290–297, Zürich, Switzerland, 2002. Springer.

Implementation of Adaptive Control Algorithms in Robot Manipulators Using Parallel Computing^{*}

Juan C. Fernández¹, Vicente Hernández², and Lourdes Peñalver³

¹ Dept. de Ingeniería y Ciencia de los Computadores, Universidad Jaume I, 12071-Castellón (Spain), Phone: +34-964-728265; Fax: +34-964-728435, jfernand@icc.uji.es

² Dept. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071-Valencia (Spain), Tel: +34 96 3877356, Fax: +34 963877359, vhernand@dsic.upv.es

³ Dept. de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, 46071-Valencia (Spain), Phone: +34-96-3877572; Fax: +34-96-3877579, lourdes@disca.upv.es

Abstract. The dynamics equation of robot manipulators is non linear and coupled. An inverse dynamic control algorithm that requires a full knowledge of the dynamics of the system is one way to solve the control movement. Adaptive control is used to identify the unknown parameters (inertial parameters, mass, etc). The adaptive control algorithms are based on the linear relationship of inertial parameters in the dynamic equation. A formulation to generalize this relationship is applied to the Johansson adaptive algorithm. The objective of this paper is to present the implementation of this relationship using parallel computing and apply it to an on-line identification problem in real-time.

1 Introduction

The dynamic equation of robot manipulator torque in open chain is determined by highly coupled and non linear differential equation systems. It is necessary to use approximate or cancelling techniques to apply some control algorithms, such as inverse dynamic control, over the full system. To apply these control techniques it is necessary to know the dynamics of the system. This knowledge allows the existing relations among the different links to be established. The links to establish the kinematic relations of the system are defined from Denavit-Hartenberg parameters. Full knowledge of the dynamics, inertial parameters, mass and inertial moments for each arm, is usually unavailable. These parameters should be estimated using least square or adaptive control techniques. Using adaptive control it is possible to solve both movement control and parameter identification problems. Some of these algorithms can be found in [1,

^{*} This work is supported by the CICYT Project TIC2000-1151-C07-06.

5,7,4]. One of the problems in applying this kind of algorithm is that of obtaining the relationship

$$\tau = Y_\tau(q, \dot{q}, \ddot{q})\theta, \quad (1)$$

where $Y_\tau(q, \dot{q}, \ddot{q})$, known as regressor, is an $n \times r$ matrix, n being the number of links and r the number of parameters; and θ is the $r \times 1$ parameter vector. Considering all different parameters, r will be $10n$.

The linear relationship for the adaptive Johansson algorithm using several algebraic properties and definitions given in [6] is employed. This formulation is a computable and general solution for any robot manipulator in open chain using Denavit-Hartenberg parameters and is an extension of the Lagrange-Euler formulation. The main problem of the Lagrange-Euler formulation is its high computational cost, but there are several studies, [8,2], where this is reduced. In this paper the parallel algorithm to obtain the linear relationship for the adaptive Johansson algorithm using the Lagrange-Euler formulation is presented.

The structure of the paper is the following: In section two the dynamic model is presented. Section three describes the Johansson adaptive control algorithm. Section four presents the parallel algorithm to obtain the linear relationship for the adaptive Johansson algorithm. The results for a Puma robot are described in section five. And finally the conclusions of this paper are presented in the last section.

2 The Dynamic Model

The dynamic equation of rigid manipulators with n arms in matrix form is

$$\tau = D(q)\ddot{q} + h(q, \dot{q}) + c(q), \quad (2)$$

where τ is the $n \times 1$ vector of nominal driving torques, q is the $n \times 1$ vector of nominal generalized coordinates, $\dot{q}$ and $\ddot{q}$ are the $n \times 1$ vectors of the first and second derivatives of the vector q respectively, D is the inertia matrix, $h(q, \dot{q})$ is the vector of centrifugal and Coriolis forces and $c(q)$ is the vector of gravitational forces.

3 Johansson Adaptive Control Algorithm

The desired reference trajectory followed by the manipulator is assumed to be available as bounded functions of time in terms of joint accelerations $\ddot{q}_r$, angular velocities $\dot{q}_r$, and angular positions q_r . A stable non linear reference model is also possible if the errors of accelerations, velocities and positions are defined as

$$\begin{bmatrix} \ddot{e} \\ \dot{e} \\ e \end{bmatrix} = \begin{bmatrix} \ddot{q} - \ddot{q}_r \\ \dot{q} - \dot{q}_r \\ q - q_r \end{bmatrix}. \quad (3)$$

The control objective is to follow a given bounded reference trajectory q_r , with no position errors e or velocity errors $\dot{e}$. Let $P_{qq}, \Omega, S \in R^{n \times n}$ and $P_{\theta\theta} \in R^{r \times r}$

be positive definite matrices and define $P_{12} = P_{qq}^{-1}\Omega$. Let $Y_J \in R^{n \times r}$ and $Y_{J_0} \in R^{n \times 1}$ be defined from the relation

$$\begin{aligned} & Y_J(q_r, q, \dot{q}_r, \dot{q}, \ddot{q}_r)\theta + Y_{J_0}(q_r, q, \dot{q}_r, \dot{q}, \ddot{q}_r) \\ &= -\frac{1}{2}\dot{D}(q)(\dot{e} + P_{12}e) + D(q)(\ddot{q}_r - P_{12}\dot{e}) + h(q, \dot{q})\dot{q} + c(q). \end{aligned} \quad (4)$$

For any choice of $P_{qq} = P_{qq}^T > 0$, $P_{\theta\theta} = P_{\theta\theta}^T > 0$, $\Omega = \Omega^T > 0$, $S = S^T > 0$, the adaptive control law is given by

$$\dot{\hat{\theta}}(q_r, q, \dot{q}_r, \dot{q}, \ddot{q}_r) = -P_{\theta\theta}^{-1}Y_J^T(\dot{e} + P_{12}e), \quad (5)$$

$$\tau(q_r, q, \dot{q}_r, \dot{q}, \ddot{q}_r, \hat{\theta}) = Y_J\hat{\theta} + Y_{J_0} - (S + P_{qq}\Omega^{-1}P_{qq})(\dot{e} + P_{12}e) + P_{qq}e. \quad (6)$$

3.1 Reformulation

Using several algebraic properties and definitions given in [6] it is possible to obtain a computable version of $Y_J(q_r, q, \dot{q}_r, \dot{q}, \ddot{q}_r)$. Considering $u = \dot{e} + P_{12}e$, the expression for the derivative of the inertial matrix is given by

$$\dot{D}(q)u = Y_{\dot{D}u}(q)\theta_{\dot{D}}, \quad (7)$$

where

$$Y_{\dot{D}u}(q) = \begin{bmatrix} rtr(B_{\dot{D}u11}) & rtr(B_{\dot{D}u12}) & \cdots & rtr(B_{\dot{D}u1n}) \\ 0 & rtr(B_{\dot{D}u22}) & \ddots & rtr(B_{\dot{D}u2n}) \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & rtr(B_{\dot{D}unn}) \end{bmatrix}, \quad (8)$$

$$\theta_{\dot{D}} = [\nu(J_1) \nu(J_2) \cdots \nu(J_n)]^T, \quad (9)$$

with

$$B_{\dot{D}uij} = \sum_{k=1}^j \sum_{l=1}^j (U_{ij} \otimes U_{lkj} + U_{lij} \otimes U_{kj}) \dot{q}_l u_k. \quad (10)$$

where J_k is the inertia tensor related to link k , U_{jk} is the effect of the movement of link k on all the points of link j and U_{kjl} is the effect of the movement of links j and l on all the points of link k . The operator rtr of a matrix is a row vector whose components are the traces of the columns in this matrix, and ν is the operator vector-column of an $m \times n$ matrix where the first m components of ν are the components of the first column of the matrix, the second m components of ν are the components of the second column of the matrix, and so on, [6].

Considering $v = \ddot{q}_r - P_{12}\dot{e}$, the expression for the inertial matrix is given by

$$D(q)v = Y_{Dv}(q)\theta_D, \quad (11)$$

where

$$Y_{Dv}(q) = \begin{bmatrix} rtr(B_{Dv11}) & rtr(B_{Dv12}) & \cdots & rtr(B_{Dv1n}) \\ 0 & rtr(B_{Dv22}) & \cdots & rtr(B_{Dv2n}) \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & rtr(B_{Dvnn}) \end{bmatrix}, \theta_D = \theta_{\dot{D}}, \quad (12)$$

with

$$B_{Dvik} = \sum_{j=1}^k (U_{ik} \otimes U_{jk}) v_j. \quad (13)$$

The expression for the centrifugal and Coriolis forces is given by

$$h(q, \dot{q}) = Y_h(q, \dot{q}) \theta_h$$

where

$$Y_h(q, \dot{q}) = \begin{bmatrix} rtr(B_{h11}) & rtr(B_{h12}) & \cdots & rtr(B_{h1n}) \\ 0 & rtr(B_{h22}) & \cdots & rtr(B_{h2n}) \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & rtr(B_{hnn}) \end{bmatrix}, \theta_h = \theta_{\dot{D}}, \quad (14)$$

with

$$B_{hij} = \sum_{k=1}^j \sum_{l=1}^j (U_{ij} \otimes U_{lkj}) \dot{q}_k \dot{q}_l. \quad (15)$$

The vector of gravitational forces can be expressed as a linear relationship with the inertial parameters

$$c(q) = Y_c(q) \theta_c, \quad (16)$$

where

$$Y_c(q) = - \begin{bmatrix} Y_{c11} & Y_{c12} & \cdots & Y_{c1n} \\ 0 & Y_{c22} & \cdots & Y_{c2n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & Y_{cnn} \end{bmatrix} = - \begin{bmatrix} g^T U_{11} & g^T U_{12} & \cdots & g^T U_{1n} \\ 0 & g^T U_{22} & \cdots & g^T U_{2n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & g^T U_{nn} \end{bmatrix}, \quad (17)$$

$$\theta_c = [m_1 \bar{r}_1 \ m_2 \bar{r}_2 \ \cdots \ m_n \bar{r}_n]^T, \quad (18)$$

where $\bar{r}_i$ is the position of the centre of mass of link i (m_i) with respect to the origin of coordinates of link i .

From the previous results, the computable version of Y_J is given by

$$Y_J = -\frac{1}{2} Y_{\dot{D}u} + Y_{Dv} + Y_h + Y_c. \quad (19)$$

The next section describes the parallel algorithm to obtain expressions (5), (6) and (19).

4 Parallel Algorithm

The parallel algorithm to obtain (19) is presented below, where n is the number of links and p is the number of processors. To determine the calculations of each processor, two parameters have been defined, i_k , the initial link, and f_k the final link of processor P_k . Then P_k computes the operations of the links $i_k, i_{k+1}, \dots, f_k$. In this case:

- Processor P_1 has the values $i_1 = 1$ and $f_1 = n - p + 1$ to obtain the matrices and vectors involved in the Johansson parallel algorithm.
- Processors $P_k, k = 2 : p$, have the values $i_k = f_k = n - p + k$ to obtain the matrices and vectors involved in the Johansson parallel algorithm.

To obtain (19) matrices Y_{Du}, Y_{Dv}, Y_h and Y_c can be computed. These matrices are calculated using matrices U_{ij} (structure U Eq. (20)) and matrices U_{ijk} (structure DU Eq. (22)). Matrices U_{ij} mean the effect of movement of link j on all the points of link i , and matrices U_{ijk} mean the effect of movement of links j and k on all the points of link i . The structure U is given by

$$U = \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1n} \\ & U_{22} & \cdots & U_{2n} \\ & & \ddots & \vdots \\ & & & U_{nn} \end{bmatrix} = \begin{bmatrix} Q_1 {}^0A_1 & Q_1 {}^0A_2 & \cdots & Q_1 {}^0A_n \\ & {}^0A_1 Q_2 {}^1A_2 & \cdots & {}^0A_1 Q_2 {}^1A_n \\ & & \ddots & \vdots \\ & & & {}^0A_{n-1} Q_n {}^{n-1}A_n \end{bmatrix}, \quad (20)$$

where Q_i is the constant matrix that allows us to calculate the partial derivative of ${}^iA_j$, the transformation matrix of a robot manipulator. To obtain U , matrices ${}^iA_j$ (structure A Eq. (21)) are necessary. This structure is given by

$$A = \begin{bmatrix} {}^0A_1 & {}^0A_2 & \cdots & {}^0A_n \\ & {}^1A_2 & \cdots & {}^1A_n \\ & & \ddots & \vdots \\ & & & {}^{n-1}A_n \end{bmatrix}, \quad (21)$$

where ${}^iA_j = {}^iA_{j-1} {}^{j-1}A_j, i = 0 : n - 2$. The matrices of the diagonal, ${}^{i-1}A_i, i = 1 : n$, are obtained from the robot parameters [3]. In the parallel algorithm P_k computes ${}^iA_j, i = 0 : f_k - 1, j = i + 1 : f_k$. This is the unique case where the values of parameters i_k and f_k are different from the parameters defined previously, in this case:

- Processor P_1 has the values $i_1 = 1$ and $f_1 = n - p + 1$ to obtain the matrices of A .
- Processors $P_k, k = 2 : p$, have the values $i_k = 1$ and $f_k = n - p + k$ to obtain the matrices of A .

Processor P_k needs ${}^0A_i, i = i_k : f_k$, to obtain matrices $U_{1i}, i = i_k : f_k$. To obtain the remaining matrices of $U, U_{ij}, j = i_k : f_k, i = 2 : j$, each processor needs ${}^0A_{i-1}$ and ${}^{i-1}A_j, j = i_k : f_k, i = 2 : j$. All these matrices have been computed previously.

To obtain matrices U_{ijk} the following structure, DU , is defined

$$DU = [DU_1 \ DU_2 \ \cdots \ DU_n]^T. \quad (22)$$

As $U_{kjl} = U_{jkl}$, it is only necessary to compute the following block of DU_i , $i = 1 : n$

$$DU_i(i : n, i : n) = \begin{bmatrix} U_{iii} & U_{iii+1} & \cdots & U_{iin} \\ 0 & U_{ii+1i+1} & \cdots & U_{ii+1n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{inn} \end{bmatrix}. \quad (23)$$

Given that P_k , $k = 1 : p$, has the required U matrices, it computes $U_{1ij} = Q_1 U_{ij}$, $j = i_k : f_k$, $i = 1 : j$.

To obtain the remaining matrices of DU , P_k , $k = 1 : p$, computes U_{ijk} , $k = i_k : f_k$, $i = 2 : k$ and $j = i : k$.

There are two situations:

- To obtain the matrices of row i , $U_{ij} = V_i Q_i {}^{i-1}A_j$, processor P_k needs V_i , $i = 2 : f_k$, and ${}^{i-1}A_j$, $j = i_k : f_k$, $i = 2 : j$. These matrices have been computed previously.
- To obtain the matrices of row $l > i$, $U_{ilj} = U_{il-1} Q_l {}^{l-1}A_j$, ${}^{l-1}A_j$ is first computed by P_k . But U_{il-1} has been computed in another processor. In order to avoid communications P_k replicates the computation of this matrix.

Then, each processor calculates the block of columns of the structures U and DU corresponding to rank $[i_k : f_k]$. P_k , $k = 1 : p$, computes $rtr(B_{\dot{D}uij})$, $rtr(B_{Dvij})$, $rtr(B_{hij})$ and Y_{cij} , $j = i_k : f_k$ and $i = 1 : j$.

As the processors have all the information to compute $B_{\dot{D}}$, B_D and B_h , no communication among them is necessary. To obtain matrix Y_c no communication is necessary because each processor has the required U matrices. With this information P_k computes expression (19).

To obtain (5), each processor P_k computes the following expression

$$\dot{\hat{\theta}}_i = \sum_{j=1}^i -P_{\theta\theta ii}^{-1} Y j_{ji}^T u_j, \quad (24)$$

for $i = i_k : f_k$.

And finally, the control law $\tau = Y_j \hat{\theta} - (S + P_{qq} \Omega^{-1} P_{qq})u + P_{qq}e$ must be computed. $P_{\theta\theta}$ and Ω can be considered as diagonal matrices. Each processor P_k computes τ_i , $i = 1 : f_k$, using the matrices $Y j_{ij}$ and $\hat{\theta}_i$ that it has calculated.

Each processor sends the computed vector τ_i to processor P_p . This processor receives these values and it obtains the final value of τ . This is the only communication in the algorithm. The term $(S + P_{qq} \Omega^{-1} P_{qq})u + P_{qq}e$ is also computed in P_p .

5 Experimental Results

The sequential algorithm is evaluated using sequential execution time, T_s . The parallel algorithms are evaluated using parallel execution time T_p (p processors), Speed-up, $S_p = T_1/T_p$ and efficiency, $E_p = S_p/p$. The results have been obtained using the parameters of a Puma 600 robot with six links. In the parallel algorithms 2, 3 and 4 processors have been used. In each case the links have been distributed among the processors according to i_k and f_k parameters. To present the results the following notation is used: $pxax \cdots x$, where px is the number of processors used and $ax \cdots x$ is the number of links computed by each processor. For example, in $p2a31$, the first three links are calculated in P_1 and the fourth link is computed in P_2 .

A Beowulf cluster with 32 nodes connected via Myrinet switch has been used. Each node is an Intel Pentium-II processor at 300MHz with 128 MBytes RAM. Communication routines in MPI and C language are used. Table (1) shows the results, in milliseconds, of computing the parallel algorithm of the adaptive Johansson control where $T_s = 9.14$, using this cluster.

Table 1. Experimental results with $n = 6$ links when a Beowulf cluster is used.

Algorithm	p	T_p	Speed-up	Efficiency
p2a33	2	8.36	1.085	54.67%
p2a42	2	7.1	1.286	64.31%
p2a51	2	5.446	1.678	83.94%
p3a222	3	5.5	1.65	55.33%
p3a321	3	4.86	1.88	62.69%
p3a411	3	4.86	1.88	62.69%
p4a3111	4	4.922	1.857	46.43%

The best efficiency is obtained using two processors, where the first processor computes links one to five and the second processor computes the last link. The shortest execution time is obtained using three processors, where the first processor calculates links one to four, the second processor computes the fifth link, and the last link is computed by processor three. Execution time increases when four processors are used because the load balance is not good. The objective is to reduce execution time even though efficiency is not good.

6 Conclusions

Given the generalized formulation for the linear relationship between variable dynamic and inertial terms it is possible to apply this formulation to the Johansson adaptive algorithm using the Lagrange-Euler formulation. Although the use of

the Lagrange-Euler formulation has a high computational cost, it is possible to reduce it in two ways:

- Eliminating the high quantity of null terms and exploiting the properties of the matrices.
- Using parallel computing to obtain the different matrices of the dynamic equation and the linear relationship of the Johansson adaptive algorithm.

Using these two techniques it is possible to obtain the linear relationship of the Johansson adaptive algorithm and apply it to an on-line identification because we have reduced the time requirements. The two techniques can be used in other adaptive algorithms, such as Slotine-Li, and in optimal control. The shortest execution time is obtained using three processors, and the best efficiency is obtained with two. The parallel algorithm can be used for a robot manipulator with more than two links. In this paper this formulation is applied to a six links Puma manipulator.

References

1. Craig, J.: Adaptive Control of Mechanical Manipulators, Addison-Wesley (1988).
2. Fernández, J.C: Simulación Dinámica y Control de Robots Industriales Utilizando Computación Paralela, Ph.D. Univ. Politécnica de Valencia (1999).
3. Fu, K.S., González, R.C., Lee, C.S.G: Robotics: Control, Sensing, Vision and Intelligence, New York, McGraw-Hill, 580 pages (1987).
4. Johansson, R.: Adaptive Control of Robot Manipulator Motion, IEEE Transactions on Robotics and Automation, 4(6), 483–490 (1990).
5. Ortega, J.M., Spong M.: Adaptive Motion Control of Rigid Robots: A Tutorial, Automatica 25(6), 877–888 (1989).
6. Peñalver, L.: Modelado Dinámico e Identificación Paramétrica para el Control de Robots Manipuladores, Ph.D. Univ. Politécnica de Valencia (1998).
7. Slotine, J.J., Li, W.: On Adaptive Control of Robot Manipulators, International Journal Robotics Research, 6(3), 49–59 (1987).
8. Zomaya, A.Y.: Modelling and Simulation of Robot Manipulators. A Parallel Processing Approach, World Scientific Series in Robotics and Automated Systems, 8, (1992).

Interactive Ray Tracing on Commodity PC Clusters

State of the Art and Practical Applications

Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek

Saarland University, Germany



Abstract. Due to its practical significance and its high degree of parallelism, ray tracing has always been an attractive target for research in parallel processing. With recent advances in both hardware and software, it is now possible to create high quality images at interactive rates on commodity PC clusters.

In this paper, we will outline the “state of the art” of interactive distributed ray tracing based on a description of the distributed aspects of the OpenRT interactive ray tracing engine. We will then demonstrate its scalability and practical applicability based on several example applications.

1 Introduction

The ray tracing algorithm is well-known for its ability to generate high quality images but has also been infamous for its long rendering times. Speeding up ray tracing for interactive use has been a long standing goal for computer graphics research. Significant efforts have been invested, mainly during the 1980ies and early 90ies, as documented for example in [11].

For users of graphical applications the availability of real-time ray tracing offers a number of interesting benefits: The ray tracing algorithm closely models the physical process of light propagation by shooting imaginary rays into the scene. Thus, it is able to accurately compute global and advanced lighting and shading effects. It exactly simulates shadows, reflection, and refraction on arbitrary surfaces even in complex environments (see Figures 1 and 2). Furthermore, ray tracing automatically combines shading effects from multiple objects in the correct order. This allows for building the individual objects and their shaders independently and have the ray tracer automatically take care of correctly rendering the resulting combination of shading effects (cf. Section 3.1). This feature is essential for robust industrial applications, but is not offered by current graphics hardware. Finally, ray tracing efficiently supports huge models with billions of polygons showing a logarithmic time complexity with respect to scene size, i.e. in the number of triangles in a scene (cf. Section 3.2). This efficiency is due to inherent pixel-accurate *occlusion culling* and *demand driven* and *output-sensitive* processing that computes only visible results.

1.1 Fast and Parallel Ray Tracing

However, ray tracing is a very costly process, since it requires to trace millions of rays into a virtual scene and to intersect those with the geometric primitives. In order to improve performance to interactive rates requires to combine highly optimized ray tracing implementations with massive amounts of computational power.

Due to its high degree of parallelism, together with its practical significance for industrial applications (e.g. for lighting simulation, visualization, and for the motion picture industry), especially parallel ray tracing has attracted significant research, not only from the graphics community (e.g. [16,5]), but also from the parallel computing community (e.g. [15]). Muuss et al. [13] and Parker et al. [14] were the first to show that interactive ray tracing is possible by massive parallelization on large shared memory supercomputers. More recently, Wald et al. [19] demonstrated that interactive frame rates can also be achieved on commodity PC clusters. In their research, Wald et. al have accelerated software ray tracing by more than a factor of 15 compared to other commonly used ray tracers. This has been accomplished by algorithmic improvements together with an efficient implementation designed to fit the capabilities of modern processors. For instance, changing the standard ray tracing algorithm to tracing packets of rays and to perform computations within each packet in breadth first order improves coherence, and enables efficient use of data parallel SIMD extensions (e.g. SSE) of modern processors. Paying careful attention to coherence in data access also directly translates to better use of processor caches, which increasingly determines the runtime of today's programs. The combination of better usage of caches and SIMD extensions is crucial to fully unfold the potential of today's CPUs, and will probably be even more important for future CPU designs.

Even though these optimizations allow some limited amount of interactive ray tracing even on a single modern CPU, one PC alone still cannot deliver the performance required for practical applications, which use complex shading, shadows, reflections, etc. In order to achieve sufficient performance on today's hardware requires combining the computational resources of multiple CPUs.

As has already been shown by Parker et al. [14], ray tracing scales nicely with the number of processors, but only if *fast access to scene data* is provided, e.g. by using shared memory systems. Today, however, the most cost-effective approach to compute power is a distributed memory PC cluster. Unfortunately they provide only low bandwidth with high latencies for data access across the network. In a related publication, Wald et al. [21] have shown how interactive ray tracing can be realized on such a hardware platform. In the following we briefly discuss the main issues of high-performance implementations in a distributed cluster environment, by taking a closer look at the distribution framework of the OpenRT interactive ray tracing engine [18].

2 Distribution Aspects of the OpenRT Interactive Ray Tracing Engine

Before discussing some of the parallel and distribution details, we first have to take a closer look at the overall system design.

2.1 General System Design

Client-Server Approach: Even though our system is designed to run distributed on a cluster of PCs, we assume this ray tracer to be used by a single, non-parallel application running on a single PC. As a consequence, we have chosen to follow the usual client/server approach, where a single master centrally manages a number of slave machines by assigning a number of tasks to each client. The clients then perform the actual ray tracing computations, and send their results back to the server in the form of readily computed, quantized color values for each pixel.

Screen Space Task Subdivision: Effective parallel processing requires to break the task of ray tracing into a set of preferably independent subtasks. For predefined animations (e.g. in the movie industry), the usual way of parallelization is to assign different frames to different clients in huge render farms. Though this approach successfully improves throughput, it is not applicable to a real-time setting, where only a single frame is to be computed at any given time.

For real-time ray tracing, there are basically two approaches: *object space* and *screen space* subdivision [16,5]. Object space approaches require the entire scene data-base to be distributed across a number of machines, usually based on an initial spatial partitioning scheme. Rays are then forwarded between clients depending on the next spatial partition pierced by the ray. However, the resulting network bandwidth would be too large for our commodity environment. At today's ray tracing performance individual rays can be traced much faster than they can be transferred across a network. Finally, this approach often tends to create *hot-spots* (e.g. at light sources that concentrate many shadow rays), which would require dynamic redistribution of scene data.

Instead, we will follow the screen-based approach by having the clients compute disjunct regions of the same image. The main disadvantage of screen-based parallelization is that it usually requires a local copy of the whole scene to reside on each client, whereas splitting the model over several machines allows to render models that are larger than the individual clients' memories. In this paper, we do not consider this special problem, and rather assume that all clients can store the whole scene. In a related publication however, it has been shown how this problem can be solved efficiently by caching parts of the model on the clients (see [21]).

Load Balancing: In screen space parallelization, one common approach is to have each client compute every n -th pixel (so-called pixel-interleaving), or every n -th row or scanline. This usually results in good load balancing, as all clients get roughly the same amount of work. However, it also leads to a severe loss of *ray coherence*, which is a key factor for fast ray tracing. Similarly, it translates to bad cache performance resulting from equally reduced *memory coherence*.

An alternative approach is to subdivide the image into quadrangular regions (called *tiles*) and assign those to the clients. Thus, clients work on neighboring pixels that expose a high degree of coherence. The drawback is that the cost for computing different tiles can significantly vary if a highly complex object (such as a complete power plant as shown in Figure 2) projects onto only a few tiles, while other tiles are empty. For *static task assignments* – where all tiles are distributed among the clients before any actual

computations – this variation in task cost would lead to extremely bad client utilization and therefore result in bad scalability.

Thus, we have chosen to use a tile-based approach with a dynamic load balancing scheme in our system: Instead of assigning all tiles in advance, corresponding to a *data driven* approach, we pursue a *demand driven* strategy by letting the clients themselves ask for work. As soon as a client has finished a tile, it sends its results back to the server, and requests the next unassigned tile from the master.

Hardware Setup: To achieve the best cost-effective setting, we have chosen to use a cluster of dual-processor PCs interconnected by commodity networking equipment. Currently, we are using up to 24 dual processor AMD AthlonMP 1800+ PCs with 512 MB RAM each. The nodes are interconnected by a fully switched 100 Mbit Ethernet using a single Gigabit uplink to the master display and application server for handling the large amounts of pixel data generated in each image. Note, that this hardware setup is not even state of the art, as much faster processors and networks are available today.

2.2 Optimization Details

While most of the above design issues are well-known and are applied in similar form in almost all parallel ray tracing systems, many low-level details have to be considered in order to achieve good client utilization even under interactivity constraints. Though we can not cover all of them here, we want to discuss the most important optimizations used in our system.

Communication Method: For handling communication, most parallel processing systems today use standardized libraries such as MPI [8] or PVM [10]. Although these libraries provide very powerful tools for development of distributed software, they do not meet the efficiency requirements that we face in an interactive environment.

Therefore, we had to implement all communication from scratch with standard UNIX TCP/IP calls. Though this requires significant efforts, it allows to extract the maximum performance out of the network. For example, consider the 'Nagle' optimization implemented in the TCP/IP protocol, which delays small packets for a short time period to possibly combine them with successive packets to generate network-friendly packet sizes. This optimization can result in a better throughput when lots of small packets are sent, but can also lead to considerable latencies, if a packet gets delayed several times. Direct control of the systems communication allows to use such optimizations selectively: For example, we turn the Nagle optimization on for sockets in which updated scene data is streamed to the clients, as throughput is the main issue here. On the other hand, we turn it off for e.g. sockets used to send tiles to the clients, as this has to be done with an absolute minimum of latency. A similar behavior would be hard to achieve with standard communication libraries.

Differential Updates: Obviously, the network bandwidth is not high enough for sending the entire scene to each client for every frame. Thus, we only send differential updates from each frame to the next: Only those settings that have actually changed from the previous frame (e.g. the camera position, or a transformation of an object) will be sent

to the clients. Upon starting a new frame, all clients perform an update step in which they incorporate these changes into their scene database.

Asynchronous Rendering: Between two successive frames, the application will usually change the scene settings, and might have to perform considerable computations before the next frame can be started. During this time, all clients would run idle. To avoid this problem, rendering is performed asynchronously to the application: While the application specifies frame N , the clients are still rendering frame $N - 1$. Once the application has finished specifying frame N , it waits for the clients to complete frame $N - 1$, displays that frame, triggers the clients to start rendering frame N , and starts specifying frame $N + 1$. Note, that this is similar to usual *double-buffering* [17], but with one additional frame of latency.

Asynchronous Communication: As just mentioned, the application already specifies the next frame while the clients are still working on an old one. Similarly, all communication between the server and the clients is handled asynchronously: Instead of waiting for the application to specify the complete scene to be rendered, scene updates from the application are *immediately* streamed from the server to all clients in order to minimize communication latencies. Asynchronously to rendering tiles for the old frame, one thread at the client already receives the new scene settings and buffers them for future use. Once the rendering threads have finished, (most of) the data for the next frame has already arrived, further minimizing latencies. These updates are then integrated into the local scene database, and computations can immediately be resumed without losing time receiving scene data.

Multithreading: Due to a better cost/performance ratio, each client in our setup is a dual-processor machine. Using multithreading on each client then allows to share most data between these threads, so the cost of sending scene data to a client can be amortized over two CPUs. Furthermore, both client and server each employ separate tasks for handling network communication, to ensure minimum network delays.

Task Prefetching: If only a single tile is assigned to each client at any time, a client runs idle between sending its results back to the server and receiving the next tile to be computed. As a fast ray tracer can compute tiles in at most a few milliseconds, this delay can easily exceed rendering time, resulting in extremely bad client utilization. To avoid these latencies, we let each client request (“prefetch”) several tiles in advance. Thus, several tiles are ‘in flight’ towards each client at any time. Ideally, a new tile is just arriving every time a previous one is sent on to the server. Currently, each client is usually prefetching about 4 tiles. This, however depends on the actual ratio of compute performance to network latency, and might differ for other hardware configurations.

Spatio-Temporal Coherence: In order to make best use of the processor caches on the client machines load balancing also considers the *spatio-temporal coherence* between rays by assigning the same image tiles to the same clients in subsequent frames whenever possible. As soon as a client has processed all of ‘its’ old tasks, it starts to ‘steal’ tasks from a random other machine.

3 Applications and Experiments

In the following, we demonstrate the potential and scalability of our system based on several practical examples. If not mentioned otherwise, all experiments will run at video resolution of 640×480 pixels. As we want to concentrate on the practical results, we will but closely sketch the respective applications. For more details, also see [18,20,3].

3.1 Classical Ray Tracing

Plug’n Play Shading: One of the main advantages of ray tracing is its potential for unsurpassed image realism, which results from its ability to support specialized shader programs for different objects (e.g. a glass shader), which can then easily be combined with shaders on other objects in a plug and play manner. This is also the reason why ray tracing is the method of choice of almost all major rendering packages. The potential of this approach can be seen in Figure 1: An office environment can be simulated with several different shaders which fit together seamlessly. For example, a special shader realizing volume rendering with a skull data set seamlessly fits into the rest of the scene: i.e. it is correctly reflected in reflective objects, and casts transparent shadows on all other objects. Using our distribution framework, the ray tracer scales almost linearly up to 48 CPUs, and achieves frame rates of up to 8 frames per second (see Figure 4).

Visualisation of Car Headlights for Industrial Applications: Being able to efficiently simulate such advanced shading effects also allows to offer solutions to practical industrial problems that have so far been impossible to tackle. For example, the limitations of todays graphics hardware force current Virtual Reality (VR) systems to make heavy use of approximation, which in turn makes reliable quantitative results impossible, or at least hard to achieve. Examples are the visualization of reflections of the car’s dash board in the side window where it might interfere with backward visibility through the outside mirror at night. Another example is the high-quality visualization of car headlights that are important design features due to being the “eyes of the car”.

Figure 1 shows an 800.000-triangle VRML model of a car headlight rendered with our ray tracing engine [3]. Special reflector and glass shaders are used that carefully model the physical material properties and intelligently prune the ray trees. The latter is essential for achieving real-time frame rates because the ray trees otherwise tend to get very large due to many recursive reflections and refractions. For good visual results we still have to simulate up to 25 levels of recursion for certain Pixels (cf. Figure 1d).

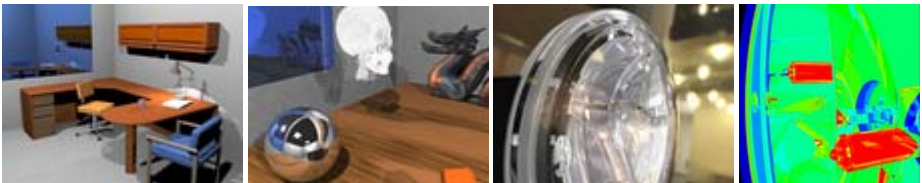


Fig. 1. Classical ray tracing: (a) Typical office scene, with correct shadows and reflections, and with programmable procedural shaders. (b) The same scene with volume and lightfield objects. Note how the volume casts transparent shadows, the lightfield is visible through the bump-mapped reflections on the mirror, etc. (c) The headlight model with up to 25 levels of reflection and refraction. (d) False-color image showing the number of reflection levels per pixel (red: 25+).

Simulating this level of lighting complexity is currently impossible to compute with alternative rendering methods. Thus, for the first time this tool allows automotive designers and headlight manufacturers to interactively evaluate a design proposal. This process previously took several days for preparing and generating predefined animations on video. Now a new headlight model can be visualized in less than 30 minutes and allows designers the long missing option of freely interacting with the model for exploring important optical effects.

3.2 Interactive Visualization of Highly Complex Models

Apart from its ability to simulate complex lighting situations, another advantage of ray tracing is that in practice it's time complexity is *logarithmic* in scene size [12], allowing to easily render even scenes with several million up to billions of triangles. This is of great importance to VR and engineering applications, in which such complex models have to be visualized. Currently, such models have to be simplified before they can be rendered interactively. This usually requires expensive preprocessing [1,2], significant user intervention, and often negatively affects the visualization quality. With our distributed ray tracing system, we can render such models interactively (see Figures 2 and 4), and – even more importantly – without the need for geometric simplifications.

Figure 2 shows screenshots from two different models: The image on the left shows three complete power plants consisting of 12.5 million triangles each. Using our ray tracer, we can easily render several such power plants at the same time, and can even interactively move parts of the power plant around. Especially the latter is very important for design applications, but is usually impossible with alternative technologies, as simplification and preprocessing usually work only in static environments.

Yet another advantage of ray tracing is its possibility to use *instantiation*, the process of re-using parts of a model several times in the same scene. For example, the second model in Figure 2 consists of only ten different kinds of sunflowers (of roughly 36,000 triangles each) and one type of tree, which are then instantiated several thousand times to form a complete landscape of roughly one *billion* triangles. Furthermore, the scene is rendered including transparency textures for the leaves, and computes even pixel-accurate shadows cast by the sun onto the leaves (see Figures 2c and 2d). Using our framework, both scenes can be rendered interactively, and achieve almost-linear scalability, as can be seen in Figures 2 and 4.



Fig. 2. Complex models: (a) Three powerplants of 12.5 million individual triangles each, rendering interactively at 23 fps. (b) A closeup on the highly detailed geometry. (b) An outdoor scene consisting of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles and is rendered including shadows and transparency. (d) A closeup of the highly detailed shadows cast by the sun onto the leaves.

3.3 Interactive Lighting Simulation

Even though classical ray tracing as described above considers only direct lighting effects, it already allows for highly realistic images that made ray tracing the preferred rendering choice for many animation packages. The next step in realism can be achieved by including indirect lighting effects computed by *global illumination* algorithms as a standard feature of 3D graphics (see Figure 3).

Global illumination algorithms account for the often subtle but important effects of indirect lighting effects in a physically-correct way [6,7] by simulating the global light transport between all mutually visible surfaces in the environment. Due to the need for highly flexible visibility queries, virtually all algorithms today use ray tracing for this task. Because of the amount and complexity of the computations, rendering with global illumination is usually even more complex than classical ray tracing, and thus slow and far from interactive, taking several minutes to hours even for simple diffuse environments. The availability of real-time ray tracing should now enable to compute full global illumination solutions also at interactive rates (see Figure 4). We will not cover the details of the algorithm here, which are described in close detail in [20] and [4].

Using our system, we can interactively simulate light propagation in a virtual scene, including soft shadows, reflections and indirect illumination. As each frame is recomputed from scratch, interactive changes to the environment can be handled well, allowing to modify geometry, lights and materials at interactive rates.

4 Results and Conclusions

In this paper, we have shown how efficient parallelization of a fast ray tracing kernel on a cluster of PCs can be used to achieve interactive performance even for high-quality applications and massively complex scenes. We have sketched the parallelization and distribution aspects of the OpenRT distributed interactive ray tracing engine [18], which uses screen-space task subdivision and demand-driven load balancing, together with low-level optimization techniques to minimize bandwidth and to hide latencies. The combination of these techniques allow the system to be used even in a real-time setting, where only a few milliseconds are available for each frame. We have evaluated the performance of the proposed system (see Figure 4) in a variety of test scenes, for three different applications: Classical ray tracing, visualising massively complex models, and interactive lighting simulation: Due to the limited bandwidth of our display server (which

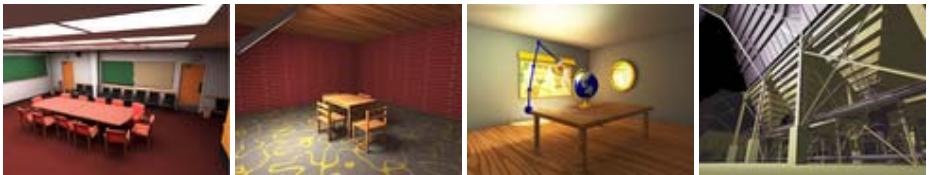


Fig. 3. An interactive global illumination application in different environments. From left to right: (a) A conference room of 280.000 triangles, (b) The “Shirley 6” scene, with global illumination including complex procedural shaders, (c) An animated VRML model, and (d) global illumination in the power plant with 37.5 million triangles. All scenes render at several frames per second.

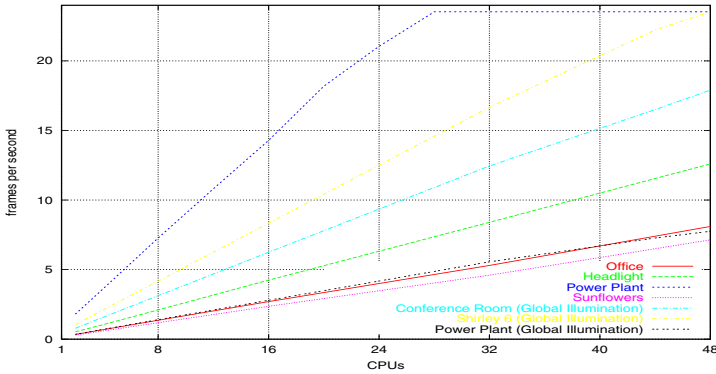


Fig. 4. Scalability of our system on a cluster of up to 48 CPUs. Note that the system is limited to about 22–24 frames per second due to the limited network connection of the display server. Up to this maximum framerate, all scenes show virtually linear scalability.

is connected to the Gigabit uplink of the cluster), our framerate is limited to roughly 22–24 frames per second at 640×480 pixels, as we simply cannot transfer more pixels to the server for display. Up to this maximum framerate however, all scenes show virtually linear scalability for up to 48 CPUs (see Figure 4).

The presented applications have also shown that a fast and distributed software implementation of ray tracing is capable of delivering completely new types of interactive applications: While today's graphical hardware solutions (including *both* the newest high-performance consumer graphics cards *and* expensive graphics supercomputers) can render millions of triangles per second, they can not interactively render whole scenes of many million to even billions of triangles as shown in Section 3.2.

Furthermore, such hardware architectures can not achieve the level of image quality and simulation quality that we have shown to be achievable with our system (see Section 3.1). This is especially true for interactive lighting simulation (Section 3.3), which – due to both its computational cost and its algorithmic complexity – is unlikely to be realized on graphics hardware any time soon. With this capability of enabling completely new applications, our system provides real value for practical applications, and is already being used in several industrial projects.

Still, typical VR applications demand even higher resolutions and even higher frame rates. Typically, resolutions up to 1600×1200 are required to drive equipment such as PowerWalls etc, and 'real-time' applications usually demand frame rates of 25 frames per second and more. This leaves enough room for even more parallelization, and also requires to eventually think about more powerful network technologies (such as Myrinet [9]) to provide the required network performance.

Acknowledgements. The RTTR/OpenRT project has been supported by Intel Corp. The Sunflowers and Powerplant models have been graciously provided by Oliver Deussen and Anselmo Lastra.

References

1. D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration. In *ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, April 1999.
2. William V. Baxter III, Avneesh Sud, Naga K Govindaraju, and Dinesh Manocha. Gigawalk: Interactive Walkthrough of Complex Environments. In *Rendering Techniques 2002*, pages 203–214, June 2002. (Proceedings of the 13th Eurographics Workshop on Rendering 2002).
3. Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive Headlight Simulation – A Case Study for Distributed Interactive Ray Tracing. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002.
4. Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. to be published at Eurographics 2003, 2003.
5. Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. AK Peters, 2002. ISBN 1-56881-179-9.
6. Micheal F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, 1993. ISBN: 0121782700.
7. Philip Dutre, Kavita Bala, and Philippe Bekaert. Advanced Global Illumination. In *SIGGRAPH 2001 Course Notes, Course 20*. 2001.
8. MPI Forum. MPI – The Message Passing Interface Standard. <http://www-unix.mcs.anl.gov/mpi>.
9. Myrinet Forum. Myrinet. <http://www.myri.com/myrinet/overview/>.
10. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, 1994.
11. Andrew Glassner. *An Introduction to Raytracing*. Academic Press, 1989.
12. Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001.
13. Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.
14. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics (I3D)*, pages 119–126, 1999.
15. Tomas Plachetka. Perfect Load Balancing for Demand-Driven Parallel Ray Tracing. In B. Monien and R. Feldman, editors, *Lecture Notes in Computer Science*, pages 410–419. Springer Verlag, Paderborn, August 2002. (Proceedings of Euro-Par 2002).
16. Erik Reinhard. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995.
17. B. Schachter. *Computer Image Generation*. Wiley, New York, 1983.
18. Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at <http://graphics.cs.uni-sb.de/Publications>.
19. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics 2001).
20. Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).
21. Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. *Rendering Techniques 2001*, pages 274–285, 2001. (Proceedings of the 12th Eurographics Workshop on Rendering).

Toward Automatic Management of Embarrassingly Parallel Applications

Inês Dutra¹, David Page², Vitor Santos Costa¹, Jude Shavlik², and Michael Waddell²

¹ Dep of Systems Engineering and Computer Science
Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil
{ines,vitor}@cos.ufrj.br

² Dep of Biostatistics and Medical Informatics,
University of Wisconsin-Madison, USA
{page,shavlik,mwaddell}@biostat.wisc.edu

Abstract. Large-scale applications that require executing very large numbers of tasks are only feasible through parallelism. In this work we present a system that automatically handles large numbers of experiments and data in the context of machine learning. Our system controls all experiments, including re-submission of failed jobs and relies on available resource managers to spawn jobs through pools of machines. Our results show that we can manage a very large number of experiments, using a reasonable amount of idle CPU cycles, with very little user intervention.

1 Introduction

Large-scale applications may require executing very large numbers of tasks, say, thousands or even hundreds of thousands of experiments. These applications are only feasible through parallelism and are nowadays often executed in clusters of workstations or in the Grid. Unfortunately, running these applications in an unreliable environment can be a complex problem. The several phases of computation in the application must be sequenced correctly: dependencies, usually arising through data written to and read from files, must be respected. Results will be grouped together, a summarised report over the whole computation should be made available. Errors, both from the application itself and from the environment, must be handled correctly. One must check whether experiments terminated successfully, and verify integrity of the output.

Most available software for monitoring applications in parallel and distributed environments, and more recently, in grid environments, concentrate on modelling and analysing hardware and software performance [8], prediction of lost cycles [9] or visualisation of parallel execution [12], to mention some. Most of them focus on parallelised applications. Few efforts have been spent on managing huge number of independent experiments and the increasing growth of interdisciplinary databases such as the ones used in biological or biomedical applications. Only recently, we have seen work in the context of the Grid such as

the GriPhyN project for Physics [1], and the development of the general purpose system Chimera [6].

In this work we present a system originally designed to support the very large numbers of experiments and data in machine learning applications. We are interested in machine learning toward data mining of relational data from domains such as biochemistry [10,4], and security [13]. Machine learning tasks are often computationally intensive. For instance, many learning systems, such as the ones that generate decision trees and logical clauses, must explore a so-called “search space” in order to find models that characterise well a set of correct (and possibly incorrect) examples. The size of the search space usually grows exponentially with the size of the problem. A single run of the learning algorithm can thus easily take hours over real data. Moreover, in order to study the effectiveness of the system one often needs to repeat experiments on different data. Splitting the original examples into different subsets or *folds* and learning on each fold is also common. Thus, a single learning task may involve several independent coarse-grained tasks, providing an excellent source of parallelism. In fact, parallelism is often the only way one can actually perform such experiments in reasonable time.

Our system has successfully run several large-scale experiments. Our goal is to extract maximum advantage of the huge parallelism available in machine learning applications. The system supports non-interactive error-handling, including re-submission of failed jobs, while allowing users to control the general performance of the application. Whenever possible, we rely on available technology: for example, we use the Condor resource manager [2] to spawn jobs through pools of machines. We have been able to run very large applications. One example included over 50 thousand experiments: in this case we consumed about 53,000 hours of CPU, but the system took only 3 months to terminate, achieving peak parallelism of 400 simultaneous machines at a time, and requiring very little user intervention.

The paper is organised as follows. First, we present in more detail the machine learning environment and its requirements. We then discuss the motivation for an automatic tool. In section 2 we present the architecture of our system and the methodology applied to the two machine learning phases: experimentation and evaluation. We then discuss some performance figures and possibilities of enhancements. Last, we offer our conclusions and suggest future work.

2 A Tool for Managing Large Numbers of Experiments

In order to be able to run the many possible combinations of experiments one needs to perform to have statistically meaningful results in machine learning applications, in a feasible time, and handle the results with least possible user intervention to avoid manual errors, we developed a tool for job management of learning applications, currently supporting Linux and Solaris environments. This tool makes use of available resource manager systems to manage idle resources available. We address the following issues: *Data Management*: each experiment

will have its own output and temporary files, but several experiments share input files. The system needs to create a directory structure such that the output and temporary files for individual experiments can be kept on separate, but easily accessed, directories for each experiment. *Control*: given a problem description our tool creates a set of build files to launch the actual jobs. Each script inputs the data required for a specific experiment, and sets the files that will be output. *Task Supervision*: Our tool must allow one to inspect successful and unsuccessful job termination. As discussed in more detail next, the probability that some jobs will fail is extremely high, so most cases of unsuccessful termination should be handled within the system. *User Interface*: a large number of results must be collected and displayed. Throughout, the user should be able to understand what is the current status of the computation, and plot the results.

In the experiments presented in this paper, we used as a resource manager, the Condor system, developed at the Computer Sciences Department of the University of Wisconsin-Madison. Condor is a specialised workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Among other things, Condor allows transparent migration of jobs from overloaded machines to idle machines and checkpointing, which permits that jobs can restart in another machine without the need to start from the beginning. These are typical tasks of a resource manager.

Putting it all Together. Our architecture is composed of two main modules: (1) the Experimentation Module and (2) the Evaluation Module. The Experimentation Module is responsible for setting up the environment to start running experiments for one or several applications, for launching jobs for **tuning** and **cross-validation**, and checking if results are terminated and complete. Once all experiments have terminated, the Evaluation Module will consult the tuning results, if any, compute accuracy functions, and plot the results. If some job leaves the queue, but the output result is incorrect or corrupted, the **check termination** program will re-submit the job either to **tuning** or to **cross-validation**.

We provide a user-friendly web interface in to the system with options for choosing different learners and learning techniques. At the moment, our system deals only with Inductive Logic Programming [11] learners, but this interface is easily extensible to other learners.

Not Everything is Neat. Several problems may arise while processing thousands or hundreds of thousands of experiments. We classify these problems as either *hardware-dependent* or *software-dependent*.

As we use off-the-shelf technology to launch our jobs, we expect such technology will deal correctly with hardware-dependent problems. In fact, the Condor system we use in our experiments deals with dynamic hardware changes such as inclusion of a new machine in the network or a machine or network failure. In the case of a new machine being included in the network, Condor updates the

central manager table to mirror this change. In the case of a machine fault, Condor employs a checkpointing mechanism to migrate the job to another “healthy” machine.

With respect to software-dependent problems, we can enumerate several sources of possible failure: jobs get lost because a daemon stops working for some reason, a job breaks because of lack of physical memory, bugs in the machine learning system, bugs in the resource manager, or even bugs at the operating system level, corrupted data due to network contention, or lack of disk space.

None of the systems that we use is totally stable: during our experiments, it has happened that the operating system was upgraded while we were performing our experiments, and the task management software was incompatible with the new upgrade. Unforeseen situations, common in large software development, can lead the execution to crash. Problems can arise from any one of these components: the machine learning system, in some cases the software that runs the machine learning system, and the resource manager. As we rely in off-the-shelf components to build our system, some of these problems will be outside our control.

We have found memory management to be a critical issue for our application. The machine learning system that we use relies on dynamic memory allocation: thus, we do not know beforehand how much memory we use, and we can expect memory requirements to grow linearly with execution time. In general, we set the experiments to only run in machines with a minimal amount of memory, say M . If we set M too low, many experiments will fail. If we set M too high, we will severely restrict parallelism. Moreover, as machines with lots of memory are likely to be more recent and powerful machines, we can expect them to be busier.

In practice we can be sure that *runs will fail*. Our approach to deal with these problems is to have a daemon that inspects the job queues, and the application output files. So, if the daemon detects that some output file is not yet generated, and the job responsible for that output is no longer in the queue, it will re-submit the job. If some output file is not yet generated and the job is taking too long to produce an output, the daemon will remove the job from the queue and re-submit the job. An output file can be generated, but be corrupted. In that case, we also need to check for output syntax to be sure the next phase will collect correct data.

Note that these steps do not need any human intervention,¹ and do not require any change to the available software being used.

3 Performance Results

We ran three sets of experiments in a period of 6 months. The experiments concerned three relational learning tasks using Inductive Logic Programming techniques. The tasks included two biocomputing problems, one concerning the

¹ Of course, if the failure rate becomes excessive, the user is informed and is allowed to terminate the experiments.

prediction of carcinogenicity tests on rodents, and the second concerning a characterisation of genes involved in protein metabolism by means of their correlation with other genes. The third task used data on a set of smuggling *events*. The goal was to detect whether two events were somehow related.

We ran our jobs in the Condor pools available at the Biostatistics and Medical Informatics and the Computer Sciences Departments of the University of Wisconsin-Madison. We used PC machines running Linux and SUN machines running Solaris. Throughout, Condor collects statistics about the progress of the jobs, but as this consumes a large amount of disk storage, not all Condor pools keep all data about jobs. Therefore in Table 1 we show statistics only for one of our pools that stores the data information about jobs progress in disk. The first column shows the month when the experiments were running. The second column shows the total number of CPU hours spent by our jobs. The third column shows the average number of jobs running, followed by the average of jobs idle, and the peak of jobs running and idle. We can observe that the system has two activity peaks in June/July and in September. During September in average the tool was able to keep almost one hundred processors busy, just in this pool. The maximum number of jobs running in this pool was around 400.

The execution priority of jobs in Condor depends on the configuration and on the pool. If a job migrates to a remote pool of machines, the priority of this job may be lowered to guarantee that local users in the remote pool are not interfered by the “foreign” jobs. The statistics shown in Table 1 are related to a remote pool.

Table 1. Statistics of Jobs

Period	Tot Alloc Time (Hours)	JobsRunning Average	JobsIdle Average	JobsRunning Peak	JobsIdle Peak
May	639.6	12.8	337.6	61.0	1057.0
Jun	39375.5	67.3	5322.8	400.0	15312.0
Jul	18406.2	70.1	2243.9	207.0	13366.0
Aug	110.1	14.9	0.1	17.0	4.0
Sep	18283.8	93.5	6713.9	397.0	11754.0
Oct	1185.8	47.0	4027.3	122.0	5933.0

In order to better understand the table, Figure 1 shows the state of the above mentioned pool during the most active month, September. The X-axis corresponds to days of the week, and the Y-axis corresponds to total machines available in the pool. The red area (bottom colour in the figure) shows user-controlled machines, the blue area in the middle, idle machines, and the green area above shows machines running Condor tasks. Pool size varies between 700 and 900 machines, most of them Linux and Solaris machines. User activity is higher during week days, where users are working on their own machine. Condor most often keeps around 400 to 500 machines busy at any one time. This provides

an insight into the maximum amount of parallelism we can take advantage of. Comparing with the results in Table 1 we can observe that there are points in time where we could actually take advantage of almost the full pool, even if the pool was shared with the whole campus.

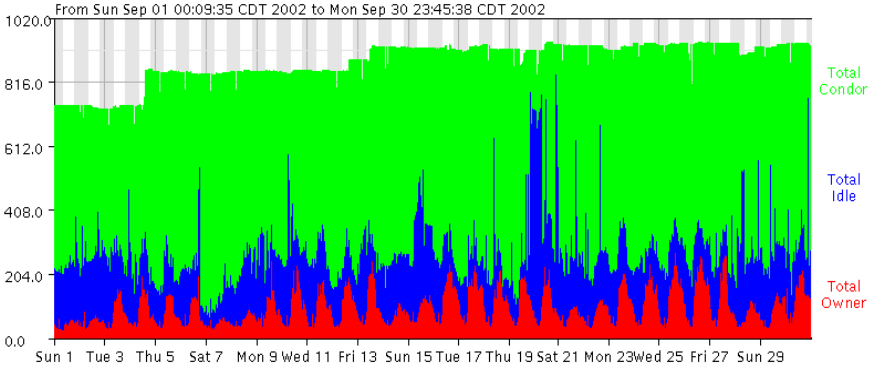


Fig. 1. The Condor Pool

Figure 2 shows that even during week days the system can provide an average of about 200 processors for long periods of time. This is about half of what we could expect from the cluster. Although we have little control over the cluster and we may be preempted by local tasks, even at the worst moments the pool can achieve above 50 machines simultaneously running our jobs.

Last, error handling was a very important issue in practice. From the 45,000 experiments we launched (three lots of 15,000 experiments for each application), around 20% failed for several reasons, and had to be re-submitted. This was done without user intervention.

4 Conclusions and Future Work

We described an automatic tool to manage large amounts of experiments in the machine learning context. The main advantage of such a tool is to provide a high-level user interface that can hide details of embarrassingly parallelisation, and that can allow for more automatic management of user experiments. Our system is capable of launching jobs automatically, check their integrity and termination, re-submit corrupted jobs, evaluate the results, plot relevant data, and inform the user about location of data and graphs. A second advantage of our tool is that we integrate off-the-shelf components in order to take advantage of already popular technologies. We used our tool to run several learning experiments with multirelational data. As an example, one of our experiments consumed about 53,000 hours of CPU, using a peak of 400 machines simultaneously.

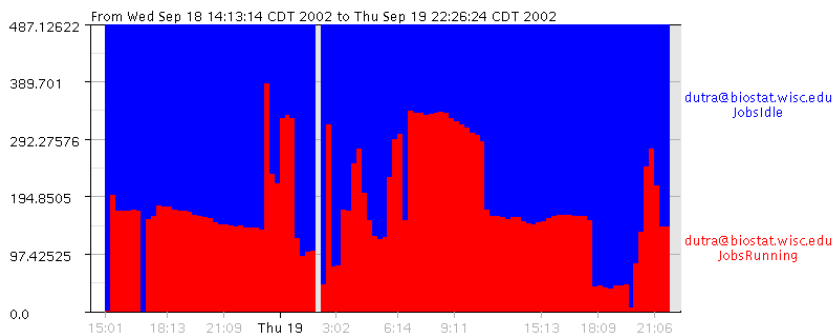


Fig. 2. Condor Activity

Most software available for parallel and distributed environments, and more recently, for grid environments, is designed to monitor applications by modelling and performance analysis, rather than managing a huge number of experiments. Condor has a limited form of handling experiments by allowing the user to express dependencies through the DAGMan tool [7]. But this tool requires that the user has some knowledge of ClassAds [14], a specification language for jobs and resources, in order to express dependencies explicitly. Chimera [6], one of the components of the Globus project [5], also has forms of automatically launching jobs depending on output produced by other jobs. Contrary to DAGMan, Chimera automatically generates a dependency execution graph, based on VDL (Virtual Data Language) [3] specifications.

Our proposal is to require minimum interference from the user, especially because many of the users in computational biology, where machine learning systems are heavily applied, have none or shallow knowledge of how to use a programming language.

As future work we intend to extend the tool to support other learning algorithms, such as the ones supported by the WEKA toolkit [15], and parallel boosting. We have been working on integrating our system with Chimera, in order to take advantage of one of the nicest features of Chimera that is to automatically infer dependencies between jobs in order to launch them without any user intervention.

Acknowledgments. This work was supported by DARPA EELD grant number F30602-01-2-0571, NSF Grant 9987841, NLM grant NLM 1 R01 LM07050-01, and CNPq. We would like to thank the Biomedical Group support staff and the Condor Team at the Computer Sciences Department for their invaluable help with Condor, Ashwin Srinivasan for his help with the Aleph system and the Carcinogenesis benchmark, and Yong Zhao from University of Chicago who has been helping us to integrate Chimera to our system.

References

1. The GriPhyN Project. White paper available at http://www.griphyn.org/documents/white_paper/index.php, 2000.
2. J. Basney and M. Livny. Managing network resources in Condor. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, Pennsylvania, pages 298–299, Aug 2000.
3. A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.
4. I. Dutra, D. Page, V. Santos Costa, and J. Shavlik. An empirical evaluation of bagging in inductive logic programming. In *Proceedings of the Twelfth International Conference on Inductive Logic Programming*, Sydney, Australia, July 2002. Springer-Verlag.
5. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.
6. I. Foster, J. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th conference on Scientific and Statistical Database Management (2002)*, 2002.
7. James Frey. Condor DAGMan: Handling Inter-Job Dependencies. <http://www.cs.wisc.edu/condor/dagman/>, 2002.
8. D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proceedings of the 8th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '00)*, 2000.
9. W. Meira Jr., T. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. In *SPDT96: SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA, ACM, pages 1–10, May 1996.
10. R. King, S. Muggleton, and M. Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647–657, 1992.
11. N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Artificial Intelligence. Ellis Horwood (Simon & Schuster), 1994.
12. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.
13. R. Mooney, P. Melville, L. P. Rupert Tang, J. Shavlik, I. Dutra, D. Page, and V. Santos Costa. Relational data mining with inductive logic programming for link discovery. In *Proceedings of the National Science Foundation Workshop on Next Generation Data Mining*, Baltimore, Maryland, USA, 2002.
14. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.
15. Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

Comparing Two Long Biological Sequences Using a DSM System

Renata Cristina F. Melo, Maria Emília Telles Walter,
Alba Cristina Magalhaes Alves Melo, Rodolfo Batista, Marcelo Nardelli,
Thelmo Martins, and Tiago Fonseca

Department of Computer Science, Campus Universitario - Asa Norte, Caixa Postal 4466,
University of Brasilia, Brasilia – DF, CEP 70910-900, Brazil
{renata, mia, albam, rodolfo, thelmo, tiago,
marcelo}@cic.unb.br

Abstract. Distributed Shared Memory systems allow the use of the shared memory programming paradigm in distributed architectures where no physically shared memory exist. Scope consistent software DSMs provide a relaxed memory model that reduces the coherence overhead by ensuring consistency only at synchronisation operations, on a per-lock basis. Sequence comparison is a basic operation in DNA sequencing projects, and most of sequence comparison methods used are based on heuristics, that are faster but do not produce optimal alignments. Recently, many organisms had their DNA entirely sequenced, and this reality presents the need for comparing long DNA sequences, which is a challenging task due to its high demands for computational power and memory. In this article, we present and evaluate a parallelisation strategy for implementing a sequence alignment algorithm for long sequences in a DSM system. Our results on an eight-machine cluster presented good speedups, showing that our parallelisation strategy and programming support were appropriate.

1 Introduction

Distributed Shared Memory (DSM) is an abstraction that allows the use of the shared memory programming paradigm in parallel or distributed architectures. The first DSM systems tried to give parallel programmers the same guarantees they had when programming uniprocessors and this approach created a huge coherence overhead [10]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared memory behaviours that are different from the traditional uniprocessor one.

In the shared memory programming paradigm, synchronisation operations are used every time processes want to restrict the order in which memory operations should be performed. Using this fact, hybrid Memory Consistency Models (MCM) guarantee that processors only have a consistent view of the shared memory at synchronisation time [10]. This allows a great overlapping of basic read and write operations that can lead to considerable performance gains. By now, the most popular MCMs for DSM systems are Release Consistency [2] and Scope Consistency [6].

JIAJIA is a scope consistent software DSM system proposed by [4] that implements consistency on a per-lock basis. When a lock is released, modifications made inside the critical section are made visible to the next process that acquires the same lock. On a synchronisation barrier, however, consistency is globally maintained and all processes are guaranteed to see all past modifications to the shared data.

In DNA sequencing projects, researchers want to compare two sequences to find similar portions of them and obtain good local sequence alignments. In practice, two families of tools for searching similarities between two sequences are widely used - BLAST [1] and FASTA[13], both based on heuristics and used for comparing long sequences. To obtain optimal local alignments, the most widely used algorithm is the one proposed by Smith-Waterman [12], with quadratic time and space complexity.

Many works are known that implement the Smith-Waterman algorithm for long sequences of DNA. Specifically, parallel implementations were proposed using MPI [9] or specific hardware [3]. As far as we know, this is the first attempt to use a scope consistent DSM system to solve this kind of problem.

In this article, we present and evaluate a parallelisation strategy for implementing the Smith-Waterman algorithm in a DSM system. Work is assigned to each processor in a column basis with a two-way lazy synchronisation protocol. An heuristic described in [11] was used to reduce the space complexity.

The results obtained in an eight-machine cluster with large sequence sizes show good speedups when compared with the sequential algorithm. For instance, to align two 400KB sequences, a speedup of 4.58 was obtained, reducing the execution time from more than 2 days to 10 hours.

The rest of this paper is organised as follows. Section 2 describes the sequence alignment problem and the optimal algorithm to solve it. In Section 3, DSM systems are presented. Section 4 describes our sequential and parallel algorithm. Some experimental results are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Smith-Waterman's Algorithm for Local Sequence Alignment

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters or substrings from the sequences [11]. We define *alignment* as the insertion of spaces in arbitrary locations along the sequences so that they finish with the same size.

Given an alignment between two sequences s and t , an score is associated for them as follows. For each column, we associate $+1$ if the two characters are identical, -1 if the characters are different and -2 if one of them is a space. The *score* is the sum of the values computed for each column. The maximal score is the *similarity* between the two sequences, denoted by $sim(s,t)$. In general, there are many alignments with maximal score. Figure 1 gives an example.

Smith-Waterman [12] proposed an algorithm based on dynamic programming. As input, it receives two sequences s , $|s|=m$, and t , $|t|=n$. There are $m+1$ possible prefixes

G	A	-	C	G	G	A	T	T	A	G	
G	A	T	C	G	G	A	A	T	A	G	
+1	+1	-2	+1	+1	+1	+1	-1	+1	+1	+1	= 6

Fig. 1. Alignment of the sequences $s = GACGGATTAG$ and $t = GATCGGAATAG$, with the score for each column. There are nine columns with identical characters, one column with distinct character and one column with a space, giving a total score $6 = 9*(+1) + 1*(-1) + 1*(-2)$

for s and $n+1$ prefixes for t , including the empty string. An array $(m+1) \times (n+1)$ is built, where the (i, j) entry contains the value of the similarity between two prefixes of s and t , $\text{sim}(s[1..i], t[1..j])$.

Fig. 2 shows the similarity array between $s = AAGC$ and $t = AGC$. The first row and column are initialised with zeros. The other entries are computed using Equation 1.

$$\text{sim}(s[1..i], t[1..j]) = \max \begin{cases} \text{sim}(s[1..i], t[1..j-1]) - 2 \\ \text{sim}(s[1..i-1], t[1..j-1]) + p(i, j) \\ \text{sim}(s[1..i-1], t[1..j]) - 2 \\ 0. \end{cases} \quad (1)$$

In equation 1, $p(i, j) = +1$ if $s[i] = t[j]$ and -1 if $s[i] \neq t[j]$. If we denote the array by a , the value of $a[i, j]$ is the similarity between $s[1..i]$ and $t[1..j]$, $\text{sim}(s[1..i], t[1..j])$.

		A	G	C
A	0	0	0	0
A	0	1	0	0
G	0	0	2	0
C	0	0	0	3

Fig. 2. Array to compute the similarity between the sequences $s = AAGC$ and $t = AGC$.

We have to compute the array a row by row, left to right on each row, or column by column, top to bottom, on each column. Finally arrows are drawn to indicate where the maximum value comes from, according to Equation 1. Figure 3 presents the basic dynamic programming algorithm for filling the array a .

Algorithm Similarity

Input: sequences s and t

Output: similarity between s and t

$m \leftarrow |s|$

$n \leftarrow |t|$

For $i \leftarrow 0$ **to** m **do**

$a[i, 0] \leftarrow i \times g$

For $j \leftarrow 0$ **to** n **do**

$a[0, j] \leftarrow j \times g$

For $i \leftarrow 1$ **to** m **do**

For $j \leftarrow 1$ **to** n **do**

$a[i, j] \leftarrow \max(a[i-1, j] - 2, a[i-1, j-1] \pm 1, a[i, j-1] - 2, 0)$

Return $a[m, n]$

Fig. 3. Basic dynamic programming algorithm to build a similarity array a .

An optimal alignment between two sequences can be obtained as follows. We begin in a maximal value in array a , and follow the arrow going out from this entry until

we reach another entry with no arrow going out, or until we reach an entry with value 0. Each arrow used gives us one column of the alignment. An horizontal arrow leaving entry (i,j) corresponds to a column with a space in s matched with $t[j]$, a vertical arrow corresponds to $s[i]$ matched with a space in t and a diagonal arrow means $s[i]$ matched with $t[j]$. An optimal alignment is constructed from right to left. The detailed explanation of this algorithm can be found in [11]. Many optimal alignments may exist for two sequences because many arrows can leave an entry.

The time and space complexity of this algorithm is $O(mn)$, and if both sequences have approximately the same length, n , we get $O(n^2)$.

3 Distributed Shared Memory Systems

Distributed Shared Memory offers the shared memory programming paradigm in a distributed environment where no physically shared memory exists. DSM is often implemented a single paged, virtual address space over a network of computers that is managed by the virtual memory system [8]. Local references usually proceed without the interference of the DSM system and only generate exceptions by protection fault. When a non resident page is accessed, a page fault is generated and the DSM system is contacted to fetch the page from a remote node. The instruction that caused the page fault is restarted and the application can proceed.

In order to improve performance, DSM systems usually replicate pages. Maintaining strong consistency among the copies was the approach used by the first DSM systems but it created a huge coherence overhead.[7]

Relaxed memory models aim to reduce this overhead by allowing replicas of the same data to have, for some period of time, different values [10]. By doing this, relaxed models provide a programming model that is complex since, at some moments, the programmer is conscious of replication.

Hybrid memory models are a class of relaxed memory models that postpone the propagation of shared data modifications until the next synchronisation point [10]. These models are quite successful in the sense that they permit a great overlapping of basic memory operations while still providing a reasonable programming model. Release Consistency (RC) [2] and Scope Consistency (ScC) [6] are the most popular memory models for software DSM systems.

The goal of Scope Consistency (ScC) [6] is to take advantage of the association between synchronisation variables and ordinary shared variables they protect. In Scope Consistency, executions are divided into consistency scopes that are defined on a per lock basis. Only synchronisation operations and data accesses that are related to the same synchronisation variable are ordered. The association between shared data and the synchronisation variable that guards them is implicit and depends on program order. Additionally, a global synchronisation point can be defined by synchronisation barriers. JIAJIA [4] is an example of scope consistent software DSM.

JIAJIA implements the Scope Consistency memory model with a write-invalidate multiple-writer home-based protocol. In JIAJIA, the shared memory is distributed among the nodes in a NUMA-architecture basis. Each shared page has a home node. A

page is always present in its home node and it is also copied to remote nodes on an access fault. There is a fixed number of remote pages that can be placed at the memory of a remote node. When this part of memory is full, a replacement algorithm is executed.

Each lock is assigned to a lock manager. The functions that implement lock acquire, lock release and synchronisation barrier in JIAJIA are `jia_lock`, `jia_unlock` and `jia_barrier`, respectively [5]. Additionally, JIAJIA provides condition variables that are accessed by `jia_setcv` and `jia_waitcv`, to signal and wait on conditions, respectively. The programming style provided is SPMD (Single Program Multiple Data) and each node is distinguished from the others by a global variable `jiapid` [5].

4 Parallel Algorithm to Compare DNA Sequences

To analyse the performance of our parallel algorithm, we implemented a sequential variant of the algorithm described in Section 2 that uses two linear arrays [11]. The bi-dimensional array was not used since, for large sequences, the memory overhead would be prohibitive. In this algorithm, we simulate the filling of the bi-dimensional array just using two rows in memory, since, to compute entry $a[i,j]$ we just need the values of $a[i-1,j]$, $a[i-1,j-1]$ and $a[i,j-1]$. So, the space complexity of this version is linear, $O(n)$. The time complexity remains $O(n^2)$.

The algorithm works with two sequences s , $|s|=m$ and t , $|t|=n$. First, one linear array is initialised with zeros. Then, each entry of the second array is obtained from the first one with the algorithm described in Section 2, but using a single character of s on each step. We denote $a[i,j]=sim(s[1..i,t..j])$ as *current score*. Each entry also contains: *initial and final alignment coordinates*, *maximal and minimal score*, *gaps*, *matches and mismatches counters* and a *flag* showing if the alignment is a candidate to be an optimal alignment. When computing the $a[i,j]$ entry, all the information of $a[i-1,j]$, $a[i-1,j-1]$ or $a[i,j-1]$ is passed to the current entry.

The *gaps*, *matches* and *mismatches counters* are employed when the *current score* of the entry being computed comes from more than one previous entry. In this case, they are used to define which alignment will be passed to this entry. We use an expression ($2*matches\ counter + 2*mismatches\ counter + gaps\ counter$) to decide which entry to use [9]. The greater value is considered as the origin of the current entry. If the values are still the same, our preference will be to the horizontal, to the vertical and at last to the diagonal arrow, in this order. At the end of the algorithm, the coordinates of the best alignments are kept on the queue *alignments*. This queue is sorted and the repeated alignments are removed. The best alignments are then reported to the user.

The access pattern presented by this variant of the Smith-Waterman algorithm leads to a non-uniform amount of parallelism. The parallelisation strategy that is traditionally used in this case is the “wave-front method” since the calculations that are done in parallel evolve as waves on diagonals.

We propose a parallel version of this variant where each processor p acts on two rows, a writing and a reading row. Work is assigned in a column basis, i.e., each proc-

essor calculates only a set of columns on the same row, as shown in figure 4. Synchronisation is achieved by locks and condition variables provided by JIAJIA [5,6]. Barriers are only used at the beginning and at the end of computation.

In figure 4, p0 starts computing and, when value $a_{1,3}$ is calculated, it writes this value at the shared memory and signals p1, that is waiting on `jia_waitcv`. At this moment, p1 reads the value from shared memory, signals p0, and starts calculating from $a_{1,4}$. P0 proceeds then calculating elements $a_{2,1}$ to $a_{2,3}$. When this new block is finished, p0 issues a `jia_waitcv` to guarantee that the preceeding value was already read by p1. The same protocol is executed by every processor p_i and processor p_{i+1} .

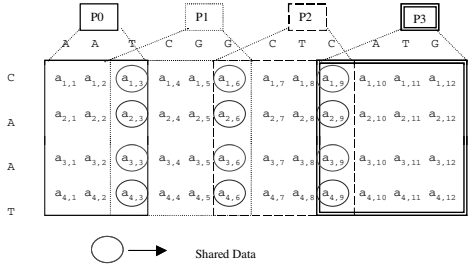


Fig. 4. Work assignment in the parallel algorithm. Each processor p is assigned N/P rows, where P is the total number of processors and N is the length of the sequence.

5 Experimental Results

Our parallel algorithm was implemented in C, using the software DSM JIAJIA v.2.1 on top of Debian Linux 2.1. We ran our experiments on a dedicated cluster of 8 Pentium II 350 MHz, 160 MB RAM connected by a 100Mbps switch. Our tests used real DNA sequences obtained from www.ncbi.nlm.nih.gov/PMGifs/Genomes. Five sequence sizes were considered (15KB, 50KB, 80KB, 150KB and 400KB). Execution times and speedups for these sequences, with 1,2,4 and 8 processors are shown in Table 1. Speedups were calculated considering the total execution time and thus include times for initialisation and collecting results.

Table 1. Total execution times (seconds) and speedups for 5 sequence comparisons

Size	Serial Exec	2 proc Exec /Speedup	4 proc Exec /Speedup	8 proc Exec /Speedup
15K x 15K	296	283.18/1.04	202.18/1.46	181.29/1.63
50K x 50K	3461	2884.15/1.20	1669.53/2.07	1107.02/3.13
80K x 80K	7967	6094.19/1.31	3370.40/2.46	2162.82/3.68
150K x 150K	24107	19522.95/1.23	10377.89/2.32	5991.79/4.02
400K x 400K	175295	141840.98/1.23	72770.99/2.41	38206.84/4.58

As can be seen in table 1, for small sequence sizes, e.g. 15K, very bad speedups are obtained since the parallel part is not long enough to surpass the amount of synchronisation inherent to the algorithm. As long as sequence sizes increase, better speedups

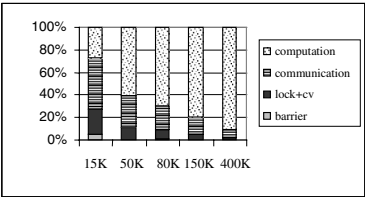


Fig. 5. Execution time breakdown for 5 sequence sizes, containing the relative time spent in computation, communication, lock and condition variable and barrier.

are obtained. This effect can be better noticed in figure 5, which presents a breakdown of the execution time of each sequence comparison.

We also compared the results obtained by our implementation (denoted GenomeDSM) with BlastN, FASTA and PipMaker [14]. For this task, we used two 50KB mitochondrial genomes, *Allomyces acrogynus* and *Chaetosphaeridium globosum*.

In table 2, we present a comparison among these four programs, showing the alignments with the best scores found by GenomeDSM. Still in table 2, the second and third best alignments were not found by FASTA. In FASTA, the query sequence had to be broken, since our version of FASTA did not compute sequences greater than 20KB. Thus, the lack of these two sequence alignments can be due to this limitation.

Table 2. Comparison among results obtained by GenomeDSM, BlastN, FASTA and PipMaker

		GenomeDSM	BlastN	FASTA	PipMaker
Alignment 1	Begin	(39109, 55559)	(39099, 55549)	(38396, 55317)	(38396, 54897)
	End	(39839, 56252)	(39196, 55646)	(39840, 56673)	(39828, 56239)
Alignment 2	Begin	(39475, 48905)	(39522, 48952)	-	(39617, 49050)
	End	(39755, 49188)	(39755, 49005)	-	(39756, 49189)
Alignment 3	Begin	(28637, 47919)	(28667, 47949)	-	(28505, 47787)
	End	(28753, 48035)	(28754, 48036)	-	(28756, 48038)

We also developed a tool to visualise the alignments found by GenomeDSM. An example can be seen in figure 6.

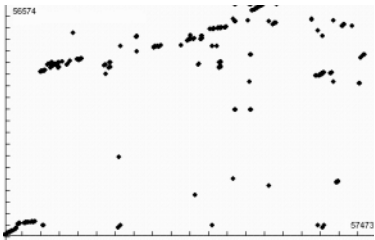


Fig. 6. Visualisation of the alignments generated by GenomeDSM with the 50KB sequences. Plotted points show the similarity regions between the two genomes.

Martins et al. [9] presented a version of the Smith-Waterman algorithm using MPI that ran on a Beowulf system with 64 nodes each containing 2 processors. Speedups

attained were very close to ours, e.g., for 800Kx500K sequence alignment, a speedup of 16.1 were obtained for 32 processors.

6 Conclusions and Future Work

In this paper, we proposed and evaluated a DSM implementation of the Smith-Waterman algorithm that solve the DNA local sequence alignment problem. Work is assigned to each processor in a column basis and the wavefront method was used.

The results obtained in an 8-machine cluster present good speedups which are improved as long as the sequence lengths increase. To compare sequences of 400KB, we obtained a 4.58 speedup on the total execution time, reducing execution time of the sequential algorithm from 2 days to 10 hours. This shows that that our parallelisation strategy and the DSM programming support were appropriate to our problem.

As future work, we intend to port the algorithm implemented in MPI proposed in [9] to our cluster and compare its results with ours. Also, we intend to propose and evaluate a variant of our approach, which will use variable block size.

References

1. S. F. Altschul et al. – *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs*. Nucleic Acids Research, v. 25, n. 17, p. 3389–3402, 1997.
2. K. Gharachorloo et al, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc. Int. Symp. On Computer Architecture*, May, 1990, p. 15–24.
3. L. Grate, M. Diekhans, D. Dahle, R. Hughey, *Sequence Analysis With the Kestrel SIMD Parallel Processor* –1998.
4. W. Hu., W. Shi., Z. Tang.: JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In Proc. of HPCN'99, LNCS 1593, pp. 463–472, Springer-Verlag, April, 1999.
5. W.Hu, W.Shi, "JIAJIA User's Manual", Technical report, CAS – China, 1999.
6. Iftode L., Singh J., Li K.: Scope Consistency: Bridging the Gap Between Release Consistency and Entry Consistency, Proc. Of the 8th ACM SPAA'96, June, 1996, pages 277–287.
7. Lamport L., *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, 1979, 690–691.
8. K. Li, "Shared Virtual Memory on Loosely Coupled Architectures", PhDThesis, Yale, 1986.
9. W. S. Martins, et al., *A Multithread Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison*, Proc. SBAC-PAD, 2001, Pirenopolis, Brazil, p.1–8.
10. Mosberger D.: Memory Consistency Models, Operating Systems Review, p. 18–26, 1993.
11. J. C. Setubal, J. Meidanis, *Introduction to Computational Molecular Biology*. Pacific Grove, CA, United States: Brooks/Cole Publishing Company, 1997.
12. T. F. Smith, M. S. Waterman, *Identification of common molecular sub-sequences* – Journal of Molecular Biology, 147 (1) 195–197–1981.
13. W. R. Pearson; D. L. Lipman, *Improved tools for biological sequence comparison*. Proceedings Of The National Academy Of Science USA, v. 85, p. 2444–2448, April 1988.
14. Schwartz Et Al. – *PipMaker – A Web Server for Aligning Two Genomic DNA Sequences* – Genome Research 10:577–586, April 2000 – <http://bio.cse.psu.edu>.

Two Dimensional Airfoil Optimisation Using CFD in a Grid Computing Environment

Wenbin Song, Andy Keane, Hakki Eres, Graeme Pound, and Simon Cox

School of Engineering Sciences
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{w.song, ajk, eres, gep, sjc}@soton.ac.uk
<http://www.geodise.org>

Abstract. In this paper, a two-dimensional airfoil shape optimisation problem is investigated using CFD within a grid computing environment (GCE) implemented in Matlab. The feature-based parametric CAD tool ProEngineer is used for geometry modelling. The industrial level mesh generation tool Gambit and flow solver Fluent are employed as remote services using the Globus Toolkit as the low level API. The objective of the optimisation problem is to minimize the drag-to-lift coefficient ratio for the given operating condition. A Matlab interface to the design exploration system (OPTIONS) is used to obtain solutions for the problem. The adoption of grid technologies not only simplifies the integration of proprietary software, but also makes it possible to harness distributed computational power in a consistent and flexible manner.

1 Introduction

Computational fluid dynamics (CFD) has been constantly developed over the past few decades and now both commercial and in-house codes can provide more and more robust and accurate results. Combined with the use of wind tunnel test data, CFD can be used in the design process to drive geometry change instead of being used mainly as a design validation tool. This aspect can be further exploited by bringing optimisation tools into the design process. Automation of the design process can significantly shorten the design cycle and lead to better designs compared to previous manual design modification approaches. Such manual approaches are still adopted by most engineers due to various reasons: lack of robustness and flexibility in automating the design process, the high computational cost associated with large numbers of iterations of high fidelity simulation codes, difficulties of collobaration in a heterogeneous computational environments, etc. In fact, the revolution brought by the World Wide Web with respect to information sharing has not yet delivered fundamental changes to engineering design practice for a number of reasons, including security problems in collaborative environments. The emerging Grid computing technologies [1] aim to fill this gap in providing a general architecture for building a geographically distributed, collaborative computational environment.

Design search and optimisation is a process that can be used to improve the design of complex engineering products, systems, and processes. There are a large number of numerical optimisation methods available for this purpose, along with various strategies that provide even more complex scenarios for design search such as hybrid methods incorporating machine learning methods. In order to evaluate the strength and/or weakness of different designs, it is often necessary to use complicated computational tools such as CFD. One of the important aspects of these tools is the high computational cost related to the solution of large numbers of simultaneous algebraic equations, such as in computational fluid dynamics or structural analysis. The combination of the repetitive nature of search processes and the constant increase in demand for computational power of high-fidelity computational models has prompted much effort in the study of various aspects of the problem solving environment. For example, approximation techniques are often used to provide surrogate models for the high fidelity codes and to decouple the strong interactions between codes from different disciplines. However, an efficient approximation framework often requires capabilities to start the analysis codes on-demand for points anywhere in the parameter space and mining new data into existing datasets. It is clear that the strong coupling between optimisation methods and domain codes partly limits the ability to prototype different complicated search strategies, and thus impedes the wider use of optimisation technologies in the engineering design offices on a daily basis.

The aim of this paper is to provide an exemplar of CFD-based shape optimisation using emerging grid technologies that address some of these issues. The paper is organized as follows. The next section gives a brief introduction on grid computing technology with a focus on the architecture and various techniques used. The third section defines a two-dimensional airfoil shape optimisation problem. The optimisation method used and some results are given in section four, with concluding remarks and future work described in section five.

2 Grid Computing

The design of increasingly complex engineering systems relies on knowledge drawn from various disciplines, and a multidisciplinary approach to tackle the interrelations between different domains. The availability of a generic infrastructure addressing the integration of software packages with a rich user interface is vital. Emerging grid computing techniques seem to be able to provide a general approach for integration and collaboration while retaining the division and autonomy of disciplinary domain experts. To address the organizational challenges that prevent a wider application of a multidisciplinary optimisation approach in a generic manner, a flexible environment supporting a powerful scripting language, rich visualization tools, and common mathematical computation capabilities is desirable. In this work, Matlab is chosen as the central stage for the problem solving environment, as it provides a broad spectrum of functions and algorithms for a wide range of applications including visualization and Graphical User In-

terface (GUI) building features. However, our approach allows that the toolkits developed can be easily integrated with other environments such as Python. [2]

The overall architecture adopted in this paper is shown in Figure 1, which is a simplified version of the general Grid-Enabled Optimisation and Design Search for Engineering (GEODISE) architecture [3] (<http://www.geodise.org/>). In the current implementation, Globus Toolkit 2.2 [4] is used to provide various low-level functionalities such as authentication, resource management, job submission, etc. These functionalities can then be exposed to end users via the interface to commodity technologies, such as Java, Python, Perl, etc.

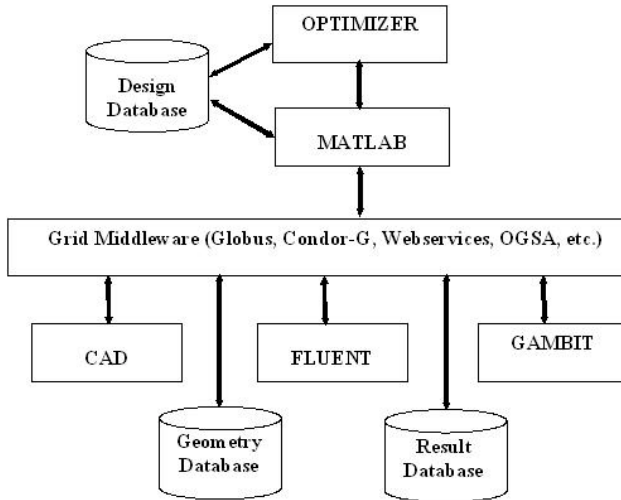


Fig. 1. Overall structure of the grid-enabled design optimisation

In the Geodise project, The Java Cog Kit [5] is used to expose Globus functionalities in the Matlab environment. Matlab is widely used in academia and industry to prototype algorithms, and to analyse and visualize data. Matlab also enables programmers to access Java classes, and therefore provides the functionality for code re-use and further developments. Detailed discussion on how various technologies such as Condor, Globus, etc., are used in Geodise can be found in [6]. Via the use of the Matlab functions provided in the Geodise Toolkit, the user is also able to submit his/her own code to computing resources, or to run software packages installed on the server. Access to database functionalities such as file archiving and retrieval and user notifications are also provided in the form of Matlab functions. Table 1 lists the functions implemented in the current version of the Geodise toolkit.

Table 1. Implemented commands in the Geodise toolkits

<code>gd.archive</code>	Stores a file in repository with associated metadata
<code>gd.proxyinfo</code>	Returns information about the user's proxy certificate
<code>gd.proxyquery</code>	Queries whether a valid proxy certificate exists
<code>gd.createproxy</code>	Creates a proxy certificate using user's credentials
<code>gd.destroyproxy</code>	Destroys the local copy of user's proxy certificate
<code>gd.getfile</code>	Retrieves a file from a remote host using GridFTP
<code>gd.putfile</code>	Transfers a file to a remote host using GridFTP
<code>gd.jobsubmit</code>	Submits a GRAM job to a Globus server
<code>gd.jobstatus</code>	Returns the status of the GRAM job specified a job handle
<code>gd.jobpoll</code>	Queries the status of a Globus GRAM job until complete
<code>gd.jobkill</code>	Terminates the GRAM job specified by a job handle
<code>gd.listjobs</code>	Returns job handles for all GRAM jobs
<code>gd.query</code>	Query metadata about a file based on certain criteria
<code>gd.retrieve</code>	Retrieves a file from the repository to the local machine

3 Two Dimensional Airfoil Design Using Orthogonal Basis Functions

A two-dimensional airfoil design optimisation problem appropriate for concept wing design is studied here, using the Geodise toolkit. Instead of using airfoil coordinates directly, as is the case in many airfoil design applications, six basis airfoil functions are used to form a set of orthogonal functions to define the airfoil shape [7]. The goal of the optimisation is to minimize the drag/lift ratio by changing the weights of the six basis functions and the thickness-to-chord ratio. The basis airfoil functions were derived from a family of nine NASA SC(2) supercritical airfoils. The first three of these six basis functions are shown in Figure 2. A similar approach of using basis functions for defining airfoil shape was adopted by Ahn J. *et al.* [8], however, the basis functions used were not orthogonal and not derived from airfoil shapes. The use of orthogonal basis functions leads to a unique mapping from parameter space to the airfoil geometries, which is a desirable feature when optimisation methods are adopted to search for good designs. The airfoil geometry is defined first by upto six weight coefficients, followed by the adjustment to the specified thickness-to-chord ratio. A detailed discussion on geometry definition, meshing and solution is given in the next sections.

3.1 Problem and Geometry Definition

Shape parameterization methods in the context of multidisciplinary design optimisation have been investigated by a number of researchers. A comprehensive overview can be found in [9]. A combined shape, topology and configuration optimisation process for structures was recently reported in [10], in which a parametric model was constructed using the CAD tool ProEngineer. In most cases, the transfer of geometry between codes is implemented using standard neutral formats such as STEP, IGES, PATRAN, etc. Here, a parametric airfoil

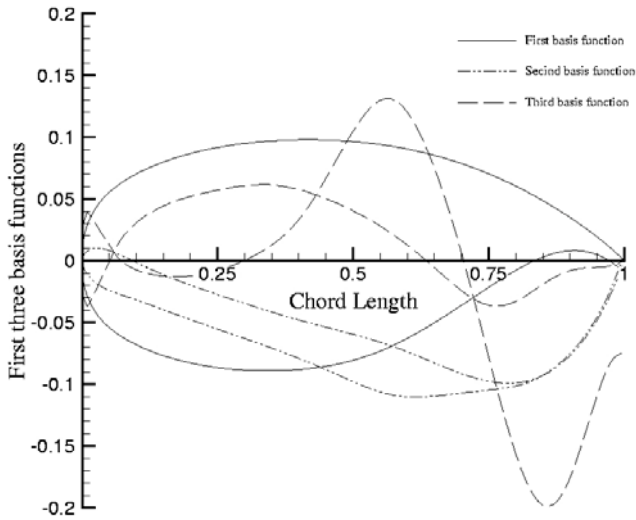


Fig. 2. First three airfoil basis functions used for airfoil geometry definition

geometry has been defined using ProEngineer. In this case the basis functions have been normalized with respect to the first member of the set and the thickness to chord ratio. Moreover, it has been shown from [7] that by adopting this approach good representation of the original airfoil can be recovered by simply varying the thickness to chord ratio and second weight, leaving the first function weight at unity and the remaining four at zero. This leads to a two-dimensional specification which is very simple to implement and interpret. These parameters, their initial values and ranges are listed in Table 2.

3.2 Mesh Generation and CFD Solution

To carry out optimization, mesh generation must be carried out in a batch mode, based on the geometry imported from ProEngineer. However, in general, a lot of work is needed to clean up the geometry in order to create robust, efficient meshes, to remove undesired features such as very short edges and surfaces (via surface merge or removal of short edges), and this process is often interactive in nature involving much graphic picking. Unique tagging can be used to replace the graphic picking and to deal with topology changes. However, this is not always possible, especially for complex models generated using a top-down design approach. To begin with here, an interactive session was used to generate a script file that is later used to run the meshing program in batch mode. The limitation of this approach is that the geometry topology must be maintained the same, and also that when a different version of meshing tool is to be used, another

Table 2. Design variable for the airfoil optimisation problem

Name	Lower bound	Initial value	Upper bound	Meaning
$w(2)$	-1.0	-0.98	1.0	Weight for the second basis function
tc_ratio	0.06	0.08	0.18	Thickness-to-chord ratio

interactive session may be needed to generate the script file. The generated mesh file is then imported into Fluent [11]. Node spacing is specified on the edges to control the mesh density. A triangular mesh is used for the flow field. Boundary layer meshes are attached to the upper and lower edges of the airfoil. A Reynolds-averaged two-dimensional compressible Navier-Stokes solver is used to obtain the flow field solutions.

Meshing jobs are submitted onto the computing server using the `gd_jobsubmit` function after a valid proxy has been established using the user’s credentials. A unique job handle is returned and later used to check the status of the meshing job. Fluent CFD jobs are then submitted using the same mechanism. Results are stored into data files on the scratch directory on the server. After the jobs finish, `gd_getfile` is used to retrieve the results files and values of lift and drag coefficients are obtained by analyzing the resulting files.

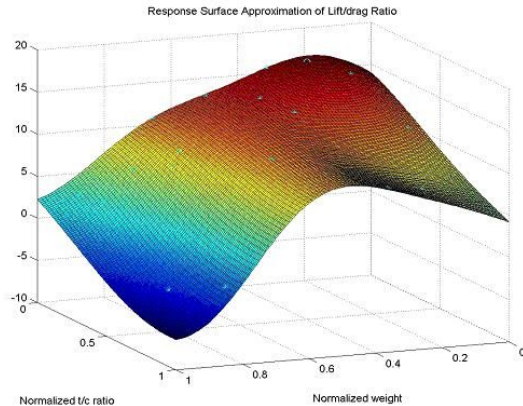


Fig. 3. Landscape of lift/drag coefficient ratio for airfoils with two design variables

4 Optimisation Using Genetic Algorithm

An implementation of a population-based genetic algorithm (GA) [12] is adopted in the search process due to its robustness in finding the location of globally optimum designs. However, the design search is not conducted on the original

landscape of the objective function, instead, a surrogate model is used in lieu of the true objective function. The surrogate model was constructed from results evaluated at twenty sample points in design space using a random Latin hypercube sampling technique. The landscape of the lift/drag ratio is shown in Figure 3 based on the results at these twenty points. Here, a kriging surrogate model [13] is constructed and searched using the Genetic algorithm. The best design found on the kriging surrogate model is then validated using a full CFD solution. The resultant airfoil shape and values of design variables and objective function are given in Figure 4. It can be seen that using a surrogate model can significantly speed up the search process and enables the exploration of multiple search methods, which would be impossible on the high fidelity codes. Furthermore, discrepancies between the landscapes of the true objective function and the approximated one could be addressed by evaluating the approximation error or introducing some adaptive update scheme based on the criteria of expected improvements. It is expected that such consideration will improve the effectiveness of approximation methods, and this will be studied in the future work.

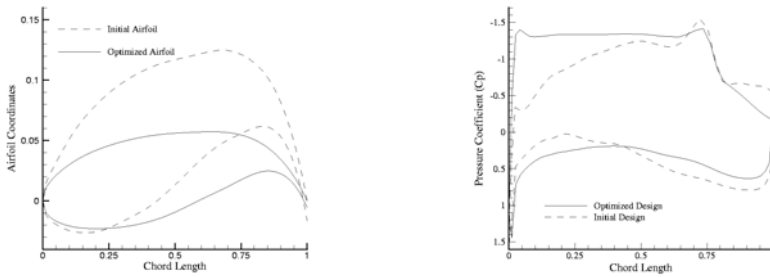


Fig. 4. Initial and Optimized airfoil (left) and corresponding pressure distribution (right). For the initial and optimized design, the weight for the second basis function is $-0.9815/-0.5985$, the thickness-to-chord ratio is $0.116/0.0716$, and lift/drag ratio is $7.8363/14.9815$

5 Concluding Remarks and Future Work

A two-dimensional airfoil design optimisation using CFD has been tackled using emerging grid technologies. Although the problem does not require as much computing power as a full-wing CFD analysis, it has demonstrated all the elements in a typical CFD-based shape optimisation problem. The Geodise computation toolkit is used in the study, along with a Genetic Algorithm and surrogate modeling methods. Grid technologies are still fast evolving, and the Open Grid Service

Architecture (OGSA) [14] seems to be the future direction for grid services developments. Other issues may involve the use of a job scheduler for job farming and monitoring. Additionally, a rich GUI is desired for novice designers as well as the scripting language for expert users and further developments. More complicated geometries and approximation and optimisation frameworks will also be studied in due course.

Acknowledgements. This work is supported by the UK e-Science Pilot project (UK EPSRC GR/R67705/01). The authors gratefully acknowledge many helpful discussions with the Geodise team. The authors also wish to thank Dr. Neil Bressloff and Mr. Andras Sobester for helpful discussions.

References

- [1] Foster I., Kesselman C., Tuecke S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications* **15** (3) (2001) 200–222
- [2] Python(2003) <http://www.python.org>
- [3] Cox, S.J., Fairman, M.J., Xue, G., Wason, J.L., Keane, A.J.: The GRID: Computational and data resource sharing in engineering optimisation and design search.(2001) IEEE Computer Society Press
- [4] The Globus Project (2003) <http://www.globus.org>
- [5] Laszewski, G., Foster I., Gawor, J., Lane P.: A Java Commodity Grid Toolkit. *Concurrency: Practice and Experience* **13** (8-9) (2001) 643–662
- [6] Pound, G., Eres, M.H., Wason, J.L., Jiao, Z., Cox, S.J., Keane, A.J.: A Grid-enabled Problem solving environment (PSE) for design Optimisation within MATLAB.(2003) International Parallel and Distributed Processing Symposium, April 22–26, 2003, Nice, France
- [7] Robinson, G.M., Keane, A.J.: Concise Orthogonal Representation of Supercritical Airfoils. *Journal of Aircraft* **38** (3) (2001) 580–583
- [8] Ahn, J., Kim, H.-J., Lee, D.-H., Rho, O.-H.: Response Surface Method for Airfoil Design in Transonic Flow. *Journal of Aircraft* **38** (2) (2001) 231–238
- [9] Samareh, J.A.: Survey of shape parameterization techniques for high-fidelity multidisciplinary shape optimisation. *AIAA Journal* **39** (5) (2001) 877–884
- [10] Langer, H., Puhlhofer, T., Baier, H.: An approach for shape and topology optimization integrating CAD parameterization and evolutionary algorithms. (2002)
- [11] Fluent (2003) <http://www.fluent.com>
- [12] Keane, A.J.: OPTIONS Design exploration system, (2003) <http://www.soton.ac.uk/~ajk/options.ps>
- [13] Jones D.R., Schonlau, M., Welch W.J.: Efficient global optimisation of expensive black-box functions. *Journal of global optimisation* **13** (1998) 455–492
- [14] Foster, I., Kesselman C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002

Applied Grid Computing: Optimisation of Photonic Devices

Duan H. Beckett¹, Ben Hiett¹, Ken S. Thomas¹, and Simon J. Cox²

¹ Department of Electronics and Computer Science,
University of Southampton, SO17

² School of Engineering Sciences, University of Southampton, SO17 1BJ

Abstract. In this paper, we present an application of grid computing to solve an important industrial problem: that of optimising the band gap of photonic crystals, which are an important technology in future generation telecomms and sensing. The computational power grid enabled months of experimentation to be performed in a weekend. Of particular interest was the necessity to run jobs on both Linux and Windows resources.

1 Introduction

Applications such as sharp angle waveguides, wave division multiplexing, single mode lasers and sensors have motivated research into photonic crystals; which are nanostructured devices that filter light at particular frequencies. Photonic crystals are modeled by finding periodic solutions to Maxwell's equations [1], [2], [3]. A typical finite element analysis will comprise: mesh generation for the domain, selection of suitable basis functions and the solution of a generalised eigenvalue problem

$$A(k)c = \lambda Bc. \quad (1)$$

The eigenvalue problem is solved for a large sample of quasi momentum vectors k in the Brillouin zone, because the spectrum of the original problem in R^2 is the union of the spectrum of (1) over $k \in B$ [3]. The band gaps are the resolvent set of the original problem.

The absolute size of the band gap is not a meaningful measurement because there is no fundamental length scale in electromagnetics [4] and so we find the configuration of the rods that gives the largest gap-midgap ratio. This optimisation and the important role played by a computational grid is the aim of this paper.

The University of Southampton has invested in grid-enabled computational facilities. We used a Beowulf cluster [5] running Linux and an experimental Condor system [6] that uses Windows. One important factor was that our chosen meshing program Geompack [7] is only available for Windows, thus requiring a heterogeneous mix of operating systems in our grid.

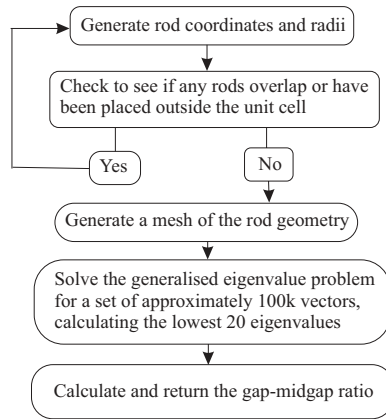


Fig. 1. Figure showing how the objective function for maximising the PC gap-midgap ratio is constructed.

2 Optimising Photonic Crystal Structures

A photonic crystal is a periodic configuration of air-filled rods in a substrate (for example Gallium Arsenide). The physical properties of a Photonic Crystal (PC) depend critically on the configuration of air rods in a substrate. A useful PC would be one such that the gap-midgap ratio [4] is very large. The objective in the optimisation is to maximize the gap-midgap ratio; the independent variables are the rod co-ordinates and the rod radii. The details of the computation are shown in Figure 1. Our approach is more direct than [8], [9], who only consider the TM mode and used a gradient method.

To investigate various PC geometries we performed a design of experiment which consisted of constructing several thousand random configurations of a number of rods in a unit cell. Two different unit cells were used: a square unit cell and a rhombic unit cell. We only report on the latter in this paper. The rhombic unit cell was varied from 1 to 10 rods, although only the radius was altered when a single rod was used as the position for a single rod is meaningless when the unit cell is tiled.

For the rhombic unit cell 100 meshes per rod radius and number of rods was generated for varying filling fractions of approximately 30% to 78%. The number of rods varied from 2 to 10. In total 1400 random meshes were created.

The meshes were generated on a computer with a Windows operating system then transferred to the Beowulf cluster to solve the generalized eigenvalue problem (a compiled MATLAB program).

The best 10 structures from each unit cell, which gave the largest gap-midgap ratio were optimised using the robust (but slow) Nelder Mead method [10] on the Condor system. The Nelder Mead algorithm was chosen for the optimisation because it only requires function evaluations and not derivatives which is ideal for our problem because the objective function has no calculable derivative. There

are a number of constraints imposed on the variables mainly to ensure that it is possible to fabricate the proposed structure.

Each iteration is computationally intensive as it requires not only meshing the photonic crystal geometry but also solving the generalized eigenvalue problem.

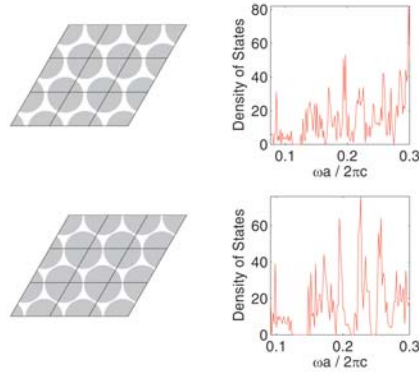


Fig. 2. Figure on the top left shows a tiled rhombic unit cell with a filling fraction of 0.753. On the top right is the density of states diagram for the unit cell. The band gap to mid-gap ratio is 0.118. On the bottom left is the optimised tiled unit cell with a filling fraction of 0.838. On the bottom right is the density of states for the optimised unit cell with a band gap to mid-gap ratio of 0.182.

The results in this section were computed with an in house FEM code. The most improved mesh increased its gap-midgap ratio by 264.3% after optimisation. A density states graph is a concise method of visualising the results and is shown in Figure 2.

3 Computer Costs

In total 8940 meshes were created. Each generalized eigenvalue problem took approximately 6 minutes to run which equates to 37 days to compute the eigenvalues of each mesh if only one computer was used. When using the Beowulf cluster the workload was split up such that each processor would compute the eigenvalues for 100 meshes. However, the nodes used were dual processors so 200 meshes could be run on each node. As a result 45 separate jobs were submitted to our cluster using the EASY scheduler each taking approximately 10 hours to run. If the cluster was not being used at all all the jobs could have been run simultaneously in roughly 10 hours. However, the rate at which all the jobs are computed is dependent on how busy the cluster is. In our case, despite a heavily used cluster, it only took a few days to calculate the generalized eigenvalue problem from all the meshes.

From the design of experiment 20 of the best PC geometries were optimised. The optimisation process was run for 12 hours for each structure. This equates to 10 days of computation on a single computer. Each of the 20 optimisation jobs were submitted to the Condor pool. With 20 free nodes, all the optimisation could be performed in only 12 hours. Once again, due to how busy the cluster was, a couple of days were required to complete all the jobs.

4 Conclusion

The randomisation proved effective in providing candidates for further analysis. Moreover, it confirmed the (known) good behaviour of the triangular lattice [4]. The Nelder Mead algorithm was successful in improving the good candidates.

The main contribution of the grid facilities was the reduction of about two months intense computation to a weekend. The scheduler on the Beowulf system was straightforward to use, but Condor had a shaper learning curve and less predictable completion times.

We have demonstrated the use of heterogeneous grid-enabled resources for an important industrial design process and in the future we hope that a widespread grid will enable the use of resources via brokering or pay-per-use and permit this sort of design to be a routine part of the process of developing new photonic devices.[4]

References

1. Axmann, W., Kuchment, P.: An efficient finite element method for computing spectra of photonic and acoustic band-gap materials – i. scalar case. *Journal of Computational Physics* **150** (1999) 468–481
2. Dobson, D.C.: An efficient method for band structure calculations in 2d photonic crystals. *Journal of Computational Physics* **149** (1999) 363–376
3. K.S.Thomas, Cox, S., Beckett, D., Hiett, B., Generowicz, J., Daniell, G.: Eigenvalue spectrum estimation and photonic crystals. In: 7th International Europar Conference, Springer (2001) 578–586
4. Joannopoulos, J., Meade, R., Winn, J.: *Photonic Crystals Molding the Flow of Light*. Princeton University Press (1995)
5. University of Southampton: Research support services: Iridis (2002)
<http://www.iss.soton.ac.uk/research/iridis/>.
6. University of Southampton: Research support services: Condor (2002)
<http://www.iss.soton.ac.uk/research/e-science/condor/>.
7. Joe, B.: Geompack mesh generating software (2001)
<http://members.attcanada.ca/bjoe/>.
8. Cox, S.J., Dobson, D.C.: Maximizing band gaps in two-dimensional photonic crystals. *Siam Journal on Applied Mathematics* **59** (1999) 2108–2120
9. Cox, S.J., Dobson, D.C.: Band structure optimization of two-dimensional photonic crystals in h-polarization. *Journal of Computational Physics* **158** (2000) 214–224
10. Brent, R.: *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ:Prentice Hall (1973)

Parallel Linear System Solution and Its Application to Railway Power Network Simulation

Muhammet F. Ercan¹, Yu-fai Fung², Tin-kin Ho², and Wai-leung Cheung²

¹ School of Electrical and Electronic Eng., Singapore Polytechnic, Singapore
mfercan@sp.edu.sg

² Dept. of Electrical Eng., The Hong Kong Polytechnic, University, Hong Kong SAR
{eeyffung, eetkho, eewlcheung}@polyu.edu.hk

Abstract. The Streaming SIMD extension (SSE) is a special feature embedded in the Intel Pentium III and IV classes of microprocessors. It enables the execution of SIMD type operations to exploit data parallelism. This article presents improving computation performance of a railway network simulator by means of SSE. Voltage and current at various points of the supply system to an electrified railway line are crucial for design, daily operation and planning. With computer simulation, their time-variations can be attained by solving a matrix equation, whose size mainly depends upon the number of trains present in the system. A large coefficient matrix, as a result of congested railway line, inevitably leads to heavier computational demand and hence jeopardizes the simulation speed. With the special architectural features of the latest processors on PC platforms, significant speed-up in computations can be achieved.

1 Introduction

An electric railway is a dynamic system with a number of sub-systems, such as signaling, power supply and traction drive, which interact with each other continuously. Because of high cost of on-site evaluation, simulation is perceived to be the most feasible means for engineering analysis. Advances in hardware and software allow enhancement on simulators, in terms of computation speed and data management [3]. Numerous simulation packages have been developed and found successful applications. However, most of the simulators were tailor-made for specific studies.

In view of this deficiency, a whole-system simulation suite, which includes the modeling of most possible functions and behavior of a railway system, is essential for accurate and reliable research and development on railway engineering. Such simulation suite should allow multi-train movement simulation under various signaling requirements and power system features on a railway network. We have designed a simulator with the above features to cater for different railway systems with user-defined track geometry, signaling system and traction equipment [4]. The computation time is crucial in a real time simulator; in this application, identifying solution of linear system equations repetitively was the most time consuming operation. Hence, we aim to speed up these operations by the SSE features provided in standard PCs.

2 Problem Modeling

The commonly adopted computing platform in the railway industry is the Intel microprocessor based PCs. Simulation imposes heavy computational demands when the railway network gets complicated and number of trains increases. Both off-line studies and real-time control prompt for more computation power. We explore the parallel computing features of Pentium CPUs to solve the linear equations.

An electrified railway line is in fact a huge electrical circuit with the substations as sources and the trains as moving loads (sources if the re-generative braking is allowed) [7]. As illustrated in Fig. 1, trains traveling on two opposite directions draw energy from the feeding substations along the line and their locations define the nodes in the electrical circuit. The supply wires and rail resistance are lumped together to form a resistor between nodes. The voltage seen by a train may vary with time and then determine its traction performance which in turn affects train movement. Thus, it is crucial to attain the voltages at certain nodes of this electrical circuit at consecutive time intervals, which requires the nodal analysis and thus the solution of a matrix equation. The size of the coefficient matrix depends upon the number of nodes in the circuit at each time interval, which is determined by the number of trains on track.

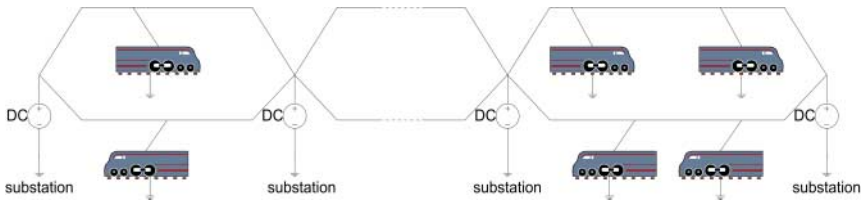


Fig. 1. Typical two- road network.

With an efficient bandwidth oriented pre-ordering scheme [2], the coefficient matrix of the power network is sparse, narrowly-banded and symmetric for a simple two-track network. The solution is obtained by Cholesky decomposition, followed by forward and backward substitutions. Unsurprisingly, this stage is the most CPU time-consuming part of the simulation. When the topology of the network becomes more complicated with branches and loops, the size and bandwidth of the coefficient matrix increases and sparsity deteriorates. Computational demand is hugely raised as a result. In order to extend the applicability of the simulator to real-time studies, it is necessary to solve the matrix equation rapidly within the limited computing resources.

3 Computational Demand and Parallel Computing

Intel microprocessor based PCs are the most commonly used platforms for railway simulators. For a simple DC railway line with two separate tracks and 20 trains on each one, the coefficient matrix is only around 50x50. The simulation time is still much faster than real-time. Power network calculation has not caused too many problems in most off-line studies even with a single processor. However, dealing with

a large, complicated and busy network (larger and less sparse coefficient matrix) and/or AC supply system (complex numbers involved) presents a formidable challenge to the simulator in the increasingly demanding real-time applications.

Parallel computing is an obvious direction to deal with the computation demand. Multiple-processor hardware is a convenient way out, but it usually requires substantial modifications within the simulator and increases the system cost. Equal workload assignment to the processors and suitable processor architecture are other critical considerations. Our objective is to exploit the SIMD computing capabilities of the latest processors so that simulator can be sped up on a single processor system with minimum alterations on the source codes. The Intel Pentium III or above processors include SSE registers where parallel data operations are performed simultaneously [6]. The SSE architecture provides the flexibility of operating with various data types, such as registers holding eight 16-bit integers or four 32-bit floating-point values. SIMD operations, add, multiply, etc., can be performed between the two registers and significant speed-up can be achieved. In particular, its support for floating point manipulations plays a key role in this application.

The matrix equation resulted from an electrical railway system is in the form of:

$$Ax = b \quad (1)$$

A is a symmetrical sparse matrix representing the admittance linking the nodes, vector x represents the unknown voltage at the nodes along the track and vector b defines the net current from each node. In our earlier work, we have demonstrated SSE based parallel algorithms for solving the above system equations with dense matrix in general [1]. Similar to many engineering problems, the system matrix of this railway network problem is also sparse. When dealing with the sparse matrix problem with SSE, the non-zero elements can be handled effectively. SSE registers can process four values at each operation; if most of the elements processed in an operation are equal to zero then the algorithm will not be efficient.

4 Experimental Results

The input to the simulator is a set of linear equations related to the electrical circuit in the railway system. Integration of the network solution algorithm to the simulator is usually realized in either modular or embedded format. The timing results, based on a 733MHz Pentium III machine, obtained from the simulator were included in Table 1. Different sizes of the matrix represent different number of trains running on the network and the matrix sizes of 102x102, 132x132, 162x162 and 192x192 correspond to 5, 10, 15, and 20 trains. From Table 1, the speedup ratio obtained is close to 2 in most cases and when the number of trains is 20. With SSE, the computing time is reduced by 8 minutes. We have, therefore, demonstrated the benefit of SSE in solving one type of engineering problem, and our current research direction is to study how SSE can be embedded in other forms of parallel mechanism.

Table 1. Timing results (in seconds) obtained from railway network simulator.

Matrix size	Non-SSE	SSE	Speed-up
102x102	210	135	1.56
132x132	436	167	2.61
162x162	764	402	1.90
192x192	1312	645	2.03

5 Conclusions

In this paper, we have utilized the latest SIMD extensions included in Pentium processors to speedup LU decomposition involved in a railway simulator. According to our results, a speedup ratio of more than two is obtained. The results are satisfactory; and more importantly such improvement is achieved simply by minor modifications to the existing software. Furthermore, additional hardware support is not required for such performance enhancement. On the other hand, drawback of SSE is the overhead, which is caused by the operations to pack data into, and then unpack data from, the 128-bit format. The number of packing and unpacking operations is proportional to N^2 , where N is the size of the matrix. The gain in performance is reduced when matrix size is larger than 400x400. Our study demonstrates that the SSE features are valuable tools for developing PC based application software.

Acknowledgement. This work is supported by the Hong Kong Polytechnic University under the grant number A-PD59.

References

1. Fung Y. F., Ercan M. F., Ho T.K., and Cheung W. L., A Parallel Solution to Linear Systems, *Microprocessors and Microsystems* **26** (2001) 39–44.
2. Goodman, C.J., Mellitt, B. and Rambukwella, N.B., CAE for the Electrical Design of Urban Rail Transit Systems, *COMPRAIL'87* (1987) 173–193.
3. Goodman, C.J., Siu, L.K. and Ho, T.K., A Review of Simulation Models for Railway Systems, *Int. Conf. on Developments in Mass Transit Systems* (1998) 80–85.
4. Ho, T.K., Mao, B.H., Yuan, Z.Z., Liu, H.D. and Fung, Y.F., Computer Simulation and Modeling in Railway Applications, *Comp. Physics Communications* **143** (2002) 1–10.
5. <http://www.pserc.cornell.edu/matpower/matpower.html>, 2003.
6. Intel C/C++ Compiler Class Libraries for SIMD Operations User's Guide, Intel, 2000.
7. Mellitt, B., Goodman, C.J. and Arthurton, R.I.M., Simulator for Studying Operational and Power-Supply Conditions in Rapid-Transit Railways, *Proc. IEE* **125** (1978) 298–303.

Topic 8

Parallel Computer Architecture and Instruction-Level Parallelism

Stamatis Vassiliadis, Nikitas Dimopoulos, Jean-Francois Collard, and
Arndt Bode
Topic Chairs

Parallel computer architecture and instruction-level parallelism are hot topics at Euro-Par conferences, since these techniques are present in most contemporary computing systems. At Euro-Par 2003, 18 papers were submitted to the topic, from which 1 distinguished, 4 regular and 4 short papers were accepted. The scope of this topic includes but is not limited to parallel computer architectures, processor architecture (architecture and micro architecture as well as compilation), the impact of emerging microprocessor architectures on parallel computer architectures, innovative memory designs to hide and reduce the excess latency, multi-threading, and impact of emerging applications on parallel computer architecture design.

The distinguished paper by George Almasi et al. gives an overview of the Blue Gene/L System Software Organization. With its 360 Teraflops of peak computing power and 65536 compute nodes it is a special purpose architecture well beyond the first entries of the TOP 500 list. The authors explain, how the system software will deal with this extreme amount of parallelism. The paper is therefore general for the future developments of parallel computer architectures.

Trace substitution is the topic of Hans Vandierendonck, Hans Logie and Koen De Bosschere from Gent University, Belgium. This technique is useful for wide-issue superscalar processors and their ability to react on branches in the instruction stream. The authors show that their new technique improves the fetch bandwidth of such processors.

Counteracting bank misprediction in sliced first-level caches again deals with possible cache misses and is presented by E.F. Torres, P. Ibanez, V. Vinals, J.M. Llaberia. The authors discuss techniques to reduce the bank misprediction penalty in future processors having sliced memory pipelines.

Juan Moure, Dolores Rexachs, Emilio Luque from the Computer Architecture and Operating Systems Group of the Universidad Aut3noma de Barcelona present a mechanism to optimize a decoupled front-end architecture: the indexed fetch target buffer (iFTB). Once again, wide issue superscalar processors are addressed and an indexed variation of the fetch target buffer, a decoupled front-end architecture is presented to increase the fetch rate of such processors.

Seong-Won Lee from the University of Southern California at Los Angeles and Jean-Luc Gaudiot from the University of California at Irvine present the technique of clustered microarchitectute simultaneous multithreading. SMT is

favourable for future generation microprocessors, since it exploits instruction-level parallelism and thread-level parallelism. The clustered SMT architecture proposed by the authors significantly reduces power consumption without performance degradation.

Spiros Kalogeropoulos from Sun proposes a global instruction scheduling technique addressing processors with moderate support for parallelism. The enhanced trace scheduler includes techniques for trace formation, a renaming scheme and a cost benefit analysis.

Masamichi Takagi and Kei Hiraki from the Department of Computer Science, University of Tokyo, propose compression in data caches with compressible field isolation for recursive data structures. The field array compression technique tends to alleviate the penalty for cache misses in program execution.

Hideyuki Miura et al. from the University of Tokyo discuss a compiler-assisted thread level control speculation for speculative multithreading execution. Two new techniques are presented.

The last paper of this topic again deals with caches in microprocessors. In this paper, Carles Aliagas et al. address the reduction of power to run the cache and show with their paper on “value compression to reduce power in data caches” the way to power reduction and reduction in die area without a performance penalty.

An Overview of the Blue Gene/L System Software Organization

George Almási¹, Ralph Bellofatto¹, José Brunheroto¹, Călin Cașcaval¹,
José G. Castaños¹, Luis Ceze², Paul Crumley¹, C. Christopher Erway¹,
Joseph Gagliano¹, Derek Lieber¹, Xavier Martorell¹, José E. Moreira¹,
Alda Sanomiya¹, and Karin Strauss²

¹ IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218

{gheorghe,ralphbel,brunhe,cascaval,castanos,pgc,erway,
jgaglia,lieber,xavim,jmoreira,sanomiya}@us.ibm.com

² Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

{luisceze,kstrauss}@uiuc.edu

Abstract. The Blue Gene/L supercomputer will use system-on-a-chip integration and a highly scalable cellular architecture. With 65,536 compute nodes, Blue Gene/L represents a new level of complexity for parallel system software, with specific challenges in the areas of scalability, maintenance and usability. In this paper we present our vision of a software architecture that faces up to these challenges, and the simulation framework that we have used for our experiments.

1 Introduction

In November 2001 IBM announced a partnership with Lawrence Livermore National Laboratory to build the Blue Gene/L (BG/L) supercomputer, a 65,536-node machine designed around embedded PowerPC processors. Through the use of system-on-a-chip integration [10], coupled with a highly scalable cellular architecture, Blue Gene/L will deliver 180 or 360 Teraflops of peak computing power, depending on the utilization mode. Blue Gene/L represents a new level of scalability for parallel systems. Whereas existing large scale systems range in size from hundreds (ASCI White [2], Earth Simulator [4]) to a few thousands (Cplant [3], ASCI Red [1]) of compute nodes, Blue Gene/L makes a jump of almost two orders of magnitude.

The system software for Blue Gene/L is a combination of standard and custom solutions. The software architecture for the machine is divided into three functional entities (similar to [13]) arranged hierarchically: a *computational core*, a *control infrastructure* and a *service infrastructure*. The I/O nodes (part of the control infrastructure) execute a version of the Linux kernel and are the primary off-load engine for most system services. No user code directly executes on the I/O nodes. Compute nodes in the computational core execute a single user, single process minimalist custom kernel, and are dedicated to efficiently run user applications. No system daemons or sophisticated system services reside on compute nodes. These are treated as externally controllable entities (i.e.

devices) attached to I/O nodes. Complementing the Blue Gene/L machine proper, the Blue Gene/L complex includes the service infrastructure composed of commercially available systems, that connect to the rest of the system through an Ethernet network. The end user view of a system is of a flat, toroidal, 64K-node system, but the system view of Blue Gene/L is hierarchical: the machine looks like a 1024-node Linux cluster, with each node being a 64-way multiprocessor. We call one of these logical groupings a *processing set* or *pset*.

The scope of this paper is to present the software architecture of the Blue Gene/L machine and its implementation. Since the target time frame for completion and delivery of Blue Gene/L is 2005, all our software development and experiments have been conducted on architecturally accurate simulators of the machine. We describe this simulation environment and comment on our experience.

The rest of this paper is organized as follows. Section 2 presents a brief description of the Blue Gene/L supercomputer. Section 3 discusses the system software. Section 4 introduces our simulation environment and we conclude in Section 5.

2 An Overview of the Blue Gene/L Supercomputer

Blue Gene/L is a new architecture for high performance parallel computers based on low cost embedded PowerPC technology. A detailed description of Blue Gene/L is provided in [8]. In this section we present a short overview of the hardware as background for our discussion on its system software and its simulation environment.

2.1 Overall Organization

The basic building block of Blue Gene/L is a custom system-on-a-chip that integrates processors, memory and communications logic in the same piece of silicon. The BG/L chip contains two standard 32-bit embedded PowerPC 440 cores, each with private L1 32KB instruction and 32KB data caches. Each core also has a 2KB L2 cache and they share a 4MB L3 EDRAM cache. While the L1 caches are not coherent, the L2 caches are coherent and act as a prefetch buffer for the L3 cache.

Each core drives a custom 128-bit double FPU that can perform four double precision floating-point operations per cycle. This custom FPU consists of two conventional FPUs joined together, each having a 64-bit register file with 32 registers. One of the conventional FPUs (the primary side) is compatible with the standard PowerPC floating-point instruction set. We have extended the PPC instruction set to perform SIMD-style floating point operations on the two FPUs. In most scenarios, only one of the 440 cores is dedicated to run user applications while the second processor drives the networks. At a target speed of 700 MHz the peak performance of a node is 2.8 GFlop/s. When both cores and FPUs in a chip are used, the peak performance per node is 5.6 GFlop/s.

The standard PowerPC 440 cores are not designed to support multiprocessor architectures: the L1 caches are not coherent and the architecture lacks atomic memory operations. To overcome these limitations BG/L provides a variety of synchronization devices in the chip: lockbox, shared SRAM, L3 scratchpad and the blind device. The lockbox unit contains a limited number of memory locations for fast atomic test-and-sets

and barriers. 16 KB of SRAM in the chip can be used to exchange data between the cores and regions of the EDRAM L3 cache can be reserved as an addressable scratchpad. The blind device permits explicit cache management.

The low power characteristics of Blue Gene/L permit a very dense packaging as shown in Figure 1. Two nodes share a node card that also contains SDRAM-DDR memory. Each node supports a maximum of 2 GB external memory but in the current configuration each node directly addresses 256 MB at 5.5 GB/s bandwidth with a 75 cycle latency. Sixteen compute cards can be plugged in a node board. A cabinet with two midplanes contains 32 node boards for a total of 2048 CPUs and a peak performance of 2.9/5.7 TFlops. The complete system has 64 cabinets and 16 TB of memory.

In addition to the 64K compute nodes, BG/L contains a number of I/O nodes (1024 in the current design). Compute nodes and I/O nodes are physically identical although I/O nodes are likely to contain more memory. The only difference is in their card packaging which determines which networks are enabled.

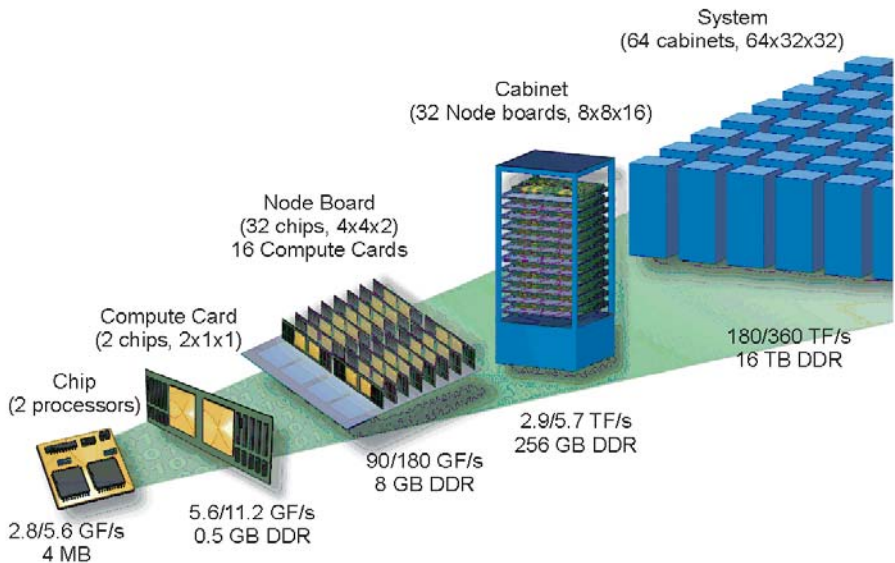


Fig. 1. High-level organization of the Blue Gene/L supercomputer. All 65,536 compute nodes are organized in a $64 \times 32 \times 32$ three-dimensional torus.

2.2 Blue Gene/L Communications Hardware

All inter-node communication is done exclusively through messages. The BG/L ASIC supports five different networks: torus, tree, Ethernet, JTAG, and global interrupts. The main communication network for point-to-point messages is a three-dimensional torus.

Each node contains six bi-directional links for direct connection with nearest neighbors. The 64K nodes are organized into a partitionable 64x32x32 three-dimensional torus. The network hardware in the ASICs guarantees reliable, unordered, deadlock-free delivery of variable length (up to 256 bytes) packets, using a minimal adaptive routing algorithm. It also provides simple broadcast functionality by depositing packets along a route. At 1.4 Gb/s per direction, the unidirectional bisection bandwidth of a 64K node system is 360 GB/s. The I/O nodes are not part of the torus network.

The tree network supports fast configurable point-to-point messages of fixed length (256 bytes) data. It also implements broadcasts and reductions with a hardware latency of 1.5 microseconds for a 64K node system. An ALU in the network can combine incoming packets using bitwise and integer operations, and forward a resulting packet along the tree. Floating point reductions can be performed in two phases (one for the exponent and another one for the mantissa) or in one phase by converting the floating-point number to an extended 2048-bit representation. I/O and compute nodes share the tree network. Tree packets are the main mechanism for communication between I/O and compute nodes.

The torus and the tree networks are memory mapped devices. Processors send and receive packets by explicitly writing to (and reading from) special memory addresses that act as FIFOs. These reads and writes use the 128-bit SIMD registers.

A separate set of links provides global OR/AND operations (also with a 1.5 microseconds latency) for fast barrier synchronization.

The Blue Gene/L computational core can be subdivided into *partitions*, which are electrically isolated and self-contained subsets of the machine. A partition is dedicated to the execution of a single job. The tree and torus wires between midplanes (half a cabinet) are routed through a custom chip called the Link Chip. This chip can be dynamically configured to skip faulty midplanes while maintaining a working torus and to partition the torus network into multiple, independent torii. The smallest torus in BG/L is a midplane (512 nodes). The tree and torus FIFOs are controlled by device control registers (DCRs). It is possible to create smaller meshes by disabling the FIFOs of the chips at the boundary of a partition. In the current packaging schema, the smallest independent mesh contains 128 compute nodes and 2 I/O nodes.

Finally, each BG/L chip contains a 1Gbit/s Ethernet macro for external connectivity and supports a serial JTAG network for booting, control and monitoring of the system. Only I/O nodes are attached to the Gbit/s Ethernet network, giving 1024x1Gbit/s links to external file servers.

Completing the Blue Gene/L machine, there is a number of service nodes: a traditional Linux cluster or SP system that resides outside the Blue Gene/L core. The service nodes communicate with the computational core through the IDo chips. The current packaging contains one IDo chip per node board and another one per midplane. The IDo chips are 25MHz FPGAs that receive commands from the service nodes using raw UDP packets over a trusted private 100 Mbit/s Ethernet control network. The IDo chips support a variety of serial protocols for communication with the core. The I^2C network controls temperature sensors, fans and power supplies. The JTAG protocol is used for reading and writing to any address of the 16 KB SRAMs in the BG/L chips, reading and writing to registers and sending interrupts to suspend and reset the cores. These services

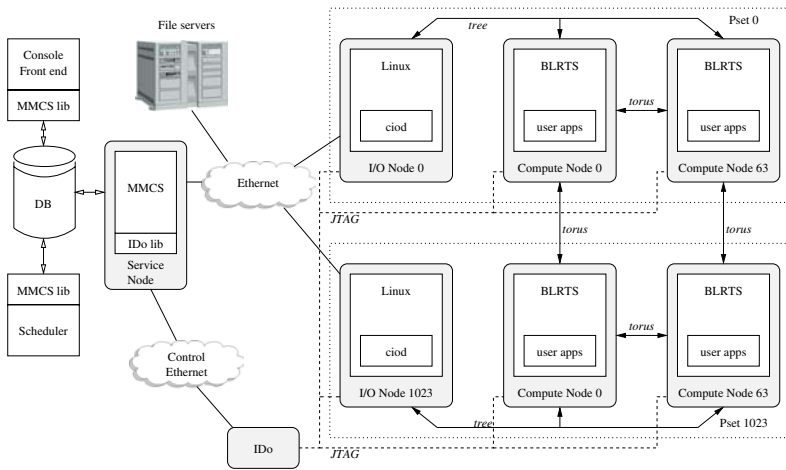


Fig. 2. Outline of the Blue Gene/L system software. The computational core is partitioned into 1024 logical processing sets (psets), each with one I/O node running Linux and 64 compute nodes running the custom BLRTS kernel. External entities connect the computational core through two Ethernet networks for I/O and low level management.

are available through a direct link between the IDo chips and nodes, and bypass the system software running on the target nodes.

3 Blue Gene/L System Software

To address the challenges of scalability and complexity posed by BG/L we have developed the system software architecture presented in Figure 2. This architecture is described in detail in this section.

3.1 System Software for the I/O Nodes

The Linux kernel that executes in the I/O nodes (currently version 2.4.19) is based on a standard distribution for PowerPC 440GP processors. Although Blue Gene/L uses standard PPC 440 cores, the overall chip and card design required changes in the booting sequence, interrupt management, memory layout, FPU support, and device drivers of the standard Linux kernel. There is no BIOS in the Blue Gene/L nodes, thus the configuration of a node after power-on and the initial program load (IPL) is initiated by the service nodes through the control network. We modified the interrupt and exception handling code to support Blue Gene/L's custom Interrupt Controller (BIC). The implementation of the kernel MMU remaps the tree and torus FIFOs to user space. We support the new EMAC4 Gigabit Ethernet controller. We also updated the kernel to save and restore the double FPU registers in each context switch.

The nodes in the Blue Gene/L machine are diskless, thus the initial root file system is provided by a ramdisk linked against the Linux kernel. The ramdisk contains shells, simple utilities, shared libraries, and network clients such as `ftp` and `nfs`.

Because of the non-coherent L1 caches, the current version of Linux runs on one of the 440 cores, while the second CPU is captured at boot time in an infinite loop. We are investigating two main strategies to effectively use the second CPU in the I/O nodes: SMP mode and virtual mode. We have successfully compiled a SMP version of the kernel, after implementing all the required interprocessor communications mechanisms, because the BG/L's BIC is not OpenPIC [6] compliant. In this mode, the TLB entries for the L1 cache are disabled in kernel mode and processes have affinity to one CPU. Forking a process in a different CPU requires additional parameters to the system call. The performance and effectiveness of this solution is still an open issue. A second, more promising mode of operation runs Linux in one of the CPUs, while the second CPU is the core of a virtual network card. In this scenario, the tree and torus FIFOs are not visible to the Linux kernel. Transfers between the two CPUs appear as virtual DMA transfers.

We are also investigating support for large pages. The standard PPC 440 embedded processors handle all TLB misses in software. Although the average number of instructions required to handle these misses has significantly decreased, it has been shown that larger pages improve performance [23].

3.2 System Software for the Compute Nodes

The "Blue Gene/L Run Time Supervisor" (BLRTS) is a custom kernel that runs on the compute nodes of a Blue Gene/L machine. BLRTS provides a simple, flat, fixed-size 256MB address space, with no paging, accomplishing a role similar to PUMA [19]. The kernel and application program share the same address space, with the kernel residing in protected memory at address 0 and the application program image loaded above, followed by its heap and stack. The kernel protects itself by appropriately programming the PowerPC MMU. Physical resources (torus, tree, mutexes, barriers, scratchpad) are partitioned between application and kernel. In the current implementation, the entire torus network is mapped into user space to obtain better communication efficiency, while one of the two tree channels is made available to the kernel and user applications.

BLRTS presents a familiar POSIX interface: we have ported the GNU Glibc runtime library and provided support for basic file I/O operations through system calls. Multi-processing services (such as `fork` and `exec`) are meaningless in single process kernel and have not been implemented. Program launch, termination, and file I/O is accomplished via messages passed between the compute node and its I/O node over the tree network, using a point-to-point packet addressing mode. This functionality is provided by a daemon called CIOD (Console I/O Daemon) running in the I/O nodes. CIOD provides job control and I/O management on behalf of all the compute nodes in the processing set. Under normal operation, all messaging between CIOD and BLRTS is synchronous: all file I/O operations are blocking on the application side. We used the CIOD in two scenarios:

1. driven by a console shell (called CIOMAN), used mostly for simulation and testing purposes. The user is provided with a restricted set of commands: run, kill, ps, set and unset environment variables. The shell distributes the commands to all the CIODs running in the simulation, which in turn take the appropriate actions for their compute nodes.
2. driven by a job scheduler (such as LoadLeveler) through a special interface that implements the same protocol as the one defined for CIOMAN and CIOD.

We are investigating a range of compute modes for our custom kernel. In *heater* mode, one CPU executes both user and network code, while the other CPU remains idle. This mode will be the mode of operation of the initial prototypes, but it is unlikely to be used afterwards. In *co-processor* mode, the application runs in a single, non-preemptable thread of execution on the main processor (cpu 0). The coprocessor (cpu 1) is used as a torus device off-load engine that runs as part of a user-level application library, communicating with the main processor through a non-cached region of shared memory. In *symmetric* mode, both CPUs run applications and users are responsible for explicitly handling cache coherence. In *virtual node* mode we provide support for two independent processes in a node. The system then looks like a machine with 128K nodes.

3.3 System Management Software

The control infrastructure is a critical component of our design. It provides a separation between execution mechanisms in the BG/L core and policy decisions in external nodes. Local node operating systems (Linux for I/O nodes and BLRTS for compute nodes) implement services and are responsible for local decisions that do not affect overall operation of the machine. A “global operating system” makes all global and collective decisions and interfaces with external policy modules (e.g., LoadLeveler) and performs a variety of system management services, including: (i) machine booting, (ii) system monitoring, and (iii) job launching.

In our implementation, the global OS runs on external service nodes. Each BG/L midplane is controlled by one Midplane Management and Control System (MMCS) process which provides two paths into the Blue Gene/L complex: a custom control library to access the restricted JTAG network and directly manipulate Blue Gene/L nodes; and sockets over the Gbit/s Ethernet network to manage the nodes on a booted partition.

The custom control library can perform:

- low level hardware operations such as: turn on power supplies, monitor temperature sensors and fans, and react accordingly (i.e. shut down a machine if temperature exceeds some threshold),
- configure and initialize IDo, Link and BG/L chips,
- read and write configuration registers, SRAM and reset the cores of a BG/L chip. As mentioned in Section 2, these operations can be performed with no code executing in the nodes, which permits machine initialization and boot, nonintrusive access to performance counters and post-mortem debugging.

This path into the core is used for control only; for security and reliability reasons, it is not made visible to applications running in the BG/L nodes. On the other hand,

the architected path through the functional Gbit/s Ethernet is used for application I/O, checkpoints, and job launch.

We chose to maintain the entire state of the global operating system using standard database technology. Databases naturally provide scalability, reliability, security, portability, logging, and robustness. The database contains static state (i.e., the physical connections between each HW component) and dynamic state (i.e., how the machine is partitioned, how each partition is configured, and which parallel application is running in each partition). Therefore, the database is not just a repository for read-only configuration information, but also an interface for all the visible state of a machine. External entities (such as a scheduler) can manipulate this state by invoking stored procedures and database triggers, which in turn invoke functions in the MMCS processes.

Machine Initialization and Booting. The boot process for a node consists of the following steps: first, a small boot loader is directly written into the (compute or I/O) node memory by the service nodes using the JTAG control network. This boot loader loads a much larger boot image into the memory of the node through a custom JTAG mailbox protocol.

We use one boot image for all the compute nodes and another boot image for all the I/O nodes. The boot image for the compute nodes contains the code for the compute node kernel, and is approximately 64 kB in size. The boot image for the I/O nodes contains the code for the Linux operating system (approximately 2 MB in size) and the image of a ramdisk that contains the root file system for the I/O node. After an I/O node boots, it can mount additional file systems from external file servers. Since the same boot image is used for each node, additional node specific configuration information (such as torus coordinates, tree addresses, MAC or IP addresses) must be loaded. We call this information the *personality* of a node. In the I/O nodes, the personality is exposed to user processes through an entry in the `proc` file system. BLRTS implements a system call to request the node's personality.

System Monitoring in Blue Gene/L is accomplished through a combination of I/O node and service node functionality. Each I/O node is a full Linux machine and uses Linux services to generate system logs.

A complementary monitoring service for Blue Gene/L is implemented by the service node through the control network. Device information, such as fan speeds and power supply voltages, can be obtained directly by the service node through the control network. The compute and I/O nodes use a communication protocol to report events that can be logged or acted upon by the service node. This approach establishes a completely separate monitoring service that is independent of any other infrastructure (tree and torus networks, I/O nodes, Ethernet network) in the system. Therefore, it can be used even in the case of many system-wide failures to retrieve important information.

Job Execution is also accomplished through a combination of I/O nodes and service node functionality. When submitting a job for execution in Blue Gene/L, the user specifies the desired shape and size of the partition to execute that job. The scheduler

selects a set of compute nodes to form the partition. The compute (and corresponding I/O) nodes selected by the scheduler are configured into a partition by the service node using the control network. We have developed techniques for efficient allocation of nodes in a toroidal machine that are applicable to Blue Gene/L [16]. Once a partition is created, a job can be launched through the I/O nodes in that partition using CIOD as explained before.

3.4 Compiler and Run-Time Support

Blue Gene/L presents a familiar programming model and a standard set of tools. We have ported the GNU toolchain (binutils, gcc, glibc and gdb) to Blue Gene/L and set it up as a cross-compilation environment. There are two cross-targets: Linux for I/O nodes and BLRTS for compute nodes. IBM's XL compiler suite is also being ported to provide advanced optimization support for languages like Fortran90 and C++.

3.5 Communication Infrastructure

The Blue Gene/L communication software architecture is organized into three layers: the *packet layer* is a thin software library that allows access to network hardware; the *message layer* provides a low-latency, high bandwidth point-to-point message delivery system; *MPI* is the user level communication library.

The packet layer simplifies access to the Blue Gene/L network hardware. The packet layer abstracts FIFOs and devices control registers into torus and tree *devices* and presents an API consisting of essentially three functions: initialization, packet send and packet receive. The packet layer provides a mechanism to use the network hardware but doesn't impose any policies on its use. Hardware restrictions, such as the 256 byte limit on packet size and the 16 byte alignment requirements on packet buffers, are not abstracted by the packet layer and thus are reflected by the API. All packet layer send and receive operations are non-blocking, leaving it up to the higher layers to implement synchronous, blocking and/or interrupt driven communication models. In its current implementation the packet layer is stateless.

The message layer is a simple active message system [12,17,21,22], built on top of the torus packet layer, which allows the transmission of arbitrary messages among torus nodes. It is designed to enable the implementation of MPI point-to-point send/receive operations. It has the following characteristics:

No packet retransmission protocol. The Blue Gene/L network hardware is completely reliable, and thus a packet retransmission system (such as a sliding window protocol) is unnecessary. This allows for stateless virtual connections between pairs of nodes, greatly enhancing scalability.

Packetizing and alignment. The packet layer requires data to be sent in 256 byte chunks aligned at 16 byte boundaries. Thus the message layer deals with the packetizing and re-alignment of message data. Re-alignment of packet data typically entails memory-to-memory copies.

Packet ordering. Packets on the torus network can arrive out of order, which makes message re-assembly at the receiver non-trivial. For packets belonging to the same message, the message layer is able to handle their arrival in any order. To restore order for packets belonging to *different* messages, the sender assigns ascending numbers to individual messages sent out to the same peer.

Cache coherence and processor use policy. The expected performance of the message layer is influenced by the way in which the two processors are used (as discussed in Section 3.2). Co-processor mode is the only one that effectively overlaps computation and communication. This mode is expected to yield better bandwidth, but slightly higher latency, than the others.

MPI: Blue Gene/L is designed primarily to run MPI [20] workloads. We are in the process of porting MPICH2 [5], currently under development at Argonne National Laboratories, to the Blue Gene/L hardware. MPICH2 has a modular architecture. The Blue Gene/L port leaves the code structure of MPICH2 intact, but adds a number of plug-in modules:

Point-to-point messages. The most important addition of the Blue Gene/L port is an implementation of ADI3, the MPICH2 Abstract Device Interface [14]. A thin layer of code transforms e.g. MPI Request objects and MPI_Send function calls into calls into sequences of message layer function calls and callbacks.

Process management. The MPICH2 process management primitives are documented in [7]. Process management is split into two parts: a process management interface (PMI), called from within the MPI library, and a set of process managers (PM) which are responsible for starting up/shutting down MPI jobs and implementing the PMI functions.

MPICH2 includes a number of process managers (PM) suited for clusters of general purpose workstations. The Blue Gene/L process manager makes full use of its hierarchical system management software, including the CIOD processes running on the I/O nodes, to start up and shut down MPI jobs. The Blue Gene/L system management software is explicitly designed to deal with the scalability problem inherent in starting, synchronizing and killing 65,536 MPI processes.

Optimized collectives. The default implementation of MPI collective operations in MPICH2 generates sequences of point-to-point messages. This implementation is oblivious of the underlying physical topology of the torus and tree networks. In Blue Gene/L optimized collective operations can be implemented for communicators whose physical layouts conform to certain properties.

- The torus hardware can be used to efficiently implement broadcasts on contiguous 1, 2 and 3 dimensional meshes, using a feature of the torus that allows depositing a packet on every node it traverses. The collectives best suited for this (e.g. Bcast, Allgather, Alltoall, Barrier) involve broadcast in some form.
- The tree hardware can be used for almost every collective that is executed on the MPI_COMM_WORLD communicator, including reduction operations. Integer operand reductions are directly supported by hardware. IEEE compliant floating point reductions can also be implemented by the tree using separate reduction phases for the mantissa and the exponent.

- Non MPI_COMM_WORLD collectives can also be implemented using the tree, but care must be taken to ensure deadlock free operation. The tree is a locally class routed network, with packets belonging to one of a small number of classes and tree nodes making local decisions about routing. The tree network guarantees deadlock-free simultaneous delivery of no more than two class routes. One of these routes is used for control and file I/O purposes; the other is available for use by collectives.

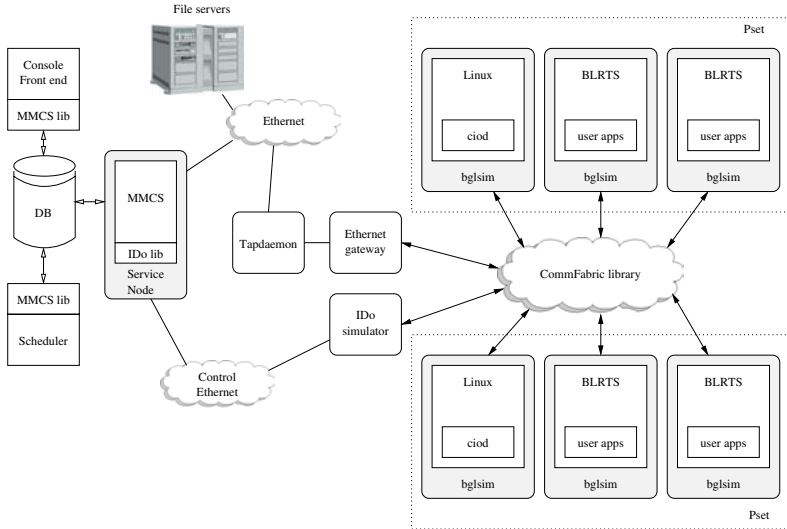


Fig. 3. Overview of the Blue Gene/L simulation environment. Complete Blue Gene/L chips are simulated by a custom architectural simulator (`bglsim`). A communication library (`CommFabric`) simulates the Blue Gene/L networks.

4 Blue Gene/L Simulation Environment

The first hardware prototypes of the Blue Gene/L ASIC are targeted to become operational in mid-2003. To support the development of system software before hardware is available, we have implemented an architecturally accurate, full system simulator for the Blue Gene/L machine [11]. The node simulator, called `bglsim`, is built using techniques described in [18,15]. Each component of the BG/L ASIC is modeled separately with the desired degree of accuracy, trading accuracy for performance. In our simulation environment, we model the functionality of processor instructions. That is, each instruction correctly changes the visible architectural state, while it takes one cycle to execute. We also model memory system behavior (cache, scratch-pad, and main memory) and all the Blue Gene/L specific devices: tree, torus, JTAG, device control registers, etc. A `bglsim` process boots the Linux kernel for the I/O nodes and BLRTS for the compute nodes. Applications run on top of these kernels, under user control.

When running on 1.2 GHz Pentium III machine, `bglsim` simulates an entire BG/L chip at approximately 2 million simulated instructions per second – a slow-down of about 1000 compared to the real hardware. By comparison, a VHDL simulator with hardware acceleration has a slow-down of 10^6 , while a software VHDL simulator has a slow-down of 10^9 . As an example, booting Linux takes 30 seconds on `bglsim`, 7 hours on the hardware accelerated VHDL simulator and more than 20 days on the software VHDL simulator.

Large Blue Gene/L system are simulated using one `bglsim` process per node, as shown in Figure 3. The `bglsim` processes run on different workstations and communicate through a custom message passing library (*CommFabric*), which simulates the connectivity within the system and outside. Additionally, different components of the system are simulated by separate processes that also link in *CommFabric*. Examples are: the IDo chip simulator, a functional simulator of an IDo chip that translates packets between the virtual JTAG network and Ethernet; the Tapdaemon and EthernetGateway processes to provide the Linux kernels in the simulated I/O nodes with connectivity to the outside network, allowing users to mount external file-systems and connect using telnet, ftp, etc. We use this environment to develop our communication infrastructure, the control infrastructure and we have successfully executed the MPI NAS Parallel Benchmarks [9].

5 Conclusions

Blue Gene/L is the first of a new series of high performance machines being developed at IBM Research. The hardware plans for the machine are complete and the first small prototypes will be available in late 2003.

In this paper, we have presented a software system that can scale up to the demands of the Blue Gene/L hardware. We have also described the simulation environment that we are using to develop and validate this software system. Using the simulation environment, we are able to demonstrate a complete and functional system software environment before hardware becomes available. Nevertheless, evaluating scalability and performance of the complete system still requires hardware availability. Many of the implementation details will likely change as we gain experience with the real hardware.

References

1. ASCI Red Homepage. <http://www.sandia.gov/ASCI/Red/>.
2. ASCI White Homepage. <http://www.llnl.gov/asci/platforms/white>.
3. Cplant homepage. <http://www.cs.sandia.gov/cplant/>.
4. Earth Simulator Homepage. <http://www.es.jamstec.go.jp/>.
5. The MPICH and MPICH2 homepage. <http://www-unix.mcs.anl.gov/mpi/mpich>.
6. *Open Firmware Homepage*. <http://www.openfirmware.org>.
7. Process Management in MPICH2. Personal communication from William Gropp.
8. N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.

9. G. Almasi, C. Archer, J. G. Castanos, M. G. X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. MPI on BlueGene/L: Designing an Efficient General Purpose Messaging Solution for a Large Cellular System. Submitted for publication to the 2003 Euro PVM/MPI workshop.
10. G. Almasi et al. Cellular supercomputing with system-on-a-chip. In *IEEE International Solid-state Circuits Conference ISSCC*, 2001.
11. L. Ceze, K. Strauss, G. Almási, P. J. Bohrer, J. R. Brunheroto, C. Caşcaval, J. G. C. nos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full circle: Simulating linux clusters on linux clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, June 2003.
12. G. Chiola and G. Ciaccio. Gamma: a low cost network of workstations based on active messages. In *Proc. Euromicro PDP'97, London, UK, January 1997, IEEE Computer Society*, 1997.
13. D. Greenberg, R. Brightwell, L. Fisk, A. Maccabe, and R. Riesen. A system software architecture for high-end computing. In *Proceedings of Supercomputing 97*, San Jose, CA, 1997.
14. W. Gropp and E. Lusk. MPICH Abstract Device Interface Version 3.3 Reference Manual: Draft of October 17, 2002. <http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf>.
15. S. A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1988.
16. E. Krevat, J. Castanos, and J. Moreira. Job scheduling for the Blue Gene/L system. In *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 38–54. Springer, 2002.
17. S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95, San Diego, CA, December 1999*, 1995.
18. M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 1995.
19. L. Shuler, R. Riesen, C. Jong, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *In Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995.
20. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference, second edition*. The MIT Press, 2000.
21. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 1995.
22. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schausser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.
23. S. J. Winwood, Y. Shuf, and H. Franke. Multiple page size support in the Linux kernel. In *Proceedings of Ottawa Linux Symposium*, Ottawa, Canada, June 2002.

Trace Substitution

Hans Vandierendonck, Hans Logie, and Koen De Bosschere

Dept. of Electronics and Information Systems

Ghent University

Sint-Pietersnieuwstraat 41

B-9000 Gent, Belgium

`{hvdieren,kdb}@elis.rug.ac.be`

Abstract. Trace caches deliver a high number of instructions per cycle to wide-issue superscalar processors. To overcome complex control flow, multiple branch predictors have to predict up to 3 conditional branches per cycle. These multiple branch predictors sometimes predict completely wrong paths of execution, degrading the average fetch bandwidth. This paper shows that such mispredictions can be detected by monitoring trace cache misses. Based on this observation, a new technique called trace substitution is introduced. On a trace cache miss, trace substitution overrides the predicted trace with a cached trace. If the substitution is correct, the fetch bandwidth increases. We show that trace substitution consistently improves the fetch bandwidth with 0.2 instructions per access. For inaccurate predictors, trace substitution can increase the fetch bandwidth with up to 2 instructions per access.

1 Introduction

Many applications have short basic blocks and branches that are difficult to predict. This property makes it difficult to sustain a high fetch bandwidth and thereby limits the maximum IPC. Trace caches offer a solution to this problem. Instructions that are executed consecutively are packed together into a trace and are cached as one unit for future reference. A multiple branch predictor predicts the followed path, which is used to retrieve the correct trace. A trace cache typically requires 3 branch predictions per cycle. The accuracy of the multiple branch predictor is of extreme importance, as trace caches are targeted at very wide-issue machines.

As branches are typically correlated to each other, certain paths or combinations of branches will be very unlikely and will almost never occur during the execution of a program. However, destructive interference in a multiple branch predictor may result in predicting these paths anyway. This paper studies the occurrence of such “obviously wrong” paths and shows that they are identified by trace cache misses. Based on this observation, we propose *trace substitution*, a simple and elegant technique to increase average fetch bandwidth. When an “obviously wrong” trace prediction is detected, a *cached* trace is substituted for the predicted trace. This technique, although very simple and requiring little additional hardware, consistently improves the fetch bandwidth.

The remainder of this paper is organised as follows. Section 2 discusses the trace cache and multiple branch predictor models and introduces trace substitution. Section 3 studies opportunities and limitations of trace substitutions and evaluates its impact for varying trace cache sizes and branch predictor sizes. Related work is discussed in section 4 and section 5 concludes the paper.

2 Trace Caches and Trace Substitution

2.1 The Trace Cache

The trace cache mechanism packs consecutively executed instructions into traces of instructions and stores them in the trace cache as a single unit [1,2]. The trace cache is typically organised as a set-associative cache. Each trace consists of at most 16 instructions and 3 conditional branches.

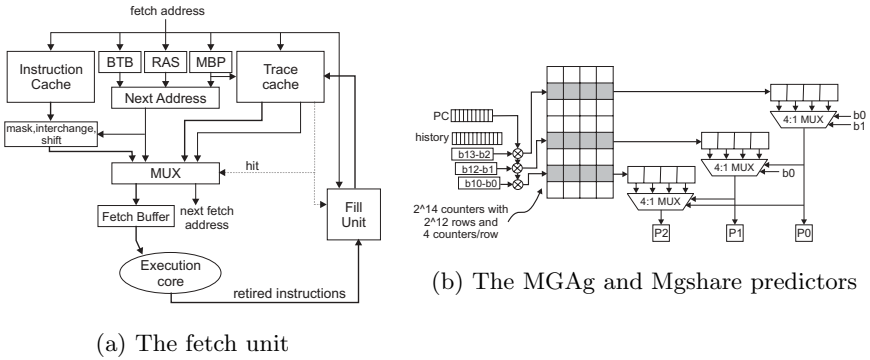


Fig. 1. Block diagram of the fetch unit containing the trace cache, the multiple branch predictor and the instruction cache.

We use the trace cache organisation of [1,3,4] (Figure 1(a)). In this organisation, the instruction cache (a.k.a. core fetch unit) and the trace cache are probed in parallel with the fetch address. The multiple branch predictor predicts the direction of three conditional branches. The branch directions and the fetch address are used to perform tag matching in the trace cache, i.e., we use path-associativity [5], meaning that multiple traces can be present starting at the same program counter. When the trace cache hits, the trace is forwarded to the processor. On a miss, 16 instructions in two consecutive cache blocks are fetched from the instruction cache. The branch target buffer (BTB) and the predicted path are used to determine how many instructions should be forwarded to the processor. The instruction cache can deliver only 1 taken branch per access.

When the trace cache misses, the fill unit is instructed to start building a trace. The fill unit only builds traces on the correct path of execution using retired instructions.

2.2 Multiple Branch Prediction

The MGAg multiple branch predictor was proposed in [1,6] (Figure 1(b)). In the MGAg, the global branch history is used as an index into a pattern history table (the PC is ignored in this case). The MGAg is a direct extension of the GAg branch predictor [7] to trace caches. Conceptually, the MGAg is a GAg that is accessed three times using the global branch history, which is updated between accesses with the predicted branch directions. This idea is implemented with three parallel pattern history table (PHT) accesses [6].

The Mshare is derived from the MGAg predictor. In a gshare predictor that is iterated three times, the PHT is accessed each time using the exclusive-or (XOR) of the branch address and the global branch history [8]. When accessing a trace cache, only one branch address is available (i.e., the starting address of a trace). Hence, we XOR this address with each of the three PHT indices.

2.3 Trace Substitution

The trace cache only stores traces along paths of execution that have actually been followed. The multiple branch predictor has to predict the future path of execution and sometimes (due to destructive interference) it predicts paths that have not been executed yet. These paths are likely to be wrongly predicted and we can use the trace cache to detect them. We show in section 3.2 that a trace cache miss is a good indicator for incorrectly predicted branches.

As a trace cache miss is a tell-tale for branch mispredictions, its occurrence can be used to improve fetch bandwidth. We argued that traces stored in the trace cache are more likely to be on the correct execution path than other traces. Thus, when a trace cache miss occurs, we return a different trace instead. In particular, we return the most recently used trace with the same starting address. We call this technique *trace substitution*.

Trace substitution requires little additional hardware. First, the hit/miss indicator of the trace cache is reused to trigger trace substitution. Secondly, a second, parallel, tag match is required to find the most recently used trace with the requested starting address. Because the trace cache is indexed using this starting address, only one cache set needs to be searched. The most recently used trace is derived from the LRU bits of the replacement policy in the trace cache. The third hardware addition is that the multiplexor that selects the returned trace has to take into account the hit/miss indicator and the second tag match. This will probably increase the critical path of the trace cache with one multiplexor delay.

The fill unit builds a trace when a trace cache miss occurs. We modify this behaviour and do not build a trace when the substituted trace is on the correct path. It is still possible for new traces to enter the trace cache. When a new trace is seen, then (i) a trace cache miss occurs, (ii) trace substitution is possibly applied and (iii) the substituted trace (if any) is on the wrong path, so the new trace is built after all and stored in the trace cache. If no trace could be found to substitute, then the trace is built and stored anyway.

3 Evaluation

We evaluate trace substitution using 6 SPECint95 benchmarks running training inputs. The harmonic mean is labelled as H-mean. The benchmarks are compiled with Compaq’s C compiler using `-O4` optimisation. At most 500 million instructions are simulated per benchmark. The baseline fetch unit has a 256-entry trace cache organised as a 4-way set-associative cache. Each trace can contain up to 16 instructions and 3 conditional branches. Traces are terminated on indirect branches and system calls. The baseline MGAg predictor uses 12 bits of global branch history and the instruction cache is 32-KB large with 32-byte blocks and is 2-way set-associative.

3.1 The Branch Predictor Changes the Trace Cache Hit Rate

The overall performance measure FIPA (fetched instructions per trace cache access) depends on both the branch prediction accuracy and trace cache miss rate in a complex way. When the branch predictor is improved and the branch prediction accuracy increases, one also expects to see the FIPA increase. This is not always the case. We investigated a 12-bit history MGAg and Mgshare, an 8-kbit hybrid predictor that is optimistically probed three times in succession and a perfect branch predictor. Still other predictors were investigated, but are not shown. The predictors are evaluated in a large 16K-entry 4-way set-associative trace cache in order to mitigate the impact of trace cache misses (Figure 2). For many benchmarks, the Mgshare has a higher branch prediction accuracy but a lower FIPA than the MGAg predictor. The trace cache miss rate seems to correlate better to the FIPA, but the trace cache hit rate for the Mgshare predictor is unproportionally low.

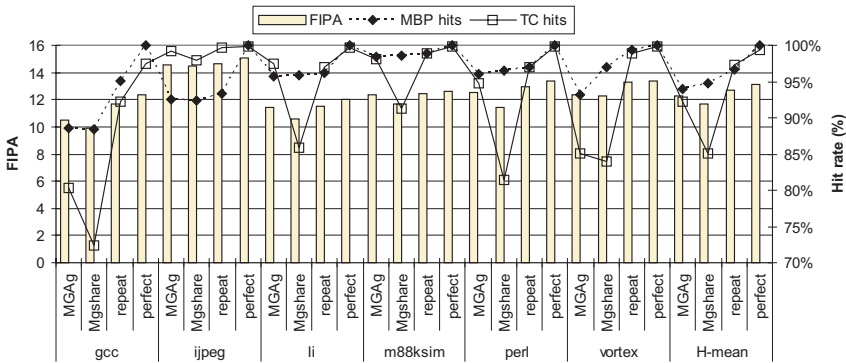


Fig. 2. Fetch throughput (FIPA), branch prediction accuracy and trace cache hit rate in a 16K-entry trace cache.

These results clearly show that the branch predictor has a strong influence on the trace cache hit rate. When the branch predictor is perfect, the trace cache almost never misses. As the trace cache is very large (16K entries), it can contain practically any trace that is required, so trace cache misses should not occur. However, when the branch predictor becomes imperfect, the number of trace cache misses increases significantly.

3.2 On a Trace Cache Miss, the Branch Predictor Is Likely Wrong

The strong dependence of the trace cache miss rate on the branch prediction suggests that the branch predictor produces, at times, predictions that are so unlikely that the trace cache cannot provide them. This effect is measured for a 12-bit history MGAg predictor (Table 1). The left half of this table shows the fraction of trace cache misses that are due to a path misprediction. When the trace cache is sufficiently large, over 80% of trace cache misses is caused by an incorrect branch prediction. In the smaller trace caches, many misses occur due to contention (capacity misses) and are not related to the branch predictor.

Table 1. Fraction of trace cache misses where the path is mispredicted (left) and the fraction of path mispredictions that are caught by monitoring trace cache misses. The trace cache size is varied and the branch history length is 12 bits.

TC size	Fraction of trace cache misses					Fraction of path mispredictions				
	64	256	1024	4096	16384	64	256	1024	4096	16384
gcc	28.3%	34.0%	51.7%	78.5%	86.7%	89.5%	85.8%	78.4%	69.0%	63.6%
jpeg	7.1%	19.5%	65.7%	84.0%	85.0%	44.1%	27.6%	16.5%	11.4%	10.5%
li	10.8%	35.6%	87.3%	92.5%	92.6%	39.0%	29.6%	24.3%	23.3%	23.2%
m88ksim	9.8%	21.4%	55.9%	81.3%	82.9%	82.5%	63.0%	53.5%	46.1%	46.2%
perl	8.7%	17.6%	51.6%	80.1%	80.4%	84.9%	71.0%	51.5%	45.7%	45.0%
vortex	15.8%	28.0%	50.7%	76.7%	82.3%	98.7%	97.8%	95.0%	91.6%	88.3%
H-mean	12.0%	25.1%	59.3%	82.0%	84.9%	68.9%	55.9%	44.9%	39.0%	37.6%

The right part of Table 1 shows the fraction of mispredicted traces that actually lead to a trace cache miss. In a small trace cache, many misses occur, so there is a large probability of detecting path mispredictions. Up to 70% of the mispredicted paths can be detected. As the trace cache becomes larger, just around 40% of the path mispredictions can be detected.

The branch prediction accuracy also has an important impact on the applicability and utility of trace substitution. We held the trace cache size constant at 256 entries and varied the branch history length of the MGAg predictor (Table 2). On average, between 14% and 71% of the trace cache misses indicate path mispredictions. The fraction of mispredicted paths that can be detected by trace cache misses decreases with increasing branch predictor size. This result is somewhat surprising, as one would expect that a higher prediction accuracy

Table 2. Fraction of trace cache misses where the path is mispredicted (left) and the fraction of path mispredictions that are caught by monitoring trace cache misses. The branch history length is varied and the trace cache is held constant at 256 traces.

hist. length	Fraction of trace cache misses					Fraction of path mispredictions				
	0	4	8	12	16	0	4	8	12	16
gcc	67.4%	63.6%	57.6%	34.0%	14.8%	88.4%	88.9%	89.4%	85.8%	77.1%
jpeg	56.0%	46.7%	34.8%	19.5%	12.8%	57.2%	51.4%	41.9%	27.6%	21.6%
li	91.7%	85.2%	63.4%	35.6%	23.2%	67.2%	66.5%	53.4%	29.6%	22.1%
m88ksim	79.4%	80.0%	50.3%	21.4%	10.5%	62.5%	95.3%	84.4%	63.0%	48.7%
perl	77.3%	77.7%	54.3%	17.6%	10.8%	90.3%	91.3%	87.6%	71.0%	58.3%
vortex	57.9%	56.7%	46.0%	28.0%	14.3%	98.8%	98.3%	98.4%	97.8%	96.7%
H-mean	70.5%	66.8%	50.2%	25.1%	13.9%	75.8%	79.9%	72.5%	55.9%	46.5%

combined with a (more or less) constant trace cache miss rate would result in an increase in the fraction of detected mispredictions. However, not all path mispredictions will miss the trace cache. Those that do are “obvious” mispredictions; those that do not are harder to detect. As the branch predictor becomes larger, the amount of “obvious” mispredictions decreases faster than the others.

Trace substitution can only work when the trace on the correct path is present in the trace cache. If it is not, then there is no point in returning another trace that is also on the wrong path. We have found that the correct path trace is present in the trace cache for 35%–87% of the mispredicted paths that miss in the trace cache, for trace caches with 64 to 16K traces and the baseline predictor. When varying the predictor size, the correct trace is available in 6%–38% of the cases. Hence, when a mispredicted path is detected, it will not always be possible to substitute the correct trace. However, even when the correct trace can not be substituted, no harm is done in trying, as both the predicted path and any other wrong-path trace are on the wrong path of execution.

3.3 Trace Substitution

Now that we have discussed the potentials and limitations of trace substitution, let us investigate its impact on performance. We measured the FIPA for a 12-bit history MGAg predictor and trace cache of sizes 64 to 16K traces, all 4-way set-associative (Figure 3(a)). The FIPA is shown for the baseline MGAg predictor, the MGAg predictor with trace substitution and a perfect branch predictor. In the small trace caches, the prominent performance bottleneck is trace cache misses related to the limited capacity of the trace cache. Hence, a perfect branch predictor does not improve performance much. As the trace cache becomes larger, the multiple branch predictor becomes a more important bottleneck. Consequently, the gap between the curves for the MGAg and perfect predictors increases with increasing trace cache size.

Trace substitution increases the FIPA obtainable with the MGAg predictor; around 20% to 40% of the gap between the baseline MGAg predictor and the

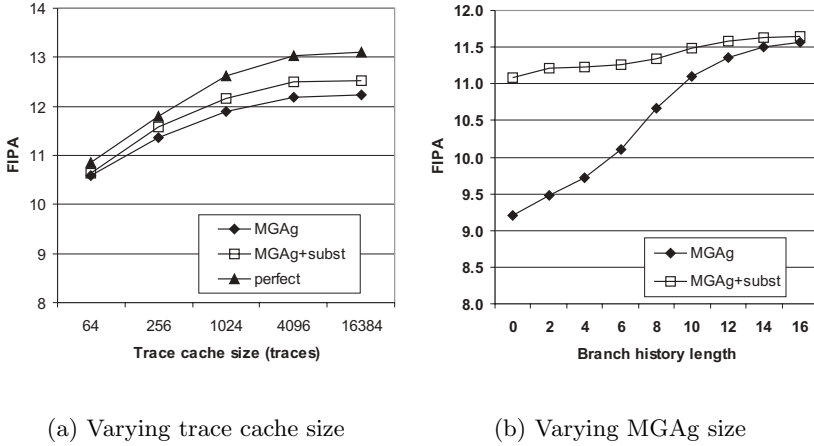


Fig. 3. The impact of trace substitution on performance (FIPA).

perfect predictor is closed. However, as trace substitution only kicks in on a trace cache miss, the obtainable performance increase strongly depends on the trace cache miss rate. Figure 3 shows that less and less of the performance potential is reached with larger trace caches. The cause of this can be found in Table 1: as the trace cache becomes larger, fewer path mispredictions are detected. The performance increase therefore stagnates as the trace cache becomes larger.

Trace substitution can hide the imperfectness of a poor branch predictor. When varying the history length (and simultaneously the size) of the MGAg branch predictor, the benefits of trace substitution become much clearer. In this experiment, we used a 256-entry trace cache (Figure 3(b)). For the small and inaccurate branch predictors, trace substitution improves performance with 1 to 2 useful instructions fetched per trace cache access. Even for the MGAg with zero bits of global branch history, trace substitution raises the FIPA to the same level as a 10-bit history MGAg without trace substitution.

For the largest of branch predictors, the FIPA is increased over a small amount, e.g., 0.2 instructions per access for the baseline predictor. More importantly, trace substitution enables one to achieve the same performance with a 16 times smaller branch predictor (e.g., 10 vs. 14 bits of history).

Trace substitution depends totally on the co-occurrence of a mispredicted path and a trace cache miss. Hence, the utility decreases for a large trace cache (fewer trace cache misses and fewer possibilities to detect mispredicted paths) and larger branch predictors (fewer path misprediction and thus lack of reason to substitute the trace). Figure 4 shows the usage and accuracy of trace substitution. The usage is defined as the fraction of trace cache accesses for which a trace was substituted. The accuracy is defined as the ratio of the correct substitutions over the total number of substitutions. The usage drops of with larger

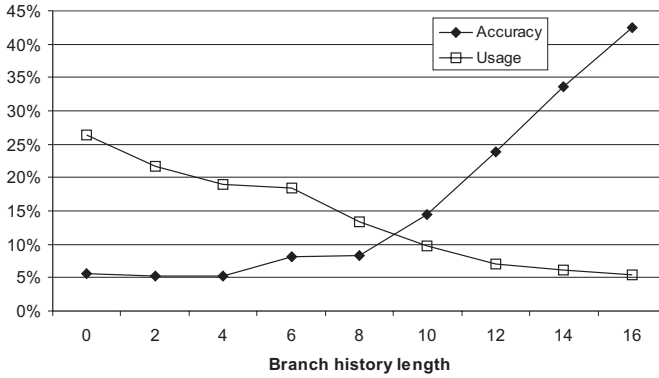


Fig. 4. Accuracy of trace substitution in the baseline trace cache.

branch predictors, as more traces are predicted correctly by the branch predictor. On the other hand, when trace substitution is applicable, it is more accurate in larger branch predictors.

Overall, the accuracy of trace substitution is low: the correct trace is returned in less than half of the cases. In the reported experiments, we always returned the most recently used trace having the correct start address. Experiments were made with other choices, e.g., returning a randomly selected trace and a state-based predictor, but these were found to have approximately the same accuracy. Notwithstanding its low accuracy, trace substitution improves the average fetch bandwidth. The reason for having a performance improvement with such low accuracy is that, assuming that the wrong path is predicted on a trace cache miss, it does not matter much whether we return a wrong-path trace from the trace cache, or fetch along another wrong path using the instruction cache. In the end, both paths are wrong.

Another reason for the low accuracy is that the correct-path trace is not always present in the trace cache. Trace substitution is applied as soon as any trace with the correct start address is found but this trace is not always on the correct path of execution. In section 3.2 we estimated that the accuracy is limited to 6%–38%, depending on the branch predictor size, exactly for this reason.

We have also performed the above experiments with the proposed Mgshare predictor. This predictor is by itself less accurate than the MGAg predictor. However, when trace substitution was added, it performed as good as the MGAg with trace substitution. Lack of space prohibits us from treating this in detail.

4 Related Work

Trace caches were introduced to increase the fetch bandwidth for superscalar processors over the single basic block limit [1,2,9]. Two approaches for multiple

branch prediction can be discerned. The first approach is to adapt a single-branch predictor to multiple branch prediction. This approach was taken in [1, 2, 5, 6, 10] and is also followed in this paper. A second approach is to predict full traces at once and has mainly been investigated in conjunction with block-based trace caches [11, 12, 13]. In [13], it is proposed to predict the next trace at completion time. This has the advantage that the predictor can be indexed using non-speculative history information. Trace substitution is not useful when only cached traces are predicted, as in [11, 12].

Selective trace storage does not store *blue* traces (i.e.: traces containing only not-taken branches) in the trace cache, as these can be retrieved equally fast from the instruction cache [3]. Trace substitution can be adapted for selective trace storage by adding additional tags to each set of the trace cache. These tags can only be used for blue traces and indicate that a specific blue path has been followed. The blue trace is still stored in the instruction cache. Trace substitution is not applied when a blue tag hits.

Trace substitution is an idea similar to branch inversion [14], a technique applicable to single-branch predictors. With branch inversion, a confidence predictor is used to detect wrong branch predictions. When a branch prediction is assigned low confidence, the predicted branch direction is inverted. Trace substitution does not require a confidence predictor, but is triggered by trace cache misses. On the other hand, trace substitution is more complicated than branch inversion because in the former a different trace has to be predicted while in the latter, it suffices to simply invert the branch direction.

5 Conclusion

Multiple branch predictors sometimes predict a path of execution that is so unlikely that the trace cache cannot provide it. This paper showed that trace cache misses can be used to detect such mispredictions. We proposed *trace substitution*, a new technique that returns a cached trace when the predicted trace misses in the trace cache. We showed that between 40%–76% of the branch mispredictions can be detected by trace cache misses, depending on the trace cache and branch predictor sizes. Furthermore, between 12%–85% of the trace cache misses are indeed caused by mispredicted branches. The low numbers correspond to small trace caches with many capacity misses.

Trace substitution increases the fetch bandwidth consistently. For small branch predictors, the FIPA is improved with 1 to 2 useful instructions per cycle, obtaining the fetch bandwidth of a 10 to 12-bit history MGAg. For large branch predictors, the FIPA is increased with 0.2 instructions per access or the same FIPA can be obtained with a 16-times smaller branch predictor.

Acknowledgements. The authors thank the reviewers for their helpful comments. Hans Vandierendonck is supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT).

References

1. E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Conference on Microprogramming and Microarchitecture*, pp. 24–35, Dec. 1996.
2. S. Patel, M. Evers, and Y. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 262–271, June 1998.
3. A. Ramírez, J. Larriba-Pey, and M. Valero, "Trace cache redundancy: Red & blue traces," in *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, Jan. 2000.
4. H. Vandierendonck, A. Ramírez, K. De Bosschere, and M. Valero, "A comparative study of redundancy in trace caches," in *EuroPar 2002*, pp. 512–516, Aug. 2002.
5. S. Patel, D. Friendly, and Y. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers*, vol. 48, pp. 193–204, Feb. 1999.
6. E. Rotenberg, S. Bennet, and J. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," Tech. Rep. 1310, University of Wisconsin - Madison, Apr. 1996.
7. T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Conference on Computer Architecture*, pp. 124–134, May 1992.
8. S. McFarling, "Combining branch predictors," Tech. Rep. WRL-TN36, Western Research Laboratory, 250 University Avenue, Palo Alto, California 94301, USA, June 1993.
9. A. Peleg and U. Weiser, "Dynamic flow instruction cache memory organized around trace segments independent of virtual address line," *U.S. Patent Number 5.381.533*, Jan. 1995.
10. T.-Y. Yeh, D. Marr, and Y. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch access cache," in *ICS'93. Proceedings of the 1993 ACM International Conference on Supercomputing*, pp. 67–76, July 1993.
11. B. Black, B. Rychlik, and J. Shen, "The block-based trace cache," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 196–207, May 1999.
12. Q. Jacobson, E. Rotenberg, and J. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Conference on Microprogramming and Microarchitecture*, pp. 14–23, Dec. 1997.
13. R. Rakvic, B. Black, and J. Shen, "Completion time multiple branch prediction for enhancing trace cache performance," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 47–58, May 2000.
14. S. Manne, A. Klauser, and D. Grunwald, "Branch prediction using selective branch inversion," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 48–56, Oct. 1999.

Optimizing a Decoupled Front-End Architecture: The Indexed Fetch Target Buffer (iFTB)

Juan C. Moure, Dolores I. Rexachs, and Emilio Luque¹

Computer Architecture and Operating Systems Group,
Universidad Autónoma de Barcelona. 08193 Barcelona (Spain)
{JuanCarlos.Moure, Dolores.Rexachs, Emilio.Luque}@uab.es

Abstract. A decoupled front-end architecture relies on a separate fetch predictor that is critical in determining the fetch bandwidth. The Fetch Target Buffer (FTB) is a multilevel fetch predictor design. In this paper we propose and analyze a variation of the FTB, the *indexed Fetch Target Buffer* (iFTB), that uses an extra, very fast, index table to speedup its operations. The iFTB achieves very accurate predictions by adapting a complex hybrid branch predictor to suit its short cycle time. With a low hardware cost, the iFTB delivers instructions between 15% and 45% faster than the FTB.

1 Introduction

The performance of a processor is limited by the performance of each of its parts. The front-end is one of the most critical elements. A *decoupled* front-end architecture is able to provide high fetch bandwidth while maintaining scalability, [13].

As shown in Fig. 1, a decoupled front-end isolates the prediction task from the rest of the front-end's tasks (fetching the actual instructions and preparing them for execution). The *fetch predictor* generates references to fetch blocks and inserts them into a decoupling *fetch target queue* (FTQ). A *fetch block* is a set of instructions stored into consecutive memory addresses, and is defined by its starting address and size. One fetch block may involve several cache accesses and decode cycles. This allows the fetch predictor to go ahead of the front-end, and create a large fetch window in the FTQ that can be used to exploit parallelism. For example, a large L1 instruction cache may be efficiently pipelined and multi-banked to provide a very high bandwidth. This organization makes the fetch predictor become the critical element of the front-end.

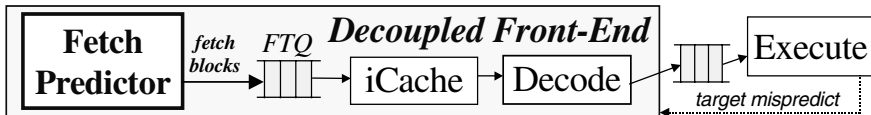


Fig. 1. The elements of a decoupled front-end and its interface with the execution core

¹ This work was supported by the MCyT-Spain under contract TIC 2001-2592 and partially supported by the *Generalitat de Catalunya – Grup Recerca Consolidat 2001* SGR-00218

This paper presents and analyzes the *indexed Fetch Target Buffer* (iFTB), a very fast and scalable fetch predictor based on the Fetch Target Buffer (FTB), [13]. It uses microarchitectural techniques, such as way prediction and overridden prediction, to aggressively reduce its cycle time at the expense of a small increase in fetch hazards.

Our work addresses fetch prediction and assumes it is the only performance bottleneck, therefore the FTQ never gets full. In this situation, simulation results indicate that the iFTB design, compared to the original FTB design, increases fetch bandwidth by between 15% and 45%. These results were obtained using CACTI to estimate cycle times and hazard penalties, [14], and using a cycle-by-cycle simulator to obtain the rate of hazards for the SPECint00 benchmark suit.

Section 2 analyzes the performance issues of the fetch predictor, reviews previous and related work and discusses the paper contribution. The iFTB design is described in full detail in section 3. Section 4 shows the experimental methodology and the main results. Section 5 outlines the conclusions and sets out future lines of research.

2 Increasing the Fetch Predictor's Performance

Equation 1 models the performance of the fetch predictor. Fetch bandwidth (BW) is augmented by increasing the number of instructions per fetch block (i_{fb}), by reducing the fetch predictor's cycle time (t_c), or by reducing the average number of cycles per fetch block (c_{fb}) (which measures the impact of hazards and the parallelism inside the fetch predictor).

$$BW = i_{fb} / (t_c \cdot c_{fb}) . \quad (1)$$

Previous Work. The branch target buffer (BTB) predicts single branches and was initially designed for single-issue processors, [8],[12]. Yeh and Patt's proposal, [17], increases fetch bandwidth by enlarging fetch blocks (i_{fb}) so as to always include a branch instruction at the end. Reinmann et al., [13], proposed the decoupled front-end architecture and the Fetch Target Buffer (FTB). The FTB is a two-level fetch predictor that achieves high performance by reducing t_c and further increasing i_{fb} . A small and fast L1-FTB reduces t_c , while the penalty of L1-FTB misses is kept down (and c_{fb} is maintained low) by using a large L2-FTB. They also proposed embedding never-taken branches into a fetch block to further increase i_{fb} .

Target mispredictions are detected in the execution stage and involve large penalties. Two-level branch predictors, [17], and hybrid branch predictors, [9], increase prediction accuracy by exploiting the growing chip area available for storing branch history. However, as noticed by Jiménez et al., [5], prediction delay may become a performance concern. In fact, increasing i_{fb} and reducing t_c puts more pressure on the latency of branch predictors. An overridden organization may be a successful way to deal with this latency. Two examples of commercial processors that use such scheme are the Alpha 21264, [7], and the Intel Itanium, [16].

Predicting several branches per cycle increases the parallelism inside the fetch predictor and reduces the number of cycles per fetch block, c_{fb} . However, current proposals, like [18],[10], rely on larger data structures and complex selection logic, which increase t_c and may also increase the target misprediction ratio.

The starting point. Each FTB's entry contains the data of a fetch block and the data to predict the next fetch block. A fetch prediction requires two steps, which determine the FTB design's cycle time (see Fig. 2.a). First, the current fetch block's address ($addr_i$) indexes the FTB to obtain the fetch block's size, target address, branch type, and branch local history. Second, the full fall-through address is assembled and the branch prediction outcome is used to select the next fetch block's address ($addr_{i+1}$).

As feature sizes shrink due to technology enhancements, the number of available transistors on a processor chip increases, but many trends indicate that circuit delays are becoming increasingly limited by the amount of wire (and memory) in the critical path, [1]. An FTB hierarchy solves this tradeoff, but we are trying to further reduce t_c .

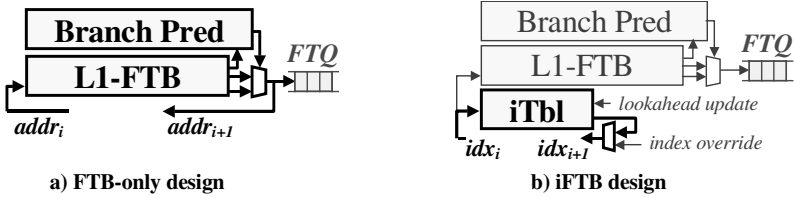


Fig. 2. Block diagram of the FTB-only and iFTB designs. Only the hit case is depicted

Our proposal. We have modified the FTB design by including a small table to provide an extra level of indirection (Fig. 2.b). The index table (iTbl) has the same number of entries as the FTB, but each entry contains a single index that points to the iTbl/FTB entry containing the next fetch block. Fetch prediction is performed at the iTbl speed (as a sequence of indexes $idx_i, idx_{i+1}, \dots$) while the FTB (which is pipelined to accommodate its longer delay) is addressed with these indexes to generate the actual fetch block addresses and sizes. The design has the following issues:

1. iTbl indexes provide a form of way prediction, hiding the delay of a set-associative access scheme whilst maintaining its low miss ratio
2. the separate iTbl allows the L1-FTB access to be taken out of the critical path
3. the single index stored into an iTbl entry provides a form of *lookahead* branch prediction. This scheme aggressively reduces the iFTB's cycle time, since the conditional branch predictor and the selection logic are taken out of the critical path
4. the branch predictor updates the iTbl's lookahead prediction on the fly, and overrides the iTbl prediction to achieve high accuracy without affecting t_c

Contribution and Related Work. Way prediction applied to the iCache was studied in [3],[7],[19]. Our design is unique in using way prediction in a decoupled front-end, and storing way predictors in a separate table. This paper extends our proposal made in [11] with a more aggressive approach and the modeling of a 2-level hierarchy.

We have adapted the hybrid local/global predictor of the Alpha 21264 to suit the very short cycle time requirements of our design. Like in [13], we use the lookahead proposal of [17] to hide the delay of the two-level local predictor. We also use an overridden scheme ([5],[7],[16]) to hide the latency of the hybrid selection. Finally, since the iFTB has a very short cycle time, access to the global and meta prediction tables may be critical. We then propose a solution to pipeline these tables using a variant of the proposal described in [18].

Our design optimizes the single branch prediction case. It could be extended to predict several branches per cycle, like in [18,10]. The use of a trace cache, [15], is also an orthogonal approach to ours. A trace cache simplifies the fetching task, reducing the penalty of FTB misses and target mispredictions. A decoupled front-end can take full advantage of a trace cache, since its larger access time may be pipelined.

Finally, a collaboration of software and hardware allows hardware complexity to be reduced and cycle time to be decreased. Our hardware-based solution can be helped by the compiler, but it does not rely on it to provide good performance.

3 The indexed Fetch Target Buffer (iFTB)

The index table provides an extra level of indirection to the FTB. It holds a small representation of the graph of the predicted transitions between fetch blocks. The graph is traversed at the iTbl speed, while a pipelined FTB generates the actual fetch block addresses and sizes and inserts them into the FTQ. Parallel with this, a complex (slow) branch predictor dynamically modifies the graph (iTbl's contents) to increase prediction accuracy. Sometimes but not often, the iTbl's outcome is wrong (fetch hazard), and the iFTB must be stopped and rolled-back, incurring a delay penalty (increasing c_{fb}).

The iFTB predicts the fetch trace as a sequence of indexes. As shown in Fig 3.a, several fetch hazards may force the index sequence to be rolled-back:

- the required fetch block is placed in a different L1-FTB way (*way miss*)
- the fetch block is not found in the L1-FTB or L2-FTB (*L1/L2 miss*)
- a new fetch block is identified at the decode stage (*miss-fetch*)
- a *target misprediction* is detected at the execution stage
- the lookahead prediction is overridden by the global predictor (*global-use*)

3.1 Way Prediction. Way and L1 misses are detected when comparing the address predicted by the L1-FTB with the address tag read from the L1-FTB at the next access. To reduce the occurrence and penalty of way misses, the L1-FTB stores the way of the fall-through and the target fetch blocks. These fields are updated after each way miss or L1 miss. After a fetch hazard, the appropriate way predictor is used. On a miss-fetch, the iTbl operation is restarted using a random way prediction. On a replacement, way predictors are copied into the L2-FTB to reduce way miss hazards.

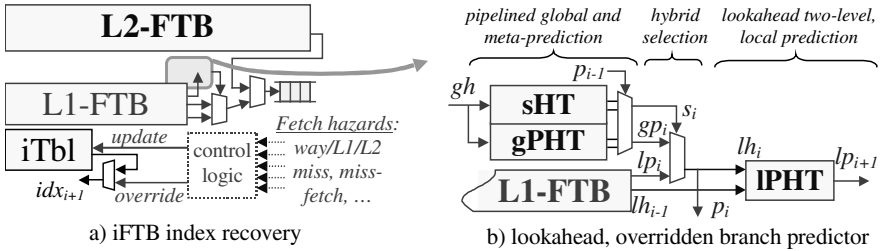


Fig. 3. iFTB design with a L2-FTB and a detailed block diagram of the branch predictor

3.2 L2-FTB. To reduce the penalty of L1 misses, the access to the L2-FTB is initiated as soon as a fetch block address is available, before concluding the L1 tag comparison. If, finally, there is no L1 miss, the L2 access is cancelled. Otherwise, the data from the L2-FTB is copied into the L1-FTB and the replaced data from the L1-FTB is copied into the L2-FTB. In both cases, a LRU replacement policy is used.

A L2 miss stalls the iFTB. Then, when the FTQ is empty, the iCache provides consecutive instructions to the decode stage until a miss-fetch is detected and a new fetch block is inserted into the L1-FTB and L2-FTB. As in [13], not-taken branches are ignored when identifying a new fetch block.

3.3 Conditional Branch Predictor. Since the penalty of branch mispredictions is extremely high, we have integrated into our design the accurate branch predictor of the Alpha 21264, [7]. Its hybrid algorithm uses a meta-predictor that chooses between a local and a global prediction. The local history stored in the FTB (10 previous results of the fetch block's ending branch) is used to index a local Pattern History Table (IPHT: 3-bit counter) and predict the branch's next occurrence. The global Pattern History Table and selection History Table (gPHT, sHT: 2-bit counters) are indexed using the previous global history (the outcome of the last 12 branches).

The high prediction accuracy of this algorithm, though, comes at the expense of an increase in prediction latency. We propose three modifications to the original algorithm to make it suit the iFTB's short cycle time:

- a lookahead scheme avoids the delay of the two-level local predictor
- the long latency to access the gPHT and sHT tables is hidden by pipelining
- an overridden scheme hides the delay of selection between local and global prediction in most of the cases

Lookahead Local Predictor. Figure 3.b. illustrates the lookahead scheme suggested in [17] to hide the delay of the IPHT. As soon as the final prediction, p_i , is generated (before it is validated), it is merged with the local history of a fetch block, lh_{i-1} , to obtain the new local history, lh_i , and indexes the IPHT to get the local prediction for the branch's next occurrence, lp_{i+1} . If the lookahead prediction changes ($lp_{i+1} \neq lp_i$) the iTbl is updated on-the-fly to reflect that change. This modification of [17] allows predicting at the iTbl's speed with the accuracy of a two-level local predictor. The prediction must be recalculated after a branch misprediction.

Pipelined gPHT/sHT. To achieve the required prediction rate, the gPHT/sHT must be pipelined. Our solution is based on the idea proposed in [18] for multiple branch prediction. The gPHT/sHT access starts before the full global history is available, using an incomplete index, so that all the adjacent entries corresponding to the missing bits are read together. Then, at the output of the gPHT/sHT, the remaining history bits (already available) are used to select the final outcome. Figure 3.b illustrates this technique assuming that there is only one missing bit (p_{i-1}).

Overridden Hybrid Predictor. The high prediction accuracy of a hybrid method relies on its ability to dynamically select the best predictor in each particular case. The iFTB design uses the two-level (lookahead) local predictor by default. Later, the hybrid prediction is calculated and overrides the previous prediction if the global

prediction differs and the meta-predictor chooses the global predictor. This hazard, that we call a *global-use hazard*, provokes an iFTB squash and a delay penalty.

3.4 Indirect Jump Prediction. Fetch blocks ending in a jump are also managed by the iTbl. Unconditional jumps are trivially predicted. Indirect jumps use the FTB’s lookahead bit as a hysteresis counter to avoid updating the iTbl until two consecutive mispredictions occur. The FTB stores the way predictors of the previous two targets.

A 32-entry Return Address Stack (RAS), [6], is used to improve the prediction of return instructions. If the FTB’s type field identifies a return, an address popped from the RAS is compared to the fetch block’s address predicted by the iTbl. If they differ, the RAS prediction overrides the iTbl resulting in a delay penalty (*ras-use hazard*).

4 Performance Evaluation

We evaluate the performance advantage of our proposal by comparing the iFTB with the FTB-only design in Fig. 2 (which is very similar to the proposal in [13]). We have used *SimpleScalar*, [2], to build a simulator modeling the iFTB components, and have used the SPECint00 benchmark suit compiled for the Alpha ISA (cc DEC 5.9, -O4).

4.1 Preliminary Analysis of Hazards. Table 1 lists some data for the SPECint00 benchmark suit, used in our experiments. For each benchmark, several billion instructions are skipped (column 3) before simulation measures are taken for an execution window of 400 million instructions.

Column 4 indicates the number of L1-FTB entries required to reduce the L1 miss rate to less than 1%. Columns 5 and 6 list the average fetch block size when never-taken branches are not embedded and when they are. The increase in the fetch block size due to embedding is 9.4% on average. Columns 7 and 8 list the misprediction rate per retired instruction due to conditional branches and indirect jumps. A 32-entry RAS reduces return mispredictions to less than 0.003%. This data provides an insight into the instruction working set and the predictability of branches for each benchmark.

Table 1. Simulation data and results (unbounded L1-FTB). Branch and jump misprediction rates are per retired instruction, while other rates are per retired fetch block

1. Bench	2. input	3. skip	4. L1miss<1%	5. i_b	6. i_b^{emb}	7. cond brnch	8. indir jmp	9. glb-use	10. ras-use
bzp	graphic	4.92	179	8.90	9.29	0.46%	0.00%	1.55%	2.23%
crafty	reference	0.77	883	8.97	9.40	0.46%	0.11%	3.27%	3.67%
eon	cook	0.16	331	8.81	10.62	0.05%	0.14%	0.88%	6.73%
gap	reference	0.17	415	15.62	17.19	0.14%	0.04%	2.40%	4.61%
gcc	166.i	2.50	1618	17.34	18.68	0.20%	0.06%	2.64%	2.77%
gzip	random	7.50	94	7.58	7.83	1.00%	0.00%	3.06%	0.00%
mcf	reference	6.00	61	4.34	4.51	2.89%	0.00%	9.78%	0.39%
parser	reference	4.92	1057	6.51	6.81	0.66%	0.00%	3.41%	3.12%
perlbmk	splitmail	0.60	556	7.97	8.66	0.13%	1.00%	0.44%	1.09%
twolf	reference	1.10	285	8.17	9.34	1.07%	0.00%	5.08%	2.08%
vortex	vortex1	4.85	1437	5.95	8.09	0.05%	0.01%	0.21%	5.58%
vpr	reference	1.50	38	9.56	10.26	0.53%	0.00%	1.15%	0.03%

Hazards specific to the iFTB. Columns 9 and 10 show the rate per retired fetch block of global-use and ras-use hazards (the average is 3.24% and 2.55%). The small penalty of global-use and ras-use hazards compared to branch misprediction hazards and the reduction of the iFTB path's complexity justifies the overridden scheme.

We have simulated the iFTB with a varying number of iFTB/L1-FTB entries (n) and associativity (a) to quantify how hazard rates are affected. Results (averaged for all benchmarks) demonstrate the great benefit of using a large L2-FTB to save branch local history and way predictors across the misses occurring in a small L1-FTB.

Figure 4.a shows how hazards increase significantly when n is reduced in a single-level FTB scheme. A hierarchy provides the low hazard rate of the large L2-FTB with the small cycle time of the L1-FTB. The only cost is the penalty of L1 misses.

Figure 4.b shows that the way miss rate is fairly low, even for a small n and a large a . Using a L2-FTB further reduces this rate to less than half.

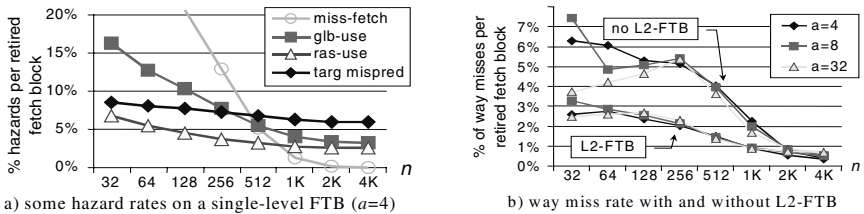


Fig. 4. Some hazard rates for a varying number of L1-FTB entries (n) and associativity (a)

4.2 Timing Model. The access time of the iFTB and the FTB determine, respectively, the cycle time of the iFTB design and the FTB-only design. In both cases, we ignore other delays in the critical design path. Our assumption benefits the FTB-only design, which includes complex logic in the critical path to combine the outputs of the FTB, branch predictor, and RAS. Conversely, the iFTB design includes a simple 2-way multiplexer.

CACTI, [14], implements a timing model of cache memories for a process technology of 0,80 μm , and scales the model to 0,10 μm , either assuming ideal wire delay scaling or taking into account the unfavorable effects due to parasitic capacitance. Using CACTI, we have gathered timing data for a process technology of 0,10 μm , using both scaling models, for memory tables with a varying number of entries, entry sizes, associative degrees, tag sizes, and read/write ports.

Combining timing data and hazard rates, we have found that using 4-way set-associativity for the FTB-only design is the best option for almost all the benchmarks (a direct-mapped FTB has a faster cycle time but very high miss rates). In fact, we have found that using a 4-bit tag, instead of a full tag (more than 20 bits) provides a good tradeoff between the slight increase in FTB conflict misses and the reduction of the FTB access time. This strategy was also analyzed in [4]. Then, we will assume a 4-way set-associativity topology for the FTB-only design.

Table 2 shows the access time for both wire-scaling models of an n -entry iFTB (entry size is $\log_2 n$), and an n -entry FTB (tag size is 4 bits, entry size is 48 bits). To allow reading and updating of the iFTB (FTB) concurrently, one read and one write ports are used. The overhead when using a single R/W port is lower than 1%.

Table 2. Access time of n -entry iTbl (direct-mapped, entry size is $\log_2 n$), and n -entry FTB (4-way set-associative, tag size is 4 bits, entry size is 48 bits) estimated using CACTI

n	32	64	128	256	512	1K	2K	4K	8K
FTB	0.43	0.44	0.44	0.46	0.47	0.49	0.55	0.60	0.67
iTbl	0.28	0.29	0.30	0.32	0.34	0.37	0.41	0.47	0.53

a) ideal wire-scaling

n	32	64	128	256	512	1K	2K	4K	8K
FTB	0.40	0.42	0.44	0.48	0.51	0.57	0.69	0.85	1.11
iTbl	0.26	0.28	0.30	0.33	0.37	0.41	0.45	0.55	0.70

b) non-ideal wire-scaling

Time penalties common to both designs. We estimate the penalty of FTB misses to be the access time of an 8K-entry, 4-way set-associative L2-FTB. The fixed delay (0.67 or 1.11 ns, depending on the scaling model) must be converted to cycles. The penalty is decreased by one cycle if the access to the L2-FTB is initiated as soon as the fetch address is available, before finishing the FTB tag comparison.

Instead of forcing time penalties to be perfect multiples of the cycle time, we assume a finer clock and admit non-integer cycle penalties to recover from hazards.

The miss-fetch penalty is determined by the delay of the FTQ, the iCache and the decode stage. Since our analysis focuses on fetch prediction, we ignore hazards that may increase the miss-fetch penalty, like iCache misses or resource conflicts. We use delays of 1.24 and 1.65 ns, assuming a 32Kbyte, 4-way set-associative iCache.

Time Penalties specific to the iFTB Design. Detecting global-use or ras-use hazards requires reading the branch type and lookahead prediction into the FTB and performing a comparison to test the hazard condition. Since the FTB data may be used without waiting for the tag check, the penalty is the difference between the access times of the iTbl and the FTB data array. Then the penalty is between 0.25 and 0.5 cycles, depending on n .

The penalty of way misses includes the time needed to perform the associative tag check and to select the FTB data from the right entry, and depends on the FTB set-associativity (a). We have found that, for large values of a , the benefit of reducing L1 misses and way misses surpasses the drawback of increasing the way miss penalty. Therefore, we have assumed a 32-way set-associativity in the FTB array for the iFTB design, and a full cycle penalty for way misses.

4.3 Fetch Bandwidth Results. The penalty of fetch hazards that occur between taken branches may be overlapped. This situation occurs because the iCache fetches sequential instructions whenever the FTQ becomes empty. Our simulation model identifies and takes into account these cases. The simulator does not model the execution core. Instead, the delay between branch decode and branch resolution is provided as a simulation parameter. Using the hazard penalties as input, the simulator generates the total number of fetch cycles.

Figures 5.a and 5.b show fetch bandwidth on the FTB-only and iFTB designs, using an 8K-entry L2-FTB and for varying values of n . Results are taken assuming an optimistic 2-cycle branch resolution delay. On both designs and for both wire-scaling models, the highest performance is obtained for $n=512$, with a 23% increase in the iFTB with respect to the FTB-only. When varying n the performance advantage remains at between 16% and 26%. The non-ideal wire-scaling model reduces fetch bandwidth in both designs ($\approx 16\%$), but affects the FTB-only scheme slightly more.

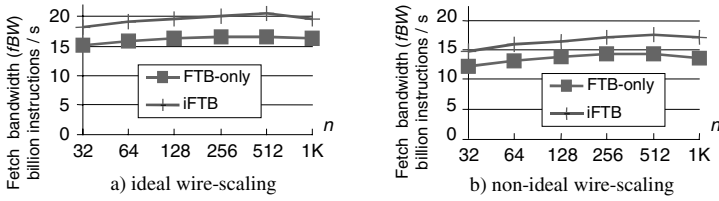


Fig. 5. Fetch bandwidth for the FTB-only and iFTB designs (averaged for all benchmarks)

We have analyzed the effect of increasing the penalty of miss-fetches and target mispredictions. In all cases, the fetch bandwidth and the relative performance advantage of the iFTB design is reduced. However, even assuming a misprediction penalty of 100 cycles, the average performance gain remains at between 4% and 6%.

Figure 6 depicts fetch bandwidth for three representative benchmarks. *Bzip* represents the set of benchmarks with a small instruction working set. They benefit extraordinarily from the use of a very small iTbl ($n \leq 64$), with a performance advantage of between 35 and 45%. Conversely, *Crafty* represents the set of benchmarks with large instruction working sets. The large number of entries required to provide optimal performance ($n \geq 512$) does not exploit the full potential of the iFTB design (20-30% enhancement). Remarkably, *Parser* is the only benchmark in the suit that matches the average fetch bandwidth curve.

These last results indicate that the iFTB scheme is more effective when the working set of the running program fits into a small iTbl, and that a higher performance may be obtained if a design is able to adapt the L1-FTB/iTBl size to that working set.

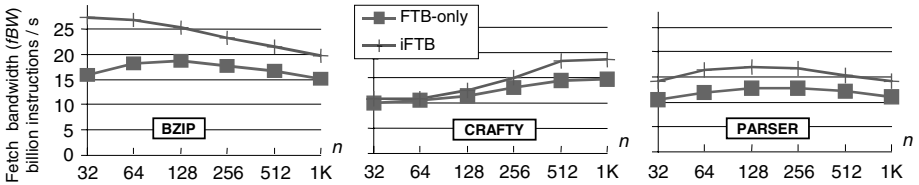


Fig. 6. Fetch bandwidth for three representative benchmarks using non-ideal wire-scaling

5 Conclusions and Future Work

We have described and analyzed the indexed Fetch Target Buffer. With a small hardware cost with respect to the FTB-only design (the index table and the extra storage for way prediction fields), we estimate the increase in the fetch bandwidth to be between 15% and 45%, depending on the benchmark. The final performance advantage depends on the ability of the execution core to extract ILP at the rate of the front-end.

The aggressive cycle time reduction achieved in the iFTB scheme is at the expense of creating new fetch hazards (way misses, lookahead, global-use, ras-use). We have shown that they are infrequent and have small penalties. Hence, they do not reduce the iFTB's cycle time advantage significantly. One key issue of the iFTB design is that the data that is used less frequently is placed out of the critical path. The other

key issue is the use of lookahead mechanisms to anticipate changes in the program's control flow without halting the iFTB operation.

Like in the FTB-only design, the harder fetch hazards are target mispredictions and L1 misses. When a program has a small instruction working set and has predictable branches, the iFTB scheme is very effective and fetch bandwidth very high. A L2-FTB, by preserving local branch history and way predictors for a large set of fetch blocks, tolerates larger working sets.

We are currently analyzing the use of fetch block prefetching to further enable the effective use of a small index table for a wider range of applications. In addition, we plan to simulate the effect of speculation and evaluate the effectiveness of techniques to avoid pollution due to early table updates, such as the speculative history queue.

References

1. V.Agarwal, M.S.Hrishikesh, S.W.Keckler, D.Burger: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. *Proc. ISCA-27*, (2000) 248–259
2. D.Burger, T.M.Austin: The SimpleScalar Tool Set. *Univ. Wisconsin-Madison Computer Science Department*, Technical Report TR-1342, 1997
3. B.Calder, D.Grunwald: Next Cache Line and Set Prediction. *ISCA-22*, (1995) 287–296
4. J.Hoogerbrugge: Cost-Efficient Branch Target Buffers. *Euro-Par 2000*, LNCS 1900, (2000) 950–959
5. D.A. Jimenez, S.W. Keckler, C. Lin: The Impact of Delay on the Design of Branch Predictors. *Proc. MICRO-33*, (2000) 67–76
6. D.R.Kaeli, P.G.Emma: Branch History Table Prediction of Moving Target Branches due Subroutine Returns. *Proc. ISCA-18* (1991) 34–41
7. R.Kessler: The Alpha 21264 microprocessor. *IEEE Micro*, (1999) 19(2):24–36
8. J.K.F.Lee, A.J.Smith: Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer* (1984) 17(2): 6–22
9. S.McFarling: Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June (1993)
10. P.Michaud, A.Seznec, S.Jourdan: Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors, *Proc. PACT-99*, (1999) 121–128
11. J.C.Moure, D.I.Rexachs, E.Luque: Speeding Up Target Address Generation Using a Self-Indexed FTB, *Euro-Par 2002*, LNCS 2400, (2002) 517–521
12. C.H.Perleberg, A.J.Smith: Branch Target Buffer Design and Optimization. *IEEE Trans. on Computers* (1993) 42(4): 396–412
13. G.Reinman, T.Austin, B.Calder: A Scalable Front-End Architecture for Fast Instruction Delivery. *Proc. ISCA-26*, (1999) 234–245
14. G.Reinman, N.Jouppi: An Integrated Cache Timing and Power Model. *COMPAQ Western Research Lab*, <http://www.research.digital.com/wrl/people/jouppi/CACTI.html> (1999)
15. E.Rotenberg, S.Bennett, J.Smith: Trace cache: a low latency approach to high bandwidth instruction fetching. *Proc. 29th Symp. Microarchitecture*, MICRO-29 (1996) 24–34
16. H.Sharangpani, K.Arora: Itanium Processor Microarchitecture. *IEEE Micro* (2000)7:24–43
17. T.-Y.Yeh, Y.N.Patt: A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *Proc. MICRO-25*, (1992) 129–139
18. T.-Y. Yeh, D.T.Marr, Y.N.Patt: Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. *Proc. 7th Int. Conf. Supercomputing* (1993) 67–76
19. R. Yung: Design Decisions Influencing the Ultrasparc's Instruction Fetch Architecture. *Proc. MICRO-29*, (1996) 178–190

Clustered Microarchitecture Simultaneous Multithreading

Seong-Won Lee¹ and Jean-Luc Gaudiot²

¹ Dept. of Electrical Engineering, University of Southern California
Los Angeles, California, USA
`seongwon@usc.edu`

² Dept. of Electrical Engineering and Computer Science,
University of California at Irvine
Irvine, California, USA
`gaudiot@uci.edu`

Abstract. The excessive amount of heat that billions of transistors will produce will be the most important challenge to the design of the future chips. In order to reduce the power consumption of microprocessors, many low power architectures have been developed. However, this has come at the expense of performance, and only few low power architectures for superscalar, such as clustered architectures, target high performance computing applications.

In this paper, we propose a new clustered SMT architecture which is appropriate for both multiple threads and single thread environments. We analyzed the proposed design is analyzed and compared it to conventional SMT architectures. We show that our approach significantly reduces power consumption without significantly degrading performance.

1 Introduction

Ever since their introduction, the complexity of microprocessors has steadily increased. Furthermore, shrinking fabrication technologies show us that next generation microprocessors will hold billions of transistors on a single die. This means that the huge amount of resources will require more efficient resource utilization than superscalar architectures are capable of. Simultaneous MultiThreading (SMT) [15], with its well known efficient utilization of hardware resources should be a good candidate to be the best architecture of future microprocessors.

On the other hand, the power consumption of microprocessor has already become a key design issue in workstation and server markets [11]. In the design of many commercial microprocessors destined for high performance computing environments, great efforts to reduce power consumption have been expended [6, 7]. Thus, the biggest challenge in designing billion transistor microprocessors will be the excessive amount of heat that they will inevitably produce. However, previous low power architectures concentrated on mobile and embedded applications [3,8] which do not demand high performance.

It has been found that one of the most significant factors to affect power consumption in superscalar architectures is the instruction issue width [16]. This is also true in SMT architectures because their implementation is a relatively simple extension of superscalar architectures [14]. In order to reduce the effect of wide instruction issue, microarchitecture clustering techniques have been used in superscalar architectures [9,16]: the instruction issue width is simply divided into several small clusters. A cluster consists of units associated with instruction issue such as instruction queue, register file, functional units and result bus. However, conventional methods to effect the partition do not fully utilize the potential of thread level parallelism available in SMT.

In this paper, we propose a new SMT architecture based on microarchitecture clustering techniques, so as to reduce power consumption without sacrificing performance. We will do this by partitioning the resources to handle instructions according to threads information. We also propose a new technique which makes our proposed SMT architecture reduce power not only with multiple threads but also with single threads.

The following section 2 describes previous research on clustering techniques. Our proposed architecture and evaluation models are introduced in 3. The simulation environment is described in section 4. Section 5 includes our simulation results and analysis. Finally, we summarize our results in section 7.

2 Background and Related Work

Increasing the issue bandwidth increases not only the complexity of the instruction queue (which is also known as issue window) but also the number of read/write ports in register files, the number of functional units, and the result bus width. By clustering the microarchitecture, the power consumption dramatically decreases and the performance can be sustained if instructions are carefully scheduled so as to minimize the dependencies between clusters (*i.e.* intercluster dependency).

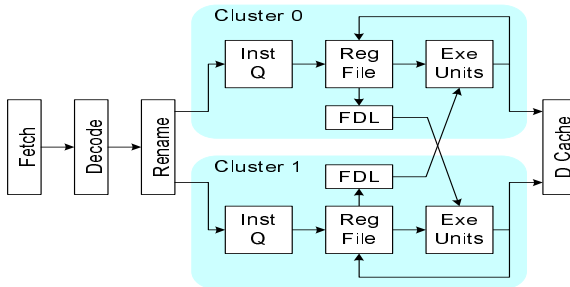


Fig. 1. Generalized Multiple Cluster Superscalar Architectures

Figure 1 represents a generalized two-cluster superscalar architecture. Each cluster has its own issue window, register file, functional units and result bus. Since superscalar architectures execute only a single thread at a time, there are data and control dependencies between instructions in the two instruction queues. Forwarding logic (FDL in Figure 1) resolves those dependencies by transferring data from one cluster to the other. Clustering the microarchitecture based on function types of instructions such as integer instructions and floating-point instructions is one possibility [9].

In order to avoid underutilizing the clusters due to the ratio of instruction types (the ratio of the number of integer over the number of floating-point instructions, in this case), some microarchitecture clustering techniques evenly partition the hardware resources of a microprocessor [4,16]. However, fair instruction distribution and resolution of intercluster dependencies become important issues in those clustering techniques. These two problems not only increase the complexity of the dispatch unit and of the forwarding logic but also demand intensive compiler support [4].

3 Clustered Simultaneous Multithreading Architectures

In order to apply the clustering techniques to SMT architectures, we propose and evaluate two approaches for implementing a clustered microarchitecture SMT. One approach is *clustering by function*. This is a conventional clustering technique for superscalar architectures [9]. There are one integer cluster and one floating-point cluster in our SMT model for this conventional clustering technique. The other approach which we propose is *clustering by threads*. Each cluster in this approach has its own integer functional units and floating point functional units. As shown in Figure 1, other units are shared by all threads. The dispatch unit distributes instructions from multiple threads to multiple clusters according to their threads. We use the term *functional cluster SMT* to describe an architecture which has separate integer and floating-point instruction queues, and the term *unified cluster SMT* to describe an architecture which has multiple unified instruction queues.

In a functional cluster SMT, different instruction issue widths of two clusters are used as in superscalar to handle instruction mixes of most applications. However, units in one cluster which are not utilized due to the lack of parallelism are idle even though operand-ready instructions in the other cluster are not issued due to the limited instruction issue width. Thread level parallelism compensates for this underutilization in a *functional cluster SMT* by issuing instructions from multiple threads. On the other hand, in a unified cluster SMT, the instruction queue can find more independent instructions with a single thread, but there are fewer threads in a single cluster. Thus, the cluster in a *unified cluster SMT* can be also underutilized. However, it should be noted that in a unified cluster SMT, instructions in one cluster do not prevent the other cluster from issuing due to intercluster dependency as instructions in a functional cluster SMT do.

It should be further noted that forwarding logic between clusters is not required in a unified cluster SMT.

4 Simulation Environment

We used our ALPSS [10] simulator to estimate the power consumption of several clustered SMT models. The ALPSS (Architectural Level Power Simulator for Simultaneous MultiThreading) is a power simulator based on SimpleScalar [1]. The power estimation models of ALPSS are based on the parameterized power estimation models of Wattch [2].

Most units in a superscalar processor of SimpleScalar were slightly modified and extended to accommodate the multiple threads of an SMT processor. Modifications from the base SimpleScalar architecture also include separation in integer and floating point queues, and addition of a reorder buffer. Other subtle features are based on the SMT architecture published by Tullsen *et al.* [14] and are described in the technical report on ALPSS [10].

We used the same parameterized power estimation models of Wattch with the extended number of partitions in ALPSS. One power estimation model in Wattch, which has only one activity counter, consists of several small models of submodules. We partitioned the big module into several small modules by giving individual activity counters to those submodules. We also modified some input parameters (such as the number of threads) of modules to extend to SMT architectures.

The configuration for a single cluster SMT architecture which has all hardware resources that any clustered microarchitecture SMTs has in a single big cluster. The performance and power consumption of the single cluster SMT are used as the reference values of all simulation results in this paper.

The detailed hardware configuration is roughly equivalent to those in a commercially designed SMT processor, the Alpha EV8 [12] (even though it has been discontinued, we select it for its SMT-like features). The issue bandwidth and the number of functional units were slightly increased in order to be equally divided for multiple clusters in our evaluation models. Architectural differences between our models and the Alpha processor could account for slight differences in the architectural parameters. All functional units are assumed to be fully pipelined. In the case of the configuration of the power model, we use the same configuration as Wattch [2].

We used the SPEC 2000 CPU benchmark suite to evaluate our clustered microarchitecture SMT models. The benchmarks were executed with the *reference* input sets. The benchmarks are Alpha EV6 binaries and compiled on a Digital Unix V4.0F. The results from the first 300 million instructions executed of each thread were discarded in order to avoid overlapping initialization routines and so as to wait until caches populations are stabilized. We then gathered results from the next 300 million instructions times the number of threads, which means 1.2 billion instructions in a simulation of four threads.

A combination of several single thread versions of the programs was used to emulate the simulation environment of multiple threads because each thread in an SMT processor is independent. In order to represent various application environments and to keep the simulation times within reasonable bounds, we chose 5 integer and 5 floating-point benchmark programs from the SPEC 2000 CPU benchmark suite based on their contents and behaviors. Out of the 10 single thread programs, 8 subsets of 2 threads and 8 subsets of 4 threads are ordered by increasing IPC and number of floating-point instructions. 4 subsets based on IPC consist of groups of programs with the highest IPC, the lowest IPC, medium IPC, and mix of the highest IPC and the lowest IPC in order to account for different program behaviors between different combinations of programs. With the order of the number of floating-point instructions instead of IPC, another 4 subsets of four thread benchmarks are selected. The eight combinations of four threads are hii (facerec, gap, apsi, gzip), loi (crafty, gcc, art, vpr), hli (facerec, gap, art, vpr), mdi (gzip, mgrid, applu, crafty), hif (applu, mgrid, art, facerec), lof (gap, crafty, gcc, gzip), hlf (applu, mgrid, gcc, gzip), and mdf (facerec, apsi, vpr, gap).

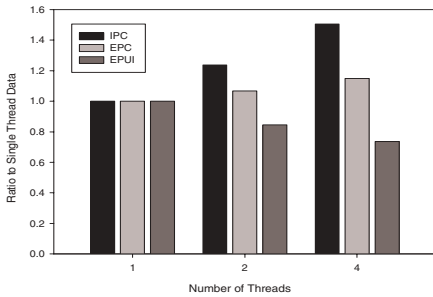


Fig. 2. Number of threads in Single Cluster SMT

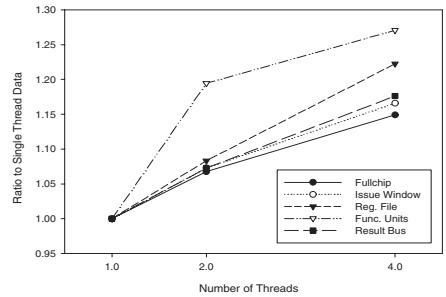


Fig. 3. Power Consumption of Each Modules

5 Performance and Power Evaluation

Figure 2 shows the effect of the number of running threads on a single cluster SMT architecture. IPC corresponds to the number of instructions per cycle, EPC is the energy per cycle (or average power consumption), EPUI is the energy per instruction completed [13] which corresponds to the efficiency of the architecture in terms of power consumption. As seen in the figure, even though EPUI decreases as the number of threads increases (that is, the more threads, the more efficient SMT becomes), the EPC increases with the number of threads.

In the Figure 3, the power consumption of modules associated with instruction issue width is shown. *fullchip* refers to the total power consumption of the whole microprocessor. All results are gathered from the simulations of a single

cluster SMT and are normalized with regard to the power consumption with a single thread. The power consumption of each module in the figure behaves differently as the number of threads increases. However, the ratio of power increase as the number of threads increases is higher than that of the whole processor. That is, the high power consumption due to the large instruction issue width is more serious in SMT processors than in superscalar processors.

We also evaluated and compared a functional cluster SMT which is similar to the conventional SMT architecture [14] to our proposed unified cluster SMT. The functional cluster SMT model which has one integer instruction queue and one floating-point instruction queue is named “separate integer and floating-point instruction queue SMT” (SIQ-SMT). On the other hand, the proposed unified cluster SMT model which has two identical unified instruction queues is named “dual unified instruction queue SMT” (DUIQ-SMT). In our evaluation models, the integer cluster of SIQ-SMT has an issue width and all integer functional units of the baseline architecture, and the floating point cluster has a 4 issue width and all floating-point functional units. The one cluster of two unified clusters in DUIQ-SMT has a 6 issue width and exactly half of the hardware resources of the baseline architecture.

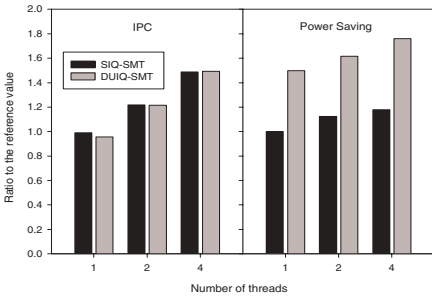


Fig. 4. Two cluster SMTs: Effect of Number of threads

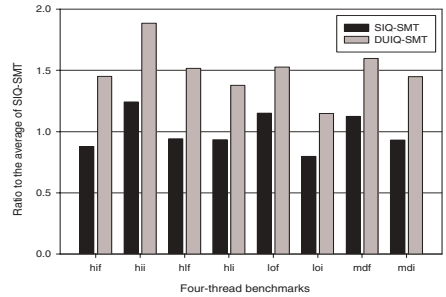


Fig. 5. Two cluster SMTs: Individual power savings

The performance and power consumption of two clustered microarchitecture SMTs are compared in Figure 4. The performance is normalized to the average IPC of single thread benchmarks on the single cluster SMT, and the power consumption saved by applying clustering techniques is shown on right side of the figure. The power saving is normalized with regard to the result of single thread benchmarks on SIQ-SMT. The performances of SIQ-SMT and DUIQ-SMT are very close to the performance of the reference (the single cluster SMT), except for the performance of DUIQ-SMT with single thread execution. The single thread performance of DUIQ-SMT drops by 3.6% compared to that of SIQ-SMT. The performance degrades because one cluster in DUIQ-SMT is idle if there is only one thread executing. Two unified clusters in DUIQ-SMT can save 47.8% more power than the power that the large-issue-width integer cluster and small-issue-

width floating-point cluster in SIQ-SMT can save. Indeed, the power saving of DUIQ-SMT is a maximum of 28.2% and an average of 11.2% over the single cluster SMT.

Individual power savings of SIQ-SMT and DUIQ-SMT with 4-thread execution are presented in Figure 5. The results show that the descending order of SIQ-SMT’s power saving is *lof*, *mdf*, *hlf*, and *hif*, and the order is the same as the ascending order of the number of floating-point instructions. Since the floating-point cluster in SIQ-SMT is isolated from the integer cluster, SIQ-SMT can effectively turn off the floating-point cluster if there are few floating-point instructions. On the other hand, DUIQ-SMT, which has unified clusters, does not show the same pattern. In fact, the results indicate that the difference of power saving between SIQ-SMT and DUIQ-SMT increases as there are more floating-point instructions.

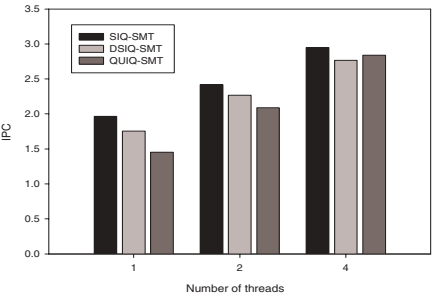


Fig. 6. 4-Cluster SMTs: Performance degradation

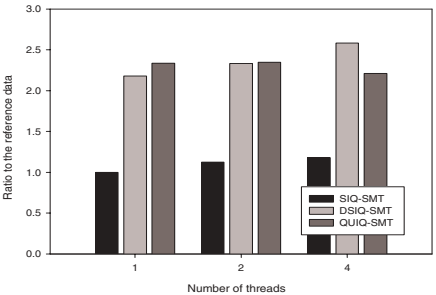


Fig. 7. 4-Cluster SMTs: Power savings

Clustered microarchitecture SMTs that have four clusters have also been investigated: a clustered microarchitecture SMT with two sets of identical integer and floating-point cluster pairs (dual integer and floating-point instruction queue SMT, denoted hereafter DSIQ-SMT) and a clustered microarchitecture SMT with four identical unified instruction queues (quadruple unified instruction queue SMT, denoted hereafter QUIQ-SMT).

Figure 6 shows the effect of multiple threads in a 4-cluster SMT architectures in terms of IPC. As seen in the Figure, in the case of DSIQ-SMT, the performance drop compared to SIQ-SMT is 9.2% on the average. The single thread performance of DSIQ-SMT is less than the multi-thread performance because one of the two integer and floating-point cluster pairs is idle. The average performance drop of QUIQ-SMT is 14.5%. However, single thread execution and two thread execution in QUIQ-SMT seriously suffers from the lack of thread-level parallelism. In the case of QUIQ-SMT, since the instruction-level parallelism of a single thread cannot utilize the hardware resources in the other clusters, the performance drop due to the small instruction issue width in a single cluster is unavoidable. As seen in Figure 7, even though the performance gain of both ar-

chitectures with up to 4 threads is less than that of SIQ-SMT, the power saving of these four cluster SMT architectures is larger than that of two cluster SMT architectures. DSIQ-SMT reduces power consumption by a factor of 114.9% over SIQ-SMT, and QUIQ-SMT saves 110.0% on the average.

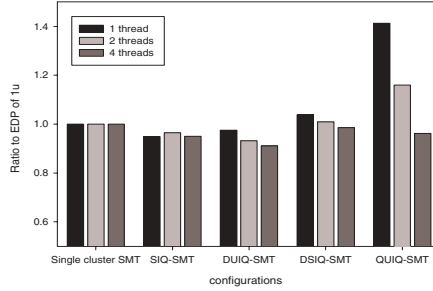


Fig. 8. Comparison of Four Clustered SMT Architectures

6 Comparison and Improvement of Clustered SMT Architectures

Power efficiency of the four models of clustered SMT architectures is compared in Figure 8. In order to consider the performance and power consumption at the same time, Energy-Delay Product (EDP) [5] is used in the comparison of the four architectures. A power efficient architecture has low energy (total power consumption) and low delay (the inverse of performance), which means a low EDP. With single thread execution, SIQ-SMT (separate integer and floating-point instruction queue SMT) has the best EDP among the four clustered SMT architectures because SIQ-SMT is optimized for single thread application environments. However, EDP does not improve much as the number of threads increases. Except for the single thread environment, DUIQ-SMT is the best clustered SMT architecture in terms of EDP. In a single thread environment, DUIQ-SMT also has a good EDP, which is close to the EDP of SIQ-SMT.

Even though DUIQ-SMT is a good solution to the high power consumption problem of SMT architectures, the performance with a single thread needs to improve. In order to simultaneously improve performance and to reduce power consumption, two unified instruction queues in DUIQ-SMT can be transformed into an architecture similar to SIQ-SMT. That is, one cluster handles the integer instructions and the other handles the floating-point instructions.

One cluster can borrow functional units in the other cluster by switching its unused functional units with the other cluster's unused functional units. Figure 9 illustrates how functional units are shared by two clusters. The remaining functional units which are not shared are also shared in the same manner. Other units such as instruction queues are omitted in order to simplify the diagram.

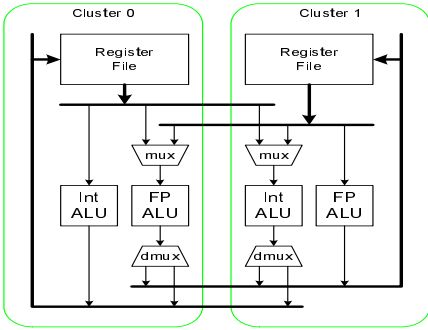


Fig. 9. Modified DUIQ-SMT Architecture

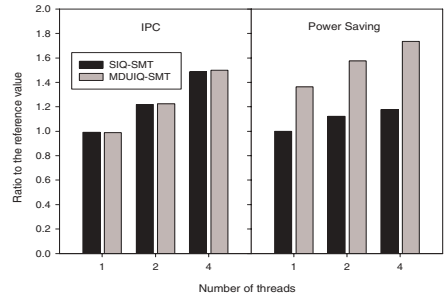


Fig. 10. Improvement of MUIQ-SMT Architecture

Figure 10 shows the results of the improvement which are the IPCs and power savings of DUIQ-SMT and modified DUIQ-SMT (abbreviated by MDUIQ-SMT) compared to the single cluster SMT. As seen in the figure, the performance of the modified DUIQ-SMT for the single thread environment clearly improves by a factor of 3.6%. However, as the number of threads increases, the improvement is marginal because the performance of DUIQ-SMT for multiple threads is already very close to that of the single cluster SMT. That is, the rate of hardware utilization in DUIQ-SMT is very high, so that one cluster cannot lend its functional units to the other cluster. The power saving of MDUIQ-SMT over SIQ-SMT slightly decreases down to 41.3% because the power consumption of the modified DUIQ-SMT increases due to the increased loads on result bus.

7 Conclusion

A multiple cluster superscalar is one type of low power microprocessor architecture which has multiple instruction queues whose instruction issue width is small. However, the same approach which is designed for single thread does not efficiently exploit thread-level parallelism in SMT architectures. In this paper, several types of clustered microarchitecture SMTs have been proposed and evaluated.

The wide issue of SMT architectures significantly increases power consumption. In order to reduce such a high power consumption while retaining the performance gain that SMT architectures provide, a symmetric, unified multi-cluster SMT has been proposed here. The functional unit sharing technique has also been proposed in order to improve the performance of single thread execution. The proposed dual unified cluster SMT architecture with shared functional units is not only more power efficient than a conventional single cluster SMT architecture, but it also outperforms the conventional functional cluster SMT in terms of power efficiency (more than 40% of additional power saving).

With appropriate compiler support, further research on the utilization of the shared functional units among multiple clusters can add significant improvements to the efficiency of our proposed architecture.

References

1. T. Austin. The SimpleScalar Architectural Research Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin-Madison, June 1997.
2. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.
3. D. Dobberpuhl. The Design of A High Performance Low Power Microprocessor. In *Proceedings of the 1996 International Symposium on Low-Power Electronics and Design*, pages 11–16, August 1996.
4. K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicenter Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, December 1997.
5. R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 21(9):1277–1284, September 1996.
6. M. Gowan, L. Biro, and D. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference*, pages 726–731, June 1998.
7. S. Gunther, F. Binns, D. Carmean, and J. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal*, February 2001.
8. R. Hakenes and Y. Manoli. A Novel Low-Power Microprocessor Architecture. In *Proceedings of the 2000 International Conference on Computer Design*, pages 141–146, September 2000.
9. R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.
10. S. Lee and J-L Gaudiot. ALPSS: Architectural Level Power Simulator for Simultaneous Multithreading, Version 1.0. Technical Report CENG-02-04, University of Southern California, April 2002.
11. T. Mudge. Power: A first class design constraint for future architectures. In *Proceedings of the 7th International Conference on High Performance Computing*, pages 215–224, December 2000.
12. R. Preston et al. Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading. In *Digest of Technical Papers, 2002 IEEE International Solid-State Circuits Conference*, pages 334–335, February 2002.
13. J. Seng, D. Tullsen, and G. Cai. Power-Sensitive Multithreaded Architecture. In *Proceedings of the 2000 International Conference on Computer Design*, pages 119–206, September 2000.
14. D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.
15. D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.
16. V. Zyuban and P. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.

Counteracting Bank Misprediction in Sliced First-Level Caches[★]

Enrique F. Torres¹, P. Ibañez¹, V. Viñals¹, and J.M. Llabería²

¹ DIIS, Universidad de Zaragoza

² DAC, Universidad Politécnica de Catalunya

Abstract. Future processors having sliced memory pipelines will rely on bank prediction to schedule memory instructions to a first-level cache split into banks. In a deeply pipelined processor, even a small bank misprediction rate may degrade performance severely. The goal of this paper is to counteract the bank misprediction penalty, so that in spite of such bank misprediction, performance suffers little. Our contribution is twofold: a new recovery scheme for latency misprediction, and two policies for selectively replicating loads to all banks. The proposals have been evaluated for 4 and 8-way superscalar processors and a wide range of pipeline depths. The best combination of our mechanisms improves IPC of an 8-way baseline processor up to 11%, removing up to two thirds of the bank misprediction penalty.

1 Introduction

Current superscalar processors have the potential for exploiting high degrees of ILP, demanding a large memory bandwidth. So, to overcome the potential bottleneck of a single cache port, concurrent accesses to non-blocking first-level caches are required [21]. An ideal implementation should not involve latency increase, because programs often have load instructions at the head of critical chains of dependent instructions and so, increasing the load-use delay could be very harmful to performance.

Several alternatives have been used in commercial processors to support such bandwidth demands. *Truly multiported* caches [17], *Mirror caches* [7] and *Virtual-multiporting* [9,14] caches have been widely studied and have drawbacks in terms of area, latency or scalability. *Multibanking* uses an interconnection component placed between address units and fast single-ported cache banks [4, 11,13]. Multibanking scales easily to multiple ports, but the interconnection component (either a crossbar network [11] or a bank scheduler [4,13]) may increase the latency of loads noticeably.

The *sliced memory pipeline* reduces the load latency of multibanked schemes by removing the interconnection component from the memory pipeline [5,18,22, 23]. Because the Instruction Issue Queue (IQ) does not know memory addresses

[★] This work was supported in part by the Ministry of Education of Spain under grant TIC 2001-0995-C02-02 and the Diputació General de Aragón (BOA 58,14/05/03).

yet and there is only one path to each bank, this design relies on *bank prediction* to schedule memory instructions to banks. A hardware loop between IQ and the address computation stage is responsible for checking the bank prediction (a *loose* loop according to Borch et al. [2]).

Speculation can be used to manage such a loose loop (load speculation). The IQ assumes the bank prediction to be correct and speculatively schedules dependent work at the proper time. If predictions are mostly correct, performance increases. However, at each misprediction the speculative work is lost and a recovery action is needed.

The aim of this papers is to counteracting the bad side effects of bank mispredictions in first-level sliced caches. We first introduce a new recovery scheme for reducing the recovery penalty. Secondly, we propose two policies for a selective replication of loads, so that the load-use delay and the number of load misspeculations are reduced.

We evaluate deeply pipelined 4 and 8-way out-of-order processors, considering that only a few logic levels will fit within the processor cycle [1]. Pentium 4 is an example, with 20 stages up to branch check, and 4 between IQ and Ex [10].

Section 2 details the processor and memory pipelines, offering an alternative for distributing the buffering needed for disambiguation. After detailing the simulation environment in Section 3, in Section 4 we isolate and analyze a number of factors that contribute to the bank misprediction penalty. Next, we propose two mechanisms for counteracting the bank misprediction penalty: Section 5 describes a new recovery mechanism and Section 6 introduces two mechanisms for spreading loads into banks in a selective way.

2 Baseline Processor and Memory System

Processor. Figure 1a shows the execution pipeline of the superscalar processor used as baseline. The front-end pipeline has eight stages from Fetch to IQ.

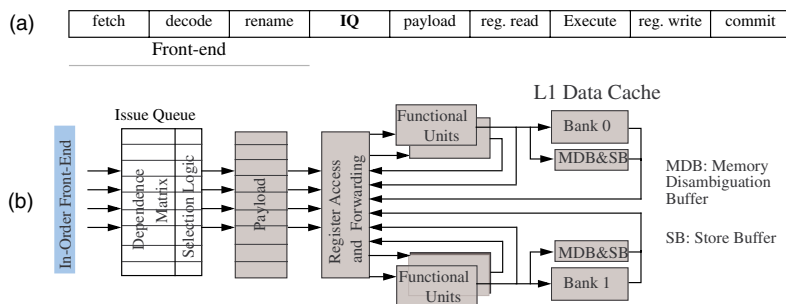


Fig. 1. Execution pipeline (a). Out-of-Order execution core with two cache banks (b).

The IQ has a Dependence Matrix and a Selection Logic [8], see Figure 1b. The Dependence Matrix encloses the dependency information among instructions and

uses wired-OR logic to find ready instructions (wake-up phase). The Selection Logic issues the oldest instructions among the ready ones (select phase). After issuing, instruction payload is read, operands are read from the register file, and execution starts.

If the trend towards rising frequency holds, the number of stages between IQ and Ex will continue increasing [1,2,10]. So, we will take this number as a parameter in all our experiments and analyze its impact on performance.

The processor can issue speculatively while waiting for the resolution of all latency predictions, namely bank prediction, cache hit prediction and disambiguation buffer non-full prediction. Each prediction has its own resolution delay. IQ assumes the cache hit latency for loads, and speculatively issues their dependent instructions during a set of cycles called the *Speculative Window*. The Speculative Window starts at the first cycle in which the direct dependents can issue and ends either when the last-prediction acknowledgement reaches IQ, or when a misprediction is notified to the Recovery stage.

IQ is used as the Recovery stage. The recovery mechanism used as a baseline is described by Morancho et al. for latency prediction in [16]. It starts recovery as soon as a misprediction notification reaches IQ. It is selective, meaning that only the (already issued) instructions that depend on a mispredicted load will be reexecuted. With *Dependent Set* we refer to the dependent instructions issued during the Speculative Window. After every issue cycle, the instructions belonging to the Dependent Set are identified and the non-speculative instructions are removed from IQ. The load itself and the Dependent Set remain in IQ until the last cycle of the Speculative Window ends. Therefore, the IQ pressure increase is only determined by the longest resolution delay.

Sliced memory pipeline. The cache is sliced into banks. Each bank is only tied to one address computation unit. We assume that bank check is made concurrently with address computation by evaluating the expression $A+B=K$ without carry propagation [6]. On a bank misprediction IQ knows about the right bank number during the cycle following bank check, see Figure 2. Then, the mispredicted instruction becomes ready to be issued again to the right bank one cycle later. We model the resolution delay for a bank prediction with a duration of up to 6 cycles, which is the case shown in Figure 2.

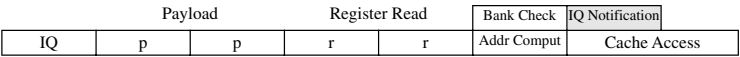


Fig. 2. Bank check delay.

Bank predictor. All memory instructions need a bank prediction. We assume that banks are line-interleaved. Therefore, the less significant bits of the line number are our prediction target. We have chosen to predict each address bit separately: 1 bit for 2 banks and 2 bits for 4 banks. As a bank predictor we use the *enhanced skewed binary predictor*, proposed by Michaud et al. to perform branch prediction [15]. We opt for a global predictor because it eases yielding several predictions per cycle [19]. Every bit predictor is sized identically: 1K

entries for the three required tables and a history length of 10. Since we are interested in recognizing the hard-to-predict loads, we further add a 2-bit confidence saturating counter to every entry in all tables.

Each individual execution of a load has been classified according to confidence (high or low) and bank prediction outcome (right or wrong). Table 1 shows the average percentages reached by the predictor for 2 and 4 banks (see workload details in Section 3).

Table 1. Average percentages reached by the bank predictor

	High confidence		Low confidence	
	right	wrong	right	wrong
2 banks	67.6	3.3	18.5	10.6
4 banks	53.7	2.6	22.8	20.9

Memory data flow. We assume memory disambiguation based on the Alpha 21264 approach [12], where loads place their addresses into a load buffer and stores place address and data at once into a store buffer. In this centralized model, the buffer entries are allocated in-order at decode, filled out-of-order at execution, and finally released at commit.

In a sliced memory system the accesses handled by the different cache banks reference disjoint memory regions. Consequently, the disambiguation unit could be distributed acting locally at each bank. Nevertheless, the entries of such distributed buffers cannot be allocated in fetch order because the bank number is unknown at decode time.

Zyuban&Kogge have suggested a distributed disambiguation mechanism for their multicluster architecture [23]. They solve the allocation problem partially by allocating an entry to every store in the buffer of every cluster at decode time. The main drawback of this approach is that the buffer of every cluster requires almost the same size of a centralized buffer.

We propose reducing the size of buffers in accordance with the number of banks and allocating buffer entries *after* the bank check (out-of-program order). A deadlock situation might arise when an allocation demand is rejected because all buffer entries have been already allocated.

The Buffer-Full condition is managed by adding a new prediction to the memory loop: it is predicted that there will be room and the memory instruction is speculatively issued. If the buffer is eventually full, a latency misprediction occurs. The faulting instruction is reissued from IQ when a signal indicates that the buffer is no longer full. On the other hand, in order to prevent deadlock it is sufficient to warrant that the oldest memory instruction in IQ can progress. This can be easily achieved by allocating the last free entry of a given buffer only if it is requested by the oldest memory instruction in IQ.

As an example, the considered 8-way processor with four banks and up to 128 in-flight loads/stores does not show performance losses when reducing the number of entries of every load buffer to 32 and of every store buffer to 16.

(a)						(b)	
	4-issue	8-issue		4-issue	8-issue	Bench.	Data set
fetch and decode width	4	8	L1 D-cache	2 cycles		bzip2	program-ref
branch predictor: hybrid (bimodal, gshare)	16 bits		banks	2	4	crafty	ref
reorder buffer entries	128	256	ports per bank	1 r/w		eon	rushmeier-ref
load/store in-flight instructions	64	128	bank size	16 KB, 1-way		gcc	166-ref
integer IQ entries	25	64	line size	32 B		gzip	program-ref
integer units	4	8	LD buffer entries	32		mcf	ref
FP IQ entries	15	32	ST buffer entries	16		parser	ref
FP units	2	4				perl	diffmail-ref
L1 I-cache	64KB, 2-way		L2 Unified Cache	1MB, 4-way, 12 cycles		twolf	ref
			main memory	80 cycles		vortex	one-ref
						vpr	route-ref

Fig. 3. Microarchitectural parameters (a). Used SPECint 2K programs (b).

3 Simulation Environment

We evaluate 4 and 8-wide issue processors having 8 stages before IQ and 1 to 4 stages between IQ and Ex. We have modified SimpleScalar 3.0c [3] in order to model a Reorder Buffer and a separate Issue Queue, looking carefully at all situations requiring latency misprediction recovery. We assume an oracle predictor for store-load independency. The microarchitectural parameters used are summarized in Figure 3a.

We are interested in integer code because it exhibits difficult-to-predict address streams. As workload we use all the integer benchmarks of SPECint 2K but 254.gap which could not be executed within our framework (see Figure 3b). We use the Alpha binaries compiled by C. Weaver (www.simplescalar.com), simulating a contiguous run of 100M instructions from the points suggested by Sherwood et al. after a warming-up of 200M instructions [20].

4 Performance Impact of Bank Misprediction

Figure 4a shows execution of a well-predicted load and two dependent instructions. We can see a load-use delay of 2 cycles (cycles 2:3) and a 4-cycle Speculative Window (cycles 4:7) inside which the Dependent Set (*add*, *sub*) is issued.

Figure 4b shows a bank misprediction. When IQ is notified a recovery action is performed during a full processor cycle (cycle 7, box with *N*). Recovery implies stalling issue because the results of both the load and the Dependent Set must be tagged as not available in the Matrix Dependence. In the same cycle, the load itself and the Dependent Set become visible to the issue logic and they will be executed again as their operands become available (cycles 8, 11, and 12 for *ld*, *add* and *sub*, respectively).

When comparing the timings of the right and wrong bank predictions, the following differences appear: the load-use delay has 7 added cycles (cycles 4:10),

the work started by the Dependent Set during the Speculative Window has been lost, and finally, a cycle is spent for the recovery action (cycle 7). Summarizing,

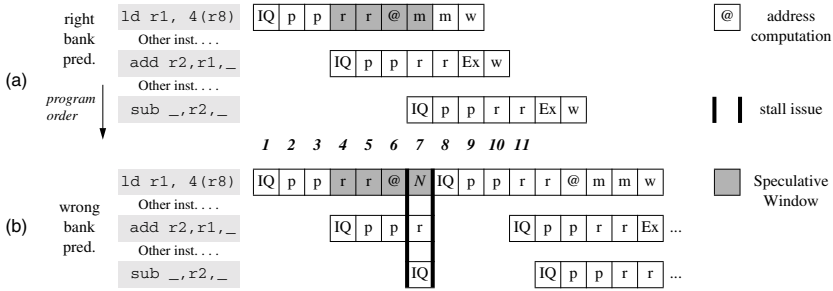


Fig. 4. Right and wrong bank predictions, assuming 4 cycles between IQ and Ex. For the sake of clarity only one instruction per cycle is issued. We show the stages of the instructions to be nullified until the cycle where IQ is notified of the misprediction.

the bank misprediction penalty is due to a number of factors, namely *the load-use delay increase*, *work loss*, and *recovery penalty*. We group the last two factors under the term *misspeculation penalty*. Notice that, as the resolution delay increases (cycles 2:7) both the load-use delay and the work lost may also increase.

To quantify this penalty, Figure 5 shows IPC for processors having from 1 to 4 stages between IQ and Ex. First, we look at the bars labelled BASE and OBP (Oracle Bank Predictor). The BASE processor has the realistic bank predictor introduced above, while OBP has an oracle that always predicts the correct bank. So the IPC difference between OBP and the BASE processor (OBP-BASE) is the bank misprediction penalty (and the room for improvement). The IPC for OBP worsens with the number of IQ-Ex stages because the branch penalty increases.

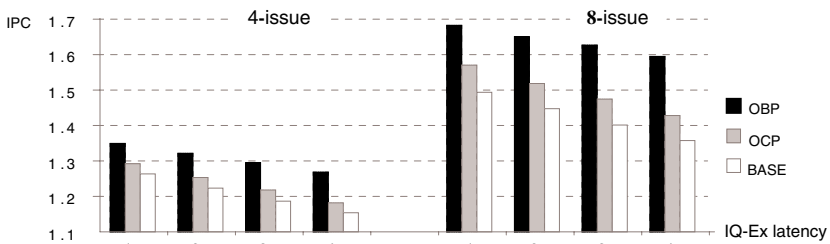


Fig. 5. Harmonic IPC mean with an Oracle Bank Predictor (OBP), an Oracle Confidence Predictor (OCP), and the real predictor (BASE).

Compared with OBP, the 4-way BASE processors experience an IPC loss ranging from 6.4% to 9.0% as the number of IQ-Ex stages increases from 1 to 4. Eight-way BASE processors have more IPC loss, ranging from 11.2% to 15.0%.

Another interesting point is breaking the misprediction penalty down into load-use delay increase and misspeculation penalty. For that we simulate the

real bank predictor again, but now if the prediction is wrong the dependent instructions are not issued (OCP, Oracle Confidence Predictor). Thus, the mis-speculation penalty is removed and so (OBP-OCP) difference is the load-use delay increase, whereas (OCP-BASE) difference is the misspeculation penalty.

Results show that both degrading factors are significant. However, load latency weighs more as the number of IQ-Ex stages increases; specifically, its contribution ranges from two thirds to three quarters as the number of IQ-Ex stages increases from 1 to 4. This is true for both 4-way and 8-way processors. In Section 6 we show how to reduce the overall misprediction penalty by replicating some loads, and in Section 5 we introduce a new recovery mechanism for reducing the recovery penalty.

5 Chained Recovery

Chained Recovery (CR) reduces the recovery penalty by not stalling issue as the baseline recovery does. Instead, the recovery process may require now several cycles because, in a given cycle, instructions that depend on previously nullified ones may be issued. In this Section the CR concept is applied to bank misprediction, but notice that it can also be used to recover from any kind of latency misprediction.

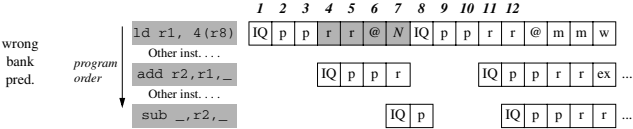


Fig. 6. Bank misprediction and Chained Recovery.

Figure 6 shows how CR acts on the code of Figure 4b. In the first recovery cycle (cycle 7), the results of the mispredicted load and the Dependent Set are tagged as not available. However, in the same cycle the Issue Logic selects an instruction (*sub*) which depends on another one that is being nullified (*add*). Next, in cycle 8, CR identifies which instructions issued in the previous cycle depends on the mispredicted load. Finally (cycle 9), the results of the instructions identified by CR in cycle 8 are tagged as not available.

From now on, in every cycle (cycles 10, 11, ...) the two actions stated above are made at once: *a*) the results of the instructions identified as dependents in the previous cycle are tagged as not available, and *b*) those instructions issued in the previous cycle and depending on nullified instructions are identified. The recovery process will eventually finish in the cycle where no issued instruction depends on any nullified instruction.

Adding a bit vector to the baseline recovery mechanism described in [16] is enough to implement CR. This bit vector check whether the instructions issued in a cycle depend or not on instructions nullified in the previous cycle.

Evaluation. Figure 7 shows results again for the OCP and BASE processors considered in Section 4. The central bar in each bar group represents a processor

enhanced with CR. As expected, the misspeculation penalty paid for CR is quite small -remember that such a penalty is the (OCP-CR) difference. So, the work lost during the Speculative Window and the issue slots wasted during the chained recovery have almost no adverse effects on performance.

Summarizing, an 8-way processor with CR outperforms the baseline from 4.8% to 3.7% as the number of IQ-Ex stages grows from 1 to 4. In 4-way processors the benefits are lower, ranging from 1.9% to 1.8%.

6 Load Replication

As shown in Section 4, bank mispredictions increase the load-use delay and the effect is more pronounced as the number of the IQ-Ex stages increase. To reduce this penalty, instead of re-issuing from IQ, Neefs et al. propose using a dedicated network *after* the bank check. The network is out of critical timing paths and re-routes the bank-mispredicted instructions towards the correct bank [18].

Our point is quite different. We consider taking conservative actions *before* knowing about the outcome of the bank check. By conservative actions we understand issuing a load to several banks in limited cases. Next we show two approaches to this idea.

The first approach is *Replication Using Free Memory Slots* (RF). It consists in making use of idle memory ports. For that, after the selection phase in IQ the oldest load is identified among the selected instructions. Then, several instances of that load are issued to all free memory issue ports, if any. As usual, the dependent instructions can be speculatively scheduled. This alternative is similar to that proposed by Yoaz et al. However, since they assume a separate IQ for the memory instruction stream, the replication policy has no interaction with the integer IQ and the sliced memory system they propose is evaluated with an analytical model, but it does not include load replication [22].

The second approach consists in adding confidence to the bank prediction and whenever it is low, switch to a conservative dispatch policy that discards the prediction. We call this approach *Replication in Dispatch* (RD), and it is accomplished by tagging low-confidence loads in order to be issued to all the banks from the IQ. All load instances are not necessarily issued in the same cycle. Each port scheduler use instruction age to select, among the ready instructions,

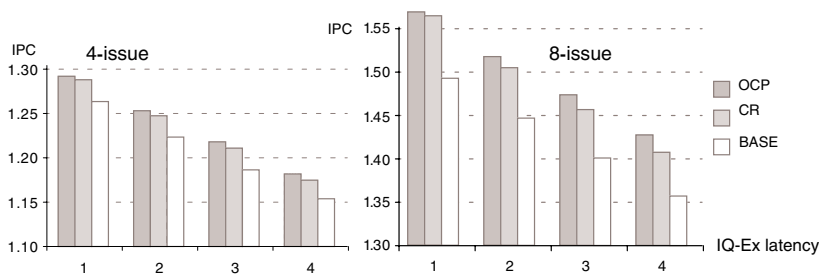


Fig. 7. Harmonic IPC mean for processors with Oracle Confidence Predictor (OCP), Chained Recovery (CR), and the actual predictor (BASE).

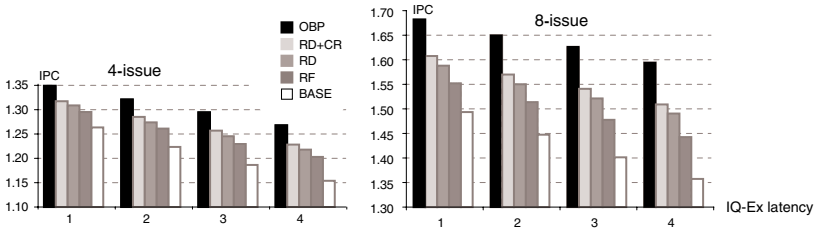


Fig. 8. Harmonic IPC mean. OBP means Oracle Bank Predictor. Replication in Dispatch (RD), Replication Using Free Memory Slots (RF) and Chained Recovery (CR).

the one to be issued. Thus, RD does not disturb the issuing of older instructions with ready operands. Besides, when IQ is notified of the bank check, the load instances pending issuing are nullified. Because a replicated load has low bank confidence, we choose not to wake up the dependent instructions speculatively. The latest load instance or an early bank check will wake them up.

RD could produce a performance loss. The low-confidence loads replicated by RD account for a 43.7% (29%) of all loads for 4 (2) banks (see Table 1), and so, younger instructions could be stalled by lack of issue slots. Moreover, the low-confidence loads that were actually well predicted see their dependent instructions delayed until the last instance is issued. On the other hand, the low-confidence loads that were actually mispredicted (e.g. 21% for 4 banks) win the replication benefit: load-use delay decrease and misspeculation penalty removal. Summing up, the results below will show that the overall effect is clearly positive.

Evaluation. As said in Section 4, counteracting bank misprediction effects offers good opportunities for improving performance, and these opportunities grow as the number of IQ-Ex stages increases; see the gap between OBP and BASE bars in Figure 8. The three bars between OBP and BASE report the performance of a selection of three enhancements of increasing complexity: RF, RD, and the combination of RD and Chained Recovery (RD+CR). Let's focus on the processors having 4 cycles from IQ to Ex and move then from the simplest alternative to the most complete. First we consider RF, which improves IPC 6.3% (4.3%) for an 8 (4)-way processor, respectively. Considering that RF adds a very small complexity in control, these are quite good numbers. Second, we consider the RD approach. We add confidence to the bank predictor by investing 6Kbits in storage, but IPC now rises to 9.8% (5.6%) for an 8 (4)-way processor, respectively. Third, if we add CR on top of RD, we are simultaneously decreasing the misspeculation penalty and the average load-use delay, achieving then 11.1% (6.5%) IPC increase for the 8 (4)-way processor, respectively (see the RD+CR bar). This extra performance comes only from the slight increase in control complexity added by CR.

As a final remark, it's important to realize that for all IQ-Ex numbers and for all issue widths, RD+CR achieves two thirds of the ideal performance given by OBP. This makes RD+CR very sound across important microarchitectural parameters and therefore an appealing cost/performance design point.

7 Conclusions

Current trends in microarchitectural design, such as issue width increase and deeper pipelining, have a negative influence on processors having sliced memory pipelines, which is increasingly dependent on the bank misprediction rate. In this paper we have introduced two complementary approaches to reduce the overall bank misprediction penalty.

The first one is *Chained Recovery*, a new recovery mechanism intended to reduce the recovery penalty after a misspeculation. We have shown that CR is very effective, achieving in the worst case 98.6% of the IPC given by an Oracle Confidence Predictor.

The second approach is a set of two specific policies to issue a load to several banks in limited cases. *Replication Using Free Memory Slots* replicate a load to free banks when there are free issue slots at issue time. It achieves a maximum 6.3% increase in performance adding only a modest control complexity. On the other hand, *Replication in Dispatch* replicates to all banks those loads with a low-confidence prediction. The increase in performance achieves 9.8% in the deepest pipelined 8-way processor, with an affordable increase in the predictor size (6K bits for our bank predictor).

Finally, if we put Chained Recovery and Replication in Dispatch together, we eliminate about two thirds of the IPC lost in bank mispredictions for all the simulated processors.

References

1. V. Agarwal et al.: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. Proc. of 27th ISCA (2000) 248–259
2. E. Borch, E. Tune, S. Manne, and J. Emer: Loose Loops Sink Chips. Proc. of 8th HPCA (2002) 299–310
3. D.C. Burger and T.M. Austin: The SimpleScalar Tool Set, Version 2.0. UW Madison Computer Science Technical Report #1342 (1997)
4. B. Case: Intel Reveals Pentium Implementation Details. MPR vol 7, n 4 (1993) 1–9
5. S. Cho, P. Yew, and G. Lee: A High-Bandwidth Memory Pipeline for Wide Issue Processors. IEEE Trans. on Computers, vol. 50, no. 7 (2001) 709–723
6. J. Cortadella and J.M. Llaberia: Evaluation of $A+B=K$ Conditions without Carry Propagation. IEEE Trans. on Computers, vol. 41, no. 11 (1992) 1484–1488
7. J. Edmondson, P. Rubinfeld, and V. Rajagopalan: Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro, vol. 15, no. 2 (1995) 33–43
8. J.A. Farrell and T.C. Fisher: Issue Logic for a 600 Mhz Out-of-Order Execution Microprocessor. IEEE J. of Solid-State Circuits, vol. 33, no. 5 (1998) 707–712
9. L. Gwennap: IBM Regains Performance Lead with Power2. MPR, vol. 7, no. 13 (1993)
10. G. Hinton et al.: The Microarchitecture of the Pentium 4 Processor. ITJ Q1 (2001)
11. P. Hsu: Design of the R8000 Microprocessor. IEEE Micro, vol. 14 (1994) 23–33
12. R. Kessler: The Alpha 21264 Microprocessor. IEEE Micro, vol. 19, no. 2 (1999) 24–36
13. A. Kumar: The HP PA-8000 RISC CPU. IEEE Micro, vol. 17, no. 2 (1997) 27–32
14. M. Matson et al.: Circuit Implementation of a 600MHz Superscalar RISC Microprocessor. Proc. of ICCD (1998) 104–110

15. P. Michaud, A. Seznec, and R. Uhlig: Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *Proc. of 24th ISCA* (1997) 292–303
16. E. Morancho, J.M. LLabería, and A. Olivé: Recovery Mechanism for Latency Misprediction. *Proc. of PACT* (2001) 118–128
17. S. Naffziger et al.: The implementation of the Itanium 2 Microprocessor. *IEEE J. Solid State Circuits*, vol. 37, no. 11 (2002) 1448–1460
18. H. Neefs, H. Vandierendonck, and K. De Bosschere: A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks. *Proc. of 6th HPCA* (2000) 313–324
19. A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides: Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. *Proc. of 29th ISCA* (2002) 295–306
20. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder: Automatically Characterizing Large Scale Program Behaviour. *Proc. of ASPLOS* (2002)
21. G.S. Sohi and M. Franklin: High-Bandwidth Memory Systems for Superscalar Processors. *Proc. 4th ASPLOS* (1991) 53–62
22. A. Yoaz, E. Mattan, R. Ronen, and S. Jourdan.: Speculation Techniques for Improving Load Related Instruction Scheduling. *Proc. of 26th ISCA* (1999) 42–53
23. V. Zyuban and P.M. Kogge: Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Trans. on Computers*, vol. 50, no. 3 (2001) 268–285

An Enhanced Trace Scheduler for SPARC Processors

Spiros Kalogeropoulos

Sun Microsystems, 16 Network Circle MS: UMPK16-203,
Menlo Park CA 94025, USA.
`spiros.kalogeropoulos@sun.com`

Abstract. In this paper an enhanced trace scheduler implementation is described which targets processors with moderate support for parallelism and medium size register file such as the *SPARC* processors *UltraSPARC(rtm) II* and *UltraSPARC(rtm) III*. The enhanced trace scheduler is a global instruction scheduler, which identifies and exploits the available instruction level parallelism in a routine, contributing to performance improvement for UltraSPARC processor systems. The enhanced trace scheduler is part of the Sun Forte 6 update 2 product compilers for *UltraSPARC II* and *UltraSPARC III* processors. The enhanced trace scheduler, in exploiting the available instruction level parallelism, attempts to address issues such as register pressure, speculation and the amount of compensation code.

1 Introduction

The *SPARC* processors *UltraSPARC II* and *UltraSPARC III* are in-order issue superscalar processors which provide moderate support for parallelism by issuing and executing up to four instructions per clock cycle. The *SPARC* processors rely on the compiler to produce an instruction schedule that extracts and exploits the available instruction level parallelism in a routine to maximize the instruction issue rate and the parallelism of memory operations by issuing prefetches and loads as early as possible.

The process of detecting and scheduling the available fine grain parallelism is usually applied on the control flow graph of a routine. Instruction schedulers use single basic blocks for detecting and scheduling the available fine grain parallelism. However, single basic blocks contain insufficient instruction level parallelism; therefore higher performance is achieved by exploiting instruction level parallelism in consecutive basic blocks.

Several global instruction scheduling techniques have been described in the literature, which perform instruction scheduling beyond the basic blocks boundaries. Trace scheduling [1], [2] selects a frequently executed sequence of contiguous basic blocks from one path in the control flow, called trace, on which instruction scheduling is performed. Most of the global scheduling techniques [1], [2], [3] target wide issue processors with a big register file and hardware support

for large amount of parallelism. The enhanced trace scheduling technique described in this paper targets the *UltraSPARC II* and *UltraSPARC III* processors which have a medium size architectural and physical register file of 32 integer registers, 32 floating point registers and support for a moderate amount of parallelism. Therefore, the performance improvement due to the exploitation of the available parallelism involves successfully addressing issues such as register pressure, speculation and the amount of compensation code. For instance, aggressive code motion might cause register spilling if the register pressure is high, which negates the benefit of utilizing the parallelism. Furthermore, aggressive speculation could hurt the performance of a program if the latency of the speculated instructions is not hidden by using idle resources of the processor.

In the following section, we present an overview of the enhanced trace scheduler. Subsequently, we describe briefly a flexible trace formation scheme. Next, the scheduling of traces and dealing with register pressure is discussed. Finally, results are presented which evaluate the enhanced trace scheduler work.

2 An Overview of the Enhanced Trace Scheduler

The enhanced trace scheduler phase could be invoked twice. The first invocation is before the register allocation during a phase named as early instruction scheduling. The second invocation could be after register allocation during the late instruction scheduling phase. The early instruction scheduling is applied on a compiler intermediate representation where values are kept in an unlimited number of virtual registers. On the other hand, the late instruction scheduling is applied on the same compiler intermediate representation after the register allocation where values reside in actual machine register.

The enhanced trace scheduler performs the following actions repetitively using the control flow of a routine until it cannot form a trace:

- It forms a trace and annotates the trace with the following information:
 - Data flow information for control flow edges entering, exiting the trace for virtual registers which are accessed but will not be renamed.
 - Information regarding the original basic block of each instruction in the trace and its properties such as execution probabilities.
- It invokes the trace instruction scheduler which addresses issues such as register pressure, speculative code motion.
- It introduces partial renaming, compensation code. Compensation code is introduced in a similar way to [1], [2].

3 Trace Formation

The enhanced trace scheduler targets a wide variety of applications with diverse enhancing performance requirements. For instance, in database applications the main requirement for performance improvement is to improve the concurrency in memory operations, since the portion of run time spent on memory operations is

high. The enhanced trace scheduler tackles the above problem by being flexible in the trace formation and by producing traces which match the performance characteristics and potential of different applications.

The trace formation phase may form three different kinds of traces, called *hot* trace, *warm* trace and *long* trace, by applying different edge execution probabilities thresholds during the trace formation.

After determining the basic blocks that will be part of the trace using a technique similar to [1], [2], the next step is to collapse all the basic blocks into a trace block. During the collapsing of the basic blocks a pseudo instruction called *join* instruction is inserted at the beginning of each basic block. The *join* instruction represents the notion of the basic block in the trace and the data flow information for control flow edges entering and exiting the trace. Furthermore, a mapping of the instructions in each basic block to its *join* instruction is maintained before and after scheduling the instructions in the trace. This information will enable us to identify the instructions that have moved from their original block after scheduling the instructions in the trace.

4 Scheduling Traces

Our approach to schedule instructions in a trace is different from other trace scheduling techniques [1], [2]. In the enhanced trace scheduler first the available parallelism is exposed. We improve the available parallelism by ignoring the data dependencies due to virtual register accesses and memory accesses in the off-trace control flow paths which enter or exit a trace and employing virtual register partial renaming on the hoisted instructions. Subsequently, its usefulness is evaluated by using a cost benefit analysis scheme which calculates the impact of speculation, compensation code and register pressure. Finally, the instructions are scheduled in the trace.

The instruction scheduler exposes the available parallelism by viewing the trace as one big basic block when it builds the necessary information for scheduling such as the DAG (directed acyclic graph). However, the instruction scheduler during scheduling views the basic blocks in the trace distinctively. Therefore the instruction scheduler is able to analyze the code motion and distinguish between a speculative code motion and a non speculative code motion. The above analysis is facilitated by maintaining information about the current basic block under scheduling and using the information of the *join* pseudo instructions. At the beginning of the scheduling, the current basic block is the first block in the trace, when all its instructions are scheduled, then the next block in the trace becomes the current basic block.

Before the scheduling commences, the instruction scheduler builds a DAG for the whole trace, where the nodes are instructions in the trace and the edges represent the data dependencies between the instructions. After building the DAG the instruction scheduler maintains height information for each instruction in the trace. The height of an instruction in a DAG is the estimated execution time of the longest path from the instruction to the exit of the DAG. Height information

is one of the factors for prioritizing the scheduling of ready for execution instructions. Furthermore, for each instruction in the trace, the information about its original block is maintained. The above information helps the instruction scheduler to find the estimated execution probability of the instruction. The execution probability is used by the cost benefit analysis scheme to determine the usefulness of the available parallelism.

The instruction scheduler is list driven and uses the DAG to build a ready list of instructions in the trace which are available for execution. Subsequently, a cost benefit analysis scheme is employed to evaluate the instructions which may belong to different basic blocks using the following heuristics:

- Earliest instruction issue time.
- A speculative instruction which blocks the further execution of instructions in the pipeline until its execution is completed, for instance a divide instruction, gets low priority.
- An instruction which will cause an amount of compensation code above a certain threshold when moved from its original block gets low priority.
- An instruction which belongs to a basic block, other than the current basic block, with execution probability relative to the head of the trace less than a certain threshold gets low priority.
- If an instruction belongs to a basic block, other than the current basic block, with low execution frequency then it may move only to a control equivalent block.

After determining the useful parallelism, instructions are prioritized further using the following main criteria: 1) Impact on register pressure. 2) Impact on machine resources. 3) Critical path height reduction.

After considering all the above criteria the most suitable instruction is scheduled and is removed from the list of the available for execution instructions.

4.1 Dealing with Register Pressure

Our approach to deal with register pressure is to exploit the available parallelism while the hardware register availability is high and focus on the register pressure impact of the instructions in the trace when the hardware register availability is close to zero.

During the scheduling of each instruction in the trace, we maintain information about the number of live virtual registers and an estimate of the available hardware registers. When our estimation shows that the availability of hardware registers drops close to zero, then the instruction scheduler for the trace gives highest priority to instructions which, if they were scheduled would free a used virtual register or not define a new virtual register. In this way we try to reduce the number of spills.

5 Results

In this section, we present results of experiments conducted to measure the performance improvement when the enhanced trace scheduler is invoked on the

benchmarks SPEC2000 INT. The compilation of the benchmarks used profile feedback where the training data set is different from the actual testing data set used by the benchmarks. The evaluation of the enhanced trace scheduler was achieved by using the peak configuration file, used in Sun’s SPEC submissions, to establish the peak SPEC2000 INT numbers. Then new peak SPEC2000 INT numbers were produced by enabling the enhanced trace scheduler.

By using the peak options without enabling the enhanced trace scheduler, the scheduling of the instructions is done by a basic block scheduler invoked before and after the register allocation. However, before the first time the basic block scheduler is invoked, a global code motion phase that redistributes the available parallelism similar to [4] is applied. The main objective of the above phase is to reduce the critical path of a basic block and its machine resource usage by trying to move some of its instructions to other basic blocks without increasing their height and utilizing the available hardware resources.

Figure 1 shows the performance improvement when the SPEC2000 INT benchmarks are compiled with an additional sequence of options enabling the enhanced trace scheduler on top of the peak flags. The enhanced trace scheduler is invoked after the global code motion phase. The binaries run on a 450 MHz *UltraSPARC II* processor with 4 Mbyte *L2* cache, and a 1050 MHz *UltraSPARC III* processor with 8 Mbytes *L2* cache.

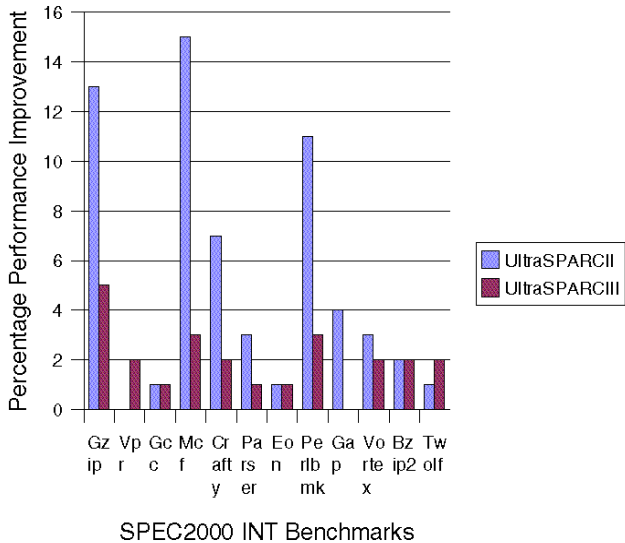


Fig. 1. Performance Improvement on *UltraSPARC II* and *UltraSPARC III* processors

The difference in performance improvement between *UltraSPARC II* and *UltraSPARC III* is due to mainly the different processor characteristics and the opportunities provided by the processors to hide the latency of integer load

instructions missing the *L1* data cache. The *UltraSPARC II* processor has a 16 Kbyte direct mapped *L1* data cache in comparison to *UltraSPARC III*'s 64 Kbytes 4 way associative *L1* data cache. Therefore there are a lot more *L1* cache misses when the SPEC2000 INT benchmarks run on *UltraSPARC II* in comparison to *UltraSPARC III*. The loads which miss the *L1* cache in *UltraSPARC II* stall the pipeline only when an instruction using the value provided by the load executes. The enhanced trace scheduler has the potential to hide the 9 cycles latency of the loads which miss *L1* cache in *UltraSPARC II*, by moving the loads far away from the instructions that use the value provided by the loads and achieve performance improvement. However, in the case of *UltraSPARC III* a load instruction missing the *L1* cache stalls the pipeline till the load instruction completes execution, therefore the code motion cannot hide the latency of the above load. Furthermore, in both processors when a load instruction hits the *L1* data cache the latency for using the load's value is 2-3 cycles. The enhanced trace scheduler can hide the above load latency by scheduling the load instruction and its data dependent instructions at an optimal cycle distance.

6 Conclusions

Most of the global scheduling techniques target processors with hardware features supporting large amount of parallelism where register pressure, speculation and compensation code is not a big issue. However, exploiting successfully the available parallelism for processors with small or medium size register files and moderate support for parallelism, such as *UltraSPARC* processors, involves successfully addressing the issues of speculation, register pressure and compensation code.

Our approach in the enhanced trace scheduler involves a flexible trace formation scheme according to the performance needs of different applications. A partial register renaming scheme is employed to improve the instruction level parallelism. Finally, the usefulness of the available parallelism is determined by using a cost benefit analysis scheme which takes into consideration the impact of speculation, compensation code and register pressure.

References

1. Lowney G., Freudenberger S., Karzes T., and et al. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing, Volume 7. p. 51-142*, 1993.
2. Ellis J. Bulldog: A compiler for VLIW Architectures. *Ph.D. thesis, MIT*, 1986.
3. Hwu W. and et al. The Superblock: An Effective Structure for VLIW and Superscalar Compilation. *The Journal of Supercomputing, Volume 7. p. 229-248*, 1993.
4. Gupta R. and Soffa M. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering, Volume 16. p. 421-431*, 1990.
5. Moon S. and Ebcioğlu K. Paralleling Non-Numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems, Volume 16. p. 1-40*, 1997.

Compiler-Assisted Thread Level Control Speculation

Hideyuki Miura, Luong Dinh Hung, Chitaka Iwama, Daisuke Tashiro,
Niko Demus Barli, Shuichi Sakai, and Hidehiko Tanaka

Graduate School of Information Science and Technology, The University of Tokyo,
7-3-1 Hongo Bunkyo-ku, Tokyo 113-8654, Japan
{hide-m,hung,chitaka,tashiro,niko,sakai,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract. This paper proposes two compiler-assisted techniques to improve thread level control speculation in speculative multithreading executions. The first technique is to increase the overall speculation accuracy by identifying threads which have exactly one successor whose address is statically determinable. The successors of these threads are then predicted using a small full associative buffer, reducing the requirement for entry in the original path-based thread predictor. The second technique is to insert information to enable early validation of thread level control speculation and to reduce the mispeculation penalty. The combination of the two techniques achieved 5.8% performance improvement on average for SPEC95int benchmark.

1 Introduction

Numerous researches have been focused on speculative multithreading (SpMT) architectures to exploit thread level parallelism from sequential programs [1, 6, 2, 4, 5, 7]. In these architectures, a sequential program is aggressively executed as speculative threads running in parallel on multiple processing units. Depending on the execution model, data speculation and/or control speculation are performed using a number of hardware and software supports.

This paper proposes two compiler-assisted techniques to improve thread level control speculation in SpMT architectures. The first technique is to identify threads which have exactly one successor whose address is statically determinable. We call these threads *fixed-successor threads*. Instead of using a sophisticated path-based predictor, successors of these threads are predicted using a small full associative buffer we call *fixed-successor cache*. This helps to reduce the requirement for entry in the path-based predictor that now can be devoted for predicting the other types of threads. The second technique is to identify control independent points of successor threads and inserts notification instructions which are used to validate the prediction previously made by the thread predictor. This technique enables control speculation of a thread to be validated even before the execution of the immediately preceding thread finishes, thus resulting in a reduced mispeculation penalty.

2 Compiler-Assisted Thread Level Control Speculation

In this paper, we employ a thread model similar to Multiscalar [6]. The compiler puts thread boundaries at loop iterations, function return points, and remaining places necessary to satisfy the thread model. We assume an SpMT Chip Multiprocessor (CMP) that consists of a thread control unit (TCU) and four processing units (PUs). During the execution, the threads are predicted by the TCU and scheduled into the PUs following the program order. Inter-thread register dependencies are synchronized while a hardware support for thread level memory speculation similar to [4] is assumed. The TCU includes a 2K-entry path-based thread predictor. It consists of a set of history registers that record addresses of recently executed threads, and a thread target buffer (TTB) that holds the addresses of the successor threads. We use a 4-way set-associative table with 4-bit tag for the TTB. The index for accessing the TTB is generated by concatenating and folding the bits from the history registers, similar to the hashing mechanism proposed in [3].

The next subsections explain the two compiler-assisted techniques we propose to improve thread level control speculation.

2.1 Fixed-Successor Threads

We define a *fixed-successor thread* as a thread which has exactly one successor whose address can be determined statically. In our thread model, the fixed-successor threads occupy 30% of the total number of threads executed. Although a successor of a fixed-successor thread is highly predictable, it may occupy multiple entries in the TTB of the path-based predictor. This contamination results in unnecessary mispredictions.

To reduce such mispredictions, we add a small full associative buffer exclusively used for predicting the successors of fixed-successor threads. We call the buffer *fixed-successor cache*. The organization of the predictor is shown in Figure 1. When predicting a successor thread, both the TTB and the fixed-successor cache are looked up. If a valid entry is found in the fixed-successor cache, the predicted address is obtained from it. Otherwise, the predicted address is obtained from the TTB. The TTB is always updated when a misprediction occurs, while the fixed-successor cache is only updated when the mispredicted thread is a successor of a fixed-successor thread. The compiler assists the hardware to identify a fixed-successor thread by putting a mark on its thread header. A fixed-successor thread can be statically determined using thread reachability analysis on the program control flow graph.

In addition to the above mechanism, the compiler also inserts the starting address of the successor into the thread header. When a fixed-successor thread is assigned to a PU, the PU decodes the header and notifies the TCU of the successor address. When the TCU detects a misprediction, it flushes the current thread and restarts execution of a new thread using the address supplied by the PU. We call this technique *early fixed-successor validation*. This technique can also be interpreted as a special case of *early validation* described shortly below.

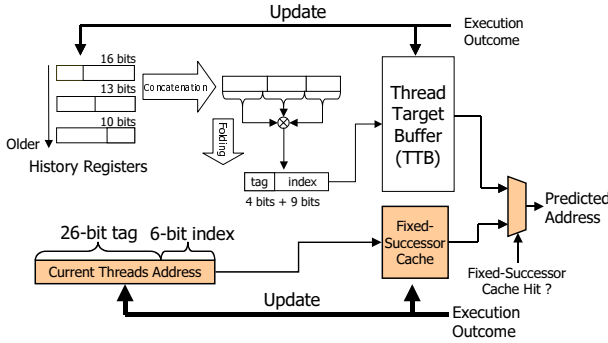


Fig. 1. Structure of the path-based thread predictor coupled with a fixed-successor cache.

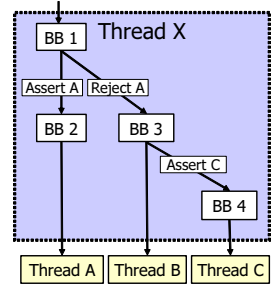


Fig. 2. Identification of assert and reject points in a thread.

2.2 Early Validation

Normally, the validation of a thread prediction must wait until the execution of the immediately preceding thread finishes. At this point, the address of the successor thread is notified to the TCU, and if a misprediction occurs, the incorrectly predicted thread is flushed and a new successor thread is restarted from the notified address. Misprediction penalty of such approach is however unnecessarily large. Thread prediction can be validated earlier since the possible paths taken by the control flow converge as the execution advances.

We propose an early validation technique for the purpose. For each statically defined thread, the compiler identifies *assert points* and *reject points* of successor thread candidates. An assert point of a successor thread candidate is a point where, if reached, the control flow is guaranteed to flow to the candidate. On the other hand, a reject point of a successor thread candidate is a point where the candidate becomes unreachable.

Assert and reject points in a thread are identified by the compiler by performing reachability analysis on the thread control flow graph. For each point, the compiler inserts a special instruction to notify the TCU about the reachability of the corresponding successor thread candidate. We call the instructions *assert instruction* and *reject instruction* respectively. The instructions carry a pointer to a register that will hold the starting address of a successor thread in consideration. Another instruction to set the thread address into the register is also inserted. This results in two additional instructions for each assertion or rejection of a thread. Figure 2 illustrates an example of the insertion of assert and reject instructions. Suppose there is a thread which comprises four basic blocks (BB1~BB4) and has three successor thread candidates (Thread A, B, C). The compiler identifies assert points of Thread A and C, a reject point of Thread C, and inserts the corresponding instructions as shown in the figure.

Assert and reject instructions are processed as follows. When a PU encounters an assert or a reject instruction, it sends the starting address of a successor

thread designated by the instruction to the TCU. The TCU then compares the notified address with the address it previously predicted. For an assert instruction, a misprediction is detected if a mismatch occurs and execution of the correct successor thread can be immediately restarted. For a reject instruction a misprediction is detected if a match occurs. In this case, since a reject instruction by itself does not tell the correct successor address, we can optionally repredict and restart execution of a new successor thread.

3 Evaluations

To evaluate the impact of our proposed techniques, we conducted simulations using a trace-based SpMT CMP simulator that models the four PUs and the TCU previously described in Section 2. Each PU is a 4-issue 10-stage out-of-order superscalar core with a 32KB L1 data and instruction cache (2-cycle latency). L2 cache shared by all PUs is assumed to have an infinite size (always hit, 6-cycle latency). Eight SPEC95int benchmark applications are used for the simulations. The input parameters are adjusted so that the execution finishes between 100-300 million instructions. We simulated the following five control speculation models.

- **TTB**: In this model, threads are predicted using a path-based predictor with a 2K-entry TTB.
- **TTB+FSC**: This model integrates a 64-entry fixed-successor cache into the TTB model.
- **TTB+FSC+EFSV**: In addition to TTB+FSC model, this model employs early fixed-successor validation. Header of a fixed-successor thread holds the address of its successor. After the header is decoded, the address is notified to the TCU for validation.
- **TTB+FSC+EFSV+ASSERT**: In addition to TTB+FSC+EFSV, this model further employs early validation technique using assert instructions.
- **TTB+FSC+EFSV+ASSERT+REJECT**: In this model, we further add early validation technique using reject instructions. When a misprediction occurs, the TCU has to repredict a new successor thread to restart the execution. Here, since our first interest is on the potential of early validation using reject instructions, we assume that the reprediction always succeeds.

Figure 3 shows the performance gain of the last four models over the baseline TTB model. The TTB+FSC model improved the performance by 0.5% on average. The performance gain is especially noticeable for *go*, *gcc* and *vortex*, whose requirement for TTB entry is relatively large. For these programs the improvements are 2.2%, 1.1%, and 1.1% respectively. These gains came from a better prediction accuracy achieved by integrating the fixed-successor cache into the thread predictor.

Employing early fixed-successor validation as in the TTB+FSC+EFSV model further increased the performance gain relative to the baseline model to 1.2% on average. Similar to the TTB+FSC model, the improvements are especially observable in *go*, *gcc*, and *vortex*. In these applications, early fixed-successor

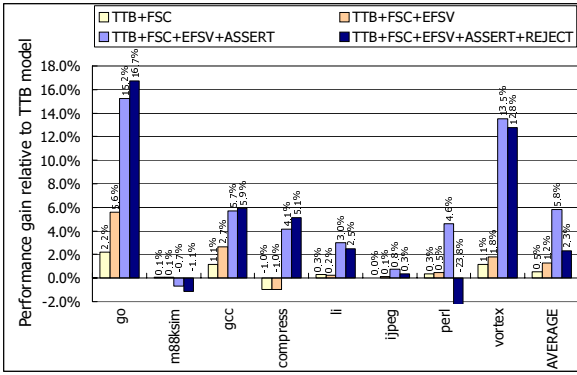


Fig. 3. Impact of the proposed techniques.

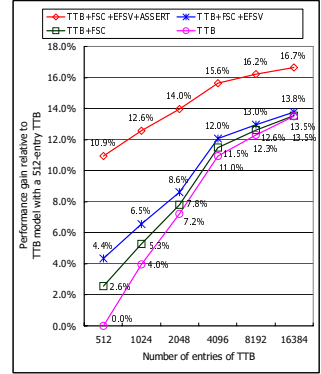


Fig. 4. Varying the size of TTB.

validation salvages the performance loss when a successor of a fixed-successor thread is mispredicted by both the TTB and the fixed-successor cache.

In the TTB+FSC+EFSV+ASSERT model, applying early validation technique using assert instruction significantly increased the performance to an average of 5.8% over the baseline model. Especially in *go* and *vortex* the gains are substantially large: 15.2% and 13.5% respectively. This is because in *go*, the control flow is hard to predict and the accuracy of baseline predictor is low. Therefore, early validation significantly reduced the penalty suffered from thread mispredictions. As for *vortex*, the largest part of mispredictions happens when predicting a successor thread of a thread exiting from a return instruction. In this case, since the return address is also notified using assert instruction, the misprediction penalty is significantly reduced.

In the TTB+FSC+EFSV+ASSERT+REJECT model, further incorporating early validation using reject instructions is not beneficial for some applications. This is mainly because the execution overhead of inserted reject instructions is larger than the effect of reduced mispeculation penalty. Although we do not show the data for all applications, in *perl* for example, the ratio of the number of reject instructions to useful instructions is 18.3%, resulting in a performance decrease of 23.8%.

Figure 4 shows the performance gain when we varied the number of TTB entries. The TTB+FSC+EFSV+ASSERT model achieved a higher performance than the TTB model by 8.6% at 1K-entry, and 2.8% at 16K-entry TTB configuration. The gain is smaller for a larger TTB, because TTB alone achieves higher prediction accuracy. It should also be noted that the effect of fixed-successor information as in TTB+FSC and TTB+FSC+EFSV models is more useful when the TTB is smaller. Furthermore, the impact is expected to be larger if a tagless direct-mapped TTB is used instead of the tagged set-associative TTB used in this paper.

Overall, incorporating the techniques we proposed, TTB size can be reduced substantially while maintaining the performance level. For example, the TTB+FSC+EFSV+ASSERT model with a 1K-entry TTB shows a better performance than the TTB model with a 4K-entry TTB.

4 Conclusion

This paper explored two compiler-assisted techniques to improve thread level control speculation in speculative multithreading executions. The first technique used the compiler to identify *fixed-successor threads*, i.e. threads which have exactly one successor whose address is statically determinable. Since the successors of these threads are easy to predict, we used a small full associative buffer for the prediction we call *fixed-successor cache*. We can then dedicate a sophisticated path-based predictor for predicting the other types of successor threads, increasing the overall accuracy of thread prediction.

The second technique used the compiler to identify *assert* and *reject* points of successor thread candidates. An assert point of a successor thread candidate is a point where, if reached, the control flow is guaranteed to flow to the candidate. On the other hand, a reject point of a successor thread candidate is a point where the candidate becomes unreachable. Using this information, the validation of a thread prediction can be performed before the execution of the immediately preceding thread finishes, thus reducing the penalty when a misprediction occurs.

Evaluation results showed that for a path-based predictor with a 2K-entry TTB, utilizing fixed-successor cache, fixed-successor early validation, and early validation using assert notification, achieved performance improvement of 5.8% on average. Reject notification, however, suffered from a substantial increase in the number of instructions, and did not contribute to further performance improvement.

References

1. H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st MICRO*, pages 226–236, 1998.
2. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th ASPLOS*, pages 58–69, 1998.
3. Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control Flow Speculation in Multiscalar Processors. In *Proc. of the 3rd HPCA*, pages 218–229, 1997.
4. V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.
5. P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th ICS*, pages 77–84, 1998.
6. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd ISCA*, pages 414–425, 1995.
7. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th ISCA*, pages 1–12, 2000.

Compression in Data Caches with Compressible Field Isolation for Recursive Data Structures

Masamichi Takagi and Kei Hiraki

Dept. of CS, Grad. School of Info. Science and Tech., Univ. of Tokyo
`{takagi-m, hiraki}@is.s.u-tokyo.ac.jp`

Abstract. We introduce a software/hardware scheme called the Field Array Compression Technique (FACT) which reduces cache misses due to recursive data structures. Using a data layout transformation, data with temporal affinity is gathered in contiguous memory, where the recursive pointers and integer fields are compressed. As a result, one cache-block can capture a greater amount of data with temporal affinity, especially pointers, improving the prefetching effect of a cache-block. In addition, the compression enlarges the effective cache capacity. On a suite of pointer-intensive programs, FACT achieves a 41.6% reduction in memory stall time and a 37.4% speedup on average.

1 Introduction

Non-numeric programs often use recursive data structures (RDS). For example, they are used to represent variable-length object-lists, trees for data repositories. Such programs using RDS make graphs and traverse them, however the traversal code often leads to cache misses. Many studies have proposed techniques effective for reducing these misses. These are (1) data prefetching [3], (2) data layout transformations, which gather data with temporal affinity in contiguous memory to improve the prefetch effect of a cache-block [1], and (3) data compression in caches, which compresses the data stored in the caches [4,5,6]. Not only does compression enlarge the effective cache capacity, but it also increases the effective cache-block size. Therefore applying it with the data layout transformation can produce a synergy effect that further enhances the prefetch effect of a cache-block. This combined method can be complementary to data prefetching. While we can compress data into $\frac{1}{8}$ or less of its original size in some programs, existing data compression methods in data caches limit the compression ratio to $\frac{1}{2}$, mainly due to the hardware complexity in the cache structure [4,5,6]. Therefore we propose a method which achieves a compression ratio over $\frac{1}{2}$.

In this paper, we propose a software/hardware scheme which we call the Field Array Compression Technique (FACT). FACT aims to reduce cache misses caused by RDS through data layout transformation of the structures and compression of the structure fields. This has several positive effects. (1) The data layout transformation improves the prefetch effect of a cache-block. (2) FACT compresses recursive pointer and integer fields of the structure, and this compression further enhances the prefetch effect by enlarging the effective cache-block

size. (3) This compression also enlarges the effective cache capacity. Since FACT utilizes a novel data layout scheme for both the uncompressed data and the compressed data in memory and utilizes a novel form of addressing to reference the compressed data in the caches, it requires only slight modification to the conventional cache structure. Therefore FACT exceeds the limit of existing compression methods, which exhibit a compression ratio of $\frac{1}{2}$.

2 Field Array Compression Technique

The detailed steps of FACT are as follows: (1) We first take profile-runs to inspect the runtime values of the recursive pointer and integer fields, and we locate fields which contain values that are often compressible. These compressible fields are the targets of the compression. (2) By modifying the source code, we transform the data layout of the target structure to isolate and gather the compressible fields from different instances in the form of an array of fields. (3) We replace the load/store instructions which access the target fields with special instructions, which also compress/decompress the data (we call these instructions `cld/cst`). (4) During runtime, the `cld/cst` instructions carry out the compression/decompression using special hardware, as well as performing the normal load/store job. Since this method utilizes a field array, we call it the Field Array Compression Technique (FACT). The compressed data is handled in the same manner as the non-compressed data, and both reside in the same cache.

2.1 Compression of Structure Fields

To compress the recursive pointer fields, we replace the absolute address of a pointer with a relative address in units of the structure size, which can be represented using a narrower bit-width than the absolute address. Figure 1 illustrates the compression. Assume we are constructing a balanced binary tree in depth-first order using RDS (1). We use the custom memory allocator which arranges the instances in contiguous memory (2). Therefore we can replace the pointers with relative orders (1). The compression/decompression is done when writing/reading pointer fields. Assume the compression ratio is $1/R$. Note that since the difference between the addresses of two contiguous instances is 8 bytes due to the data layout transformation, the relative order is equal to the address difference of the two instances divided by 8, and we use this as the $64/R$ -bit codeword. We use a special codeword to indicate incompressibility, and which is handled differently by the `cld` instruction if the difference is outside the range that can be expressed by a standard codeword. As to the integer field compression, FACT utilizes two methods [7]. As to the selection of the compression target fields in a program, we use profiling method, whose details are described in [7]. In addition, we choose one compression ratio and one integer compression method used in a program using this profiling.

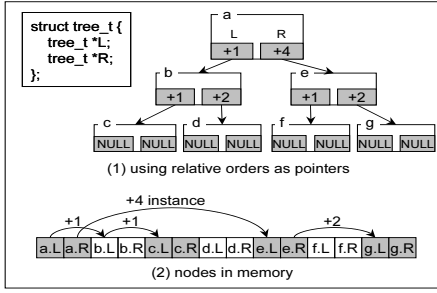


Fig. 1. Pointer compression.

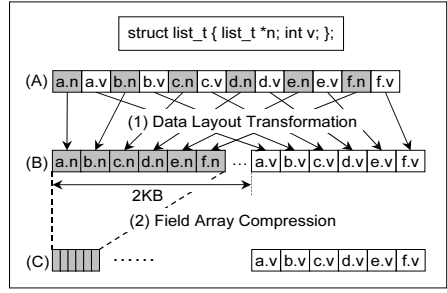


Fig. 2. Instance Interleaving (I2).

2.2 Data Layout Transformation for Compression

We transform the data layout of RDS to make it suitable for compression. The transformation is the same as Instance Interleaving (I2) proposed in [1]. We implement I2 by modifying the source code [7]. Since the compression shifts the position of the data, accessing the compressed data in the caches requires a memory instruction to translate the address for the uncompressed data into the address which points to the compressed data in the caches. When we use the different address space in the caches for the compressed data, the translation can be done by shrinking the address for the uncompressed data by the compression ratio. However, the processor must know the ratio which varies with the structure types. To solve this problem, we transform the data layout of RDS to isolate and group the compressible fields away from the incompressible fields. Assume as an example compressing a structure which has a compressible pointer n and an incompressible integer v . Figure 2 illustrates the isolation. Since field n is compressible, we group all the n fields from the different instances of the structure and make them contiguous in memory. We segregate n and v as arrays (B). Assume all the n fields are compressible at the compression ratio of $\frac{1}{8}$, which is often the case. Then we can compress the entire array of pointers (C). In addition, the address translation becomes a simple division-by-eight. We can also exploit temporal affinity between fields through the transformation. Assume two n pointers contiguous in memory have temporal affinity and each instance requires 16 bytes without I2. Using a 64-byte cache-block, 1 cache-block can hold 4 n pointers with temporal affinity (A). With I2, it can hold 8 of these pointers (B), and with compression at a ratio of $\frac{1}{8}$, it can hold 64 of these pointers (C). This transformation enhances the prefetch effect of a cache-block.

2.3 Address Translation for Compressed Data

Since we attempt to compress data which changes dynamically, we find it is not always compressible. Therefore we need area for both the compressed and uncompressed data. We allocate space for the both initially. This layout requires an address translation from the uncompressed data to the compressed data. We

can calculate it using an affine transformation with the following steps: FACT uses a custom allocator, which allocates a memory block (for example, 2 KB) and divides it into two for the compressed data and the uncompressed data. When using a compression ratio of $\frac{1}{8}$, it divides the total block into a 1 : 8 ratio for the compressed data block and the uncompressed data block. This layout also provides the compressed data with spatial locality. However, this layout restricts the position of the compressed data, thus causing cache conflicts. Therefore we prepare new address space for the compressed data in the caches. We add a 1-bit tag to the caches to distinguish the address spaces. Figure 3 illustrates these address spaces. Consider the uncompressed data D and its physical address A (1), the compressed data d and its physical address a (3), and the compression ratio of $\frac{1}{R}$. We utilize $\frac{A}{R}$ in this new address space to point to d in the caches (2). We need only to shift A to get $\frac{A}{R}$. That is, `cld/cst` instructions that access D shift given address A into $\frac{A}{R}$ and access d in the caches using $\frac{A}{R}$ (X). When the compressed data needs to be written-back to or fetched from main memory, we translate address $\frac{A}{R}$ into address a (Y).

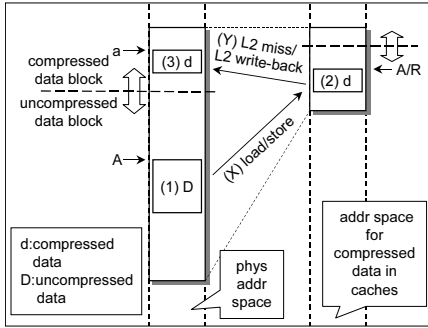


Fig. 3. Address translation in FACT.

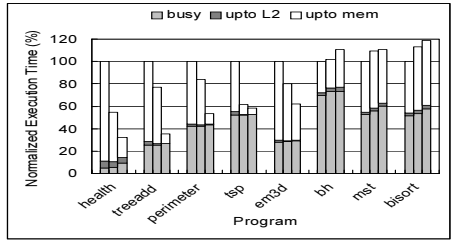


Fig. 4. Execution time comparison. In each group, each bar shows from the left, execution time of the baseline configuration, with I2, and with FACT, respectively.

2.4 Deployment and Action of `cld/cst` Instructions

FACT replaces the load/store instructions that access the compression target fields with `cld/cst` instructions. There are three types of `cld/cst` instructions, corresponding to the compression targets and methods, and we choose an appropriate type for each replacement. The `cst` instruction checks whether its data is compressible, and if it is, `cst` compresses it and puts it in the cache after the address translation which shrinks the address by the compression ratio. When `cst` encounters incompressible data, it stores the codeword indicating incompressibility, and then it stores the uncompressed data to the address before translation. The details of the operation of the `cst/cld` instruction are described in [7].

3 Evaluation Methodology, Results, and Discussions

We assume architecture employing FACT uses a superscalar 64-bit micro-processor, which uses 7-stage pipeline and whose load-to-use latency is 3 cycles. We assume the decompression performed by the `cld` instruction requires one additional cycle to the load-to-use latency. When `cst` and `cld` instructions handle incompressible data, they must access both the compressed data and the uncompressed data. We assume the penalty in this case is at least 4 cycles for `cst` and 6 cycles for `cld`. We assume other compression operations of `cld/cst` instructions do not require additional latency. We developed an execution-driven, cycle-level software simulator of a superscalar processor to evaluate FACT. Table 1 shows its parameters. We used 8 programs from the Olden benchmark [2], `health`, `treeadd`, `perimeter`, `tsp`, `em3d`, `bh`, `mst`, `bisort`. Table 2 shows characteristics and the compression ratio used. They make graphs using RDS as their nodes. Note that different input parameters are used for the profile-run and the evaluation-run. All programs were compiled using Compaq CC version 6.2-504 on Linux Alpha, using optimization “-O4”.

First we show the dynamic memory accesses of the compression target pointers (A_{target}) normalized to the total dynamic accesses (access rate), and the dynamic accesses of compressible pointers normalized to A_{target} (success rate).

Table 1. Simulation parameters

Fetch, Decode, Issue, Retire	up to 8 insts, 128-entry inst. window, 64-entry load/store queue, 256-entry ROB
Exec unit	4 INT, 4 LD/ST, 2 other INT, 2 FADD, 2 FMUL, 2 other FLOAT
L1 data cache	32 KB, 2-way, 64 B blk size 3-cycle load-to-use latency
L2 cache	256 KB, 4-way, 64 B blk size 13-cycle load-to-use latency
Memory	200-cycle load-to-use latency

Table 2. Program used in the evaluation. Note that we modified `perimeter` and `bh` for simulation convenience [7]

Name	Input param. for evaluation	Inst. count	Cmp. ratio
health	lev 5, time 300	69.5M	1/4
treeadd	1M nodes	89.2M	1/8
perim.	16K×16K img	159M	1/8
tsp	64K cities	504M	1/8
em3d	32K nodes, 3D	213M	1/8
bh	4K bodies	565M	1/4
mst	1024 nodes	312M	1/4
bisort	256K integers	736M	1/4

Table 3. Dynamic memory access rate and compression success rate of compression target recursive pointers

Prog.	Access (%)			Success (%)		
ratio→	1/4	1/8	1/16	1/4	1/8	1/16
health	31.1	1.45	1.45	94.6	76.8	76.5
treeadd	11.6	11.6	11.5	100	98.9	96.5
perim.	17.6	17.5	17.6	99.8	95.9	85.6
tsp	10.2	10.2	10.2	100	96.0	67.1
em3d	.487	.487	.487	100	99.6	99.6
bh	1.56	1.56	.320	88.2	51.3	52.2
mst	5.32	5.32	0	100	28.7	0
bisort	43.0	41.2	41.0	90.8	65.6	59.2

Table 3 summarizes the results with various compression ratios. `treeadd`, `perim`, `em3d`, and `tsp` exhibit high success rates. This is because they organize

the nodes in memory in the same order as the traversal. In these programs we can compress many pointers into a single byte. On the other hand, **bh**, **bisort**, **health**, and **mst** exhibit low success rates, because they organize the nodes in a different order to the traversal order. Figure 4 compares the execution times of the programs using the baseline configuration, with I2, and with FACT. Each component in the bar shows from the bottom, busy cycles other than stall cycles due to cache misses (busy), stall cycles due to accesses to the secondary cache (upto L2), and due to accesses to main memory (upto mem). FACT reduces the stall cycles due to cache misses by 41.6% on average. If the traversal order and the memory order of the nodes in the programs are the same, I2 can exploit temporal affinity between fields, which FACT can exploit further by compression. This is the case in **health**, **treeadd**, **perim**, and **em3d**. I2 distributes the fields within one data structure among multiple cache-blocks. It recovers this inefficiency by gathering the fields with temporal affinity in one cache-block. When the memory order and the traversal order of the nodes in the programs are different, I2 cannot recover this inefficiency thus lowering the utilization ratio of a cache-block, resulting in degradation of performance. This is the case in **bh**, **mst**, and **bisort**.

4 Summary

We proposed the Field Array Compression Technique (FACT) which reduces cache misses caused by recursive data structures. Through software simulation, we showed that FACT yields a 37.4% speedup and a 41.6% reduction of memory stall time on average. This paper has four main contributions. (1) FACT achieves the compression ratio of $\frac{1}{8}$. This ratio exceeds $\frac{1}{2}$, which is the limit of existing compression methods. (2) We represent that we can compress many recursive pointer fields into 8 bits. (3) We represent the notion of a split memory space, where we allocate one byte of compressed memory for every 8 bytes of uncompressed memory. Each uncompressed element is represented in the compressed space with a codeword placeholder. This provides compressed data with spatial locality and simplifies the address translation from uncompressed data address to compressed data address. (4) We represent the notion of a new address space for compressed data in the caches. which simplifies the addressing of compressed data in the caches and avoids cache conflict.

References

1. D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. *In Proc. of PACT*, pp. 322–329, Oct. 1998.
2. A. Rogers et al. Supporting dynamic data structures on distributed memory machines. *ACM TOPLAS*, 17(2):233–263, Mar. 1995.
3. A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. *In Proc. of ASPLOS*, pp. 115–126, Oct. 1998.
4. J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. *In Proc. of MICRO*, pp. 258–265, Dec. 2000.

5. S. Y. Larin. Exploiting program redundancy to improve performance, cost and power consumption in embedded systems. Ph. D. Thesis, ECE Dept., North Carolina State Univ., Raleigh, North Carolina, Aug. 2000.
6. Y. Zhang et al. Data compression transformations for dynamically allocated data structures. *In Proc. of Int. Conf on CC*, LNCS 2304, pp. 14–28, Apr. 2002.
7. M. Takagi et al. Compression in data caches with data layout transformation for recursive data structures. TR03-01, Dept. of CS, Univ. of Tokyo, May 2003.

Value Compression to Reduce Power in Data Caches

Carles Aliagas¹, Carlos Molina¹, Montse Garcia¹, Antonio Gonzalez^{2,3}, and
Jordi Tubella²

¹ Departament d'Enginyeria Informàtica i Matemàtiques,
Universitat Rovira i Virgili. Tarragona-SPAIN
{caliagas, cmolina, mgarciaf}@etse.urv.es

² Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya. Barcelona-SPAIN
{antonio, joridit}@ac.upc.es

³ Intel Barcelona Research Center,
Intel Labs-Universitat Politècnica de Catalunya. Barcelona-SPAIN

Abstract. Cache memory represents an important percentage of the total energy consumption of today's processors. This paper proposes a novel cache design based on data compression to reduce the energy consumed by the data cache. The new scheme stores the same amount of information as a conventional cache but in a smaller physical storage. At the same time, the cache latency is preserved, thus no performance penalty is introduced. The benefits of energy vary from 15.5% to 16%, and the reduction in die area ranges from 23% to 40%.

1 Introduction

Caches occupy a very important part of die area in most processors. Percentages of around 50% are relatively common. On the other hand, some authors have reported that caches may be responsible for 10%-20% of total energy consumed by a processor.

The objectives of this work are twofold: to reduce the area of the cache and its energy consumption while maintaining the same amount of information. We present a novel cache architecture that provides significant advantages in terms of these metrics.

The key idea is based on the observation that many values stored in the data cache can be represented with a small number of bits instead of the full 32-bit or 64-bit common representation. Using less bits to store the data, together with a simple encoding/decoding scheme, allows the processor to reduce the storage required for a given amount of data, and thus, reduces the area and energy consumption of the memory.

The rest of the paper is organized as follows. Section 2 discusses and evaluates schemes for compressing data values. The proposed cache architecture is presented in section 3 and evaluated in section 4. Section 5 outlines some related work and finally, section 6 summarizes the main conclusions of this work.

A detailed description of section 2 and 3 is in the following technical report <http://www.ac.upc.es/pub/reports/DAC/2003/UPC-DAC-2003-10.ps.Z>

2 Data Value Compression

Data values have significant redundancy in their representation. In this section we quantify this phenomenon and discuss some different types of data compression that can be used to take advantage of this redundancy.

In general, we focus on 64-bit and 32-bit wide data values. This allows for a huge rang of values to be represented. However, the significant bits of these values are relatively few. We avoid to store non-significant bits using: integer data compression and address data compression.

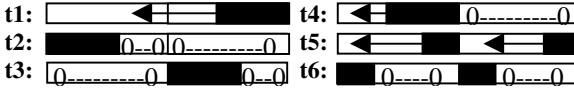


Fig. 1. Integer data compression

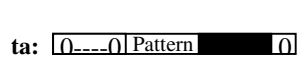


Fig. 2. Address data compression

Integer data compression is applied when the data can be represented with a few bits and the rest of the bits are all ones or zeros. Figure 1 shows the different types of data compression applied to 64 bits. We will only store in the cache the *black* part (significant bits). The arrow indicates sign extension, and the “0..0” means that the field is all zeros.

Another type of data are addresses stored in memory. In this case, we apply address data compression. There are three sources of addresses: data pointers, code pointers and stack pointers. The compiler distributes the memory logical space in three major components: code, data and stack. These addresses are 64-bit wide, but only a part of their bits is variable. The rest of bits remain unaltered and are common to all addresses. We try to recognize the pattern of the common part and store only the variable part. Figure 2 depicts the components of the address data compression. We have experimentally observed that with 8 patterns we can represent 83% of total value addresses.

These 7 types of data compression allow us to compress data form 64 to 22 bits. This can be done for 70% of all data stored in a 16KB data cache, being *type 1*(t1) and *type address*(ta) the most significant. (see framework described in section 4)

3 Pattern Cache

The Pattern Cache is proposed to store data values in a compressed form. Thus the proposed cache architecture reduce the storage area and energy consumption.

The Pattern Cache is divided into two areas: the tag and the data areas. The tag area or Location Table (LT) stores compressed values and pointers to the data area or Value Table (VT). LT is indexed as a conventional data cache. Each LT entry has two fields: a tag that identifies the memory address stored in each line, as in a usual cache, and the location field that determines for each 64-bit word in the line where and how

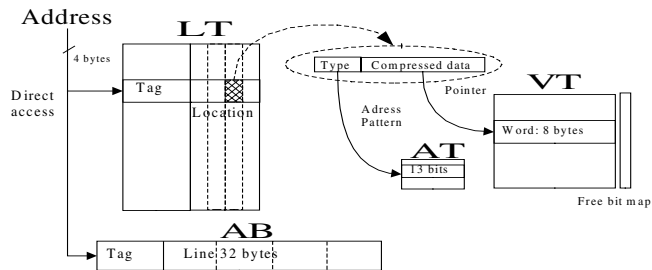


Fig. 3. Pattern cache block diagram

to find its value. Each location field has two parts: type and compressed data. The first one determines how to find the value and the second one is a compressed data or a pointer to VT (Table 1 summarizes the different types of compression). On the other hand, a VT entry has a full 64-bit data value.

Table 1. Compression Type Codification

Type	Information	Type	Information	Type	Information	Type	Information
0000	Pointer	0100	Data: type4	1000	Address: prefix0	1100	Address: prefix4
0001	Data: type1	0101	Data: type5	1001	Address: prefix1	1101	Address: prefix5
0010	Data: type2	0110	Data: type6	1010	Address: prefix2	1110	Address: prefix6
0011	Data: type3	0111	Not used	1011	Address: prefix3	1111	Address: prefix7

There are three ways of obtaining the data values: a) a value may be stored in the LT entry itself using an integer compressed form; b) a value may be stored partly in the LT entry and partly in the Address Table (AT) using an address compressed form; and c) the value may be stored in the Value Table (VT) in a non-compressed form. The behavior of the Pattern Cache may be summarized as follows:

1. Miss on Read or Write: the new line from memory is stored in LT + VT or the Assisting Buffer (AB). Those 64-bits words that can be compressed are stored directly in LT, the others are stored in VT. The AB store the full line when one of 64-bits words cannot be store in LT-VT. The AB also works as a victim cache for lines replaced from the LT [7]
2. Hit on read: if there is a hit in LT, the location field provides either the value or a pointer to the VT. If the value is compressed it is necessary to restore the full value. With Integer Data Compression (Figure 1) only sign extension and zero concatenation is necessary. With Address Data Compression it is necessary to access the Address Table (AT in Figure 3) to obtain the pattern to restore the full value.
3. Hit on write: there are four cases: a) The replaced value is in the VT and the new value cannot be compressed, the latter is stored in the location of the former. b) The replaced value is in the VT and the new value can be compressed, the later is stored in the LT and the VT entry is invalidated. c) The replaced value is in the LT and the new value can be compressed, the later is stored in the location of the former. d) The replaced value is in the LT and the new value cannot be compressed, the value

is tried to be stored in an empty entry of the VT. If there is no space in the VT, all the line is moved to the AB and the entries in the LT and the VT are invalidated.

The compression test requires very few hardware effort and only on Miss or Write. For the integer type compression it is necessary an AND gate to detect a set of ones and an OR gate to detect a set of zeros. For address data compression, an AND gate with some entries inverted is needed for each different pattern. We consider those gates and the AT table (8 entries of 13 bits) negligible in terms of area and energy consumption.

4 Performance and Evaluation

4.1 Static Analysis

This subsection presents an evaluation of the cache area, access time and energy consumption of the Pattern Cache. These evaluations are done by means of the CACTI tool version 3.0 [9]. The input parameters of the CACTI tool have been adapted to model the structure of the Pattern Cache. The assumed technology is $0.09\mu\text{m}$.

The baseline cache is a 16KB direct-mapped cache with 32-bytes line size, with a Victim Cache (VC) of 16 entries full associative. For the Pattern cache: the LT has the same number of lines and associativity as the base cache. Both tags are identical. The location field has 22 bits for each word in a line. And the VT is managed as a direct mapped cache with 8-byte line size. The AB has the same structure of the VC. VT_{xx} means that the VT capacity has been reduced to $xx\%$ of the baseline size.

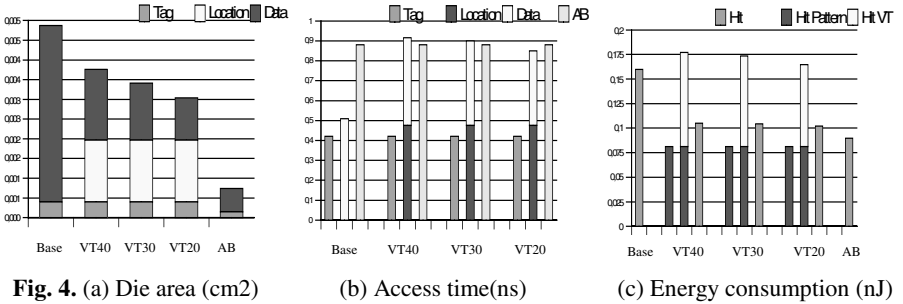


Fig. 4. (a) Die area (cm²)

(b) Access time (ns)

(c) Energy consumption (nJ)

Figure 4.a shows the total die area. Different colors in each bar represent the contribution of *tags*, *location field* and *data*. The rightmost bar represents the area of the VC or AB, that has to be added to all cache configurations, including the baseline. As expected, the achieved area reduction is significant in all cases. For instance, VT30 achieves a reduction of 26% with respect to the baseline cache. Below, it is discussed how these reductions interact with the miss ratio.

The access time on Hit is shown in Figure 4.b. Notice that, every cache configuration has three bars. First bar determines the time required for tag check. Second bar determines the time required to access the value (in a compressed form or through VT). Finally, last bar determines the access time to the VC or AB. In order to determine the total access time, the maximum of those three bars has to be considered. The main

conclusion of this study is that the AB determines the access time. This is only true for caches of less than 128KB.

Finally Figure 4.c shows the energy consumption for a cache hit. In particular, the *Hit pattern* corresponds to the energy consumed by an access in the Pattern Cache that hits in LT. *Hit VT* corresponds to the energy consumed by the Pattern Cache when the value is obtained from VT (LT + VT). *Hit* is the average consumption of the Pattern Cache considering the percentage of *Hit pattern* and *Hit VT* during the execution of programs. Simulations have shown that on average, 75% of the time the value is obtained from LT.

Statistics for cache capacities ranging from 4KB to 64 KB are detailed in Table 2.

4.2 Dynamic Analysis

The simulation environment is built on top of the SimpleScalar [4] Alpha toolkit that has been modified to model the Pattern Cache.

The following Spec2000 benchmarks have been considered: crafty, eon, gcc, gzip, mcf, parser, twolf, vortex and vpr from the integer suite; and ammp, apsi, art, equake, mesa, mgrid, sixtrack, swim and wupwise from the FP suite. The programs have been compiled with maximum level of optimization. The reference input set was used and statistics were collected for 1000 million of instructions after skipping the initializations.

4.3 Analysis of Results

This subsection presents and analyzes the results obtained for different memory level hierarchy configurations in terms of die area, energy consumption and miss ratio.

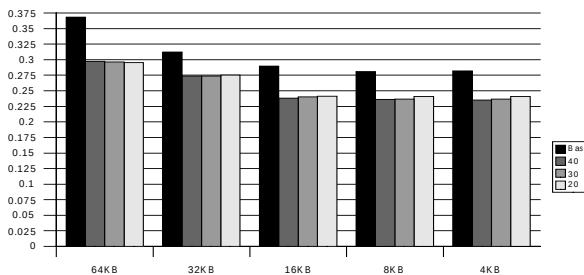


Fig. 5. Energy Consumption

Miss Rate vs. Power Consumption

In order to determine the performance benefits of the cache design proposed in this paper, the scenario described in previous subsection is considered. The size of cache varies from 4KB to 64KB.

Figure 5. shows the energy consumption of the memory hierarchy. Each bar denotes the average energy consumed per access for different cache configurations, which is obtained through the following formula:

$$EC = (\text{Hit\_ratio} * DL1\_EC) + (\text{Miss\_ratio} * (2 * DL1\_EC+ DL2\_EC)) \tag{1}$$

DL1_EC is the average energy consumed by an access to the first level data cache. DL2_EC is the energy consumed by an access to the second level cache. This cache is assumed to be 512KB. Note that this expression computes the total energy by considering the consumption of the first level data cache on hit, and the consumption of the first and the second level data cache on miss. A miss produces two accesses to DL1 (one for detecting the miss, and another for storing the values) and one access to DL2 to obtain the data.

As expected, the Pattern Cache outperforms the base model in terms of energy consumption. The main contribution to this reduction is due to the LT, which provides the data in 75% of the hits. Note also that the VT reduction hardly affects power dissipation since decreasing the VT size increases miss ratio and more data must be provided by the second level cache.

From these numbers and the results in the previous section, we can conclude that the Pattern Cache is an effective architecture for first level data caches in terms of power, die area and access time. Table 2 summarizes all the statistics for each cache configuration being considered. For instance, a reduction of VT to 40% produces an average die area reduction of 19%, a energy consumption reduction of 16% and a very minor (1.5%) increase in the miss ratio.

Table 2. Statistics for different capacities

Cache	Die Area Reduction			Energy Consumption Reduction			Miss Ratio Increment		
	VT40	VT30	VT20	VT40	VT30	VT20	VT40	VT30	VT20
4KB	16%	18%	27%	16.5%	16%	14.5%	1.2%	2.4%	8.6%
8KB	22%	26%	36%	16%	15.5%	14%	1.4%	2.8%	10.0%
16KB	20%	26%	33%	18%	17%	16.5%	1.6%	3.2%	11.0%
32KB	17%	27%	35%	12.5%	12%	12%	1.7%	5.2%	12.2%
64KB	20%	28%	36%	19%	19.5%	20%	1.8%	5.5%	12.9%
AMEAN	19%	25%	33%	16%	16%	15.5%	1.5%	3.8%	10.9%

5 Related Work

Zhang et al. [13] propose the design of the FCV (Frequent Value Cache). The FVC only contains frequently accessed values stored in a compact encoded form and it is used in conjunction of a traditional Direct-Mapped Cache. Yang et al [12] present a similar cache design and evaluation called CC (Compression Cache) where each line can hold one uncompressed line or two cache lines compressed to at least half. A modification of the FVC is proposed in [11] in order to improve energy efficiency.

Significance compression is used by Brooks et al. [3] and Canal et al. [5] to reduce power dissipation, not only in data cache but in the full pipeline. Brooks et al. present a mechanism that dynamically recognizes whether the most significant bits of values are just sign extension and in this case, the functional units operate just on the least significant bits. Canal et al. propose a new significance compression scheme that ap-

pendes two or three extension bits to each data value to indicate the significant byte positions. Other different data compression are also presented in [1],[8] and [10].

6 Conclusions

We have introduced a novel data cache architecture called Pattern Cache. The proposed cache applies different types of compression to values in order to reduce the required storage. Simulation results show that in a data cache, close to 70% of the values stored in 64-bit may be compressed to 22 bits. We have shown that the Pattern Cache significantly reduces power dissipation and die area with a minimum loss in terms of miss ratio while maintaining the same access time as a conventional cache.

References

1. B. Black, B. Rychlik, and J. Shen, "The Block-Based Trace Cache". In Proc. of the 26th ISCA, 1999
2. D. Brooks, V. Tiwari and M. Martonosi, "Watch: A Framework for Architectural-Level Power Analysis and Optimization". In Proc. of the 27th ISCA, 2000
3. D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", In Proc. of the 5th Int. Sym. on HPCA. 1999
4. D. Burger, T.M. Austin and S. Bennet, "Evaluating Future Microprocessors: The SimpleScalar Tool Set". Technical Report CS-TR-96-1308. University of Wisconsin, 1996
5. R. Canal, A. González, and J.E. Smith, "Very Low Power Pipelines using Significance Compression". In Proc. of the 33th Annual Int. Sym. on Microarchitecture, 2000
6. M.Gowan, L.Brio and D. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor". In Proc. of the 35th Design Automation Conference, 1998
7. N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associate Cache and Prefetch Buffers". In Proc. of the 17th ISCA, 1990
8. S. Y. Larin, "Exploiting Program Redundancy to Improve Performance, Cost and Power Consumption in Embedded Systems", Ph.D. Thesis, ECE, North Carolina State U., 2000
9. P. Shivakumar, N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model". Western Research Lab (WRL), Research Report 2001/2
10. L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction". In Proc. of the 33th Annual Int. Sym. on Microarchitecture, 2000
11. J. Yang, and R. Gupta, "Energy Efficient Frequent Value Data Cache Design", In Proc. of the 35th Annual Int. Sym. on Microarchitecture, 2002
12. J. Yang, Y. Zhang and R. Gupta, "Frequent Value Compression in Data Caches". In Proceedings of the 33th Annual International Symposium on Microarchitecture, 2000
13. Y. Zhang, J. Yang and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design". In Proc. of the 9th Annual Int. Conference on ASPLOS, 2000

Topic 9

Distributed Algorithms

Jayadev Misra, Wolfgang Reisig, Michael Schoettner, and Laurent Lefevre
Topic Chairs

The wide acceptance of the internet standards and technologies, the emerging Grid structures makes it hard to imagine a situation in which it would be easier to argue about the importance of distributed algorithms than it is today. Distributed algorithms cover a wide area of topics including but not limited to:

- design and analysis of distributed algorithms
- resource sharing in distributed systems
- real-time distributed algorithms and systems
- distributed algorithms in telecommunications
- distributed mobile computing
- cryptography, security and privacy in distributed systems
- distributed fault-tolerance
- efficient algorithms based on shared memory and distributed memory

Although a lot of significant research has been done in the past, there continues to be a strong need for design and analysis of algorithms especially for efficiency, reliability, security, and of course for new requirements of emerging architectures.

Out of 11 submitted papers to this topic, 5 have been accepted as regular papers (45,5%), and 2 as short papers.

Credits: We would like to take the opportunity of thanking all contributing authors as well as all reviewers for their work. We owe special thanks to the additional reviewers beside the topic chairs: S. Frenz, M. Wende, R. Goeckelmann, S. Dolev, T. A. Nolte, B. Sanders, H. Attiya, E. T. Pierce, T. Herman, S. Lakshminarasimhan, Cong-Duc Pham, and Jean-Patrick Gelas.

Multiresolution Watershed Segmentation on a Beowulf Network

Syarraieni Ishar¹ and Michel Bister²

¹Faculty of IT, Multimedia University, 63100 Cyberjaya, Malaysia,
syarraieni.ishar@mmu.edu.my

²Faculty of Engineering, Multimedia University, 63100 Cyberjaya, Malaysia,
mrbister@ieee.org

Abstract. Among the many existing multiresolution algorithms, the scale-space approach offers the benefits of strong mathematical and biological foundation and excellent results, but the serious drawback of a heavy computational load. Parallel implementation of this category of algorithms has never been attempted. This article presents a first experiment, using the multiresolution watershed segmentation as algorithm, and an 8-node Beowulf network as hardware platform. First, the classical approach is followed whereby the image is divided in several regions that are separately allocated to different nodes. Each node performs all the calculations for his region, at any level of resolution. Next, a truly multiresolution approach is followed, allocating the workload to the processors per resolution levels. Each node is allocated a number of resolution levels in the scale space, and performs the calculations over the whole image for the particular resolution levels assigned to it. The implementation in the latter approach is clearly much more straightforward, and its performance is also clearly superior. Although the experiments using the region-wise assignment were only done by splitting up the image in rows, and not in columns or in quadrants, the difference in the results is so dramatic that the conclusions can easily be generalized, pointing to the fact that scale space algorithms should be parallellised per resolution level and not per image region.

1 Introduction

1.1 Scale Space

Over the past few years, multiresolution algorithms have become a dominant approach in digital image processing, and this for two good reasons: their excellent results, and their biological motivation. It has indeed been shown that the human visual system is largely a multiresolution system. The scale space approach is an alternative to other several approaches that have been suggested: Laplacian pyramids, filterbanks, wavelets, etc [1].

The scale space approach first develops its theory in the continuous domain, and adds a scale axis to the spatial axes of the image. For practical applications, the scale axis has to be sampled, just as the spatial axis is sampled. The whole development of the theory is aimed at studying the evolution of the image features along scale. This is termed the "deep structure".

Since an N -dimensional image is extended over an additional dimension (scale), it is obvious that this approach is both memory- and computational intensive - hence the greater popularity of sub-sampled alternatives like wavelets. However, the scale space has much stronger mathematical motivations, as it can theoretically be deduced from basic assumptions in many different ways, including causality, dimension analysis, entropy maximization, etc. Also, the correlation between the findings from the scale space approach and the human front end vision (the uncommitted part of the human visual system) is striking. Finally, results are impressive. Most of the "classical" digital image processing techniques (e.g. deblurring, texture analysis, gradient analysis, optical flow, stereo analysis) have all been implemented using scale space techniques [2, 4, 5, 8].

Taking into account the heavy demands of any scale space algorithm, it is obvious that the scale space community would benefit greatly from parallel implementations. Hence, it is surprising that no parallel implementation of the scale space approach has been attempted to date. This paper is a first attempt at filling this void. Taking into account the vast amount of different scale space algorithms, it was necessary to limit the selection.

1.2 Watershed Segmentation

The basic principle of watershed segmentation is to let the pixels in the image link to one another, whereby each pixel links to its neighbor with the least gradient magnitude - hence away from the strongest gradient. This linking scheme defines a classification, whereby all the pixels linking mutually define one class, or segment. In practice, the links are propagated from one pixel to another: if pixel A links to pixel B, and pixel B links to pixel C, then pixel A is linked directly to pixel C. This is iterated until there are no more changes. Pixels who link to themselves (e.g. pixel X links to pixel Y, and pixel Y links to pixel X, so after one iteration pixel X links to itself) are called roots. Roots are numbered, and all pixels are labeled with the number of the root to which they link.

This algorithm is illustrated in Figure 1. It is called watershed segmentation for the following reason. If we consider the gradient image as a landscape, with the magnitude of the gradient as a measure of the elevation of the point, then the algorithm simulates the flow of rain and the splitting up of the landscape in watersheds. Several implementations exist, not all following the simple and intuitive procedure outlined here.

The main problem with watershed segmentation is its inherent over-segmentation: the smallest local minimum in the gradient produces a new root, hence a new segment. Many methods have been proposed in the literature to deal with this problem, including the low-pass filtering of the image to reduce the noise and hence the number of local minima in the gradient image. The question is then: how much filtering should be applied? The scale space approach gives an elegant answer to this question by relating the results obtained at different levels of resolution.

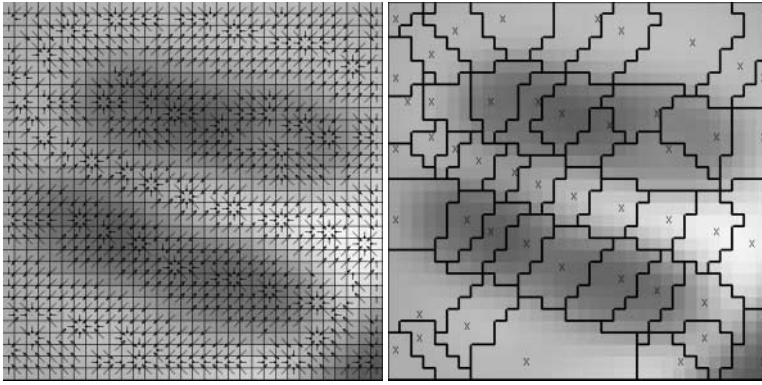


Fig. 1. Watershed algorithm of a bacteria image. Left: linking scheme (pixels link toward smallest gradient). Right: roots are marked by the letter 'x', and dark lines show the border of the segments.

1.3 Scale Space Watershed

Let us imagine an image blurred with two different levels of blurring, σ_1 and σ_2 . Each image is segmented independently using the watershed algorithm. The segmentation using σ_1 is blurred just enough to have sufficiently smooth borders but too many segments, while the segmentation using σ_2 is too blurred for the borders (the borders have shifted) but just enough to avoid over-segmentation. The purpose would be to have the segments as defined in the σ_2 image but with the borders as determined in the σ_1 image. The problem is to trace the borders of the segments from the higher level in the scale space (associated with σ_2) to a lower level (associated with σ_1). As tracing the borders of a segment corresponds to tracing the segment, the maximum overlap algorithm, as illustrated in Figure 2, is used as the solution.

The segments at level σ_1 are named S_i , $0 \leq i < N_{s1}$, and the segments at level σ_2 are named T_j , $0 \leq j < N_{s2}$. Then, the amount of overlap $V(S_i, T_j)$ is equal to the number of pixels that belong both to segment S_i at level σ_1 and to segment T_j at level σ_2 . Segment S_i is linked to segment T_j if and only if $V(S_i, T_j) > V(S_i, T_k) \forall k \neq j$. Finally, all segments at level σ_1 that link to the same segment at level σ_2 are merged, resulting in a number of segments equal to the N_{s2} but with segment borders defined at level σ_1 .

This procedure can be propagated throughout the scale space, as in a loop, so as to have segments defined at any level of scale with borders defined at another level of scale. This is the algorithm that we want to implement here. Hence, several steps have to be considered: blurring, gradient calculation, watershed segmentation, and merging. Actually, we want to leave the merging as a parameter to the user, so we want to generate the structure of segment linking from one level to another.

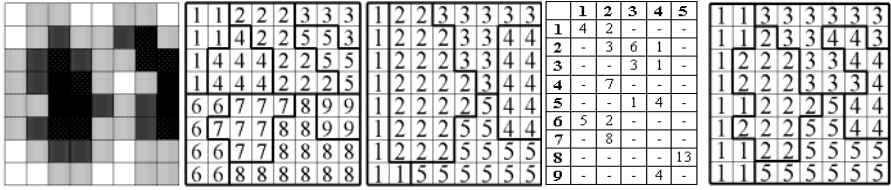


Fig. 2. Scale space merging algorithm. From left to right: gray scale input image; result of segmentation on level σ_1 – segment borders accurate, but over-segmentation; result of segmentation on level σ_2 – reduced over-segmentation, but segment borders inaccurate due to blurring; overlap table between the two segmentations; resulting of assigning to each segment at level σ_1 the value of the segment at level σ_2 with which it has the highest overlap.

1.4 Beowulf Networks

The Beowulf system provides a low-cost high-performance parallel computing environment. This system is ideal for loosely coupled parallel processing which involves a set of independent processors, each with its local memory, typically with their own copy of the OS, working in parallel to solve a problem. The system consists of one server node interconnected with clients via a high-speed network, allowing each processor to work on its own data [9].

Beowulf cluster is used to run the SPMD (Single Program Multiple Data) algorithm using either of the two following popular libraries: Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) [10]. MPI is a popular message passing interface used in numerous homogeneous hardware environments ranging from distributed memory parallel machines to networks of workstations [3, 7, 10].

2 Implementation

In our parallel implementation of the scale space watershed algorithm, we use a Beowulf cluster consisting of 4 Dual Processors Intel Pentium III 800MHz workstations (1 master and 7 slaves) with 512Mb, 20.4Gb, running RedHat Linux 7.0 operating systems. These workstations are interconnected via a Myrinet switch.

2.1 Region-Wise Implementation

There are several parallel processing approaches for image data such as segment processing, frame processing, column-wise processing, row-wise processing, etc. For solving the parallel watershed problem, the row-wise approach was first implemented. Distribution of the working area over the processors or slaves is based on the number of rows in the input image. Each slave works on a sub-image for the watershed segmentation for all scale levels.

Determining of links only requires the sharing of one row of pixels along the border of the sub-region. Hence this poses no problems. Likewise, the numbering of

the roots can be done easily, with just a little precaution to make sure that each sub-region gets unique root numbers.

The real problem comes from the link following part of the algorithm. Several possible situations are illustrated in Figure 3, where an 8x6 image was divided among 3 slaves. A first linking list is without problem: $\text{link}(A) = B$, $\text{link}(B) = C$, so the instruction $\text{link}(A) = \text{link}(\text{link}(A))$ results in $\text{link}(A) = C$, after which iterations stop because $\text{link}(\text{link}(A)) = \text{link}(A)$. All of this can be done within the region assigned to slave 1. The second case (pixels D-E-F-G) illustrates the need for message passing between processors: but F is outside the region assigned to slave 1, so slave 1 needs information from slave 2 to continue following the list and put $\text{link}(D)$ to G. One could consider letting each slave first iterate until no further changes *within its assigned region*, and then exchange all the link information from the border pixels. In this case, slave 1 would already know that $\text{link}(D) = F$ and slave 2 know that $\text{link}(F) = G$, so passing this information to slave 1 would allow this one to put $\text{link}(D)$ equal to G.

Further problems are illustrated in the third case: repeated crossing of the border line (from H to Q...) or linking between regions that are not even adjacent (from R to W...), which illustrate that 1) just linking within each region and then sharing border information is not enough; 2) sharing information once between adjacent regions is not enough - repeated sharing OR sharing between all regions is necessary.

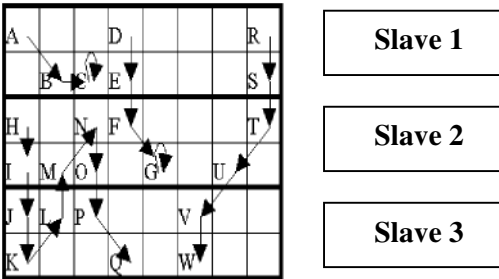


Fig. 3. Possible situation encountered when following linked lists of pixels between regions assigned to different slaves (see text for explanation).

Since the linking algorithm requires updated information of other pixels that are scattered throughout the image, and not only in the neighboring sub-images, many message-passing activities would occur. Every slave would communicate to each other slave to gather other sub-images information. Many synchronization points would be added for this implementation. To simplify the implementation, instead of many individual message passing steps, one single centralised message passing step is implemented: after slaves have completed the watershed segmentation for every level, they will send their result to master. Once gathered, master will display the findings.

2.2 Scale-Wise Implementation

In a second implementation, the task is distributed among the processors based on the scale levels. The master will first calculate the number of scale levels based on the

following formula and distribute the range of scale levels to all slaves as in Figure 4 [4]:

$$\text{Scale level } (\sigma) = \log(\text{Number of extrema in input image})/2 * \text{rate}$$

Each slave will then perform a sequential watershed algorithm. Once the watershed segmentation is completed for all levels assigned to them, the slaves send their results to the master. A few synchronization points were added to the code to ensure that the master receives the correct segmentation level. The master will then display the watershed segmentations.

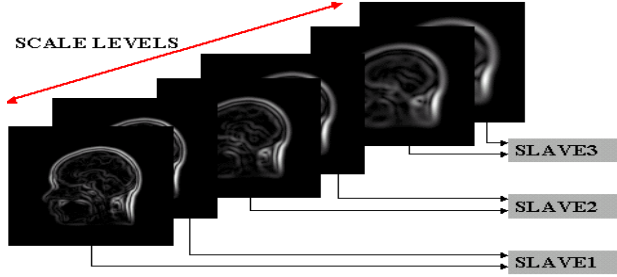


Fig. 4. Parallel implementation of the scale space watershed algorithm per level.

3 Results

The two implementations were applied on many different popular test images in digital image processing, but due to the space constraints, only the results for 8 of them are reported here: 4 with resolution 512x512 (Airplane, Boat, Crowd, Peppers) and 4 with resolution 256x256 (Camera, Couple, House, Orca). Thumbnails of these images are given in Figure 5. For each of these images, the experiments were repeated 4 times for more reliable results.

The average execution time for each of these images and for all three implementations is recorded: serial (for reference), per row, and per level. From this, the average speedup factor was determined using the formula: $\text{Speedup } S(n) = \text{Execution Time (1 processor)} / \text{Execution Time}(n \text{ processors}) = t_s/t_p$ [10]. All the results are shown in Figure 6.



Fig. 5. Thumbnail of 8 test images used in the experiments.

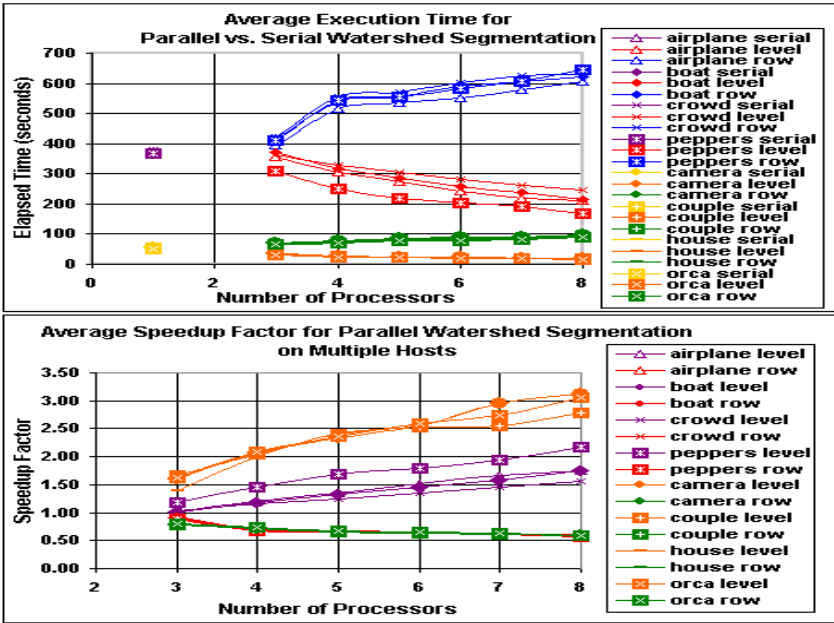


Fig. 6. Results for each test image in function of number of processors: a) average execution time; b) average speedup factor.

4 Discussion

From the results it is clear that the implementation by scale levels is much superior to the implementation by rows. The implementation by rows systematically slows down the process. This is due to the intensive message-passing during the linking phase of the watershed algorithm. As was shown previously, at this stage, each processor needs a lot of interaction with all the other processors. This creates a communication bottleneck, which only increases with increased number of processors.

On the other hand, the implementation by scale levels systematically speeds up the process. This is the logical choice, since the watershed is typically performed independently at each level of the scale space, minimized message passing. As a result, we have a speedup for the implementation by scale levels, but an actual speed-down in the implementation by rows.

The performance is strongly dependent on image size and image structure. For small images, obviously the execution time is shorter, but also the speedup factor for the implementation by level is higher. For the 512x512 images, the results vary more, with the lowest to the highest speedup, (speedup of 1.56 (Crowd) to 2.18 (Peppers) with 8 processors). Regardless the image size, the highest speedup is the Orca (factor 3.06). It is tempting to explain this by the lower complexity of the Orca image (large homogenous regions) and the high complexity of Airplane and Crowd images.

5 Conclusion

The large number of tests performed and the consistency of the results are enough to show that the parallelisation of the scale space watershed algorithm should be done by level and not by row. The reason for the failure of implementation by rows is the intense message-passing required during the linking phase of the watershed algorithm. Any other region-wise implementation (by quadrant, by column, etc) would face the same problem.

On the other hand, the implementation by resolution levels is coarsely grained because the processing is done independently per resolution level. As the number of processors increases, fewer and fewer levels are assigned per processor. The speedup curve levels off as one approaches one level per processor - beyond this limit, it is clear that no additional speedup could be expected any more. Particularly for the smaller images, with a smaller number of levels, the leveling of the speedup curve is clearly visible.

Since many deep structure algorithms work independently on individual levels or on pairs of levels, similar performance would be expected, and the logical choice would definitely be the implementation of scale space algorithms by levels of resolution.

References

1. Bister M, Cornelis J, Rosenfeld A, A critical view on pyramid segmentation algorithms. *Pattern Recognition Letters*, Vol. 11, pp. 605–617, 1990.
2. Florack, L.M.J. *Image Structure. Computational Imaging and Vision Series*. Kluwer Academic Publishers, Dordrecht. (1997)
3. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: *MPI- The Complete Reference: Volume 2, The MPI Extensions*. The MIT Press, Cambridge London. (1998)
4. Romeny, B.M. ter Haar (ed): *Front-End Vision and Multiscale Image Analysis*. Kluwer Academic Publishers, Dordrecht. (2003)
5. Romeny, B.M. ter Haar (ed): *Geometry-Driven Diffussion in Computer Vision*. Kluwer Academic Publishers, Dordrecht. (1994)
6. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: *MPI – The Complete Reference: Volume 1, The Core*. 2nd edn. The MIT Press, Cambridge London. (1998)
7. Sporing, J., Nielsen, M., Florack, L., Johansen, P. (eds): *Gaussian Scale-Space*. Kluwer Academic Publishers, Dordrecht. (1996)
8. Sterling, T.L., Salmon, J., Becker, D.J., Savarese, D.F.: *How to Build a Beowulf Cluster: A guide to the Implementation and Application of PC Clusters*. The MIT Press, Cambridge London. (1998)
9. Wilkinson, B., Allen, M.: *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice Hall, New Jersey. (1999)

*i*RBP – A Fault Tolerant Total Order Broadcast for Large Scale Systems

Luiz Angelo Barchet-Estefanel*

Laboratoire ID – IMAG
51, Avenue Jean Kuntzmann
38330 Montbonnot St. Martin, France
Luiz-Angelo.Estefanel@imag.fr

Abstract. Fault tolerance is a key aspect on the development of distributed systems, but it is barely supported on large-scale systems due to the cost of traditional techniques. This paper revisits RBP, a Total Order Broadcast protocol known by its efficiency that presents some very interesting characteristics for scalable systems. However, we found a membership flaw on RBP that can lead to inconsistencies among correct processes. Hence, we propose *i*RBP, an improvement to the RBP algorithm that not only circumvents the membership weaknesses using recent membership techniques, but also improves its scalability aspects.

1 Introduction

Fault tolerance is a key aspect on the development of distributed systems, but it is barely supported on large-scale systems like GRIDs due to the cost of traditional techniques. A good alternative would be RBP (Reliable Broadcast Protocol), a well-known protocol for Total Order Broadcast [3,5]. Its structure around a logical ring allows the protocol to efficiently reduce the number of exchanged messages [11], which is very interesting in large-scale systems.

However, we observed that RBP is still vulnerable to some failure patterns. Correct processes may leave and rejoin the group due to incorrect suspicions, but nothing ensures that such processes will be kept consistent with the rest of the group members. This problem, known as *time-bounded buffering problem*, is not easy to solve (or has a high cost).

We propose *i*RBP [1], an improvement to the RBP protocol that integrates new membership and failure detection techniques to solve the *time-bounded buffering problem*. We also tried to improve its performance when the number of processes grows up. Based on the membership techniques implemented on *i*RBP, we propose a simple but efficient strategy to bound the latency levels of the protocol, an important scalability concern.

Our observations candidate *i*RBP as a good and efficient alternative to Total Order Broadcast protocols like Atomic Broadcast based on Consensus [2], that have scalability limitations due to the number of messages they exchange.

* Supported by grant BEX 1364/00-6 from CAPES - Brazil

This paper is organized as follows. Section 2 defines the system model and properties considered through this paper. Section 3 presents RBP and how its membership flaws may lead to system inconsistency. Section 4 presents the techniques we used to solve the *time-bounded buffering problem*, and how they were integrated with iRBP internal structure. Section 5 analyses the iRBP performance through the number of exchanged messages and the delivery latency. Finally, Section 6 concludes the paper.

2 System Model and Definitions

We consider an asynchronous distributed system where a set of processes $\pi(t) = \{p1;p2;...;pn\}$ interacts by exchanging point-to-point messages through unreliable (*fair-lossy*¹) communication channels. For simplicity, **multicast primitives (including broadcast)** are constructed by sending **several point-to-point messages**.

Processes are only subjected to crash failures without recovery. A process that does not crash during all the execution is called “correct”. *Eventually strong* ($\Diamond S$) failure detectors, as defined by Chandra and Toueg [2], are used to suspect failed processes.

The Total Order Broadcast (Multicast) problem may be informally explained as how to ensure that remote processes deliver messages in the same order, while keeping a system consistent even if there are process failures. It is defined through four properties [6]:

- **VALIDITY** - If a correct process broadcasts a message m to $Dest(m)$, then some correct process in $Dest(m)$ eventually delivers m .
- **AGREEMENT** - If a correct process in $Dest(m)$ delivers a message m , then all correct processes in $Dest(m)$ eventually deliver m .
- **INTEGRITY** - For any message m , every correct process p delivers m at most once, and only if (1) m was previously broadcast by $sender(m)$, and (2) p is a process in $Dest(m)$.
- **TOTAL ORDER** - If correct processes p and q both deliver messages m and m' , then p delivers m before m' if and only if q delivers m before m' .

3 Revisiting RBP

RBP (Reliable Broadcast Protocol) is a well-known Total Order Broadcast [3,5]. It uses the moving sequencer strategy [6] to agree with the Total Order property: only one single process can define the delivery order, while the role of sequencer moves around the processes. It can be easily implemented using token passing: the process that holds the token is the only one that can assign sequence numbers to the messages. Once it has sequenced a message (or a group of messages, if

¹ **Fair-lossy channel** – if p send a message m to q an infinite number of times and q is correct, then q receives m from p an infinite number of times.

there are many in the buffer [11]), the token is passed to the next process in the ring using unreliable communication primitives.

When a process receives the token (i.e., it was indicated to be the next sequencer), it *accepts* the token if and only if it has all data messages previously sequenced. If the process has already all previous messages, it simply assigns a sequence number to a new message (this message acts as an acknowledgement to the previous sequencer). However, if a process detects that it has lost some messages, it uses negative acknowledgements to explicitly request retransmission. In fact, if channels “behave well”, i.e., they do not lose many messages, the protocol can efficiently deliver messages with a minimum of message exchange [11].

However, the use of fair-lossy channels has a drawback: processes are not allowed to deliver messages immediately because a simple crash may violate the *Agreement* property. Instead, messages should be delivered only after the token was passed among an arbitrary number of processes, what is called k -resiliency. Actually, RBP proposes the use of n -resiliency, where the token must pass among all processes (sometimes called “total resiliency”).

3.1 Fault Tolerance, Membership, and RBP Limitations

Fault tolerance is essential on RBP, because a single failure may prevent the token passing, blocking the protocol. Thus, at the occurrence of a simple failure, the token list should be reconstructed, removing suspect processes, in a procedure called “Reformation Phase” [3].

During the Reformation Phase, each process that detects a failure (which is called “originator”) contacts the other processes to start a new token list. The originator proposes a new token list composed by all processes that acknowledged its call. A three-phase commit (3PC) protocol coordinated by the originator [8] conducts the commitment of the new token list, which should be composed by a majority of processes.

However, as detection is not simultaneous to all processes, multiple processes can invoke the reformation concurrently, and a correct non-suspected process may be excluded from the token list just because it was the originator of a concurrent token list. In [3] it is assumed that such excluded processes may rejoin the token list in a later Reformation, and eventually the token list will contain all correct processes. We observed that this assumption is not enough to guarantee the *Agreement* property, leading to inconsistencies among the processes. In fact, such assumption induces the *time-bounded buffering problem* [4], where a process p cannot safely remove the message m unless if it knows that all processes in $Dest(m)$ have either received m or crashed.

4 *i*RBP, an *improved* RBP Protocol

4.1 Solving the *Time-Bounded Buffering Problem*

The *time-bounded buffering problem* represents a serious flaw on RBP. Hopefully, this problem was already studied in the context of the Primary-Backup replica-

tion model with View Synchronous Communication (VSC) [5]. Two techniques to solve the *time-bounded buffering problem* revealed to be highly adapted to the RBP structure. These techniques, presented below, were integrated on the protocol that we called *iRBP*, for *improved* RBP.

Program-Controlled Crash. In [4] it was claimed that no implementation of Reliable Broadcast (a problem weaker than Total Order Broadcast, see [7]) over fair-lossy links could solve the *time-bounded buffering problem* unless it uses a *Perfect* failure detector (*P*) [2]. As *P* failure detectors are impossible to be implemented in asynchronous systems, a practical solution is to use *program-controlled crash*. Program-controlled crash gives the ability to kill other processes or to suicide, ensuring time-bounded buffering (if all suspect processes will “crash”, then message *m* can be safely discarded from the buffers).

The use of program-controlled crash together with membership control is not a new technique (it was already employed on the ISIS system [5]), but it is not very popular due to its non-negligible cost. Every time a process *q* is forced to crash due to an incorrect suspicion, a membership change is executed to update the view. Hence, is common to rely on failure detectors with conservative timeout values, which can block the system for a long period.

Two views. In typical group membership architectures, solving efficiently both problems of time-bounded buffering and blocking prevention at the same time is not possible, because they are linked to the same failure detection scheme. In order to better explore both issues, Charron-Bost [4] proposed the use of two levels of Group Membership Service (GMS).

There, *membership views* (or simply *views*) are identical to the views of View Synchronous Communication, while *ranking views* (or *rk-views*) are installed between membership views. If membership views are denoted by $v_0, v_1, \dots$, the *rk-views* between v_i and v_{i+1} are denoted as $v_i^0, v_i^1, \dots, v_i^{last}$. The membership of all ranking views $v_i^0, \dots, v_i^{last-1}$ is the same as the membership of v_i , differing only in the order that processes are listed in the view. For example, $v_i = v_i^0 = \{p, q, r\}$, $v_i^1 = \{q, r, p\}$, etc. As *rk-views* are composed by the same set of processes, they do not require program-controlled crash.

This model in two layers allows us to solve the dilemma between fail over time and program-controlled crash. Membership views are generated by suspicions resulting from conservative timeouts, while *rk-views* are generated by suspicions resulting from aggressive timeouts. This way, *rk-views* contribute to avoid blocking situations, while membership views ensure time-bounded buffering of messages.

4.2 iRBP Membership

To solve the membership problems from RBP, we constructed a Two-Level View-Synchronous membership algorithm to replace the *Reformation Phase*. We did

not rely on the model suggested by Charron-Bost [4] to represent the views and ranking views (“move the suspect coordinator to the end of the list”) because it was not suitable for our ring-based protocol. Instead, we suggest a different approach to implement *rk*-views, where a view is composed by two subsets of processes:

$$\{(\text{"core group"}),(\text{"external group"})\}$$

Broadcasts are sent to the whole group, but the token is passed only among the “*core group*”. Once a process is suspected, an *rk*-view change moves it from the *core group* to a contiguous set, called “*external group*”. As a suspected process does not participate in the sequencing process (it is not in the token ring), it does not block the token passing. As it still belongs to the view, it receives all broadcasts, can send messages to the group and request retransmissions.

While algorithm for Membership View Change is only a modified version of the traditional View Synchronous Membership that includes *program-controlled crash* (Algorithm 1), the *rk*-view change was optimized, as an *rk*-view change does not need to exchange information about delivered messages (processes just need to know who is active), and no *program-controlled crash* is required. Consequently, Algorithm 2 presents an optimized version of the View Change algorithm, adapted to *rk*-view changes.

Algorithm 1 Membership View Change with *program-controlled crash*

Upon suspicion of some process in TL_i by a conservative Failure Detector

RBroadcast (reformation, i)

Upon R-Deliver (reformation, i) by p_k for the first time

1. send $seqQ_k$ to all /* sends the “unstable” list of messages */
 2. $\forall p_i$, wait until receive $seqQ_i$ from p_i or p_i suspected
 3. let $initial_k$ be the tuple $(TL, Msgs_k)$ s.t.
 - TL means $\{(core_k), ()\}$, and $core_k$ contains all processes that sent their $seqQ$
 - $Msgs_k$ is the union of the $seqQ$ sets received
 4. execute Consensus among TL_i processes, with $initial_k$ as the initial value
 5. let (TL, Msg) be the consensus decision
 6. $stableQ \leftarrow Msg$, $seqQ \leftarrow \{\}$ /* the set Msg becomes stable */
 7. if $p_k \in TL$, then “install” TL as the next view TL_{i+1}
 else suicide
-

4.3 Resiliency Levels

As commented on Section 3, RBP [3] suggests the use of total resiliency, which guarantees the maximum level of fault tolerance. However, total resiliency induces a high latency, representing a time cost that should be reduced whenever it is possible. On *i*RBP, the resiliency level on depends on the number of processes in the *core group*, and consequently, to control the number of core processes is a simple technique to efficiently bounds the latency of the protocol.

Algorithm 2 Optimised rk -view change

Upon suspicion of some process in TL_i^j by an aggressive Failure Detector
 RBroadcast (rk-view, j)
 Upon R-Deliver (rk-view, j) by p_k for the first time

1. send ack to all
2. $\forall p_i$, wait until receive ack from p_i or p_i suspected
3. let $initial_k$ contain $\{(core_k), (external_k)\}$ s.t.
 - $core_k$ is the list of all processes that sent ack
 - $external_k$ is the list of all processes that does not sent ack
4. execute Consensus among TL_i^j processes, with $initial_k$ as the initial value
5. let (TL) be the consensus decision
6. if $p_k \in TL$, then "install" TL as the next rk -view TL_i^{j+1}

In the next Section we evaluate how different resiliency levels influence the *delivery latency* from iRBP.

5 Performance Analysis

5.1 Number of Messages

Due to the ability of RBP to piggyback acknowledgements and token passing in a single message, iRBP uses very efficiently the network resources. If no message is lost, iRBP needs only two broadcasts: the first one, sent by the message source, submits the message; the second one comes from the sequencer, which assigns a sequence number to that messages, at the same time that passes the token to the next sequencer. Thus, if no messages are losts, the protocol requires only $2(n-1)$ messages (assuming that broadcasts transmit $n-1$ point-to-point messages) for each message to be delivered.

Comparatively, the Atomic Broadcast based on the Consensus [2] may exchanges up to $(4+2n)(n-1)$ messages in a single execution step. There are other consensus algorithms than Chandra and Toueg Consensus, as for example the Early Consensus [9], but they mainly focus on the reduction of communication steps, while the number of exchanged messages remains high (still generating $O(n^2)$ messages, against $O(n)$ from iRBP).

5.2 Latency, Core-Resiliency, and Scalability

One important element on iRBP performance analysis is its *latency*. We consider latency as the time elapsed between the broadcast of a message m by the source process and the first time it is delivered to the application, i.e., the *delivery latency*. On iRBP, latency depends on the resiliency level, because a message is only delivered after this resiliency level is achieved. If the number of processes grows up, the latency may reach undesirable levels.

To prevent such effects, we use the own structure of the Two-levels membership from iRBP to bound the latency. By controlling the number of processes in

the *core* group, it is possible to limit the latency levels even if the system scales up.

We implemented *i*RBP in Java using the Neko framework [10]. The experiments were conducted on the **ID/HP i-cluster** from the ID laboratory Cluster Computing Center², where machines are interconnected by a switched Ethernet 100 Mbps network. The tests were executed with 4, 8, 16, 32 and 64 machines (one process/machine), and we compared RBP's performance against *i*RBP with different *core*-resiliency levels. Figure 1 presents our results when messages arrive according to a Poisson process of rate 10 messages per second.

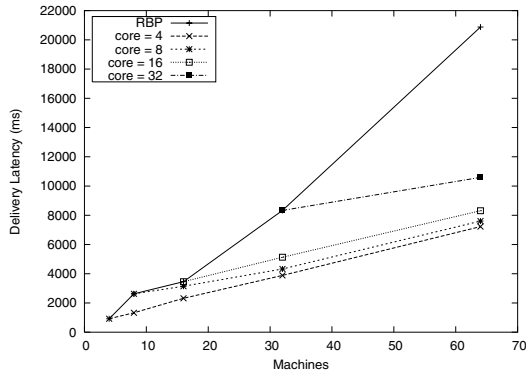


Fig. 1. Comparison between RBP and *i*RBP delivery latencies

We can observe that in the RBP model, the latency increases almost linearly with the number of processes in the token list. By fixing the number of elements in the core group, we obtain latency levels smaller than those from RBP. This experiment shows that bounded delivery latency is possible with the use of simple techniques like the *core*-resiliency.

The choice of a small core-group allows the protocol to deliver messages with latency levels similar to conventional techniques (we should not forget that the Atomic Broadcast based on the Consensus requires from 6 to 8 communication steps), while the number of exchanged messages is far reduced. In addition, future implementations may take advantage of network low-level primitives, like Ethernet-Broadcast or IP-Multicast, optimizing all *i*RBP communications, which is not the case with the Atomic Broadcast based on the Consensus.

6 Conclusions

In the first part of this paper, we demonstrate that RBP, a well-known Total Order Broadcast protocol, allows some situations that can violate the consistency

² <http://www-id.imag.fr/grappes.html>

of processes. It is an interesting observation, because there are several works that explore RBP efficiency, but to our knowledge, none has examined the consistency problem as we did.

We looked for feasible solutions that could be applied to our “improved RBP” protocol, that we called *iRBP*. Specially, we identified the origins of the inconsistencies as the *time-bounded buffering problem*, and we focused on solving efficiently this problem. We replaced the original RBP membership control, allowing our protocol to deal with the *time-bounded buffering problem* while reducing the cost involved in the use of *program-controlled crash*.

In the second part of this paper, we focused on the performance of *iRBP*. The results from our practical experiments demonstrate that *iRBP* is a good candidate to large-scale distributed computing, like GRID environments, where fault tolerance is a major concern. *iRBP* is an efficient protocol for Total Order Broadcast, and can be used as a building block to provide fault-tolerant support on large scale systems.

References

1. Barchet-Estefanel, L. A.: Analysing RBP, a Total Order Broadcast Protocol for Unreliable Networks. Technical Report, IC-EPFL – Switzerland. (2002)
2. Chandra, T. D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2). ACM Press New York, NY, USA (1996) 225–267
3. Chang, J., Maxemchuck, N.: Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3). ACM Press New York, NY, USA (1984) 251–273
4. Charron-Bost, B., Défago, X., Schiper, A.: Broadcasting Messages in Fault-Tolerant Distributed Systems: the benefit of handling input-triggered and output-triggered suspicions differently. *Proceedings of the 21st Int’l Symposium on Reliable Distributed Systems*, Osaka, Japan. (2002)
5. Chockler, G., Keidar, I., Vitenberg, R.: Group Communication Specifications: a comprehensive study. *ACM Computing Surveys*, 33(4). ACM Press New York, NY, USA (2001) 427–469
6. Défago, X.: Agreement-related Problems: from semi-passive replication to totally ordered broadcasts. PhD Thesis, EPFL – Switzerland. (2000)
7. Jalote, P.: *Fault Tolerance in Distributed Systems*. Prentice-Hall (1994)
8. Maxemchuck, N., Shur, D.: An Internet Multicast System for the Stock Market. *ACM Trans. on Computer Systems*, 19(3). ACM Press New York, NY, USA (2001) 384–412
9. Schiper, A.: Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3) Springer-Verlag, Berlin Heidelberg New York (1997) 149–157
10. Urbán, P., Défago, X., Schiper, A.: Neko: A single environment to simulate and prototype distributed algorithms. *Proceedings of the 15th Int’l Conf. on Information Networking*, Beppu City, Japan (2001)
11. Whetten, B., Montgomery, T., Kaplan, S.: A High Performance Totally Ordered Multicast Protocol. In: Birman, K. P., Mattern, F., Schiper, A. (eds.): *Theory and Practice in Distributed Systems: International Workshop*. Springer-Verlag, Berlin Heidelberg New York (1995) 33–57

Computational Models for Web- and Grid-Based Computation^{*}

Joaquim Gabarró¹, Alan Stewart², Maurice Clint², Eamonn Boyle², and
Isabel Vallejo¹

¹ Dep. LSI, Universitat Politècnica de Catalunya, Jordi Girona, 1-3,
Barcelona 08034, Spain.

² School of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN,
Northern Ireland.

Abstract. Two models of web- and grid-based computing are proposed. Both models are underpinned by the notion of computing with “approximate” data values. The models utilise an asynchronous form of communication which blocks neither sender nor receiver. The idea of computing with approximate data is illustrated using a number of examples including a client server web computation and a partial eigensystem co-algorithm. The results of some preliminary experiments showing the feasibility of the approach for computing partial eigensolutions are presented.

Keywords. CPOs, approximations, Web and Grid computation, UNITY, co-algorithms.

1 Introduction

The areas of web and grid computation have been driven largely by technological advances – however, such computations have not been well supported by appropriate abstractions. In practice this means that design is often realised at a low-level. Notable exceptions are mobility theories (see, for example, the π -calculus or ambient calculus [1]) which provide a means of reasoning about dynamically generated links. In this paper a different aspect of Web- and Grid-based computing, namely, computing with partial information, is explored. The notion of processing partial information is intrinsic to web computing – for example, it is normal to display incomplete web pages. In a similar way, (multiple) iterative methods can be used to generate a series of partial approximations to a required fixed point. The convergence rates of individual methods may be improved by exchanging, at appropriate points in the computations, the set of available approximate solutions.

^{*} Work partially supported by: Spanish CICYT under grant TIC2002-04498-C05-03 (Tracer), IST program of the EU under contract IST-2001-33116 (Flags), Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and the European Social Fund.

Traditional forms of communication overwrite a memory location – the receiver has no knowledge of the nature of the incoming data. However, in specialised situations, communications may *improve* (rather than just overwrite) the value stored in a location. Grab and Go communication [7,8] is based on the notion of utilising the best available approximation immediately (rather than waiting for pending communications). Thus synchronization overheads can be avoided. In this approach approximations are viewed as elements of CPOs. A CPO [5] comprises a domain D of values, a least element, $\perp$, an ordering relation $\sqsubseteq$ and a least upper bound operator $\sqcup$.

Grab and Go communication is used to underpin two models for web- and grid-based computing. The models are described in UNITY [3] and can be viewed as a variants of the inflationary fixed point problem (common meeting time) [3].

A model of web computing and an integration example are given in §2 and §3. A co-algorithm approach to exploiting grid resources is described in §4. An example of a co-algorithm which computes partial eigensolutions is given in §5. Some preliminary results are reported in §6.

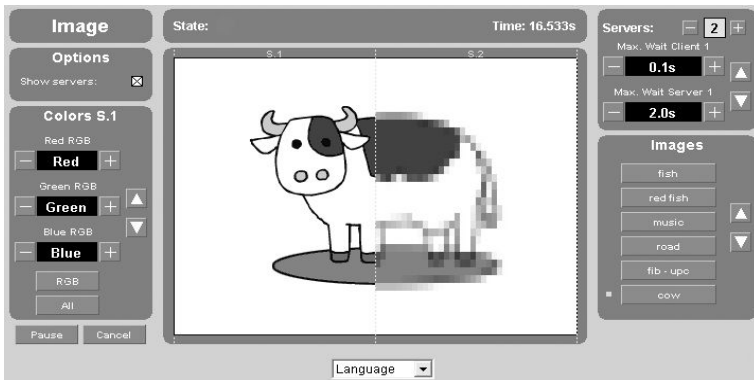


Fig. 1. A Java simulation of a web-based computation. Two servers send approximations of parts of a single image. The first server has transmitted completely the first half while the other server has transmitted only a crude approximation of the second half.

2 A Model of Web-Based Computation

Consider the situation in which a client computer displays and processes information stored on, or generated by, servers. A server may forward information to the client piecemeal. At a given instant the client may only have received partial information from the servers (see Fig 1). In the context of web computing it is desirable that this partial information should be displayed and perhaps processed by the client.

A simple web scenario can be modelled by the generation, by two servers – *server1* and *server2*, of two streams of input information and a computation, by a *client*, which processes both of these data streams. Let the servers

communicate information sets f and g , respectively. The sets f and g are constructed piecewise from two static sequences, ssf and ssg . In practice ssf and ssg may be representations of either (i) persistent data, decomposed into packets for transmission or (ii) data directly generated by computation – the latter possibility suggests the use of sequences to represent ssf and ssg . For example, a large image may be broken down into segments for transmission, or a live video image may become available in discrete ordered frames. This situation is dynamic and, at a particular instant, only *parts* of the complete server information may be available. Data available for transmission is inserted in either f or g . This nondeterministic situation can be described in UNITY by:

$$\begin{array}{ll} \{send\ f\} & ssf, f := tail(ssf), f \cup \{head(ssf)\} \quad \text{if } ssf \neq null \\ \parallel & \{send\ g\} \quad ssg, g := tail(ssg), g \cup \{head(ssg)\} \quad \text{if } ssg \neq null \end{array}$$

At some instant the client will input this information ($f \cup g$) through a port variable p and, perhaps, apply some function h to it. The best data available to the client can be generated by combining the new data $f \cup g$ with previously received data stored in p :

$$\begin{array}{ll} \{client\ receive\} & p, f, g := \sqcup (p, f \cup g), \{\}, \{\} \\ \parallel & \{client\ apply\} \quad z := h(p) \end{array}$$

For example, a better approximation to an image may be generated by adding new image fragments to a previously available partial image. In this situation a read operation by the client has the effect of *augmenting* the data stored in a variable rather than overwriting it. A client port variable is used to represent the best data approximation available. This variable is represented by an element of a CPO: no information is modelled by a least element, $\perp$, while the addition of new information is captured by applying the least upper bound operation to a pair comprising newly received information and previously transmitted data.

Let the client process compute

$$h\left(\sqcup (rng(ssf) \cup rng(ssg))\right) \quad (1)$$

– where rng returns the set of elements in its sequence argument. Suppose that i and j pieces of information are available from ssf and ssg , respectively. Then the client may compute an approximation to (1) using:

$$h\left(\sqcup \{ssf(1), \dots, ssf(i), ssg(1), \dots, ssg(j)\}\right)$$

It is necessary that h be monotonic to ensure that an increase in the quality of the input approximations improves the quality of the result generated by the client.

The entire client server system is:

Program *web_computation*
initially $f, g, p, z = \{\}, \{\}, \perp, \perp$

```

assign
  {send f}      ssf, f := tail(ssf), f ∪ {head(ssf)}   if ssf ≠ null
  || {send g}    ssg, g := tail(ssg), g ∪ {head(ssg)}   if ssg ≠ null
  || {client receive} p, f, g := ⊔ (p, f ∪ g), {}, {}
  || {client apply} z := h(p)
end {web_computation}

```

The system may make progress in three ways:

- a server may make available more information;
- the client may read from f and g ; or
- the client may improve its approximation to the required result (1).

The client-server system may be implemented using Grab and Go communication [7] whereby many messages (or none) may be read from an asynchronously modified port. Note that the port extraction statement

$$p, f, g := \sqcup (p, f \cup g), \{\}, \{\}$$

is unconditional. Thus, execution of this action need not be delayed while awaiting input – the port is processed immediately whatever its status (empty, having received one input, or having received many inputs).

A Java simulation [9] of web-based computation has been developed. The system [14] (see Figs 1, 2) can be executed from <http://how.to/grabandgo>. Partial orders are represented abstractly by an interface **Cpoable**. Ports retain only the best available information: port output, $p.write(e)$, is implemented as $p := p \sqcup e$ and port input, $p.read(x)$, is implemented as $x := p \sqcup x$. A port $p(D, \perp, \sqsubseteq, \sqcup)$ is parametrized by the data type, D .

3 Web Based Integration

Consider a system (see Fig 2) which computes and displays approximations to the integral $\int_a^c f(x)dx$ using the property

$$\int_a^c f(x)dx = \int_a^b f(x)dx + \int_b^c f(x)dx \text{ where } a < b < c$$

Let a and b be two real numbers with $a < b$, and let n be an integer where $n > 0$. Define $h = |b-a|/n$ and $x_i = a + ih$. Then $\int_a^b f(x) dx \approx \text{trapezoidal}(f, a, b, n)$ [2] where

$$\text{trapezoidal}(f, a, b, n) = \frac{h}{2} \left[f(a) + 2 \sum_{i=1}^{n-1} f(x_i) + f(b) \right]$$

Given that f is increasing and convex a sequence of approximations generated by the trapezoidal rule are ordered according to:

$$n \leq n' \Rightarrow \text{trapezoidal}(f, a, b, n) \sqsubseteq \text{trapezoidal}(f, a, b, n')$$

The least upper bound of two approximations is defined, for increasing and convex f , as:

$$\text{trapezoidal}(f, a, b, n) \sqcup \text{trapezoidal}(f, a, b, n') = \text{trapezoidal}(f, a, b, \max(n, n'))$$

A system *integral* which generates approximations to $\int_a^c f(x)dx$ is defined below:

Program *integral*

initially

$i, j, \text{best} = 0, 0, 0$

$\text{firstPart}, \text{secondPart} = \{\}, \{\}$

$\text{portFirstPart}, \text{portSecondPart} = \perp, \perp$

assign

$i, \text{firstPart} := i + 1, \text{firstPart} \cup \{\text{trapezoidal}(f, a, b, 2^i)\}$

$j, \text{secondPart} := j + 1, \text{secondPart} \cup \{\text{trapezoidal}(f, b, c, 2^j)\}$

$\text{portFirstPart}, \text{firstPart} := \sqcup (\text{portFirstPart}, \text{firstPart}), \{\}$

$\text{portSecondPart}, \text{secondPart} := \sqcup (\text{portSecondPart}, \text{secondPart}), \{\}$

$\text{best} := \text{portFirstPart} + \text{portSecondPart}$

end *{integral}*

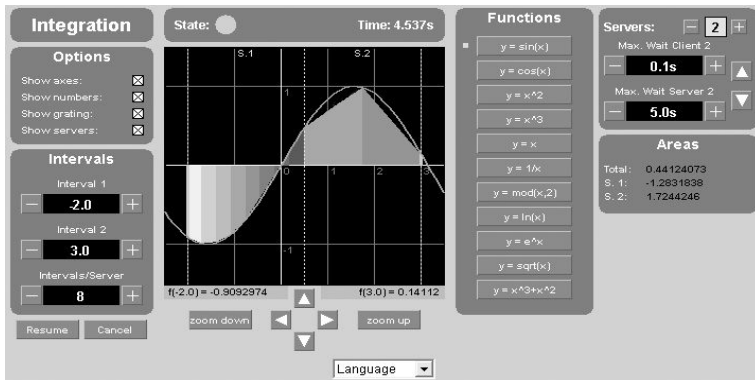


Fig. 2. Two servers compute approximations to disjoint parts of an integral.

4 Co-algorithms

It is widely recognised that the *grid computing paradigm* can give rise to radically different ways of organising computations. Algorithms which give poor performance on conventional architectures may be reorganised for efficient implementation on computational grids. The concept of grid computing makes traditional measures of parallel efficiency to some extent redundant. The view taken here is that computation need no longer be constrained to utilise a particular machine efficiently – rather it may exploit grid resources extravagantly in order to reduce computation time. Consider a situation in which a problem may be solved by either of two iterative algorithms repeatedly applying functions f and g . The

convergence characteristics of the algorithms may be data dependent. Thus it is not possible to say, in general, which algorithm has the better convergence behaviour. Further, one algorithm may converge at a uniform rate whereas the other may initially converge very slowly but, when a good approximate solution has been found, may converge rapidly. It may be profitable to execute the algorithms concurrently, allowing them to share information from time to time with the goal of improving overall performance. Suppose that f generates a series of improving approximations to a desired solution:

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \cdots \sqsubseteq f^k(\perp).$$

where $f^i(\perp)$ is the approximation generated on the i th iterative cycle, f is a monotonic function and $\perp$ is an initial approximation. Suppose that g possesses the same properties. It may be useful to allow the processes applying the functions f and g periodically to exchange information in order to improve overall performance. Such a combination of processes is called a *co-algorithm*. A UNITY template for a co-algorithm comprising processes which apply f and g , is:

```

Program co_algorithm
  initially  $x, y, best = \perp, \perp, \perp$ 
  assign
     $x := f(x) \parallel best := best \sqcup x \parallel x := x \sqcup best$ 
     $\parallel y := g(y) \parallel best := best \sqcup y \parallel y := y \sqcup best$ 
  end {co_algorithm}
```

Here the iterative processes which apply f and g may continue in isolation until they reach their (common) fixed-point. Alternatively, each process may update or read an approximation to the best available data, $best$. Thus, one process may utilise another's approximate solution, offering the possibility of achieving the common fixed point more rapidly. The system can be extended in obvious ways to incorporate more than two iterative functions.

A simple concrete co-algorithm to compute a minimum spanning tree is developed below. The purpose of the exercise is to demonstrate that, for a well understood problem, intermediate approximations generated by components of the system may differ in character from approximations generated by independently executing non-interfering processes.

Example 1. A solution to the minimum spanning tree problem may be generated by using either Kruskal's or Prim's algorithm. In order to ensure that both algorithms are generating the same solution it is necessary to assume that the weights of edges of an input graph, G , are strictly positive and unique. In such cases the minimum spanning tree T_G of G is unique. A co-algorithm which combines these algorithms and which periodically shares partial solutions generated by them is:

```

Program min_spanning
  initially  $k, p = \{\}, \{n\}$ 
```

```

assign
     $k := \text{KruskalStep}(k, G)$    $\parallel$    $k := k \cup p$ 
     $\parallel$    $p := \text{PrimStep}(p, G)$      $\parallel$    $p := p \cup k$ 
end  $\{min\_spanning\}$ 

```

Here the approximation in k (*Kruskal*) need not contain only (acyclic) edges of least weight nor need the approximation in p (*Prim*) be a tree. For a random graph the edges in a partial solution generated by Kruskal's algorithm are distributed "uniformly" over the whole input graph G . This property is not enjoyed by the partial solutions generated by Prim's algorithm. Thus, in general, the two algorithms compute *different* approximations to T_G . Combining partial solutions generated by Kruskal's and Prim's algorithms is a valid operation since there is a unique minimum spanning tree and

$$edges(F) \subset edges(T_G) \wedge edges(T) \subset edges(T_G) \Rightarrow edges(F \cup T) \subseteq edges(T_G)$$

This co-algorithm could be extended by including multiple copies of Prim's algorithm. For example, consider a graph G with n nodes. A co-algorithm may comprise Kruskal's algorithm and n instances of Prim's algorithm, each with a different start node in G .

5 Eigensystem Co-algorithm

Consider a co-algorithm to determine the m ($\leq n$) dominant eigenpairs of a real symmetric matrix $A \in \mathbb{R}^{n \times n}$. An eigenvalue-eigenvector pair (λ, x) of A satisfies $Ax = \lambda x$. Computing subsets of eigenvalues and eigenvectors of very large, often sparse, matrices is an essential feature of many important applications. The only feasible way to perform this task is to use an iterative method in which the matrix of interest, A , is used solely as a multiplier. A number of independent iterative algorithms are available for the solution of the problem. The co-algorithm proposed here is constructed from a selection of these and includes an extended power method and a number of subspace iteration methods [6]. For all of these algorithms it is assumed that the eigenpairs are extracted in order of decreasing eigenvalue magnitude. This assumption simplifies the treatment of communication ports below. The extended power method below computes the eigenpairs one at a time; when an eigenpair has been isolated the method is restarted with an initial vector which is orthogonal to previously computed eigenvectors. A sequential version of the algorithm is:

Program *Extended_Power*

initially $x(1), k = e_1, 1$

assign

$x(k) := g(x(k), A)$ if $\neg converged(x(k))$

$\parallel$ $x(k+1), k := ortho(e_{k+1}, x(1), \dots, x(k)), k+1$ if $converged(x(k)) \wedge k < m$

end $\{Extended\_Power\}$

Here $ortho(x, X)$ orthogonalises a vector x with respect to the set of vectors X , e_i denotes the unit vector of length n whose i th component is 1, $converged(x)$ is true if x is a sufficiently accurate approximation to an eigenvector of A and g denotes the iterative body of the extended power method – which includes the orthogonalisation of eigenvector approximations with respect to already converged eigenvectors.

A second method – subspace iteration – repeatedly refines m approximations to the required eigenvectors using an iterative function, h . Each iteration of this method is more computationally intensive than an iteration of the extended power method. On each iteration approximations to *all* of the required m eigenpairs are available. The eigenvector approximations are modified according to $y(1), \dots, y(k) := h(y(1), \dots, y(k), A)$.

There are many ways in which to construct co-algorithms based on the methods above and their variants. Other co-algorithms may be built by including, for example, variants of the Lanczos algorithm [13]. Co-algorithms use partial eigenpair information to improve the overall convergence rate. Here, only a simple instance is presented in which information generated by a *single* subspace iteration method and the extended power method is shared. An appropriate CPO for ordering vectors can be constructed using a metric $\mathbf{M}$ given by $\mathbf{M}(x) =_{def} \| (Ax - \bar{\lambda}x) \|_2$. Here the eigenvalue approximation $\bar{\lambda}$ is given by $\bar{\lambda} = x^T Ax$ and the norm $\|x\|_2$ is defined as $\sqrt{x^T x}$. The vector approximations x and y are ordered according to $x \sqsubseteq y$ iff $\mathbf{M}(x) \geq \mathbf{M}(y)$. The least upper bound of two vectors x and y is given by:

$$x \sqcup y =_{def} \begin{cases} x & \text{if } \mathbf{M}(x) \leq \mathbf{M}(y) \\ y & \text{otherwise} \end{cases}$$

Then a co-algorithm to compute the m dominant eigenvectors of A is:

Program *Extended_Power_Subspace*

initially $x(1), k, y(1), \dots, y(m) = e_1, 1, e_1, \dots, e_m$

assign

```

     $y(1), \dots, y(m) := h(y(1), \dots, y(m), A)$ 
     $\parallel$   $best(1), \dots, best(m) := best(1) \sqcup y(1), \dots, best(m) \sqcup y(m)$ 
     $\parallel$   $y(1), \dots, y(m) := y(1) \sqcup best(1), \dots, y(m) \sqcup best(m)$ 
     $\parallel$   $x(k) := g(x(k), A)$  if  $\neg converged(x(k))$ 
     $\parallel$   $best(k) := best(k) \sqcup x(k)$ 
     $\parallel$   $x(k) := x(k) \sqcup best(k)$ 
     $\parallel$   $x(k+1), k := ortho(best(k+1), x(1), \dots, x(k)), k+1$ 
if  $converged(x(k)) \wedge k < m$ 

```

end {*Extended_Power_Subspace*}

Algorithm interaction can be restricted by limiting the non-determinism in the co-algorithm above: for example, the best available solution may be updated only when an eigenvector has converged to the required accuracy.

6 Results

Some preliminary experimental results are presented for a co-algorithm for computing the first m dominant eigenpairs of a real symmetric matrix. The co-algorithm used comprises the following components:

- an extended power method which computes the m dominant eigenvectors *one at a time* in order of decreasing eigenvalue magnitude;
- six variants of subspace iteration each of which uses a different approach to improving *simultaneously* m eigenvector approximations.

The co-algorithm has been implemented in FORTRAN 90 and uses the MPI communication interface. It has been executed on an IBM SP2 with a single processor node allocated to each of its constituent algorithms. The main points of interest of the implementation are:

- A client/server model is used. Each algorithm is a client. The server acts as the ports.
- The p th port is associated with the p th eigenvector. Only the least upper bound of all received approximations and its accuracy are stored.
- Each client retains a local copy of port data;
- The system employs blocking *sends* to transfer metrics and control signals. Eigenvector transmission utilises non-blocking communication.
- A *send* inspects the accuracy of the local buffer before querying the server; data is transferred only if the information being sent is superior to the best port data.
- A *receive* inspects its local buffer first before querying the server. If the server holds more accurate information, a non-blocking *receive* is initiated.
- Non-blocking transfers use buffers. A local buffer is updated with any completed receive transfers at the beginning of each receive call. A receive transfer initiated on one call can only be used in subsequent calls to the function.
- To reduce communication the clients and server are restricted at any one instant to conducting one non-blocking send and one non-blocking receive for a given port.

Eigenpairs are checked for convergence one at a time, beginning with the dominant pair. A *send* is issued only when an eigenpair has converged. A *receive* is issued for the most dominant non-converged eigenvector after a fixed number of iterations have been performed. The system has been tested using a number of sparse matrices (selected from Matrix Market [10] or constructed from an example taken from the literature [4]). In Table 1 execution times are presented for (i) algorithms operating in isolation and (ii) a co-algorithm in which its constituent algorithms exchange eigenvector approximations. The matrices used are real symmetric, sparse of order n with nz non-zero elements; m eigenpairs are sought. *TimeI* and *TimeC* are the execution times for the fastest single algorithm and the co-algorithm respectively. *UP* denotes the number of updates made to a client's work area (beneficial receives); *CI* denotes the ratio of the execution time of the co-algorithm to that of the fastest single algorithm.

These preliminary experimental results indicate that:

Table 1. Co-algorithm Results (times in secs)

Matrix name	n	nz	m	$TimeI$	$TimeC$	UP	CI
BCSSTK09	1083	9760	4	10.4	9.6	2	.93
BCSSTK25	15439	133840	4	177.7	177.9	0	1.00
12Jennings	10^5	15×10^6	11	946	880	3	.93
12Jennings	10^5	45×10^6	11	2030	1833	4	.9
12Jennings	10^6	9×10^4	11	3689	3318	4	.9
12Jennings	10^6	10^8	11	7061	6538	3	.93
12Gear	10^5	10^5	8	465	465	0	1
salvr1	238	11280	50	1098	50	36	.045

- in the cases of BCSSTK25 and 12Gear no beneficial communications are made to the fastest algorithm. However, the performance of the co-algorithm system coincides with that of the fastest individual algorithms.
- the co-algorithm performs particularly well in the case of Salvr1. In this case all of the *individual* algorithms converged slowly – with some failing to converge within the allocated time. The constituent algorithms benefit greatly from sharing information.
- for the other examples the gains are more modest with the co-algorithm outperforming the best single algorithm by, on average, 8%.

7 Future Work

Constraint satisfaction problem (CSP) involves computing with *partial information* using *propagate-and-search* [12] techniques – for example, $25 \leq X \leq 100$ and $0 \leq Y \leq 75$ is a constraint on the values of X and Y . Partial information may be processed by *local deduction* to generate improved approximations. In the example above the addition of the clause $X \leq Y$ allows the deduction $25 \leq Y \leq 75$. Connections with our approach remain unexplored.

Misra [11] has shown that commutative systems can be realised as (distributed) loosely coupled systems. In general, the actions of the web and co-algorithm templates are *not* commutative. For example, the actions of the web system which update the shared variables f and p

$$\begin{aligned} \{\mathbf{send}\}ssf, f &:= tail(ssf), f \cup \{head(ssf)\} \quad \text{if } ssf \neq null \\ \{\mathbf{receive}\}p, f, g &:= \bigsqcup (p, f \cup g), \{\}, \{\} \end{aligned}$$

do not commute. In one case $head(ssf)$ is first sent and is then read by p ; in the other p is updated before $head(ssf)$ is sent. However, although the web and co-algorithm templates are non-commutative they are still amenable to implementation on distributed message passing architectures. For example, although the actions above do not commute one (*receive* followed by *send*) is a subcomputation of the other in the sense that:

$$(send ; receive) = (receive ; send) ; receive$$

Thus (*receive* ; *send*) can progress, after one further action, *receive*, to the state realised by (*send* ; *receive*). It may be possible, therefore, to broaden the definition of loosely coupled systems to include shared variables which are accessed only by Grab and Go communications.

The preliminary experimental results suggest that the practical realisation of Grab and Go systems merits further investigation. Further work is in progress to determine the practicability of using partial eigensystem co-algorithms. In particular, the investigation is being broadened in three respects:

1. more algorithms (for example Lanczos variants) are being incorporated in the co-algorithm system;
2. alternative interference strategies are being investigated (the current system only sends eigenvector approximations on convergence); and
3. experimental results on a range of architectures are being compiled.

Acknowledgement. The authors are grateful to the referees whose comments have led to improvements in this paper and to C.A.R. Hoare for his remarks on a draft of an earlier paper on Grab and Go communication.

References

1. Bowman H.; Derrick, J.: *Formal Methods for Distributed Processing, a Survey of Object-Oriented approaches*, Cambridge, 2001.
2. Burdern R., Faires, D.: *Numerical Analysis*, sixth edition, Brooks/Cole, 1997.
3. Chandy K.M., Misra J.: *Parallel Program Design: a Foundation*, Addison-Wesley, 1988.
4. Clint M., Jennings A.: The evaluation of eigenvalues and eigenvectors of real symmetric matrices by simultaneous iteration, *Comp. Jnl.*, 13, 76–80, 1970.
5. Davey B.A.; Priestley H.A.: *Introduction to Lattices and Order*, second edition, Cambridge, 2002.
6. Demmel J. W.: *Applied Numerical Linear Algebra*, SIAM, 1997.
7. Gabarró J., Stewart A., Clint M.: Grab and Go systems: a CPO approach to concurrent web- and grid-based computation, *Foundations of wide area network computing (F-WAN)*, ENTCS Vol. 66.3, 2002.
8. Gabarró J., Stewart A., Clint M.: The iterative solution of large-scale problems in numerical linear algebra: a grid-oriented Grab and Go model, PMAA'02, Neuchâtel Switzerland, Abstracts, 29, 2002.
9. Magee, J.; Kramer, J.: *Concurrency, State Models and Java Programs*, Wiley, 1999.
10. Matrix Market available at: <http://math.nist.gov/MatrixMarket/>.
11. Misra J.: Loosely Coupled Processes, *Future Generations Computer Systems* 8, 269–286, North-Holland, 1992.
12. Roy, P.V.; Haridi, S.: *Concepts, Techniques, and Models of Computer Programming*. To appear, <http://www.sics.se/~seif>.
13. Szularz M., Weston J., Clint M.: Explicitly restarted Lanczos algorithms in an MPP environment, *Parallel Computing*, Vol 25, No. 5, 613–631, 1999.
14. Vallejo I.: Visualización de sistemas Grab-And-Go (in Spanish), Final Studies Project, UPC, (to appear).

CAS-Based Lock-Free Algorithm for Shared Deques

Maged M. Michael

IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA
magedm@us.ibm.com

Abstract. This paper presents the first lock-free algorithm for shared double-ended queues (deques) based on the single-address atomic primitives CAS (Compare-and-Swap) or LL/SC (Load-Linked and Store-Conditional). The algorithm can use single-word primitives, if the maximum deque size is static. To allow the deque's size to be dynamic, the algorithm employs single-address double-width primitives. Prior lock-free algorithms for shared deques depend on the strong DCAS (Double-Compare-and-Swap) atomic primitive, not supported on most processor architectures. The new algorithm offers significant advantages over prior lock-free shared deque algorithms with respect to performance and the strength of required primitives. In turn, lock-free algorithms provide significant reliability and performance advantages over lock-based implementations.

1 Introduction

The double-ended queue (deque) object type [11] supports four operations on an abstract sequence of items: PushRight, PushLeft, PopRight and PopLeft. PushRight (PushLeft) inserts a data item onto the right (left) end. PopRight (PopLeft) removes and returns the rightmost (leftmost) data item, if any. A shared object is *lock-free* [8] if it guarantees that whenever a process executes some finite number of steps towards an operation on the object, *some* process (possibly a different one) must have completed an operation on the object, during the execution of these steps. Therefore, unlike conventional lock-based objects, lock-free objects are immune to deadlock even with process (fail stop) failures, and offer robust performance even with arbitrary process delays.

Prior lock-free algorithms for shared deques [1,4,5] depend on the strong DCAS (Double-Compare-and-Swap) atomic primitive, which is not supported on most current processor architectures, and its simulation using weaker widely-supported primitives such as CAS (Compare-and-Swap)¹ or LL/SC (Load-Linked and Store-Conditional)² entails significant performance overhead.

¹ CAS(addr,exp,new) performs {if (*addr ≠ exp) return false; *addr ← new; return true;} atomically. DCAS is similar to CAS, but operates on two independent memory locations.

² LL(addr) returns *addr. SC(addr,new) performs {if (*addr was written by another process since the last LL(addr) by the current process) return false; *addr ← new;

This paper presents the first CAS-based lock-free algorithm for shared deques. The general structure of our implementation is a doubly-linked list, where each node contains pointers to its right and left neighbors, and includes a data field. The two ends of the doubly-linked list are pointed to by two anchor pointers. The two anchor pointers along with a three-value status tag occupy one memory block that can be manipulated atomically by CAS or LL/SC. The status tag indicates whether the deque is stable or not. When a process finds the deque in an unstable state, it must first attempt to take it to a stable state before attempting its own operation. The algorithm can use single-word CAS or LL/SC, if the maximum size of the deque is static. To allow the deque's size to be dynamic, the algorithm uses single-address double-width versions of these primitives.

```
// Code in bold type is for memory management.
// Types and structures
define StatusType = { STABLE, RPUSH, LPUSH }
structure NodeType { Right, Left: (*NodeType, integer); Data: DataType; }
define AnchorType = (*NodeType, *NodeType, StatusType)
// Shared variables
Anchor : AnchorType; // initially Anchor = (null, null, STABLE)
```

Fig. 1. Types and structures.

2 The Algorithm

2.1 Structures

Figure 1 shows the data structures used by the algorithm. The deque is represented as a doubly-linked list. Each node in the list contains two link pointers, Right and Left, and a data field. A shared variable, Anchor, holds the two anchor pointers to the leftmost and rightmost nodes in the list, if any, and a three-value status tag. Anchor must fit in a memory block that can be read and manipulated using CAS or LL/SC, atomically. Initially both anchor pointers have null values and the status tag holds the value STABLE, indicating an empty deque.

The status tag serves to indicate if the deque is in an unstable state. When a process finds the deque in an unstable state, it must first attempt to take it to a stable state before attempting its own operation.

The algorithm can use single-word CAS or LL/SC. Nodes can be preallocated in an array, which length can be chosen such that both anchor pointers and the status tag can fit in a single-word. However, in order to allow the deque to be completely dynamic, with arbitrary size, we employ single-address double-width CAS or LL/SC.

return true;} atomically. However, due to practical considerations, processor architectures that support LL/SC prohibit nesting or interleaving LL/SC pairs, and occasionally but not infinitely often, allow SC to fail spuriously.

2.2 Valid States

Figure 2(a) shows the three stable deque states. A deque is stable only if the status tag holds the value `STABLE`. A deque can be stable only if it is coherent. That is, for all nodes $x^\wedge$ in the deque's list $x^\wedge.Right^\wedge.Left = x$, unless $x^\wedge$ is the rightmost node; and $x^\wedge.Left^\wedge.Right = x$, unless $x^\wedge$ is the leftmost node. Empty and single-item deques are always stable (states S_0 and S_1 , respectively). A deque is in the S_{2+} state if it is stable and contains two or more items.

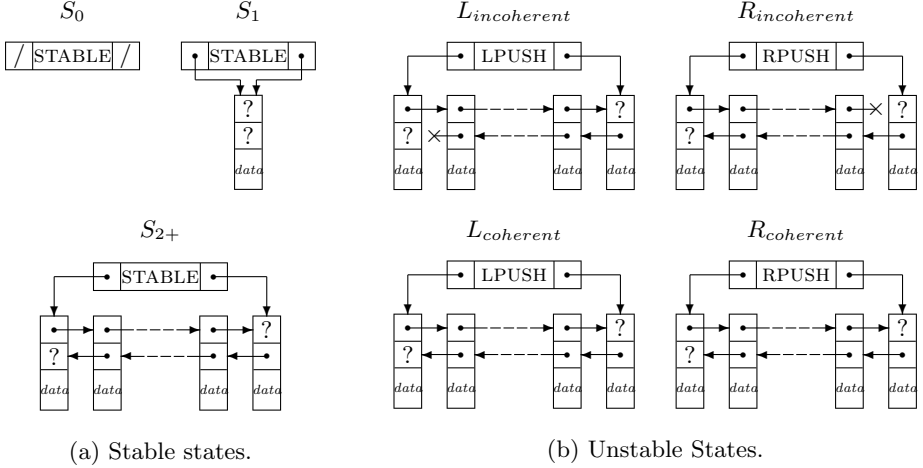


Fig. 2. Valid states.

Figure 2(b) shows the four unstable but valid deque states. A deque can be unstable only if it contains two or more items. It is acceptable for one end of the deque to be incoherent, as long as the status tag indicates an unstable deque on the same end. The list, excluding the end nodes, is always coherent. The deque is right-incoherent if $r^\wedge.Left^\wedge.Right \neq r$, where $r^\wedge$ is the rightmost node. The deque is left-incoherent if $l^\wedge.Right^\wedge.Left \neq l$, where $l^\wedge$ is the leftmost node. The deque is in state $R_{incoherent}$ ($L_{incoherent}$) when the deque is right-incoherent (left-incoherent), the rest of the list is coherent, and the status tag is `RPUSH` (`LPUSH`). At most one end of the deque can be incoherent at a time. Unstable states result from push operations, but never from pop operations, hence the naming of the unstable status tag values. It is also acceptable for the status tag to indicate an unstable deque even when the deque is in fact coherent. Such is the case for the $R_{coherent}$ and $L_{coherent}$ states.

2.3 Operations

For clarity, we assume for now perfect memory management. However, due to the limited available space and the importance of memory management we include

code (in bold type) related to other memory management methods, that we discuss only in the following subsection.

```

PushRight(data:DataType) {
  // For simplicity, assume NewNode() always returns a new node.
  node ← NewNode();
  node.Data ← data;
  while true {
1:   ⟨l,r,s⟩ ← Anchor;
     if r = null {
2:     if CAS(&Anchor,⟨l,r,s⟩,⟨node,node,s⟩) return;
     } else if s = STABLE {
3:     node.Left ← ⟨r,anyvalue⟩;
4:     if CAS(&Anchor,⟨l,r,s⟩,⟨l,node,RPUSH⟩)
       {StabilizeRight(⟨l,node,RPUSH⟩); return;}
     } else Stabilize(⟨l,r,s⟩);
  }
}

```

Fig. 3. PushRight algorithm.

Figure 3 shows the algorithm for the PushRight operation. If the deque is empty (i.e., in state S_0), the operation is completed in one atomic step (line 2) by setting both anchor pointers to the address of the new node containing the new item. Otherwise, if the deque is unstable, it must be stabilized first using the Stabilize routine in Figure 4, before attempting the push operation.

If the deque is stable and contains one or more items (i.e., in states S_1 or S_{2+}), the push operation is completed in multiple steps. The first step (line 4) is to swing the right anchor pointer to the new node and to indicate a right-unstable deque in the status tag, atomically, after setting the Left pointer of the new node to point to the rightmost node in the deque (line 3). After this step, the deque is unstable and most likely right-incoherent. It is possible for a successful CAS in line 4 to take a stable deque in state S_1 or state S_{2+} to state $R_{coherent}$ directly, if the Right pointer of the old rightmost node in the list happens to hold the address of the newly pushed node. The next step of a PushRight operation is to stabilize the deque by calling StabilizeRight.

If either of the CAS operations in lines 2 and 4 fails, the process restarts the push attempt by reading a fresh snapshot of Anchor in line 1. Throughout the algorithm, the state of the deque changes only on the success of CAS operations.

The success of the CAS in line 2 is the linearization point of a RightPush into an empty deque, and the success of the CAS in line 4 is the linearization point of a RightPush into a non-empty deque.

The StabilizeRight routine (Figure 4) involves two main stages. The first stage (lines 5–9) serves to guarantee that the Right pointer of the old rightmost node (i.e., the left neighbor of the newly pushed node) points to the new node.

At the end of that code segment, either the deque is in state $R_{coherent}$, or it must have been in that state at some time since the success of the latest push in line 4 by the current process. The final stage in `StabilizeRight` is to attempt to set the status tag to `STABLE` in line 10 and take the deque to state S_{2+} .

```

Stabilize( $\langle l, r, s \rangle$ :AnchorType) {
  if  $s = \text{RPUSH}$ 
    StabilizeRight( $\langle l, r, s \rangle$ );
  else //  $s = \text{LPUSH}$ 
    StabilizeLeft( $\langle l, r, s \rangle$ );
}

StabilizeRight( $\langle l, r, s \rangle$ :AnchorType) {
  // hp0, hp1 and hp2 are private pointers to three of the process'
  // hazard pointers.
  *hp0  $\leftarrow l$ ; // for memory management only.
  *hp1  $\leftarrow r$ ; // for memory management only.
  if Anchor  $\neq \langle l, r, s \rangle$  return; // for memory management only.
5:   $\langle prev, dontcare \rangle \leftarrow r.^{\wedge}.Left$ ;
  *hp2  $\leftarrow prev$ ; // for memory management only.
6:  if Anchor  $\neq \langle l, r, t \rangle$  return;
7:   $\langle prevnext, t \rangle \leftarrow prev.^{\wedge}.Right$ ;
  if  $prevnext \neq r$  {
8:    if Anchor  $\neq \langle l, r, s \rangle$  return;
9:    if  $\neg \text{CAS}(\&prev.^{\wedge}.Right, \langle prevnext, t \rangle, \langle r, t+1 \rangle)$  return;
  }
10: CAS( $\&Anchor, \langle l, r, s \rangle, \langle l, r, \text{STABLE} \rangle$ );
}

```

Fig. 4. Stabilize and StabilizeRight routines.

In order to avoid race conditions that may corrupt the object, whenever a change in the deque is detected, the process ends its stabilization attempt. The algorithm requires any process that finds the deque in an unstable state to stabilize it first before attempting its intended operation. Thus, any changes in Anchor detected in line 6 or line 8 guarantee that some other processes must have already stabilized the deque.

The order of steps in the algorithm is very delicate, and especially in the `StabilizeRight` (and `StabilizeLeft`) routine. The condition in line 6 guarantees that the pointer value $prev$ read in line 5 is a valid pointer as it guarantees that at that time, the deque contained two or more nodes, and that the node $r.^{\wedge}$ was part of the deque.

The condition in line 8 is needed to prevent the process from corrupting the deque if, for instance, between executing lines 6 and 7, some other process stabilized the deque, popped the node pushed by the original process, and then completed another `PushRight` operation taking the deque to state S_{2+} . Without

the condition in line 8, if the original process resumes execution, it will reach line 9 and its CAS operation will succeed, resulting in the corruption of the deque. The deque would be corrupted as its status tag becomes STABLE while its list is right-incoherent. This is not a valid state. A succession of PopLeft operations can cause the left anchor pointer to point to a node that is no longer part of the deque's list.

```

PopRight():DataType {
  while true {
11:   $\langle l, r, s \rangle \leftarrow \text{Anchor}$ ;
      if  $r = \text{null}$  return EMPTY;
      if  $r = l$  {
12:    if  $\text{CAS}(\&\text{Anchor}, \langle l, r, s \rangle, \langle \text{null}, \text{null}, s \rangle)$  break;
      } else if  $s = \text{STABLE}$  {
          *hp0  $\leftarrow l$ ; // for memory management only.
          *hp1  $\leftarrow r$ ; // for memory management only.
          if  $\text{Anchor} \neq \langle l, r, s \rangle$  continue; // for memory management only.
13:     $\langle \text{prev}, \text{dontcare} \rangle \leftarrow r.^{\wedge}\text{Left}$ ;
14:    if  $\text{CAS}(\&\text{Anchor}, \langle l, r, s \rangle, \langle l, \text{prev}, s \rangle)$  break;
      } else Stabilize( $\langle l, r, s \rangle$ );
  }
  data  $\leftarrow r.^{\wedge}\text{Data}$ ;
  RetireNode(r); // definition depends on memory management method
  return data;
}

```

Fig. 5. PopRight algorithm.

Figure 5 shows the PopRight algorithm. If the deque is empty (i.e., in state S_0) a value EMPTY is returned. Pop operations on non-empty deques take exactly one successful CAS operation to complete. If the deque contains one item (i.e., in state S_1), the operation is completed in one atomic step (line 12) by setting both anchor pointers to null. Otherwise, if the deque is unstable it is stabilized first before attempting the pop. If the deque is stable and contains two or more items (i.e., in state S_{2+}), the pop operation is completed in one atomic step (line 14) by swinging the right anchor pointer to the left neighbor of the rightmost node.

Reading Anchor in line 11 is the linearization point of a RightPop operation on an empty deque. The success of the CAS in line 12 is the linearization point of a RightPop from a single-item deque, and the success of the CAS in line 14 is the linearization point of a RightPop from a multiple-item deque.

The deque algorithm is symmetric. The PushLeft, PopLeft, and StabilizeLeft routines are similar to the corresponding right side routines.

On architectures that support LL/SC (PowerPC, MIPS and Alpha) but not CAS, implementing $\text{CAS}(\text{addr}, \text{exp}, \text{new})$ using the following routine suffices for

the purposes of this algorithm. *{ repeat { if $LL(addr) \neq exp$ return false; } until $SC(addr, new)$; return true; }*

2.4 Memory Management

The freeing and reuse of nodes removed (popped) from a deque object involve two related but different issues: memory reclamation and ABA prevention. The memory reclamation problem is how to allow the arbitrary reuse of removed nodes, while still guaranteeing that no process accesses free memory [13]. The ABA problem can occur if a process reads a value A from a shared location, then other processes change that location to B and then back to A, later the original process performs an atomic comparison (e.g., CAS) on the location and the comparison succeeds where it should fail, leading to corrupting the object [9]. For this algorithm, the ABA problem occurs only if a node is popped and then reinserted while a process holds a reference to it with the intent of using that reference as an expected value of an ABA-prone comparison.

The simplest method for preventing the ABA problem is to include a tag with each pointer to dynamic nodes that is prone to the ABA problem, such that both are manipulated atomically, and the tag is incremented when the pointer is updated [9]. In such a case, an otherwise ABA-prone CAS succeeds only if the tag has not changed since the current process last read the location (assuming that the tag has enough bits to make full wraparound between the read and the CAS practically impossible). This solution allows removed nodes to be reused immediately, but if used by itself it prevents memory reclamation for arbitrary reuse.

Unless atomic operations on three or more words are supported, the inclusion of ABA tags with the Anchor variable in an atomic block prevents the anchor pointers from holding arbitrary pointer values, thus limiting the maximum size of the deque. In such a case, nodes available for insertion in the deque can be allocated in an array. The size of the array can be chosen such that both anchor pointers, status tag, and ABA-prevention tag fit in an atomic block.³

Other memory management methods do not require the inclusion of an ABA-prevention tag with Anchor and thus allow dynamic deque size using double-width single-address primitives.

If automatic garbage collection is available, it suffices to nullify the contents of removed nodes to prevent accidental garbage cycles from forming, in case the fields of the removed node hold the addresses of other memory blocks that would be otherwise eligible for recycling. Garbage collection techniques prevent the ABA problem for this algorithm.

The hazard pointer method [13] allows the safe reclamation for arbitrary reuse of the memory of removed nodes and provides a solution to the ABA-problem for pointers to dynamic nodes without the use of per pointer tags or

³ For example, if the atomic block size is 64 bits, then 32 bits can be dedicated to the ABA-prevention tag, two bits are occupied by the status tag, and each anchor pointer can occupy up to 15 bits. Thus, the deque size is at most $2^{15} - 1$ nodes.

per node reference counters. The method requires the target lock-free algorithm to associate a small number of *hazard pointers* (three for this algorithm) with each participating process. The method guarantees that no removed node is reused or freed as long as some process' hazard pointer has been pointing to it continuously from a time when it was in the object. By preventing the ABA problem for Anchor without using tags, it allows the anchor pointers to hold arbitrary (two-byte-aligned) pointer values, and hence allows the size of the deque and its memory use to grow and shrink arbitrarily, using double-width single-address CAS or LL/SC. The figures in the previous subsections show (in bold type) code related to memory management that uses hazard pointers, and for simplicity uses ABA tags for the side pointers of each dynamic node. Combined with hazard pointers, ABA tags do not prevent removed nodes from being freed for arbitrary reuse.

2.5 Complexity

In the absence of contention, each operation on the deque takes a constant number of steps. Under contention, we use the total work complexity measure. Total work is defined as the worst-case total number of primitive steps executed by P processes, during any execution of an arbitrary sequence of r operations on the shared object, divided by r [10]. The total work of any operation of the new algorithm, under contention by P processes is $O(P)$. Intuitively, the useful work to complete an operation is constant, and at most constant work for each of the other $P - 1$ concurrent operations is deemed useless as a result of the success of the completed operation.

3 Discussion

3.1 Related Work

Universal lock-free methodologies (e.g., [8,17]) can be used to transform sequential object implementations including deques into shared lock-free implementations. However, the resulting lock-free implementations for almost all object types, including deques, are extremely inefficient. This motivated the development of object-specific algorithms.

The first lock-free shared deque algorithms were the linear (array-based) algorithms of Greenwald [5] using DCAS. Agesen *et. al.* [1] presented two DCAS-based algorithms. The first algorithm is array-based, offering improvements over Greenwald's. The other algorithm is dynamic and requires three DCAS operations per push and pop pair, in the absence of contention. Later, they presented a dynamic DCAS-based algorithm [4] that requires two DCAS operations per push and pop pair, in the absence of contention.

DCAS is not supported on most current processor architectures. Implementations of DCAS using CAS or LL/SC were proposed in the literature (e.g., [3, 10,17]). However, they all entail substantial overhead for every DCAS operation,

prompting many researchers to advocate the implementation of DCAS efficiently in hardware [5]. In fact, Agesen *et al.* [1,4] state as the main motivation for developing their algorithms, justifying the need for hardware support for DCAS on future architectures. This paper shows that DCAS is not needed for the efficient implementation of lock-free shared dequeues.

It is worth noting that the algorithms of Agesen *et al.* [1,4] allow execution overlap between one operation from the right with one operation from the left. On the other hand the new algorithm serializes operations on the deque as operations are serialized on the Anchor variable. However, even without concurrency, the new algorithm promises much higher throughput than the former algorithms, which at most allow the concurrency of two operations. Basic queuing theory determines that one fast server is guaranteed to result in higher throughput than two slow servers, if the response time of each of the slow servers is more than twice that of the fast server. Operations of the former algorithms, even without contention, are substantially slower than those of the new algorithm, taking into account simulation of DCAS using CAS or LL/SC.

Finally, it is worth noting that sometimes single-producer work queues are referred to in the literature as dequeues [2]. However, these objects do not actually support the full semantics of the shared deque object (i.e., without restricting the number or identity of operators). Furthermore, such objects do not even support the semantics of simpler shared object types such as LIFO stacks and FIFO queues that are subsumed by the semantics of the deque object type.

3.2 Summary

This paper was motivated by the significant reliability and performance advantages of lock-free algorithms over conventional lock-based implementations, and the practicality and performance deficiencies of prior lock-free algorithms for shared dequeues, which depend on the strong DCAS primitive. We presented the first lock-free algorithm for shared dequeues based on the weaker primitives CAS or LL/SC. The algorithm can use single-word CAS or LL/SC, if the nodes available for use in the deque are preallocated statically. In order to allow the size of the deque to grow and shrink arbitrarily, we employ single-address double-width CAS or LL/SC. The new algorithm offers significant advantages over prior lock-free algorithms for shared dequeues with respect to the strength of required primitives and performance. With this algorithm, the deque object type joins other object types with efficient CAS-based lock-free implementations, such as LIFO stacks [9], FIFO queues [14,15,19], list-based sets and hash tables [6,12,16], work queues [2,7], and priority queues [18].

References

1. Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. DCAS-based concurrent dequeues. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, July 2000.

2. Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, June 1998.
3. Hagit Attiya and Eyal Dagan. Improved implementations of binary universal operations. *Journal of the ACM*, 48(5):1013–1037, September 2001.
4. David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. Even better dcas-based concurrent dequeues. In *Proceedings of the 14th International Symposium on Distributed Computing, LNCS 1914*, pages 59–73, October 2000.
5. Michael B. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
6. Timothy L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing, LNCS 2180*, pages 300–314, October 2001.
7. Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 280–289, July 2002.
8. Maurice P. Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
9. IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.
10. Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.
11. Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.
12. Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.
13. Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.
14. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.
15. Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.
16. Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, July 2003.
17. Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
18. Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, April 2003.
19. John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, October 1994.

Energy Efficient Algorithm for Disconnected Write Operations in Mobile Web Environments^{*}

Jong-Mu Choi, Jin-Seok Choi, Jai-Hoon Kim, and Young-Bae Ko

College of Information and Communication,
Ajou University, Republic of Korea
{jmc, shinechoi79}@dmc.ajou.ac.kr, {jaikim, youngko}@ajou.ac.kr

Abstract. Because wireless links are subject to disturbance and failures, supporting disconnected operations is important issues in mobile computing research. Many algorithms have been proposed for supporting the efficient disconnected operations to continue performing task with a low communication cost and are focused on how the mobile client-server model can be adapted to the dynamic wireless environments. However, energy efficient algorithms are also important in mobile computing environments since mobile devices are powered by battery. In this paper, we propose energy efficient algorithm for disconnected write operations in mobile Web environments and develop analytical models with the goal of saving energy-consumption.

1 Introduction

Mobile computing and wireless networks are fast growing technologies. Since the mobile computation on the wireless networks is subject to disturbance and failures, it is essential to support disconnected operations in mobile computing. When a wireless link is expected to disconnect, process and/or data can be transferred to a mobile host (MH) to continue its operations during the disconnection. The MH is in the *hoarding* state before entering disconnection. In the hoarding state, data expected to be used during the disconnection is moved into the MH. When a wireless link is disconnected, the MH enters the *disconnected* state. During the disconnected state, the MH can continue operation using local data only including the hoarding data prefetched during the hoarding state. Data modification at the MH is logged on a stable storage. If the MH requests data not available locally in the disconnected state, then the MH enters a *reintegration* state. The MH can't execute further instructions in the reintegration state. These three states can be used to support disconnected operations [1,2,3].

^{*} This work is supported by grant for Upcoming Research Program of the Korea Research Foundation(KRF-2002-003-D00262), the Ministry of Education of Korea(Brain 21 Project Supervised by Korea Research Foundation), and University Fundamental Research Program supported by Ministry of Information & Communication in Republic of Korea(2001-103-3).

In case of mobile Web environments, a number of algorithms supporting disconnected operations have been proposed in the past years [4,5,6]. These algorithms assume read-only property in Web browsing during disconnections. To support Web page write operation, a few different approaches have been proposed [7,8]. However, all of these algorithms supporting disconnected operations on mobile Web are mostly focused on how to support disconnected operations for mobile Web environments and to optimize their algorithms in the aspect of computation and communication cost (mostly they use computing and communication time as a cost factor). Because these algorithms do not consider energy consumption in their algorithms, it can cause energy-shortage since mobile devices are powered by battery. Thus, in mobile environments, energy efficiency in the MHs (mobile hosts) are stressed as another important considering factor in the disconnected operations. To solve this problem and to further improve the energy performance of existing disconnected operations, we propose a new energy efficient algorithm for disconnected write operations in mobile Web environments. Using our proposed algorithm to reduce energy consumption, the number of messages transferred in the MHs are reduced and work load of disconnected operations for mobile Web service is performance on the Web server instead of MH. Web server does not suffer from energy-shortage since plenty of energy can be constantly supplied.

2 Related Works

In this section, we briefly summarize some of existing disconnected operations in mobile Web environments. We motivate our works by pointing out energy consumption problem not considered in these algorithms.

Many algorithms have been proposed to support disconnected operations efficiently [1,2,3]. These are based on cache management to support disconnection states. Files to be used during the disconnected state are prefetched at the MH (mobile host). In hoarding state, users can prefetch the files manually or system prefetches the files automatically by utilizing file access history.

Recently, algorithms to support disconnected operations for mobile Web environments are proposed and these are based on read-only property in Web files [4,5,6]. These algorithms assume that Web data is not changed or added in the MH. Reference [4] describes the eNetworks Web Express systems which uses paired proxies to optimize communication in low-bandwidth networks and to enable Web browsing in which both disconnected and asynchronous operations are supported. For optimizing the Web prefetching scheme, there are various algorithms. Among those algorithms, Reference [5] introduces efficient access probabilities and prefetch thresholds to reduce access delay, and prepare for disconnection. The prefetching scenario discussed in [5] is adopted to verify the performance improvement of our algorithm. To support intermittent or weak connections ARTour Web Express [6] permits updates to be gradually and asynchronously submitted to a server and all requests are served when connectivity is re-established.

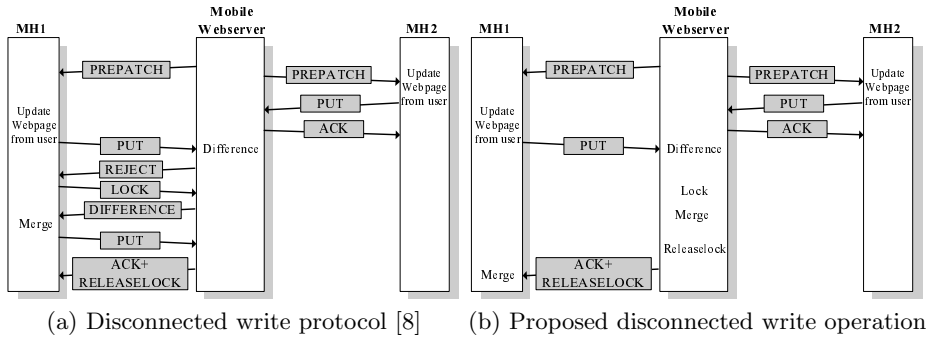


Fig. 1. Disconnected write operation in [8] and proposed algorithm

More recently, reference [8] proposed disconnected write operations for wireless Web access in mobile client-server environments. Their algorithm is depicted in Fig. 1(a). According to their scenario, two mobile hosts (MH1 and MH2) cache the Web page from mobile Web Server beforehand, then MH1 enters the disconnected state. First, two MH users modify the cached Web pages during the disconnected state of MH1. After MH1 is reconnected to mobile Web server, MH1 sends PUT request message which contains the differences between the modified Web page and the original Web page, and the version number of original Web page. Now, two different cases (**Case 1** and **Case 2**) can be occurred by Web server[8].

Case 1 (No update conflict): The Web page in the mobile Web server has not been updated by other MH (e.q., MH1). Thus, the Web server sends an ACK message to MH2 to notify the acceptance of the PUT request.

Case 2 (Update conflict): The Web page in the server has been modified by other MH (e.q., MH2). It can be detected by comparing the version numbers.

In Case 2, the mobile Web server sends the REJECT message to MH1. When MH1 receives the REJECT message, it needs to follow the special operations listed below to synchronize with the Web server[8].

1. MH1 sends the LOCK message to mobile Web server to lock the Web page.
2. Mobile Web server generates DIFFERENCES message which contains information about difference between prefetched data to MH1 and updated data from MH2.
3. On receiving DIFFERENCES message from mobile Web server, MH1 performs *merging* algorithm to update the Web page.
4. After updating the Web page, MH1 resends PUT request message which contains information about updated Web page.
5. Finally, mobile Web server releases locking on their Web page then it replies ACK message to MH1.

The above algorithm, a mobile host oriented approach, may result in generating too many messages and wasting the energy consumption in mobile host due to exchanging many messages and performing merge algorithm. We think that these problems occurred because the existing disconnected operations algorithms such as reference [8] do not consider the energy consumption as a cost metrics in mobile computing. Thus, we attempt to reduce the number of message transfer and the amount of computation in mobile host by shifting work load to mobile Web server, which is free from energy limitation and has enough computation power.

3 Proposed Disconnected Write Operation

Our proposed algorithm for disconnected write operations are depicted in Fig. 1(b). The operations of MH2 are same as in Fig. 1(a). The Web page in the mobile Web server is not updated by other MH until receiving PUT request message. However our proposed algorithm operates differently when the Web page in the server has been modified by other MH. In this situation, mobile Web server does not send REJECT message to MH1 but voluntarily merges and updates its own Web page using update information from PUT request message sent by MH1. When the version information on receiving PUT message from MH1 and on server are different, the mobile Web server operates as follows:

1. Lock the Web page to prevent the updating operation from other MH.
2. Find difference between the data prefetched to MH1 and updated from MH2.
3. Merge the difference from the previous step and update data according to PUT request message.
4. Release the locked Web page.
5. Send the ACK and DIFFERENCE, which contains merging data, to MH1.

On receiving DIFFERENCE message, MH1 can update its prefetched data. By doing so, MH1 synchronize the content of date with mobile Web server. Transferring more work load to mobile Web server, our proposed algorithm can dramatically reduce the number of message needed for disconnected write operations. Mobile Web server may be somewhat over loaded and consumes more energy than the previous algorithms as shown in Fig. 1(a). However, as Web server consists of high performance computer and enough system resources such as CPU, memory, and storages, it can endures the work loads. By supplying eternal power, energy problems can also be resolved since mobile Web servers are electrically connected to power supply construction. However, in some applications, merge algorithm can't be performed in Web server (e.g., mobile users need to decide their choice when their update operation is conflicted on the Web server). We think that mobile user can cancel the merge operation in the Web server, and send PUT message again after receiving PREFETCH message.

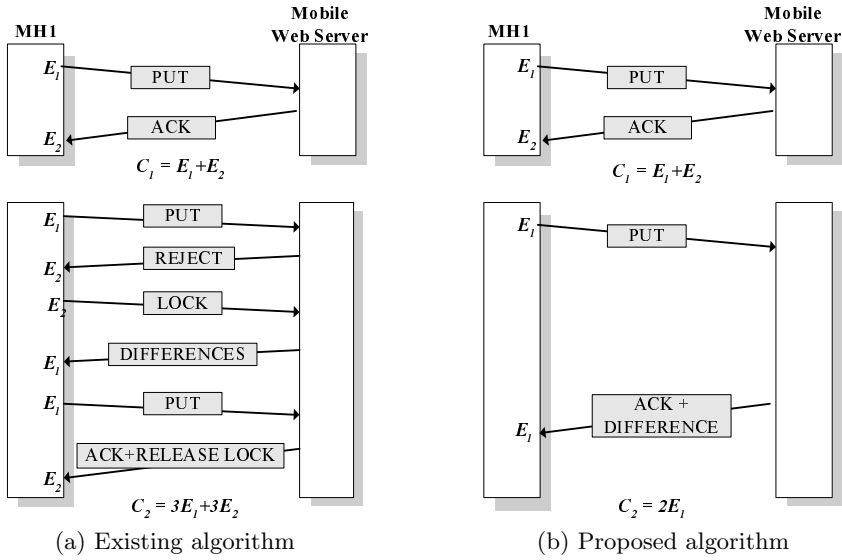


Fig. 2. Energy const model for existing algorithm and proposed algorithm

4 Performance Evaluation

The objective of our proposed algorithm is to minimize the energy consumption during disconnected write operations in mobile Web environments. Thus we use the energy consumption as a performance metrics. Proposed algorithm is compared to the existing disconnected write operations[8] for mobile Web environment by means of energy consumption. Let the energy cost on the MH be E , which include the energy to transmit the message as well as receive the message. According to the first-order radio model in [9],

$$E = E_{TX} + E_{RX} = N_{TX} \times \epsilon_{TX} + N_{RX} \times \epsilon_{RX} \quad (1)$$

where N_{TX} and N_{RX} is the number of bits in message for transmissions and receiving, respectively, and ϵ_{TX} and ϵ_{RX} is the energy consumed to transmit and receive a bit via a wireless link, respectively. If ϵ_{TX} and ϵ_{RX} are assumed to be the same, these two parameters can be denoted as ϵ . We also assume two energy cost parameters, E_1 and E_2 , which represent the energy cost for exchanging message of different size. These two parameters can be classified whether it is update message or control message. Namely, E_1 is the energy cost for message transfer which contains update data such as PUT/DIFFERENCE and E_2 is the energy cost for control message such as ACK/REJECT/LOCK. These two parameter can be represented by $E_1 = p_m s_1 \epsilon$ and $E_2 = s_2 \epsilon$. where p_m is the average fraction of Web page being modified and s_1 and s_2 is the average size of Web page and control message, respectively.

Finally, we assume some parameters related to update behavior. The Web page in server can be updated by any MH. In this case, the update rate by any MH for Web page is denoted as λ , and update rate by a specific MH itself is denoted as λ_m . If MH1 updates data during a disconnected state, MH1 sends PUT request message after the end of disconnected state. Assume that other MHs also update the Web page in the mobile Web server during the disconnected state of MH1 (L), then the next PUT request message from MH1 may be denied. Namely, the **Case 2** in section 2 is occurred. To model this case, probability that an update request is denied by the mobile Web server (r), is given by $r = 1 - e^{-(\lambda - \lambda_m)L}$ where $\lambda - \lambda_m$ is the rate of update by other MHs and L is duration of disconnected state. We assume that the arrival of Web page update at the system is governed by the Poisson process. We represent energy cost as C_1 and C_2 , for **Case 1** and **Case 2**, respectively. Thus, by applying the above equations, the average energy consumption involved in update propagation from the MH to the Web server as shown in Fig. 1(b) can be modeled following Eq.(2).

$$C_e = r \times C_2 + (1 - r) \times C_1 \tag{2}$$

Fig. 2 shows the energy cost model for existing algorithm and our proposed algorithm. We evaluate only the aspect of the MH since mobile Web server does not concern the energy limitation. As shown in Fig. 2, we can compute the four different energy costs classified in Table 1. In the Case 1, energy cost is the same for both algorithms, $C_1 = E_1 + E_2$ (E_1 for PUT and E_2 for ACK message). However, in the Case 2, two algorithms require different energy cost as two algorithms operate differently. Existing algorithm requires $3E_1$ for three control messages and $3E_2$ for two PUT messages and one DIFFERENCES message. Proposed algorithm requires $2E_2$ for PUT message and ACK+DIFFERENCE message. By applying these four energy costs to Eq.(2), we can obtain the two equations, $C_e^{EA} = 3r(E_1 + E_2) + (1 - r)(E_1 + E_2)$ and $C_e^{PA} = 2rE_1 + (1 - r)(E_1 + E_2)$, for each energy consumption behavior of existing algorithm and proposed algorithm, respectively.

Table 1. Energy cost for different cases

	Existing algorithm	Proposed algorithm
Case 1: No update conflict (C_1)	$E_1 + E_2$	$E_1 + E_2$
Case 2: Update conflict (C_2)	$3E_1 + 3E_2$	$2E_1$

We have analyzed under following conditions (similar to [8] and [9]). Initially, the average rate of updating the Web page by a MH is 8/3600 and any MH is 10/3600, respectively – i.e., update process have been occurred 8 and 10 times per hour for each. We fix ϵ (radio electronics) And p_{Mm} (average fraction of the Web page modified) is 0.1. Thus, s_2 can be $0.1 \times s_1$. For different performances analysis, we varied λ_m/λ (ratio between rates of updating the Web page by a MH and any MH), ϵ , and $p_m \times s_1$ (size of updated page). Varying these three values

intends to observe the effect of these parameters on energy consumption when the disconnected write operations are executed in the mobile Web environment. The value of λ_m/λ is varied from 0.2 to 1 in our first analysis. The value of ϵ is varied from 0.2 to 1 mJ/KB . And at the last analysis, we also vary $p_m \times s_1$ from 5 KB to 25 KB .

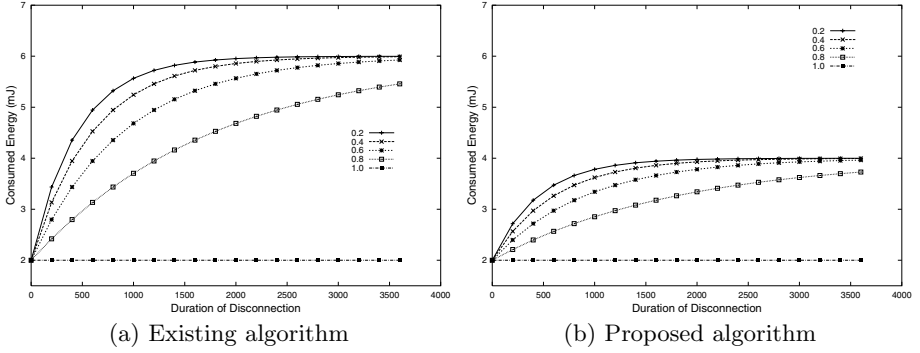


Fig. 3. Energy consumption varying λ_m/λ

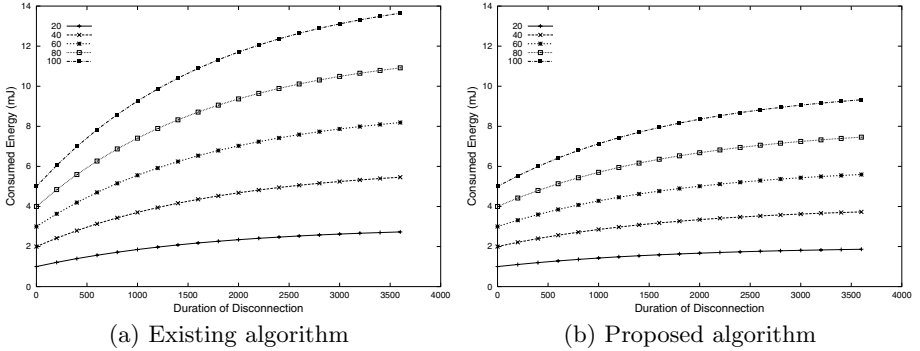


Fig. 4. Energy consumption varying ϵ

The effect of varying the values λ_m/λ , ϵ , and $p_m \times s_1$ are shown in Fig. 3, 4, and 5, respectively. As these parameters are increased, the energy cost begins to increase for both algorithms. However, proposed algorithm requires a less energy (up to 30 %) than existing algorithm. The reason is that proposed algorithm reduce the number of message for disconnected operations by giving more work load to the mobile Webserver when update conflict occurs.

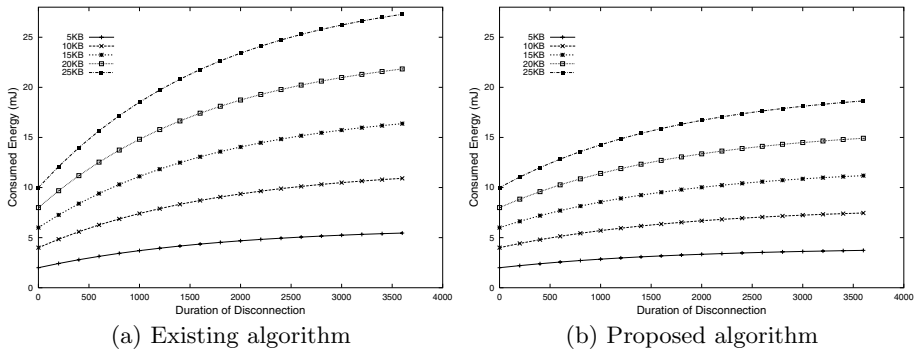


Fig. 5. Energy consumption varying $p_m \times s_1$

5 Conclusion

Many approaches have been proposed to support disconnected operation, these previous algorithms are focused on minimizing the response time. However, energy efficiency is also important in mobile Web environment since mobile host are powered by battery. In this paper, we proposed disconnected write operations for mobile Web environments to save energy consumption by reducing the exchange of unnecessary messages and transferring the work load to Web server. To verify excellence of proposed algorithm, we have developed simple analytical model to compare our algorithm with existing algorithm. The result of performance comparison shows that proposed algorithm can save the energy consumption up to 32 % over the existing algorithm.

References

1. E. Pitoura: Data Management for Mobile Computing. *Kluwer Publishers*, (1998).
2. S. Jeong and J.-H. Kim: Modeling and Performance Analysis of Disconnected Operation for Mobile Computing. *1st Annual ICCIS*, Oct. (2001).
3. J. Jing, A. Helal, and A. Elmagarmid: Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, Vol. 31, No. 2, Jun. (1999).
4. R. Floyd, and B. Housel: Mobile Web Access Using eNetworks Web Express. *IEEE Personal Commn.*, Vol. 5, No. 5, Oct. (1998).
5. Z. Jiang and L. Kleinrock: Web Prefetching in a Mobile Environment. *IEEE Personal Commn.*, Vol. 5, No. 5, Oct. (1998).
6. H. Chang, et al.: Web Browsing in a Wireless Environments: Disconnected and Asynchronous Operation in ARTour Web Express. *ACM Mobicom '97*, Sep. (1997).
7. M. S. Mazer and C. L. Brooks: Writing the Web While Disconnected. *IEEE Personal Commn.*, Vol. 5, No. 5, Oct. (1998).
8. I.-R. Chen, N. A. Phan, and I.-L. Yen: Algorithms for Supporting Disconnected Write Operation for Wireless Web Access in Mobile Client-Server Environments. *IEEE Transaction on Mobile Computing*, Vol. 1, No. 1, Jan. (2002).
9. W. Heinzelman, A. Chandrakasan, and H. Balakrishnan: Energy-efficient Communication Protocols for Wireless Microsensor Networks. *HICSS*, Jan. (2000).

Distributed Scheduling of Mobile Priority Requests

Ahmed Housni, Michel Lacroix, and Michel Tréhel

LIFC/CNRS-FRE UFR Sciences et Techniques Université de Franche Comté 16, route de gray
25030 Besançon Cedex France
{housni, lacroix, trehel}@lifc.univ-fcomte.fr

Abstract. We present an adaptation of the Naïmi-Tréhel mutual exclusion algorithm to manage a prioritized access to a shared resource. Each resource request has a priority level, and is recorded in a global distributed queue. The distributed queue scheduling needs a modification of the algorithm data structures, additional messages between requesting sites (users), and request mobility between requesting sites.

1 Introduction

This paper presents an adaptation of an algorithm by Naïmi-Tréhel [2] to a prioritized environment, by making it possible to associate priority with each request. The term request refers to the use of a shared resource, which is protected by a mutual exclusion algorithm. In the initial mutual exclusion algorithm by Naïmi-Tréhel [2], waiting requests are pushed in a global distributed queue : each requesting site (user) may issue only one request at a time. Associating priorities to requests and processing those requests according to these priorities, leads to sort a global distributed queue.

The first section presents the initial algorithm by Naïmi-Tréhel [2]. In the second section, we present our work, which introduces priority in that algorithm. The last section is for conclusion.

2 Presentation of the Algorithm by Naïmi-Tréhel

Naïmi-Tréhel Algorithm [2] is based on the unicity of a token which moves on a dynamic tree, this tree being extracted from a completed communication graph. In this algorithm, a site (a user), which wishes to enter the critical section, sends its request message to another predefined site: its father in the rooted tree, if it has one. Every site knows its father in the rooted tree: its father will receive all the requests it issues. When a site receipts a request, it deactivates the communication line which connects it with its son, who sent this request, and then, activates the communication line which connects it to the request sender site. Requests are pushed in a distributed global queue, and processed according to the order they enter this queue. This simple linked

queue is made up of the *successor* variables associated with each site. The queue head is the token holder site.

2.1 Data Structures

Naïmi-Tréhel algorithm uses two data structures:

- **A logical rooted tree:** this tree is modified according to the evolution of the different requests for the critical section. A request transits along the path leading from its issuing site to the root. This root site may be in one of the following four states:
 1. It possesses the token, but does not use it: the requesting site may then get the token and enter the critical section.
 2. It possesses the token and uses it; the requesting site is put in a waiting state.
 3. It is waiting for the token; the requesting site is put in a waiting state.
 4. It does not possess the token and doesn't request the token: the requesting site is linked to the site which is the token holder.
- **A distributed global queue:** when the root site is in a state corresponding to the states 2 or 3, it records the site identity of the request author site in its *successor* variable. The linkage of the *successor* variables values forms a distributed global linked queue (Fig. 1).

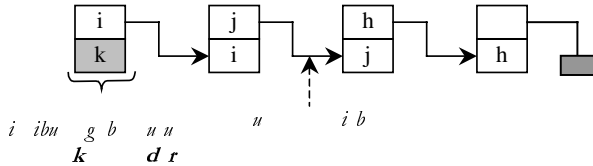


Fig. 1. The distributed global queue

2.2 Messages

Naïmi-Tréhel algorithm uses two message types:

1. A "Request" message, which represents the critical section request.
2. A "token" message, which represents the token and allows to enter the critical section.

3 Prioritized Naïmi-Tréhel Algorithm

Our contribution associates a priority to each request. Requests are ordered according to an increasing priority, and are processed according to a decreasing order. With regard to the dynamic property of the request rooted tree, the request sorting method is an insertion sort.

3.1 Data Structures

To preserve the Naïmi-Tréhel modification mechanism of the request rooted tree, we modify the distributed global queue structure by adding, for each site, a variable which denotes its *predecessor* in the request waiting queue. Hence, the queue becomes a double linked queue. Every requesting site knows its successor and its predecessor, if they exist (Fig. 2). The head of the request global queue is the token holder site. When a site goes out of the critical section, it passes on the token to its successor in the distributed global queue, if there is one of it. The token movement does not modify the routing requests rooted tree.

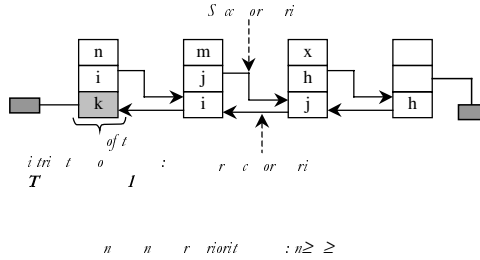


Fig. 2. Structure of the distributed global queue

3.2 Messages

To sort this distributed queue, we modify the structure of the existing messages (*request* and *token*), and introduce additional messages (*Find_Place*, *Exchange*, *Ack* and *MAJ_PRED*). A special message, (*MAJ_POS*), corrects the algorithm dysfunction when it occurs: i.e. when the root of the request graph is neither a token user nor a token holder site.

3.2.1 Messages for the Critical Section Management

a) **The token message** contains the following information :

- *Request_token* , is the site which will use the token,
- *Lender_token* , is the site which lent the token. This information allows to return the token backwards to the root of the tree (if the site, which last used the token, is not the root).
- *issuing_token* , is the site which sent the token towards another token requesting site. This information allows forwarding a request to the next token user, when the queue head has already sent the token to its successor.

b) **The Request message** is modified and mapped through a *message(Request, Request_site, Priority)* :

- *Request_site* is the requesting site which is the author of the request.
- *Priority* is the request priority.

3.2.2 Messages for the Global Queue Management

Associating priority to requests requires the participation of the requesting sites in managing the global distributed queue. Requesting sites exchange messages to sort requests in that queue:

a) The *Find_Place message*, This message runs right across the global queue to find where to insert a request j in the queue. It contains :

- the request j , with its priority: $(j, \text{Priority})$,
- "*issuing_msg_site*" is the site identity of the site which issues this message. This allows to stop the *ping pong* between two requesting sites when:
 - The first site contains a request with a lower priority than request j 'one in the local queue,
 - The second site contains a request with a higher priority than request j 'one in the local queue.

b) The *Exchange message* : this message is used to exchange requests between the site which issued the request and the one which receives this message, when the place of the request to place is found. This message contains the request to exchange and the identity of its issuing site.

c) The *MAJ_PRED message*: this message updates the link (the "*predecessor*" variable) of the double linked queue.

d) The *MAJ_POS message*: this message updates the "*token_holder*" variable of the root site.

3.2.3 Synchronization Messages between Requesting Sites

a) The *message Ack* : to avoid the possible loss of waiting requests in two requesting sites during the queue management, we add a synchronization *message (Ack, issuing_site)*. This message indicates that the site which sends the *message (Exchange, {j, Priority}, issuing_msg_Echange)*, must not modify the local queue contents as long as the site which receipts the *message (Exchange, {j, Priority}, issuing_msg_Echange)*, did not send the *message (Ack, issuing_site)*.

b) The *message (Ack_sup, successor of the site)* : Before a site kills the waiting request of its successor in its queue, it must receive an acknowledgment from its successor (a *message (Ack_sup, successor of the site)*) as soon as this last one receives the token. This allows forwarding a request to the successor of i .

3.2.4 Message to Correct the Dysfunction of the Algorithm

Two cases are added which correspond to new situations created by the introduction of the priority in Naïmi-Tréhel [2] algorithm. Indeed, the association of priority to requests and their management introduce two dysfunctions :

- * How could the token go backwards to the root of the request routing rooted tree, when there are no current request, and when this root is not the last element of the distributed global queue?
- * How to insert a request into the distributed global queue, when this request reaches the root of the request routing rooted tree, this root being either a token user or a token holder?

These two dysfunctions appear when the root r of the request routing rooted tree passes in the third of the successive states enumerated below:

- + The root r is the head of the distributed global queue.
- + The root r is in the critical section and does not receive requests.
- + The root r left the critical section and gives on the token to its successor in the distributed global queue.

To correct these dysfunctions, we introduced two variables and an additional message in the classical Naïmi-Tréhel algorithm:

* The variables:

- If the tree root is not the tail of the queue, the "*Lender*" variable allows this element to send the token backwards to the tree root.
- If the tree root is not the requesting site, the "*token_holder*" variable indicates to this root which site can take care of the request and insert it into the distributed queue.

* The *MAJ_pos* message and the *token_holder* variable correct the dysfunction of the algorithm. This message contains information relative to the token (*Lender_token_site* and *token_holder*). The message is issued by the token holder site and is intended for the root, when this one transits in the state 3 (the token holder knows the root and its state). This message is issued as the token holder goes out of the critical section. It allows the root to know the new token holder and to update the *token_holder* variable. The root state changes when it receives an other request.

3.3 Algorithm Performances

In Naïmi-Tréhel [2] without priority, the number of messages needed for a request satisfaction equals the step number this request made, from its author site to the site which emitted the previous request, plus 1 (to book the token displacement message). Let us denote $D(i, j)$ the distance between site i which is the author of a request and the site j which emitted the preceding request. $D(i, j)$ = number of sites crossed by a site i request before it reaches site j . The number of necessary messages for a request satisfaction is: $(D(i, j) + 1)$ messages. The number of messages necessary for a request satisfaction, in that case, integrates, besides the number above, the requests displacement messages between sites and the request placement messages on requesting sites (of the distributed global queue) plus three messages (a message for the permutation of two requests, a synchronization message, a message to place the token). Let us note $P(i, j)$ the number of requesting sites between i and j in the distributed global queue. The number of messages necessary for a request satisfaction is: $(D(i, j) + P(i, j) + 3)$ messages. Yet an exception: if the token holder emits a request, the necessary number of messages is zero.

4 Conclusion and Perspectives

This paper presents a distributed scheduling of mobile prioritized requests in a dynamic communication structure. If Naïmi-Tréhel [2] algorithm without priority ap-

pears to be more effective than Mueller's one [3], Mueller algorithm results remains better than those obtained with Naïmi-Tréhel [2] algorithm when it includes priority. A distributed sort is very expensive. We note that, when sites have identical fixed priorities, it seems interesting to cluster these sites in order to optimize the token movement. This led us to be concerned with groups [1], with the following main idea: looking, among algorithms, for two algorithms the most adapted to the communication, based on the distinction between: Communication between groups (an algorithm handling priority) and Communication inside the groups (an algorithm without priority).

References

1. A. Housni, "L'exclusion mutuelle entre les membres de deux groupes structurés de téléconférenciers", *revue RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles)*, Ed. Hermès, vol. 12, n° 5–6, pp. 565–592, 2000.
2. M. Naïmi, M. Tréhel and A. Arnold, "A $O(\log(n))$ distributed mutual exclusion algorithm based on path reversal", *Parallel and distributed computing*, 34, pp. 1–13, 1996.
3. F. Mueller "Prioritized Token-Based Mutual Exclusion for Distributed Systems", in *Workshop on Parallel and Distributed Real-Time Systems*, pages 72–80, April 1997.

Parallel Distributed Algorithms of the β -Model of the Small World Graphs^{*}

Mahmoud Rafea, Konstantin Popov, Per Brand, Fredrik Holmgren, and
Seif Haridi

SICS, Box 1263, SE-16429 Kista, Sweden
{mahmoud,kost,perbrand,fredrikh,seif}@sics.se

Abstract. The research goal is to develop a large-scale agent-based simulation environment to support implementations of Internet simulation applications. The Small Worlds (SW) graphs are used to model Web sites and social networks of Internet users. Each vertex represents the identity of a simple agent. In order to cope with scalability issues, we have to consider distributed parallel processing. The focus of this paper is to present two parallel-distributed algorithms for the construction of a particular type of SW graph called β -model. The first algorithm serializes the graph construction, while the second constructs the graph in parallel.

1 Introduction

The idea of small worlds (SW) is based on a phenomenon, which formalizes the anecdotal notion that "you are only ever six 'degrees of separation' away from anybody else on the planet." In this paper, the β -model of SW graphs is going to be described but other graphs are out of the scope. For more information about SW graphs we refer to [1].

The β -model graph is used to model the social network of Internet users where each vertex represents a user and the neighbors of this vertex represent his friends. Each vertex in the graph represents a simple agent [2]. Those agents are used to implement behavior models that simulate emerging phenomena on the Internet. The sequential agent-based simulation environment can be found in [3]. Also, one of the implemented behavior models can be reviewed in [4]. The parallel environment is under publication [5].

The platform is a rack-mounted 16-nodes in a chassis. Each node has: an AMD Athlon (1900+) processor on a Gigabyte 7VTXH+ motherboard, which contains half Gigabyte DDR memory. All the nodes are running Linux and having the same installation. The implementation language is Mozart [6]. The next section (2) introduces the β -model. The motivations of implementing the parallel-distributed graph, approaches and problem are described in section (3). In section (4), the methodology of distributed graphs verification is described. In

^{*} This work has been done within the *iCities* project, funded by European Commission (Future and Emerging Technologies, IST-1999-11337)

section (5) the first effort of partitioning the vertexes using a serialized algorithm is described. In section (6), an algorithm for parallel construction of the graph is described.

2 β -Model of SW Graph

If u and w are vertexes of a graph G , then an edge of the form (u, w) is said to join or connect u and w . An adjacency list is used to represent the graph. The adjacency list contains all vertexes of the graph and next to each vertex is all the vertexes adjacent to it, is called neighbors. A graph in which all vertexes have precisely the same number of edges (k) is called k -regular or just regular. Examples are demonstrated in Fig. 1.

In β -model, the first step is to construct a regular graph. So any vertex v ($1 \leq v \leq N$) is joined to its neighbors, u_i and w_i , as specified by: $u_i = (v-1-i+N)(mod\ N)+1$, AND $w_i = (v-1+i)(mod\ N)+1$. Where $1 \leq i \leq k/2$, and it is generally assumed that $k \geq 2$.

(a)	<table><tr><th>Id</th><th>Neighbors</th></tr><tr><td>1</td><td>2,3,5,6</td></tr><tr><td>2</td><td>1,3,4,6</td></tr><tr><td>3</td><td>1,2,4,5</td></tr><tr><td>4</td><td>2,3,5,6</td></tr><tr><td>5</td><td>3,4,6,1</td></tr><tr><td>6</td><td>1,2,4,5</td></tr></table>	Id	Neighbors	1	2,3,5,6	2	1,3,4,6	3	1,2,4,5	4	2,3,5,6	5	3,4,6,1	6	1,2,4,5
Id	Neighbors														
1	2,3,5,6														
2	1,3,4,6														
3	1,2,4,5														
4	2,3,5,6														
5	3,4,6,1														
6	1,2,4,5														
(b)	<table><tr><th>Id</th><th>Neighbors</th></tr><tr><td>1</td><td>2,6</td></tr><tr><td>2</td><td>1,3</td></tr><tr><td>3</td><td>2,4</td></tr><tr><td>4</td><td>3,5</td></tr><tr><td>5</td><td>4,6</td></tr><tr><td>6</td><td>5,1</td></tr></table>	Id	Neighbors	1	2,6	2	1,3	3	2,4	4	3,5	5	4,6	6	5,1
Id	Neighbors														
1	2,6														
2	1,3														
3	2,4														
4	3,5														
5	4,6														
6	5,1														

Fig. 1. Regular graph representations: (a) $N = 6, k = 4$ and (b) $N = 6, k = 2$

The construction of SW graph passes in two phases. The first is the construction of a regular graph. The second is graph rewiring. Rewiring is done for a number of vertexes selected randomly using probability β . It is accomplished by deleting an edge, and inserting a newly randomly generated edge. The new edge can be the deleted edge but never any existing one. A refinement of the algorithm is to store only the rewired vertexes because neighbors can be easily computed using the vertex id and the value of k . Consequently, the algorithm complexity is $O(k \times N \times \beta)$.

3 The Motivation, Approaches, and Problems

The motivation of this work is to construct SW graphs with very large number of vertexes, on the scale of 1,000,000. We have used distributed parallel processing to cope with scalability issues. There are two approaches. The first is to serialize the rewiring, which does not need to modify the algorithm of the sequential one. The second is to rewire the graph in parallel, which needs to modify the rewiring

algorithm. Implementation wise, the challenge is to have an efficient messages passing between partitions on the scale of $k/2 \times \beta \times N \times (P - 1)/P$, where P is number of partitions, and $(P - 1)/P$ is the probability to cross the process boundary, e.g., $10/2 \times 0.2 \times 1000000 \times 9/10 = 900000$ messages.

Algorithm analysis reveals two problems. Consider the regular graph shown in Fig. 1.(b). If the vertexes 1, and 4 are going to be rewired concurrently, the edges (1, 2) and (4, 5) will be deleted. The possible edges to insert are: $vertex_1 = (1, 2), (1, 3), (1, 4), (1, 5)$; and $vertex_4 = (4, 5), (4, 6), (1, 4), (2, 4)$. The edge (1, 4) can be added by both processes. This creates a graph with a number of edges less than the sequential one (problem 1). Also, using the same regular graph, consider vertexes 3 and 4. The rewiring of 3 may delete the edge (3, 4). This means that while rewiring the vertex 4 in sequence the edge (3, 4) will be considered as one of the possible edges to insert, but this is not the case in parallel rewiring (problem 2).

4 Distributed Graphs Verification

The graphs are verified by: (1) checking that the graph is correctly connected, i.e., for all graph vertexes, if vertex w is a neighbor of u , then u must be a member in neighbors of w , (2) counting the total number of edges, which should not change after rewiring, and (3) statistically, comparing graphs that are constructed sequentially with ones that are distributed using the minimum and maximum degree of vertexes; this is needed because rewiring changes a regular graph to an irregular graph. The results reveal that there is no significant difference between them. It should be remarked that random number generator initialization is a key factor to get correct distributed graphs. Initially, we missed this, which leads to graphs with higher maximum value.

5 Distributed Serialized β -Model of SW Graph

A graph G is partitioned into P partitions with equal size S . The vertexes of the P partitions are: 1 to S , $S+1$ to $2S$, ... and $(P-1) \times S+1$ to $P \times S$. The rewiring is then started in each partition process. Unlike the sequential algorithm, this requires cooperation among processes, which takes the form of remote calls. The parallelization strategy is based on: 1) Detecting the calls that need to cross the process boundary, 2) Collecting those calls in a list, 3) Partitioning the list into sub-lists such that each sub-list contains calls to one process, and 4) Parallel processing of those sub-lists using remote procedure calls (RPC).

This strategy has dual benefits: first, minimizing physical messages sent (RPCs), and second, minimizing thread creation and management. The total number of physical messages is $k/2 \times P \times (P - 1)$. There is only one thread per process that handles the intra-partition rewiring and another thread that handles the inter-partition rewiring.

The rewiring loop is synchronized so that the first loop iteration starts by rewiring the first partition meanwhile; the execution of all other partitions is

suspended. Generally, in the same loop iteration, partition $(x + 1)$ is suspended; until the rewiring of partition (x) is completed ($1 \leq x < P$). The synchronization data structure is constructed from unbound variables that are organized in tuples. The number of tuples equals the number of partitions (P). The arity of the tuple equals $k \text{ div } 2$.

The last part is the edge rewiring code, Fig. 2, which is modified so that procedure calls that need to cross the process boundary are handled using the parallelization strategy. In this way, the distributed rewiring behaves exactly similar to the sequential one. Notice that the time complexity is the same as the sequential algorithm.

The experimental study shows that this approach gives the required scalability. On the scale of 2000000 vertexes, there is no plenty on performance. It even performs slightly better. Statistically, this slight improvement is insignificant. Using efficiently hardware resources may explain these results.

```

proc {RewireEdge Id Index}
  OldNeighbor = (Id + Index) mod N
  Neighbors = {Subtract {GetNeighbors Id} OldNeighbor}
  NewNeighbor = {GenerateRandomNeighbor Id Neighbors} in
  if OldNeighbor \= NewNeighbor then
    {RemoveNeighbor Id OldNeighbor}
    {PutNeighbor Id NewNeighbor}
    {RemoveNeighbor OldNeighbor Id}
    {PutNeighbor NewNeighbor Id}
  end
end
end

```

Fig. 2. Edge rewiring

```

proc {PutNewNeighbor U W}
  if {AlreadyNeighbor U W} then {PutNeighbor U {GenrateNeighbor U}}
  else {PutNeighbor U W} end
end

```

Fig. 3. Adding new neighbor in parallel rewiring

6 Parallel-Distributed Algorithm

It differs from the serialized algorithm only in synchronization. All RPCs are invoked after the rewiring loop is terminated. This means the number of physical messages, directly generated from rewiring, is only $P \times (P - 1)$. The edge rewiring is the same, Fig. 2, but new procedures for adding and removing edges need to be added to handle the two problems that lead to incorrect graphs, which are described above (section 3).

```

proc {RemoveOldNeighbor U W}
  if {Problem2Exist U W} then
    V = {SelectRandom {NewlyAdded U}} in
    {RemoveNeighbor U W} {RemoveNeighbor U V}
    local Z = {GenerateRandomNeighbor U} in
      if Z == W then {PutNeighbor U W}
      else {PutNeighbor U V} end
    end
  end
end

```

Fig. 4. Removing old neighbor in parallel rewiring

The first new procedure is called **PutNewNeighbor/2** (Fig. 3). The existence of w in neighbors of u and the condition $w < u$ detects the first problem and generating a new edge corrects it. Using the same example described in section 4, adding the vertex 1 to the adjacency list of vertex 4 can be detected because the edge already exists and $1 < 4$. So, selecting an edge from the possible ones and adding it, is exactly what will happen in the sequential rewiring. Notice that a new edge may cross the process boundary.

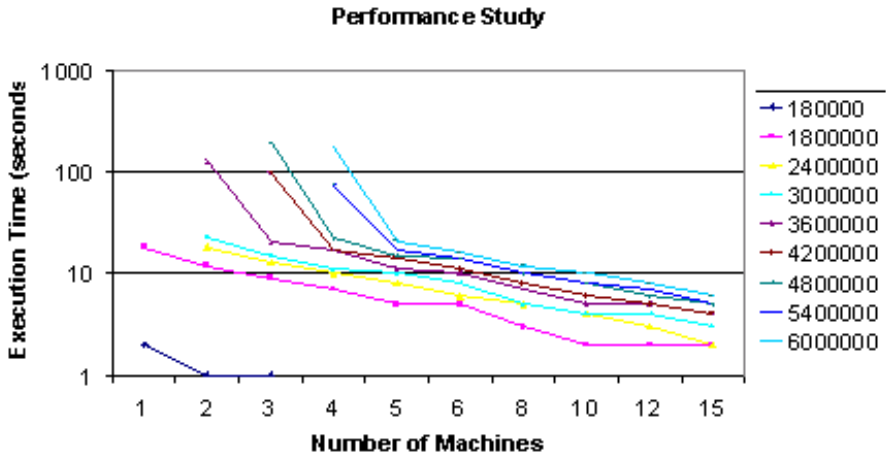


Fig. 5. Performance in relation to the number of vertexes and the number of machines

The second new procedure is called **RemoveOldNeighbor/2** (Fig. 4). If u has been rewired AND $w < u$ AND $u - w \leq k/2$ detects the second problem. The solution is based on selecting randomly one vertex (v) from newly added neighbors. Removing w and v from the neighbors of u , prepares for a state that is identical to the sequential rewiring. Now the value of a randomly generated vertex, z , will determine the next actions. If $z = w$, we need to correct the edges,

else just remove w from the neighbors of u . To correct the edges: delete edge (u, v) and keep the edge (u, w) .

The time complexity is the same as the previous algorithms. The results of an experimental study of performance in relation to the number of vertexes and the number of machines are depicted in the graph of Fig. 5. The performance is measured in seconds. The values shown are the mean values obtained from at least 10 experiments. An experiment is terminated, if it exceeds 300 seconds. The results prove that the developed parallel algorithm gives a reasonable speed up. It also shows that increasing the number of vertexes can be efficiently handled by increasing the number of machines.

7 Conclusion

The β -model of SW graph could be constructed in parallel while distributed in different machines, using two approaches. The first serializes the rewiring, while the second is fully parallel. The key factor is an efficient parallelization technique to cope with the heavy messages load. The parallel implementation gives a very good speed up. The parallelization strategy is based on minimizing the overhead of the number of physical messages sent between processes, and second, minimizing the thread creation and management overhead. A detailed version of this paper can be found at:

www.sics.se/~mahmoud/ICities/papers/ParallelDistributedSmallWorldGraph.pdf.

References

1. Watts, D.J.: Small worlds: the dynamics of networks between order and randomness. Princeton University Press, New Jersey (1999) ISBN 0-691-00541-9.
2. Epstein, J.M., Axtell, R.: Growing artificial societies: social science from the bottom up. The MIT Press, Cambridge, Massachusetts (1996) ISBN 0-262-05053-6.
3. Rafea, M., Holmgren, F., Popov, K., Haridi, S., Lelis, S., Kavassalis, P., Sairamesh, J.: Application architecture of the internet simulation model: Web word of mouth (wom). In: IASTED International Conference on Modelling and Simulation (MS2002), Marina del Rey, USA (2002)
4. Lelis, S., Kavassalis, P., Sairamesh, J., Haridi, S., Holmgren, F., Rafea, M., Hatis-tamatiou, A.: Regularities in the formation and evolution of information cities, the second kyoto meeting on digital cities. In: The Second Kyoto Meeting on Digital Cities, Kyoto Research Park, Kyoto, Japan (2001)
5. Rafea, M., Holmgren, F., Popov, K., Haridi, S., Lelis, S., Kavassalis, P., Sairamesh, J.: Large scale agent-based simulation environment. submitted to Journal of Systems Analysis Modelling Simulation (2002)
6. Mozart Consortium: The Mozart Programming System (2003) <http://www.mozart-oz.org>.

Topic 10

Parallel Programming: Models, Methods, and Programming Languages

José Cunha, Marco Danelutto, Christoph Herrmann, and Peter Welch
Topic Chairs

This topic deals with advances in algorithmic and programming models, design methods, languages, and interfaces which are needed to produce correct, portable parallel software with predictable performance on different parallel and distributed architectures.

Of the 22 papers submitted to this topic, 11 have been accepted as regular and one as distinguished. Two papers delegated from other topics could not be accepted due to the high quality standards of this topic.

The topic has been divided into four sessions of three talks each:

1. Concurrent programming languages

They are dealing with communication mechanisms like channels in Eden (Haskell) and Concurrent ML, or multiple-way receives in the Java extension JR.

2. Functional programming methods and cost models

Functional techniques are applied in the skeleton approaches of the distinguished paper of Double Scan and of tree skeletons, as well as for a kind of parallel composition in BSP/OCaml.

3. Systems and environments

This session presents Java-based grid systems, a software solution for distributed shared memory and an environment for structured parallel programming.

4. Nondeterminism and analytical cost models

Nondeterminism is addressed by order sensitive computations in Parallel Prolog and by an API for highly-irregular computations in C++. One paper discusses the relation between theoretical and practical performance analysis.

The distinguished paper by Bischof, Gorlatch and Kitzelmann presents a composition of their Double-Scan skeleton (generalization of tridiagonal system solving) by blockwise combinators, which permits a parallel implementation of linear cost, i.e., cost-optimal complexity.

Acknowledgements. The topic chairs would like to thank the external reviewers for their help. Special thanks go to Harald Kosch for his frequent, extremely ambitious and always friendly support and advice.

Cost Optimality and Predictability of Parallel Programming with Skeletons

Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann

Technical University of Berlin, Germany
{bischof|gorlatch|jemanuel}@cs.tu-berlin.de

Abstract. Skeletons are reusable, parameterized components with well-defined semantics and pre-packaged efficient parallel implementation. This paper develops a new, provably cost-optimal implementation of the DS (*double-scan*) skeleton for the divide-and-conquer paradigm. Our implementation is based on a novel data structure called *plist* (pointed list); implementation's performance is estimated using an analytical model. We demonstrate the use of the DS skeleton for parallelizing a tridiagonal system solver and report experimental results for its MPI implementation on a Cray T3E and a Linux cluster: they confirm the performance improvement achieved by the cost-optimal implementation and demonstrate its good predictability by our performance model.

1 Introduction

A promising approach to improve and systematize the parallel programming process is to use well-defined patterns of parallelism, called *skeletons* [4]. The programmer expresses an application using skeletons as reusable, parameterized components, whose efficient implementations for particular parallel machines are provided by a compiler or library. A rich collection of both very basic and complex parallel skeletons have been proposed and implemented in systems like P3L, HDC, Skil, SKELib etc. [3,6,7,11], and have a potential to be actively used in future problem solving environments [5].

This paper addresses two questions that have a major impact on the choice and use of skeletons in the programming process: (1) whether the parallel cost of a particular skeleton implementation, i.e. its time-processor product [15], is optimal, and (2) whether the performance of a skeleton-based program can be predicted in advance early on in the design process. We answer these questions for two fairly complicated skeletons introduced by ourselves earlier, both embodying the divide-and-conquer paradigm: DH (distributable homomorphism) [8] and DS (double-scan) [1].

The contributions and organization of the paper are as follows:

- We introduce basic data-parallel skeletons and our two skeletons DH and DS, and prove conditions for the generic parallel implementations of both DH and DS to be cost-optimal (Section 2).

- We introduce a novel data structure called *plists* (*pointed lists*), develop a new parallel DS-implementation that uses plists internally, estimate the runtime of this implementation and prove its cost optimality (Section 3).
- We report experimental results for a tridiagonal system solver based on the DS-skeleton on a Cray T3E and a Linux cluster using MPI. They demonstrate both a substantial performance improvement achieved by our cost-optimal implementation and the high quality of the analytical runtime prediction (Section 4).

We conclude the paper by discussing our results in the context of related work.

2 Skeletons and Their Cost Properties

A skeleton can be formally viewed as a higher-order function, customizable for a particular application by means of functional parameters that are provided by the application programmer. The first parallel skeletons studied in the literature were traditional second-order functions known from functional programming: map, reduce, scan, etc. They are defined on non-empty lists as follows, function application being denoted by juxtaposition, i.e. $f x$ stands for $f(x)$:

- Map: Applying a unary function f to all elements of a list:

$$\text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n]$$

- Zip: Component-wise application of a binary operator $\oplus$ to a pair of lists of equal length (it is similar to the Haskell function `zipWith`):

$$\text{zip}(\oplus)([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1 \oplus y_1), \dots, (x_n \oplus y_n)]$$

- Scan-left and scan-right: Computing prefix sums of a list by traversing the list from left to right (or vice versa) and applying a binary operator $\oplus$:

$$\text{scanl}(\oplus)([x_1, \dots, x_n]) = [x_1, (x_1 \oplus x_2), \dots, (\dots(x_1 \oplus x_2) \oplus x_3) \oplus \dots \oplus x_n]$$

$$\text{scanr}(\oplus)([x_1, \dots, x_n]) = [(x_1 \oplus \dots \oplus (x_{n-2} \oplus (x_{n-1} \oplus x_n) \dots)), \dots, x_n]$$

We call these second-order functions “skeletons” (or patterns) because each of them describes a whole class of functions, obtainable by substituting application-specific operators for parameters $\oplus$ and f .

Our basic skeletons have obvious data-parallel semantics: the asymptotic parallel complexity is constant for *map* and *zip* and logarithmic for both scans if $\oplus$ is associative. If $\oplus$ is non-associative, then the scans have to be computed sequentially with linear time complexity.

The need to manage important classes of applications led to the introduction of more complex skeletons, e.g. different variants of divide-and-conquer, etc. In [8], we defined one such skeleton, the DH (distributable homomorphism):

Definition 1. *The DH skeleton is a higher-order function with two parameter operators, $\oplus$ and $\otimes$, defined as follows for arbitrary lists x and y of equal length, which is a power of two:*

$$\begin{aligned} dh(\oplus, \otimes)[a] &= [a], \\ dh(\oplus, \otimes)(x \mathbin{++} y) &= zip(\oplus)(dh\ x, dh\ y) \mathbin{++} zip(\otimes)(dh\ x, dh\ y) \end{aligned} \quad (1)$$

The DH skeleton is a special form of the well-known divide-and-conquer paradigm: to compute dh on a concatenation of two lists, $x \mathbin{++} y$, we apply dh to x and y , then combine the results elementwise using zip with the parameter operators $\oplus$ and $\otimes$ and concatenate them. For this skeleton, there exists a family of generic parallel implementations, directly expressible in MPI [9].

In [1], we introduced a more special skeleton, *double-scan* (DS), which is more convenient than DH for expressing some application classes:

Definition 2. *For binary operators $\oplus$ and $\otimes$, two double-scan (DS) skeletons are defined:*

$$scanrl(\oplus, \otimes) = scanr(\oplus) \circ scanl(\otimes) \quad (2)$$

$$scanlr(\oplus, \otimes) = scanl(\oplus) \circ scanr(\otimes) \quad (3)$$

where $\circ$ denotes function composition from right to left, i. e. $(f \circ g)\ x = f(g(x))$. Both double-scan skeletons have two functional parameters, which are the base operators of their constituent scans. If the parameter operator $\oplus$ is associative, then as shown in [1], DS can be parallelized.

An important efficiency criterion for parallel algorithms is their *cost*, which is defined as the product of the required time and the number of processors used, i. e. $c = t \cdot p$ (see [15]). A parallel implementation is called *cost-optimal* on p processors, iff its cost equals the cost on one processor, i. e. $p \cdot t_p \in \Theta(t_{seq})$. Here and in the rest of this paper, we use the classical notations o , O , ω and Θ to describe asymptotic behaviour (see, e. g., [12]).

Our first question is whether the known implementations of DH [8] and DS [1] are cost-optimal. We are especially interested in a skeleton's generic implementation, which can be applied to all particular instances of the skeleton.

The *generic DH-implementation* partitions the input list into p blocks of approximately the same size, computes DH locally in the processors, and then performs $\log_2 p$ rounds of pairwise communications and computations in a hypercube-like manner [8]. Its time complexity is $\Theta(t_{seq}(m) + m \cdot \log p)$, where $t_{seq}(m)$ is the time taken to sequentially compute DH on a block of length $m \approx n/p$. There always exists an obvious sequential implementation of the formula (1) on a data block of size m , which has a time complexity of $\Theta(m \cdot \log m)$. So, in the general case, the time complexity of the generic DH-implementation is $\Theta(n/p \cdot \max\{\log(n/p), \log p\})$.

The sequential time complexity of a particular DH instance may also be asymptotically smaller than $m \cdot \log m$. An example relevant for us is the DS skeleton. On the one hand, as proved in [1], the DS skeleton can be expressed as

an instance of the DH skeleton for particular values of parameters $\oplus$ and $\otimes$, with additional pre- and postcomputations performed locally. On the other hand, the sequential time complexity of DS is obviously linear.

The following theorem shows how the cost optimality of the generic DH-implementation, used for a particular instance of the DH-skeleton, depends on the optimal sequential complexity of this instance, i. e. particular application:

Theorem 1. *The generic parallel implementation of DH, when used for a DH instance with optimal sequential time complexity of t_{seq} , is*
 (a) *cost-optimal on $p \in O(n)$ processors, if $t_{seq}(n) \in \Theta(n \cdot \log n)$, and*
 (b) *non-cost-optimal on $p \in \omega(1)$ processors, if $t_{seq}(n) \in o(n \cdot \log p)$.*

Here is the proof sketch: (a) $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq O(n \cdot \log n)$, and
 (b) $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq \Omega(n \cdot \log p)$.

From Theorem 1(b) it follows that the generic DH-implementation is not cost-optimal for the DS skeleton, whose $t_{seq}(n) \in \Theta(n)$. Our next question is whether there exists a cost-optimal implementation of the DS skeleton? Since there are applications that are instances of the DS skeleton and that still have cost-optimal hand-coded implementations, we can strive to find a generic, cost-optimal solution for all instances. This motivates our further search for a better parallel implementation of DS.

3 Towards a Cost-Optimal Double-Scan

In this central section of the paper, we develop a new generic, cost-optimal implementation of the DS skeleton. The implementation makes internal use of a novel special data structure, which, however, remains invisible to the programmer using the skeleton.

3.1 Plists and Functions on Them

We introduce a special intermediate data structure – *pointed lists (plists)*. A k -*plist*, where $k > 0$, consists of k conventional lists, called segments, and $k - 1$ points between the segments:

$$\underbrace{l_1} \quad \underbrace{a_1 \quad l_2}_{\bullet} \quad \underbrace{a_2 \quad l_3}_{\bullet} \quad \underbrace{a_3 \quad \dots \quad a_{k-1} \quad l_k}_{\bullet}$$

If parameter k is irrelevant, we simply speak of a plist instead of a k -plist. Conventional lists are obviously a special case of plists. To distinguish between functions on lists and plists, we prefix the latter with the letter p , e. g. $pmap$. To transform between lists and plists, we use the following two functions:

- $list2plist_k$ transforms a list into a plist, consisting of k segments and $k - 1$ points. It partitions an arbitrary list into k segments:

$$list2plist_k(l_1 \uparrow [a_1] \uparrow \dots \uparrow [a_{k-1}] \uparrow l_k) = [l_1, a_1, \dots, a_{k-1}, l_k]$$

Our further considerations are valid for arbitrary partitions but, in practice of parallelism, one tries to obtain segments of approximately the same size.

- $plist2list_k$ is the inverse of $list2plist_k$, transforming a k -plist into a conventional list:

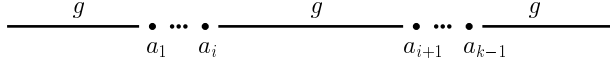
$$plist2list([l_1, a_1, \dots, a_{k-1}, l_k]) = l_1 \uplus [a_1] \uplus \dots \uplus [a_{k-1}] \uplus l_k$$

We now develop a parallel implementation for a distributed version of *scanrl*, function *pscanrl*, which computes *scanrl* on a plist:

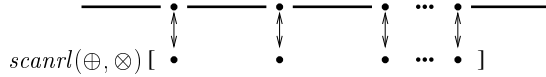
$$scanrl(\textcircled{1}, \textcircled{2}) = plist2list \circ pscanrl(\textcircled{1}, \textcircled{2}) \circ list2plist_k$$

We introduce the following auxiliary skeletons as higher-order functions on plists, omitting their formal definitions and illustrating them instead graphically:

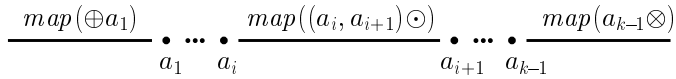
- $pmap\_l\ g$ applies function g , which operates on conventional lists, to all segments of a plist:



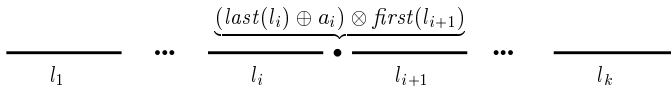
- $pscanrl\_p(\oplus, \otimes)$ applies function $scanrl(\oplus, \otimes)$, defined in (2), to the list containing only the points of the argument plist:



- $pinmap\_l(\odot, \oplus, \otimes)$ modifies each segment of a plist depending on the segment's neighbouring points, using operation $\oplus$ for the left-most segment, $\otimes$ for the right-most segment and operation $\odot$ for all inner segments, where operation $\odot$ is a three-adic operator, i. e. it gets a pair of points and a single point as parameters:



- $pinmap\_p(\oplus, \otimes)$ modifies each single point of a plist depending on the last element of the point's left neighbouring segment and the first element of the point's right neighbouring segment.



The implementations and time complexity analysis of these functions are provided in the next section.

3.2 A Cost-Optimal Implementation of Double-Scan

In this section, we present a theorem that shows that the distributed version of the double-scan skeleton can be expressed using the auxiliary skeletons introduced above. We use here the following definition: a binary operation $\circledast$ is called to be *associative modulo* $\odot$, iff for arbitrary elements a, b, c it holds: $(a \circledast b) \circledast c = (a \odot b) \circledast (b \circledast c)$. Usual associativity is associativity modulo operation *first*, which yields the first element of a pair.

Theorem 2. *Let $\textcircled{1}$, $\textcircled{2}$, $\textcircled{3}$ and $\textcircled{4}$ be binary operators, such that $\textcircled{1}$ and $\textcircled{3}$ are associative, $\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{scanrl}(\textcircled{3}, \textcircled{4})$, and $\textcircled{2}$ is associative modulo $\textcircled{4}$. Moreover, let $\textcircled{5}$ be a three-adic operator, such that $(a, a \textcircled{3} c) \textcircled{5} b = a \textcircled{3} (b \textcircled{1} c)$. Then, the double-scan skeleton pscanrl on plists can be implemented as follows:*

$$\begin{aligned} \text{pscanrl}(\textcircled{1}, \textcircled{2}) &= \text{pinmap\_l}(\textcircled{5}, \textcircled{1}, \textcircled{3}) \circ \text{pscanrl\_p}(\textcircled{1}, \textcircled{2}) \\ &\quad \circ \text{pinmap\_p}(\textcircled{2}, \textcircled{4}) \circ (\text{pmap\_l scanrl}(\textcircled{1}, \textcircled{2})) \end{aligned} \quad (4)$$

For the theorem's proof, see [2]. To help the user in finding the operator $\textcircled{5}$, we show in [2] how $\textcircled{5}$ can be generated if $(a \textcircled{3})$ is bijective for arbitrary a and if $\textcircled{3}$ distributes over $\textcircled{1}$. Since $\text{pscanrl}(\textcircled{1}, \textcircled{2}) = \text{pscanrl}(\textcircled{3}, \textcircled{4})$, equality (4) holds also for $\text{pscanrl}(\textcircled{3}, \textcircled{4})$.

Let us analyze the pscanrl implementation (4) provided by Theorem 2. On a parallel machine, we partition plists so that each segment and its right “border point” are mapped to a processor. The last processor contains no extra point because there is no point to the right of the last segment in a plist. We further assume that all segments are of approximately the same size.

The right-hand side of (4) consists of four stages executed from right to left, whose parallel time complexity we now study, relying on the graphical representation in Section 3.1:

1. The function $\text{pmap\_l scanrl}(\textcircled{1}, \textcircled{2})$ can be computed by simultaneously applying $\text{scanrl}(\textcircled{1}, \textcircled{2})$ on all processors, provided that the argument plist is partitioned among p processors as described above. Because $\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{scanrl}(\textcircled{3}, \textcircled{4})$, we can apply either of both on all processors, so the minimum of their runtimes should be taken. Thus, the time complexity is $T_1 = (n/p - 2) \cdot \min\{t_{\textcircled{1}} + t_{\textcircled{2}}, t_{\textcircled{3}} + t_{\textcircled{4}}\} \in O(n/p)$, where $t_{\textcircled{1}}, \dots, t_{\textcircled{4}}$ denote the time for one computation with operators $\textcircled{1}, \dots, \textcircled{4}$, respectively.
2. To compute $\text{pinmap\_p}(\textcircled{2}, \textcircled{4})$, each processor sends its first element to the preceding processor and receives the first element from the next processor. Then operations $\textcircled{2}$ and $\textcircled{4}$ are applied to the last element of each processor:

```
/* first buffers 1st element of each processor */
/* first_next receives 1st element of following processor */
MPI_Sendrecv(first, preceding, ..., first_next, following, ...);
if ( !last_processor ) {
    otwo(last, point);
    ofour(point, first_next);
    /* last denotes the last element of each segment */ }
```

The resulting time complexity is $T_2 = t_s + t_w + t_{\textcircled{2}} + t_{\textcircled{4}} \in O(1)$, where t_s is the communication startup time and t_w the time needed to communicate one element of the corresponding datatype.

3. As described in Section 3.1, $p\text{scanrl\_}p(\textcircled{1}, \textcircled{2})$ applies $\text{scanrl}(\textcircled{1}, \textcircled{2})$ on the points of the argument plist, which are distributed across the first $p - 1$ processors. Since DS is an instance of the DH skeleton, the generic DH implementation developed in [8] can be used for this step:

```
for (dim=1; dim<p-1; dim<=<=1){
  neighbour = my_rank^dim;
  MPI_Sendrecv(data, neighbour, ..., tmp, neighbour, ...);
  if (my_rank < neighbour) oplus( tmp, data);
  else otimes( tmp, data);}

```

The shown code is a sequence of $\log_2 p$ swaps iterating over the dimensions of the virtual hypercube. A swap consists of pairwise, two-directional communications between neighbouring nodes, followed by a computation in each processor. It takes time $T_3 = \log_2(p-1) \cdot (t_s + 3t_w + t_{\oplus/\otimes}) \in O(\log p)$, $t_{\oplus/\otimes}$ denoting the time for one computation with operator $\oplus$ or $\otimes$ of the DH-skeleton. Operators $\oplus$ and $\otimes$ are defined on triples of values using $\textcircled{1}, \dots, \textcircled{4}$ (see [1]).

4. To compute $\text{pinmap\_}l(\textcircled{5}, \textcircled{1}, \textcircled{3})$, each processor sends its last element to the next processor. Then operation $\textcircled{5}$ is applied to the elements of the “inner” processors. The elements of the first processor are manipulated by $\textcircled{1}$, and the elements of the last processor by $\textcircled{3}$:

```
/* last_prec receives last element of preceding processor */
MPI_Sendrecv(last, following, ..., last_prec, preceding, ...);

/* elements denotes the datablock of each processor */
if ( first_processor )           // oone will be applied
  for (i=0; i<(m-1); i++)
    oone(elements[i], last);
else if ( last_processor )       // othree will be applied
  for (i=0; i<(m-1); i++)
    othree(last_prec, elements[i]);
else                             // ofive will be applied
  for (i=0; i<(m-1); i++)
    ofive(last_prec, last, elements[i]);

```

The computations in the processors are mutually independent, which results in a time complexity of $T_4 = t_s + t_w + (n/p - 1) \cdot t_{\textcircled{5}} \in O(n/p)$, where $t_{\textcircled{5}}$ denotes the time required by $\textcircled{5}$.

To obtain the overall time complexity of the implementation (4), we sum up the times of the four stages discussed above:

$$\begin{aligned}
t &= T_1 + T_2 + T_3 + T_4 \\
&\approx (n/p-1) \cdot (\min\{t_{\textcircled{1}} + t_{\textcircled{2}}, t_{\textcircled{3}} + t_{\textcircled{4}}\} + t_{\textcircled{5}}) + (2 + \log_2(p-1)) \cdot (t_s + 3t_w + t_{\oplus/\otimes}) \\
&\in \Theta(n/p + \log p)
\end{aligned} \tag{5}$$

Recall that, as shown in Section 2, the time complexity of the generic DH-implementation is $\Theta(n/p \cdot \max\{\log(n/p), \log p\})$, so we have substantially improved the asymptotic complexity.

The cost of our new DS-implementation is obviously $\Theta(n + p \cdot \log p)$, which by choosing an appropriate value of p can be made equal to the (linear) cost of DS in the sequential case. Thus, we have proved the following proposition:

Proposition 1. *The parallel implementation (4) of the double-scan skeleton is cost-optimal if $p \in O(n/\log n)$ processors are used.*

Though in principle a very important characteristic of the quality of parallel implementation, cost optimality characterizes asymptotic behaviour. In practice, the actual performance for a particular application and particular machine are very important. This will be our topic in the next section.

4 Performance Prediction and Case Study

Programming with skeletons offers a major advantage in terms of performance prediction: performance has to be estimated once for a skeleton on a target architecture. In this section, we study how the performance of an application using DS can be predicted based on the estimates for our cost-optimal DS-implementation. This is done by tuning the generic estimate obtained in the previous section to a particular machine and/or application. The order of tuning steps can be chosen depending on the specific goals of the application programmer.

We start with the generic performance estimate for the cost-optimal DS (5) and tune it first to two particular machines, a Cray T3E and a Linux cluster, and then to a particular application case study (tridiagonal system solver).

4.1 Tuning Estimates to Particular Machines

Variables t_s and t_w in (5) are machine-dependent. On a Cray T3E, t_s is approximately $16.4 \mu s$. The value of t_w also depends on the size of the data type: $t_w = d \cdot t_B$, where t_B is the time needed to communicate one byte and d is the byte count of the data type. Measurements show that for large array sizes the bidirectional bandwidth on our machine is approximately 300 MB/s, i.e. $t_B \approx 0.0033 \mu s$. Inserting these values of t_s and t_B into (5), we obtain the following runtime estimate, t_{Cray} , for the double-scan skeleton on a Cray T3E:

$$t_{\text{Cray}} = (n/p - 1) \cdot (\min\{t_{\textcircled{1}} + t_{\textcircled{2}}, t_{\textcircled{3}} + t_{\textcircled{4}}\} + t_{\textcircled{5}}) + (2 + \log_2(p-1)) \cdot (16.4 \mu s + d \cdot 0.0099 \mu s + t_{\oplus/\otimes}) \quad (6)$$

On a Linux Pentium IV cluster with SCI interconnect we measured $t_s \approx 25 \mu s$ and $t_B \approx 0.0123 \mu s$ for large data sizes. Substituting these values into (5), we obtain the following runtime estimate, t_{Cluster} , for the double-scan skeleton on a Linux cluster:

$$t_{\text{Cluster}} = (n/p - 1) \cdot (\min\{t_{\textcircled{1}} + t_{\textcircled{2}}, t_{\textcircled{3}} + t_{\textcircled{4}}\} + t_{\textcircled{5}}) + (2 + \log_2(p-1)) \cdot (25 \mu s + d \cdot 0.0369 \mu s + t_{\oplus/\otimes}) \quad (7)$$

4.2 Tuning Estimates to an Application

By way of an example application, we consider the solution of tridiagonal system of equations, described in detail in [1]. The given system is represented as a list of rows consisting of four values: the value on the main, upper and lower diagonal and the value of the right-hand-side vector. A typical sequential algorithm for solving a tridiagonal system is Gaussian elimination (see e.g. [15,12]) which eliminates the lower and upper diagonal of the matrix in two steps: (1) eliminates the lower diagonal by traversing the matrix from top to bottom according to the *scanl* skeleton using an operator $\textcircled{2}_{tds}$; (2) eliminates the upper diagonal of the matrix by a bottom-up traversal, i.e. using the *scanr* skeleton with an operator $\textcircled{1}_{tds}$. We can alternatively eliminate first the upper and then the lower diagonal using two other row operators, $\textcircled{3}_{tds}$ and $\textcircled{4}_{tds}$, see [1] for a formal definition.

Being a composition of two scans, the tridiagonal system solver is a natural candidate to be treated as an instance of the DS skeleton. In [2] we proved that the conditions of Theorem 2 are indeed satisfied, so that our new cost-optimal DS-implementation can be used for the tridiagonal solver. For the solver, implementation (4) operates on lists of rows, which are quadruples of double values, thus having a size of $d = 32$ Bytes. We have measured the operations $\textcircled{1}_{tds}, \dots, \textcircled{5}_{tds}, \oplus, \otimes$ of the tridiagonal system solver by executing the operations in a loop over a large array. Operators $\oplus$ and $\otimes$, presented in [2], are the DH operators, defined using $\textcircled{1}, \dots, \textcircled{4}$. Measurement results are presented in Table 1.

Table 1. Measured times in μs for operations of the tridiagonal system solver

architecture	$t_{\textcircled{1}}$	$t_{\textcircled{2}}$	$t_{\textcircled{3}}$	$t_{\textcircled{4}}$	$t_{\textcircled{5}}$	$t_{\oplus/\otimes}$
Cray T3E	0.24	0.36	0.37	0.27	0.43	2.44
Cluster	0.10	0.15	0.15	0.10	0.14	0.73

Inserting operations' times and the value of d into (6) and (7), we obtain:

$$\begin{aligned} t_{\text{Cray-tds}} &= n/p \cdot 1.03 \mu s + \log_2(p-1) \cdot 19.16 \mu s + 37.28 \mu s \\ t_{\text{Cluster-tds}} &= n/p \cdot 0.39 \mu s + \log_2(p-1) \cdot 26.91 \mu s + 53.43 \mu s \end{aligned} \quad (8)$$

The next tuning step is to substitute a particular problem size n into (8): e. g. by substituting $n = 2^{19} \approx 5 \cdot 10^5$, we obtain the following runtime estimate depending on the number of processors:

$$\begin{aligned} T_{\text{Cray-tds-}n=2^{19}} &= 1/p \cdot 0.540 \text{ s} + \log_2(p-1) \cdot 19.16 \mu\text{s} + 37.28 \mu\text{s} \\ T_{\text{Cluster-tds-}n=2^{19}} &= 1/p \cdot 0.204 \text{ s} + \log_2(p-1) \cdot 26.91 \mu\text{s} + 53.43 \mu\text{s} \end{aligned} \quad (9)$$

Another tuning sequence is to fix the number of processors and vary the problem size. E.g., the estimate for the runtime on 17 processors depending on the problem size is:

$$\begin{aligned} T_{\text{Cray-tds-}p=17} &= n \cdot 0.061 \mu\text{s} + 113.92 \mu\text{s} \\ T_{\text{Cluster-tds-}p=17} &= n \cdot 0.023 \mu\text{s} + 161.07 \mu\text{s} \end{aligned} \quad (10)$$

Here, we assume that one MPI process runs on a processor of a parallel machine.

4.3 Experimental Results

We conducted experiments with the tridiagonal system solver on two machines: (1) a Cray T3E machine with 24 processors of type Alpha 21164, 300 MHz, 128 MB, using native MPI implementation and (2) a Linux cluster with 16 nodes, each consisting of two processors of type Pentium IV, 1.7 GHz, 1 GB, using an SCI port of MPICH.

We are interested in two questions: (1) What is the performance improvement achieved by our cost-optimal implementation as compared with the generic, non-cost-optimal implementation? (2) How well is the performance of the implementation predicted by our analytical model, in particular by formulae (9)–(10)?

The first question is answered by Fig. 1. The cost-optimal solution demonstrates a substantial time improvement compared with the generic, non-cost-optimal version: up to 13 on 17 processors of the Cray T3E and up to 18 on 9 processors of the cluster. Note that we use a logarithmic scale for the time axis.

To answer the second question, we compare in Fig. 1 the predicted runtime using (9) with the measured times for a fixed problem size. The quality of prediction is quite good: the difference to the measured times is less than 12% on both machines.

The number of processors in our generic DS implementation is $p = 2^k + 1$ for some k , because the third step of algorithm (4), $p\text{scanrl\_}p(\textcircled{1}, \textcircled{2})$, applies the generic DH implementation to $p - 1$ points of the argument plist and $p - 1$ is always a power of 2 in the generic DH implementation.

Another interesting point is the absolute speedup, i. e. the parallel version compared to the optimal sequential implementation. Our estimates yield the speedup of approximately $0.6 p$, which has been confirmed by our measurements.

5 Related Work and Conclusions

Our main contribution is a new, cost-optimal parallel implementation of the DS-skeleton, which is directly applicable for the whole class of applications that

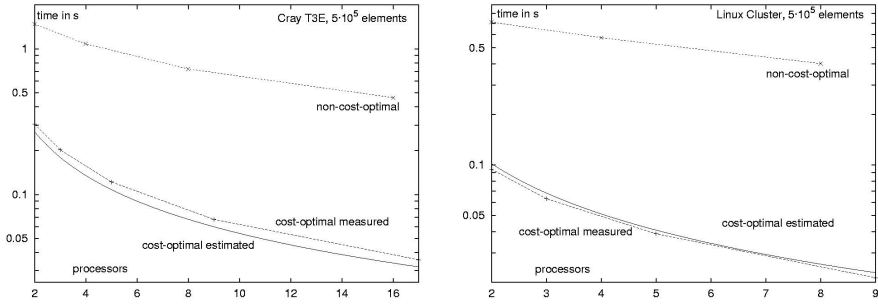


Fig. 1. Runtimes of the tridiagonal solver: cost-optimal vs. non-cost-optimal implementation vs. predicted runtime. Left: on the Cray T3E; Right: on the Linux cluster.

(or their parts) are instances of DS. We have expressed the implementation in MPI to make it portable over different parallel architectures and exploitable in practical skeleton-based programming systems. The cost optimality guarantees not only good runtime but also economical use of processors. We have also formulated a general condition for the cost optimality of the generic implementation of a more general DH skeleton. Cost optimality of divide-and-conquer algorithms was studied earlier in the context of the HDC system [10].

Our implementation makes internal use of the data structure called *plist*. The *plist* data structure is new, to the best of our knowledge. A closer look reveals that *plists* are present, albeit implicitly, in some parallel algorithms: if a block distribution is used, the border points of the blocks being handled specifically, then the data structure used can often be seen as a *plist*. Our contribution in this respect is in defining and treating explicitly a data structure that traditionally has remained hidden in the design of algorithms.

O'Donnell presented in [14] a bidirectional scan that combines a left and a right scan, independent of each other. By contrast, our DS describes a composition of scans, where the result of the first scan is used by the second scan.

An important advantage of the skeleton approach is good predictability of performance early in the design process. We have demonstrated that the performance of a particular application can be predicted by stepwise tuning of the generic performance estimate of the DS skeleton to a particular target machine and problem size. We have verified experimentally a quite good prediction quality: the error is less than 12%.

The parallelization of our case study – the tridiagonal system solver – is known to be a non-trivial task owing to the sparse structure and restricted amount of potential concurrency [12,13]. It is interesting to observe that our generic DS-implementation, when customized for the tridiagonal solver, is very similar to the implementation by Wang and Mou [17], based on Wang's algorithm [16], which is today probably the solution most widely used in practice. The good speedup of the obtained solution is confirmed by experiments on a Cray T3E and a Linux cluster.

Acknowledgements. We are grateful to the referees, and especially to Christoph A. Herrmann, for improving the original manuscript, to Roman Leshchinskiy and Martin Alt for many fruitful discussions, and to Phil Bacon who helped a great deal in brushing up the presentation.

References

1. Bischof, H., Gorlatch, S.: Double-scan: Introducing and implementing a new data-parallel skeleton. In Monien, B., Feldmann, R., eds.: Euro-Par 2002. Volume 2400 of LNCS., Springer (2002) 640–647
2. Bischof, H., Gorlatch, S., Kitzelmann, E.: The double-scan skeleton and its parallelization. Technical Report 2002/06, Technische Universität Berlin (2002)
3. Botorog, G., Kuchen, H.: Efficient parallel programming with algorithmic skeletons. In Bougé, L., et al., eds.: Euro-Par'96: Parallel Processing. Lecture Notes in Computer Science 1123. Springer-Verlag (1996) 718–731
4. Cole, M.I.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. PhD thesis, University of Edinburgh (1988)
5. Cunha, J.C.: Future generations of problem solving environments. In: Proceedings of the IFIP WG 2.5 Conference on the Architecture of Scientific Software. (2000)
6. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for data parallelism in P3L. In Lengauer, C., Griebel, M., Gorlatch, S., eds.: Euro-Par'97. Volume 1300 of LNCS., Springer (1997) 619–628
7. Danelutto, M., Stigliani, M.: SKELib: parallel programming with skeletons in C. In Bode, A., Ludwig, T., Karl, W., Wismüller, R., eds.: EuroPar 2000. Volume 1900 of LNCS., Springer (2000) 1175–1184
8. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y., eds.: Euro-Par'96: Parallel Processing, Vol. II. Lecture Notes in Computer Science 1124. Springer-Verlag (1996) 401–408
9. Gorlatch, S., Lengauer, C.: Abstraction and performance in the design of parallel programs: overview of the SAT approach. *Acta Informatica* **36** (2000) 761–803
10. Herrmann, C.A.: The Skeleton-Based Parallelization of Divide-and-conquer Recursions. PhD thesis (2000) ISBN 3-89722-556-5.
11. Herrmann, C.A., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters* **10** (2000) 239–250
12. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publ. (1992)
13. López, J., Zapata, E.L.: Unified architecture for divide and conquer based tridiagonal system solvers. *IEEE Transactions on Computers* **43** (1994) 1413–1424
14. O'Donnell, J.: Bidirectional fold and scan. In: Functional Programming: Glasgow 1993. Workshops in Computing (1993) 193–200
15. Quinn, M.J.: Parallel Computing. McGraw-Hill, Inc. (1994)
16. Wang, H.H.: A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software* **7** (1982) 170–183
17. Wang, X., Mou, Z.: A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In: Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press (1991) 810–816

A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms

K. Villaverde¹, E. Pontelli¹, H-F. Guo², and G. Gupta³

¹ Dept. Computer Science, New Mexico State University

² Dept. Computer Science, University of Nebraska at Omaha

³ Dept. Computer Science, University of Texas at Dallas

Abstract. We propose a novel methodology, based on stack splitting, to efficiently support order-sensitive computations (e.g., I/O, side-effects) during search-parallel execution of non-deterministic languages on Beowulf platforms. The methodology has been validated in the context of the PALS Prolog system and results on a Pentium Beowulf are discussed.

1 Introduction

Parallelism has been identified as one of the most promising avenues to provide efficient execution of *non-deterministic* languages [4]. The essence of this class of systems—non-determinism—provides a natural approach for the automatic exploitation of parallelism: non-determinism implies the presence of multiple alternatives that have to be explored in the computation—such alternatives can be explored in parallel. Our goal is to explore methodologies to support the exploitation of search-parallelism from non-deterministic languages on Beowulf platforms. Although a number of schemes to handle search parallelism have been studied [4], only a few of them meet the following properties [11]: (1) they do not require shared data structures; (2) they allow the use of effective scheduling strategies; (3) they are applicable across different classes of languages. In our previous research [3,11] we proposed a general methodology, called *Stack Splitting*, which meets these criteria. Stack splitting is an evolution of stack-copying [1,8] and it has been practically validated in the context of *PALS*—the first Prolog on Beowulfs—and in non-monotonic reasoning systems [6].

Sequential non-deterministic systems frequently include features that allow the programmer to introduce a component of sequentiality in the execution. These may be in the form of facilities to express side-effects (e.g., I/O) or constructs to control the order of construction of the computation (e.g., pruning operations, user-defined search strategies). Such *Order Sensitive Components (OSC)* need to be performed in the *same order as in a sequential execution*; if this requirement is not met, the parallel computation may reproduce a semantics different from the one indicated by the programmer [4]. A correct parallel execution of a program containing *OSC* will reduce the amount of parallelism

available—parts of the program have to be sequentialized. Nevertheless, programs with moderate number of *OSC* still provide sufficient parallel work to keep processors busy. Existing proposals on handling *OSC* are either inefficient [7] or inadequate to interact with advanced methodologies such as stack splitting. We extend stack splitting to handle *OSC*. The solution can be efficiently implemented and requires less communication than the other schemes.

Order-sensitive Computations and Prolog: A parallel Prolog system that maintains Prolog semantics reproduces the behavior of a sequential system (same solutions, in the same order, and with the same termination properties). In the context of Prolog, there are three different classes of *OSC*: *side-effects* predicates (e.g., I/O), *meta-logical* predicates (e.g., test the instantiation state of variables), and *control* predicates (e.g., for pruning branches of the search tree). In the context of search-parallelism only certain classes of *OSC* require sequentialization across parallel computations—only side-effects and control predicates. The presence of *OSC* does not require a sequentialization of the whole execution involved, only the *OSC* themselves need to be sequentialized. If the *OSC* are infrequent and spaced apart, good speedups can be obtained, even in a Distributed Memory Platform (DMP). The correct order of execution of *OSC* corresponds to an in-order traversal of the computation tree. A specific *OSC* α can be executed only if all the *OSC* that precede α in the traversal have been completed. Detecting when all the *OSC* to the left have been executed is an undecidable problem, thus requiring the use of approximations. The most commonly used approximation is to execute an *OSC* only when the branch containing it becomes the left-most branch in the tree [5]. Thus, we approximate the termination of the preceding *OSC* by verifying the termination of the *branches* that contain them. Most of the schemes proposed [7,4] rely on traversals of the tree, where the computation attempting an *OSC* walks up its branch verifying the termination of all the branches to its left. These approaches can be realized [5,1,9] in presence of a shared representation of the computation tree—required to check the status of other executions without communication. These solutions do not scale to the case of DMP, where a shared representation of the computation tree is not available. Simulation of a shared representation is infeasible, as it leads to unacceptable bottlenecks [10]. Some attempts to generalize mechanisms to handle *OSC* to DMPs have been made [2], but only at the cost of sub-optimal scheduling mechanisms. It is unavoidable to introduce a communication component to handle *OSC* in a distributed setting. We demonstrate that stack splitting can be modified to solve this problem with minimal communication. The modification is inspired by the optimal algorithms for *OSC* studied in [7].

Stack Splitting: In the traditional stack-copying technique [1,8], each choice-point explored by multiple processors has to be *shared*—i.e., transferred to a common area accessed in mutual exclusion—to make sure that the selection of its untried alternatives by different processors is serialized, so that no two processors can select the same alternative. As discussed in [3,11] this method allows the use of very efficient scheduling mechanisms—such as the scheduling on bottom-most choice-point—but may cause excessive network traffic if realized on a DMP [10].

However, there are other simple ways of ensuring that no alternative is simultaneously selected by multiple workers: *the untried alternatives of a choice-point can be split between the two copies of the choice-point stack*. This operation is called *Stack Splitting*. This will ensure that no two workers pick the same alternative. Choice-points are alternatively exchanged between the two processors, so that after the sharing each processor maintains half of the original set of choice-points. Different strategies to split the choice-points between the sharing agents can be selected depending on the structure of the computation. The major advantage of stack splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting the different or-parallel threads become fairly independent of each other, and hence communication is minimized during execution. This makes the stack splitting technique highly suitable for DMP. The possibility of parameterizing the splitting of the alternatives based on additional information (granularity, non-failure, user annotations) can further reduce the likelihood of additional communications due to scheduling. The results in [3] indicate that stack splitting obtains better speedups than MUSE on shared memory architectures (SMP)—thanks to reduced interaction between workers. Stack splitting has provided positive results in terms of speedups and efficiency [11] on DMPs.

Optimal Algorithms for Order-sensitive Executions: The problem of handling *OSC* during parallel executions has been pragmatically tackled in the literature [4]. Only recently the problem has been formally studied [7]. The computation tree is dynamic; the updates can be described by two operations: **expand** which adds a (bounded) number of children to a leaf, and **delete** which removes a leaf. Whenever a branch encounters a side-effect, it needs to verify whether the branch containing the side-effect is currently the leftmost active computation in the tree. If n is the current leaf of the branch where the side-effect is encountered, its computation is allowed to continue only if $\mu(n) = \text{root}$, where $\mu(n)$ indicates the highest node m in the tree (i.e., closest to the root) such that n is in the leftmost branch of the subtree rooted at m . $\mu(n)$ is also known in the parallel logic programming community as the *subroot node* of n [5]. Checking if a side-effect can be executed requires the ability of performing the operation **find_subroot**(n) which, given a leaf n , computes $\mu(n)$. From the data structures studied in [7] it is possible to derive the following: any sequence of **expand**, **delete**, and **find_subroot** operations can be performed in $O(1)$ time per operation on pointer machines—i.e., without the need of complex arithmetic (i.e., the solution does not rely on the use of “large” labels). The data structure used to support this optimal solution is based on maintaining a dynamic list which represents the frontier of the tree. The dynamic list can be updated in $O(1)$ time each time leaves are added or removed (i.e., when expanding a branch and performing backtracking). Subroot nodes can be efficiently maintained for each leaf—in particular, each delete operation affects the subroot node of at most one other leaf. Identification of the computations an *OSC* α depends on can be simply accomplished by traversing the list of leaves right-to-left from α . Leftmostness can be verified in $O(1)$ time by simply checking whether the subroot of the leaf points to the root of the tree.

2 Stack Splitting and Order-Sensitive Computations

Determining the executability of an *OSC* α in a distributed memory setting requires two coordinated activities: (a) determining *what are* the computations to the left of α in the computation tree—i.e., who are the processors that have acquired work in branches to the left of α ; (b) determining what is the *status* of the computations to the left of α . On DMPs, both steps require exchange of messages between processors. The main difficulty is represented by step (a)—without a shared structure, discovering the position of the processors requires arbitrary localization messages exchanged between all the processors. We propose a simple modification of stack splitting—dictated by the ideas in Sect. 1—that guarantees that processors are aware of the position of their subroot nodes. Thus, instead of having to locate the subroot nodes whenever an *OSC* occurs, these are implicitly located whenever a sharing operation is performed (a very infrequent operation, compared to the *OSC* steps). Knowledge of the position of the subroot nodes allows processors to maintain an approximation of the frontier of the tree, which in turn can be used to support the execution of step (b) above.

In the original stack splitting procedure [3], during a sharing operation the parallel choice-points are alternatively split between two processors. The processor that is giving the work keeps the odd-numbered choice-points, while the processor that receives work keeps the even-numbered ones. We have demonstrated [3,11] that this splitting strategy is effective and leads to good speedups for large classes of benchmarks. The alternation in the distribution of choice-points is aimed at reducing the danger of focusing a processor on a set of fine-grained computations. This strategy for splitting a computation branch between two processors has a significant drawback w.r.t. execution of *OSC*, since the processors, via backtracking, may arbitrarily move left or right of each other. This makes it impossible to know a-priori whether one processor affects the position of the subroot node of other processors, thus complicating the detection of the composition of the frontier of the tree. From Sect. 1 we learn that a processor operating on a leaf of the computation tree can affect other processors' subroot nodes only in a limited fashion. The idea can be easily generalized: if a processor limits its activities to the bottom part of a branch, then the number of leaves affected by the processor is limited and well-defined. This observation leads to a modified splitting strategy, where the processor giving work keeps the lower segment of its branch as private, while the processor receiving work obtains the upper segment of the branch. This modification guarantees that the processor receiving work will be always to the right of the processor giving the work. Since the result of a sharing operation is broadcast to all processors—to allow them to maintain an approximate view of work distribution—this method allows each processor to have an approximate view of the composition of the frontier of the tree. Algorithms and their correctness have been analyzed in [10].

2.1 Implementation

Data Structures: In order to support the new splitting strategy and use it to support *OSC* steps, each processor will require only two additional data structures: (1) the *Linear Vector* and (2) the *Waiting Queue*. Each processor keeps and updates a *linear vector* which consists of an array of processor Ids that represents the linear ordering of the processors in the search tree—i.e., the respective position of the processors within the frontier of the computation tree (section 1). The idea behind this *linear vector* is that whenever a processor wants to execute an *OSC*, it first waits until there are no processors Ids to its left on the *linear vector*. Such a status indicates that all the processors that were operating to the left have completed their tasks and moved to the right side of the computation tree, and the subroot node has been pushed all the way to the root of the tree. Once this happens, the processor can safely execute the *OSC*, being left-most in the search tree. Initially, the linear vector of all processors contains only the Id of the first running processor. In the original bottom-most scheduler developed for stack splitting [11], every time a sharing operation is performed, a **Send_LoadInfo** message is broadcast to all processors; this is used to inform all processors of the change in the workload and of the processors involved in the sharing. For every **Send_LoadInfo** message, each processor updates its linear vector by moving the Id of the processor that received work immediately to the right of the Id of the processor giving work. Each processor also maintains a *waiting queue* of Ids, representing all the processors that are waiting to execute an *OSC* but are located to the right of this processor. Whenever a processor enters the *scheduling* state to ask for work, it informs all processors in its waiting queue that they no longer need to wait on it to execute their *OSC*.

The Procedure: In stack splitting [11], a processor can only be in one of two states: *running state* or *scheduling state*. In order to handle *OSC*, we need another state: the *order-sensitive* state. All processors wanting to execute an *OSC* will enter this state until it is safe for them to execute their *OSC*. The transition between the states requires the introduction of three types of messages: (1) **Request_OSC**, (2) **OSC_Acknowledgment**, and (3) **Reply_In_OSC**. A **Request_OSC** will be used by a processor that wants to execute an *OSC*, and sent to all processors to its left in its linear vector. When a processor receives a message of this type, and it is not located to the left of the sending processor, a reply message with tag **OSC_Acknowledgment** will be sent back. This message informs the *OSC* processor that it no longer needs to wait on this processor. When a processor is in running state and gives work to another processor, a **Send_LoadInfo** message is sent to all other processors informing them of the new load information. Each processor receiving the **Send_LoadInfo** message updates its own linear vector by placing the Id of processor that received the work to the right of the processor giving work. When a processor is in scheduling state and receives work from another processor p , it will also update its linear vector by placing its Id to the immediate right of p . Once a processor arrives to the order-sensitive state, it first sends a **Request_OSC** to all the processors to its left in its linear vector. It then waits for an **OSC_Acknowledgment** message from each of them.

An `OSC_Acknowledgment` is sent by a processor when it is no longer to the left of the processor wanting to execute the *OSC*. When this message is received, the Id of the processor sending it will be removed from the linear vector. Notice that when the processor is waiting for these messages, it may receive `Send_LoadInfo` messages. If this happens, the processor has to update its linear vector. In particular, if due to this sharing operation a processor moves to its left, a `Request_OSC` message needs to be sent to this processor as well. Once the processor receives an `OSC_Acknowledgment` from all these processors, it can safely perform the *OSC*. Processors in the order-sensitive state are not allowed to share work; requests to share work are denied with the `Reply_In_OSC` message. When a processor is in running state and receives a `Request_OSC` message, it consults its linear vector and reacts in the following way. If the Id of the processor wanting to execute an *OSC* is to its right in the linear vector, the Id of the requesting processor is inserted in the waiting queue. When the running processor runs out of work and moves to the scheduling state, an `OSC_Acknowledgment` message will be sent back to the processor wanting to execute the *OSC*. If the Id of the processor wanting to execute the *OSC* is to its left, an `OSC_Acknowledgment` message is immediately sent back to the processor wanting to execute the *OSC*. This means that the running processor is no longer to the left of the processor wanting to execute the *OSC*. When a processor enters the scheduling state, it dequeues all the Ids from its waiting queue and sends an `OSC_Acknowledgment` to all these processors, informing them that it is no longer to their left. Similarly, the linear vector of a processor in any of the states that receives a `Request_Work` and cannot give away work can also be easily updated. In this case, the Id of the processor requesting work can be safely removed from the linear vector. Just as we attach load information to messages in the traditional scheduling algorithm [11], we also attach updated load information to these three new messages.

2.2 Implementation Details

Partitioning Ratios: The stack splitting modification divides the stack of parallel choice-points into two contiguous partitions, where the bottom partition is kept by the processor giving work and the upper partition is given away. This stack splitting modification guarantees that the processor that receives work will be to the immediate right of the other processor. The question is what is the partitioning ratio that will produce the best results? We first tried using a partition where the processor that is giving work keeps the bottom half of the branch and only gives away the top half. After experimenting with lots of different partition ratios, we found out that with a partition ratio of $3/4 - 1/4$ where the processor that is giving work keeps the bottom $3/4$ of the parallel choice-points and gives away the top $1/4$ of the parallel choice-points, our benchmarks without side-effects obtain excellent speedups—similar than our original alternating splitting [11]. When we run our benchmarks with side-effects, the partition ratio of $3/4 - 1/4$ performed superior to the partition ratio of $1/2$. The explanation is that it is better for processors to get smaller pieces of work with fewer side-effects than large pieces with lots of side-effects.

Messages Out of Order: `Send_LoadInfo` messages may arrive out of order and then the linear vectors may be outdated. E.g., processor 2 receives from processor 0 a `Request_Work` message but decides not to share work. Since processor 0 is requesting work, processor 2 removes 0 from its linear vector. Later on, processor 0 gets work from processor 1, and processor 1 broadcasts a `Send_LoadInfo` message. Afterwards, processor 0 gives work to processor 3 and also broadcasts a `Send_LoadInfo` message. Now, suppose that processor 2 receives the second `Send_LoadInfo` message first and the first `Send_LoadInfo` next. When processor 2 tries to insert 3 to the immediate right of 0 in the linear vector, 0 is not located and therefore 3 cannot be inserted. MPI (used in our system for processor communication) does not guarantee that two messages sent from different processors at different times will arrive in the order that they were sent. The scenario presented above can be avoided if, in every sharing operation, both the involved processors broadcast a `Send_LoadInfo` message to all the other processors. In this case every processor will be informed that a sharing operation occurred either by the giver or by the receiver of work. Processor 2 in the above scenario will first know that processor 0 obtained work from processor 1, and then will know that processor 0 gave work to processor 3. Duplication of `Send_LoadInfo` messages is handled through the use of two dimensional arrays *send1* and *send2* of size N^2 , where N is the total number of processors; *send1*[*i*][*j*] (*send2*[*i*][*j*]) is incremented when a sharing message from *i* to *j* is received from processor *i* (*j*). The linear vector will be updated only if *send1*[*i*][*j*] > *send2*[*i*][*j*] (*send2*[*i*][*j*] > *send1*[*i*][*j*]) and the message comes from *i* (*j*).

Table 1. Benchmarks (Time in sec.)

Benchmark	Timings	Number of solutions
<i>9 Costas</i>	412.579	760
<i>Knight</i>	159.950	60
<i>Stable</i>	62.638	2
<i>10 Queens</i>	4.572	724
<i>Hamilton</i>	3.175	80
<i>Map Coloring</i>	1.113	2594

3 Implementation Results

We implemented the above described technique to handle *OSC* in our PALS Prolog system and tested it with the following benchmarks. *Stable* is an engine to compute stable models of a logic program. *Knight* finds a path of knight-moves on a chess-board visiting every square on the board. *9 Costas* computes the Costas sequences of length 9. *Hamilton* finds a closed path through a graph such that all the nodes of the graph are visited once. *10 Queens* places a number of queens on a chess-board so that no two queens attack each other. *Map Coloring*

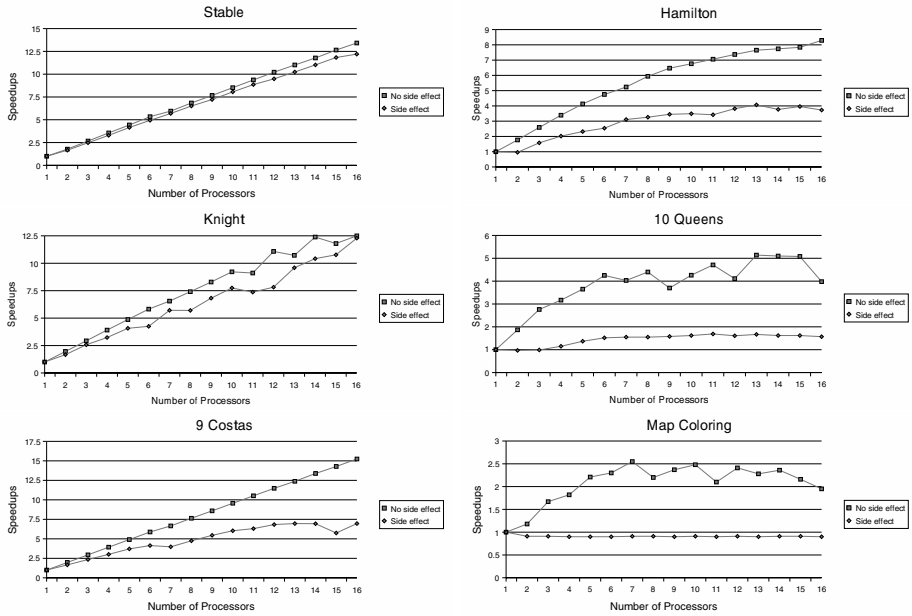


Fig. 1. No Side-Effects vs. Side-Effects (1) **Fig. 2.** No Side-Effects vs. Side-Effects (2)

colors a planar map. These benchmarks compute all solutions and execute side-effect predicates, e.g., *write* to describe the computations.

Figs. 1 and 2 show the speedups obtained by this technique under the label *side-effect*. The figures also show the speedups obtained when running these benchmarks without treating the *write* predicate as a side-effect but using the modified stack splitting approach described above. Two main observations arise from these experiments. First of all, the speedups obtained using the modified scheduling scheme are not that different from those observed in our previous experiments [10]; this means that the novel splitting strategy does not deteriorate the parallel performance of the system. For benchmarks with substantial running time and with the fewest number of printed solutions (*Knight*, *Stable*) the speedups are very good and close to the speedups obtained without handling of *OSC*. We also observe that for benchmarks with smaller running time but larger number of side-effects (*Hamilton*) the speedups are still good but less close to the speedups obtained without side-effects. Note that for benchmarks with small running time and the greatest number of printed solutions (*Map Coloring*, *10 Queens*), the speedups deteriorate significantly and may be less than 1. This is not surprising; the presence of large numbers of side-effects (proportional to the number of solutions) implies the introduction of a large sequential component in the computation, leading to reduced speedups. *9 Costas* has the largest number of solutions, but its speedups are good.

The results obtained are consistent with our belief that DMP implementations should be used for programs with coarse-grained parallelism and a mod-

est number of OSC . Coarse-grained computations are even more important if we want to handle large numbers of side-effects where it is necessary that the OSC be spaced far apart. For programs with small-running times there is not enough work to offset the cost of exploiting parallelism and even less for handling of OSC . Nevertheless, our system is reasonably efficient given that it produces good speedups for large and medium size benchmarks with even a considerable number of OSC , and produces no slow downs except for benchmarks with huge numbers of side-effects and small running times. Even in presence of OSC , the parallel overhead observed is substantially low—on average 6.5% and seldomly over 10%. As a comparison, Fig. 3 compares with the speedups for some benchmarks obtained using a variant of the Muse system [1] on SMP. The results highlight the fact that, for benchmarks with significant running time, our methodology is capable of approximating the best behavior on SMPs.

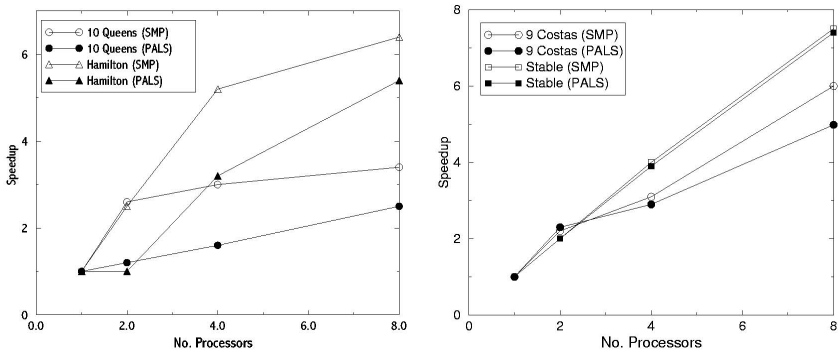


Fig. 3. Comparison with Muse

4 Conclusion

In this paper we presented a novel technique to handle exploitation of search-parallelism from non-deterministic languages on DMP. The technique is an instance of the optimal Stack Splitting methodology and provides an effective methodology to handle OSC (e.g., side-effects) without unnecessary communication and without excessive loss of parallelism. The methodology has been implemented in the PALS Or-parallel Prolog system and excellent results have been observed—in particular, the degradation of performance w.r.t. optimal methodologies used in shared memory systems is minimal. Work is in progress to validate the methodology in different systems (e.g., non-monotonic reasoning systems).

References

1. K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-Parallelism in Muse. In *International Journal of Parallel Programming*, 19(6), 1990.

2. L. Araujo. Full Prolog on a Distributed Architecture. In *EuroPar*, Springer, 1997.
3. G. Gupta et al. Techniques for Implementing Or-parallelism on DMPs. *ICLP*, MIT, 1999.
4. G. Gupta et al. Parallel Execution of Prolog Programs. *TOPLAS*, 23(4), 2001.
5. B. Hausman et al. Cut and Side-Effects in Or-Parallel Prolog. *Conference on Fifth Generation Computer Systems*, 1988. Springer Verlag.
6. E. Pontelli. Experiments in Parallel Execution of ASP. *IEEE IPDPS*, 2001.
7. D. Ranjan et al. Data Structures for Order-Sensitive Predicates. *ACTA Informatica*, 37(1):21–43, 2000.
8. R. Rocha et al. Parallel Prolog based on Stack Copying. *EPIA*, Springer, 1999.
9. P. Szeredi. Using dynamic predicates in an or-parallel prolog. *ILPS*, MIT, 1991.
10. K. Villaverde. *Methodologies for OR-Parallelism on DMPs*. PhD, NMSU, 2002.
11. K. Villaverde et al. Incremental Stack Splitting Mechanisms. *IEEE ICPP*, 2001.

From Complexity Analysis to Performance Analysis

Vicente Blanco, Jesús A. González, Coromoto León, Casiano Rodríguez, and
Germán Rodríguez

Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
La Laguna, 38271, Spain
casiano@ull.es
<http://nereida.deioc.ull.es/~call/>

Abstract. This work presents a new approach to the relation between theoretical complexity and performance analysis of parallel programs. The study of the performance is driven by the information produced during the complexity analysis stage and supported at the top level by a complexity analysis oriented language and at the bottom level by a special purpose statistical package.

1 Introduction

Researchers have successfully developed simple analytical models like BSP [11] or LogP [3] to obtain qualitative insights and bounds on the scalability of parallel programs as a function of input and system size. In contrast, tools for more detailed performance analysis have been primarily based on measurement and simulation rather than on analytical modeling. This paper introduces a general framework where different performance models can be instantiated, develops an approach that combines analytical modeling with measurement and discuss a set of software tools giving support to the methodology.

The next section presents the idea of complexity analysis driven measurement. It establishes the concepts supporting the analytical model for the proposed methodology. Section 3 describes the software environment. Examples of use are shown in section 4. Some conclusions and open questions are formulated in section 5.

2 The Meta-model

Assuming that a processor has an instruction set $\{I_1, \dots, I_t\}$ where the i -th instruction I_i takes time p_i , an approximation to the execution time $T^{\mathcal{P}}(\Psi)$ of an algorithm $\mathcal{P}$ with sizes of its input data $\Psi = (\Psi_1, \dots, \Psi_d) \in \Omega \subset \mathbb{N}^d$ is given by the formula:

$$T^{\mathcal{P}}(\Psi) \simeq \sum_{i=1}^t w_i(\Psi) \times p_i \quad (1)$$

where $w_i(\Psi)$ is the number of instructions of type I_i executed by the algorithm $\mathcal{P}$ with input data of size Ψ .

On the other side, a complexity analysis can be considered as the task of classifying the sequence of operations executed by an algorithm in sets, so that all sentences in a set are executed approximately the same number of times. The “algorithm designer” counts these high level operations in algorithm $\mathcal{P}$ and provides a complexity formula:

$$f^{\mathcal{P}}(\Psi) = \sum_{i=0}^n A_i \times f_i(\Psi) \quad (2)$$

which is an approximation to the execution time of the program expressed in terms of its input data. The f_i functions depend only on the input sizes Ψ and summarize the repetitions of the different segments of code. The constants A_i stand for the sets of machine instructions corresponding to the “component” function $f_i(\Psi)$. These constants are architecture dependent and can be computed through multidimensional regression analysis on the gathered execution times. Unfortunately, it is not uncommon to observe a variation of the values A_i with the input size, that is, the A_i are not actually real fixed numbers but functions

$$A_i : \Omega \subset \mathbb{N}^d \rightarrow \mathbb{R}$$

Usually the assumption is that A_i is a step-wise function of the input size $\Psi \in \Omega$.

Let us denote by $\Gamma = \{\gamma_0, \dots, \gamma_m\}$ the set of parameters which describe an architecture. These sets of parameters can be based on benchmarking tools, as PMB [4] or ParBench [7] or model oriented programs as the `bsp_probe` included with the BSPlib library [5]. These programs are usually known as probes. Each index γ_i given by a probe program evaluates a “Universal Instruction Class” (UIC). The set of UICs evaluated on a machine $\mathcal{M}$ constitutes its “machine profile”, $(\gamma_i^{\mathcal{M}})$. For example, the BSP model uses the set $\Gamma = (s, g, L)$, which corresponds to the universal instruction classes “computing”, “communication” and “synchronization”. The last one also stands for startup latency.

We assume that each coefficient A_i in Eq. (2) can be expressed as a function of the architecture parameters Γ ,

$$A_i = X_i(\gamma_1, \dots, \gamma_m) \quad i = 0, \dots, n \quad (3)$$

A computational model determines the definition and construction of the triples (Γ, X, A) i.e. not only defines the nature of the parameters Γ and application coefficients A but the set of procedures to determine them and the methodology to build the functions X_i that permit to characterize the performance of a given algorithm.

Though expressions (3) are not always linear functions, in not few cases the linear behavior is a good approach. Equation (3) then becomes:

$$A_i = \sum_{j=0}^m x_{i,j} \times \gamma_j \quad i = 0, \dots, n \quad (4)$$

where $x_{i,j}$ denotes the contribution of the j -th UIC to the i -th “component” function f_i . Computational models can be classified as *linear* models, when they follow Eq. (4) and they assume A_i to be constants in the whole range of the input size Ω . Otherwise

we will call them non-linear models (i.e. they provide a non linear function X_i or they assume a functional nature of the coefficients).

Assuming a linear model, we can substitute the coefficients in the complexity formula (2) by their expressions in terms of a machine profile (4):

$$f^{\mathcal{P}}(\Psi) = \sum_{i=0}^n \sum_{j=0}^m x_{i,j} \times \gamma_j \times f_i(\Psi) \quad (5)$$

$$f^{\mathcal{P}}(\Psi) = \sum_{j=0}^m \left(\sum_{i=0}^n x_{i,j} \times f_i(\Psi) \right) \times \gamma_j \quad (6)$$

$$\sum_{i=0}^n A_i \times f_i(\Psi) = \sum_{j=0}^m \omega_j(\Psi) \times \gamma_j \quad (7)$$

where

$$\omega_j(\Psi) = \sum_{i=0}^n x_{i,j} \times f_i(\Psi) \quad (8)$$

is the frequency of the j -th UIC during the execution of program $\mathcal{P}$ with input Ψ . Executing with a range of different inputs, Eq. (7) leads in a system of equations. Now the coefficients A_i can be obtained by taking into account the following facts:

1. the UICs γ_j are provided by the probe program,
2. the values $\omega_j(\Psi)$ can be obtained instrumenting the program, and
3. the functions $f_i(\Psi)$ are provided by the algorithm designer.

3 The Software Support

The previous mathematical approach is supported by a set of software tools. *CALL* and *LLAC* [6] had been developed to cooperate in the framework shown in Fig. 1.

CALL consists of a pragma based language, its compiler and measurement libraries. Through the use of pragmas, complexity formulas of the form of Eq. (2), are embedded in a source code. For example, the classical matrix product algorithm can be annotated as follows:

```
#pragma cll M M[0]+M[1]*N+M[2]*N*N+M[3]*N*N*N
  for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
      sum = 0;
      for(k = 0; k < N; k++)
        sum += A(i, k) * B(k, j);
      C(i,j) = sum;
    }
  }
#pragma cll end M
```

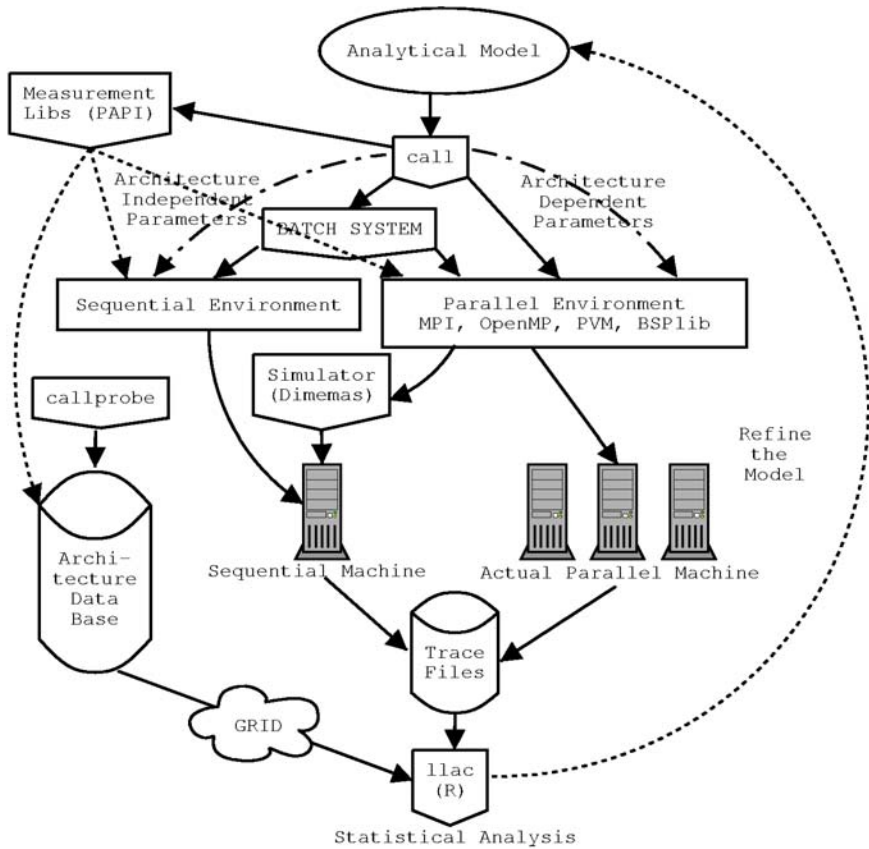


Fig. 1. Framework for *CALL* and *LLAC* tools

Where the prefix *c11* is the signal for the *CALL* compiler. The identifier *M* denotes an experiment enclosed between matching pragmas, and coefficients $M[0]$, $M[1]$, $M[2]$ and $M[3]$ stand for the algorithm computational constants. The *CALL* compiler translates these pragmas into calls to measurement routines. It generates an instrumented source code that can be compiled and executed on both sequential and parallel environments. The current version contain drivers for sequential, MPI, PVM, OpenMP and also for the Oxford BSPlib and PUB BSP oriented libraries [1,10].

Trace files containing the experiment measurements are produced by executing an instrumented code. *LLAC* provides an interactive environment that allows visualization and statistical processing of these files. Using multidimensional regression, it computes the coefficients in Eq. (2). *LLAC* also deals with a machine profile database generated by a probe program to solve the system of equations (7).

Though the most common observable is execution time, *CALL* relies on portable and architecture dependent performance measurement libraries as PAPI [2], to provide access to other execution magnitudes. The line labeled “Architecture Dependent Measurements”

in Fig. 1 corresponds to such kind of magnitudes, as cache hits and misses, number of floating point operations, etc. *CALL* also allows the observation of algorithm dependent magnitudes as the number of exchanges and swaps in a sorting function or the number of visited nodes in a branch and bound search. These second kind of observables correspond to the “Architecture Independent Measurements” label in Fig. 1.

4 Examples of Use

This section intends to be a brief introduction to the use of the model proposed in section 2 and its accompanying tools, *CALL* and *LLAC*. At the same time, it illustrates its advantages. For this, we will solve the 0-1 Knapsack Problem (*0-1KNP*) using the Branch and Bound technique.

In the *0-1KNP(N,C)* we have a set of N objects and a knapsack of total capacity C . Each object $i \in \{1, \dots, N\}$ has a profit, denoted by p_i and a weight denoted by w_i . You want to find the combination of objects that maximizes the total profit but whose total weight does not exceeds the knapsack total capacity C :

$$\max\left\{\sum_{i=1}^N x_i \times p_i : x \in \{0,1\}^N \wedge \sum_{i=1}^N x_i \times w_i \leq C\right\} \quad (9)$$

Fig. 2 presents a C main function solving the knapsack problem.

```

1 ....
2 number N, M;
3 number w[MAX], p[MAX];
4 number bestSol = -INFINITY;
5 #pragma cll code double numvis;
6 ....
7 int main(int argc, char ** argv) {
8     number sol;
9     readKnap(data);
10 #pragma cll kps numvis = kps[0]*unknown(N)
11     sol = knap(0, M, 0);
12 #pragma cll end kps
13     printf("\nsol = ", sol);
14 #pragma cll report all
15     return 0;
16 }
```

Fig. 2. B&B main code with *CALL* annotations

The call to the function `knap(0, M, 0)` at line 11 is enclosed between *CALL* pragmas to establish an experiment named `kps`. The complexity formula embedded at line 10 states that we want to observe the variation of the variable that holds the number of

nodes visited by the branch and bound function: `numvis`. It also states that we don't know the complexity formula that relates the number of visited nodes with the number of objects (that is the meaning of the special *CALL* function `unknown`) but that we think it depends on N . The variable `numvis` is updated at line 4 in figure 3. Observe how through

```

1 number knap(number k, number C, number P) {
2   number L, U, next;
3   if (k < N) {
4     #pragma cll code numvis++;
5     lowerUpper(k,C,P,&L,&U);
6     if (bestSol < L) {
7       bestSol = L;
8     }
9     if (bestSol < U) { /* L <= bestSol <= U */
10      next = k+1;
11      knap(next, C - w[k], P + p[k]);
12      knap(next, C, P);
13    }
14  }
15  return bestSol;
16 }
```

Fig. 3. B&B function annotated with *CALL* pragmas

the use of pragmas the semantic of the original program is preserved, so that compilation without the intervention of *CALL* will behave as the initial un-instrumented program.

Figure 4 a) was produced using *LLAC*. To make the generated instances of this well known NP-complete problem more difficult, we followed [8], and generate them so that weights and profits were strongly correlated. The capacity was fixed to half the sum of the weights. For number of objects between 1000 and 5000, 10 different problems were generated. A fact that follows from the observation of the figure is that the separation between the best and worse problems for a given size diverges with N . This was something we expected.

The dashed line shows the average number of visited nodes. The continuous line is the fit of the averages to a quadratic function: $A_2 \times N^2$. The “average” number of visited nodes produced by this generator of “difficult” *0-1KNP* instances seems to vary almost perfectly with the square of N . This was something we didn't expect!

Figure 4 b) shows the results for a master-slave parallelization of the code in figure 3 on an heterogeneous Linux cluster with 8 processors. The two first processors (0 and 1) have a clock rate of 800MHz. The clock rate of the other 6 is 500MHz. Therefore, there is a roughly relation of 5/8 between the slow and fast processors. The figure shows the number of nodes visited by each processor. There were five executions. Since processor 0 plays the role of the master, it always visits 0 nodes. The ratio of visited nodes between processors 1 and 2 approaches its 5/8 performance ratio. But the ratio of processor 1 with the other processors is much worse, indicating a potential problem. The master

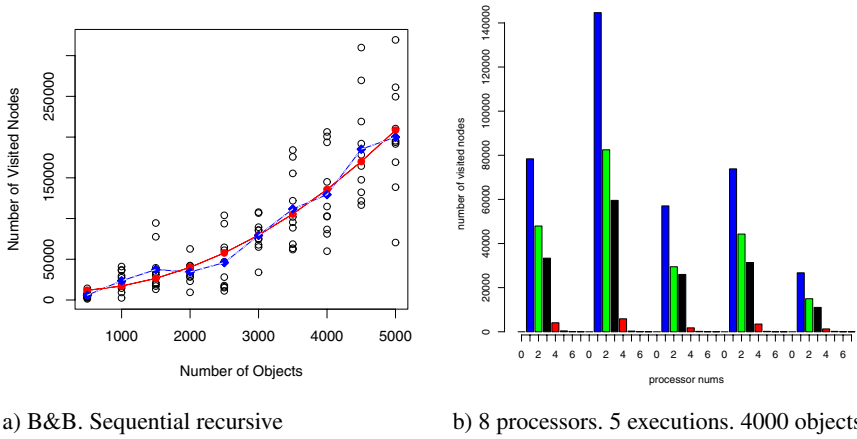


Fig. 4. B&B 0-1 KNP

distributes problems on a “processor with lowest index idle is first served” basis. That partially explains why the two first processors work more than the others. Since the same profile appears for *0-1KNP* problems with large values of N we conclude that the profile is not a consequence of dealing with an small N but a performance “bug”.

It is remarkable that the 5 executions appearing in figure 4 b) correspond to the same input. Observe how the search trees differ from one execution to the next. It is known that this curious phenomena appears in parallel branch and bound [9].

5 Conclusions and Future Work

A Performance Model has been proposed. It can be applied to sequential, distributed memory and shared memory parallel platforms. The model, through the support of software tools, improves the current state of analytical models regarding to prediction accuracy [12].

Many open questions remain. How to improve the detection of UICs? What is the more appropriate design for a probe? What is the best algorithm to detect the regions of validity?

Acknowledgements. We would like to thank CIEMAT and EPCC (TRACS EC funded project). This work has been supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04498-C05-05 and TIC2002-04400-C03-03).

References

1. Olaf Bonorden, Ben Juulink, Ingo von Otto, and Ingo Rieping. The Paderborn University BSP (PUB) Library—Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, 1999.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Application*, 14(3):189–204, 2000.
3. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.
4. Pallas GmbH. Pallas MPI Benchmarks version 2.2. <http://www.pallas.com/e/products/pmb>.
5. M. Goudreau, J. Hill, K. Lang, B. McColl, S. Rao, D. Stefanescu, T. Suel, and T. A. Tsantilas. Proposal for the BSP worldwide standard library, 1996.
6. G. Rodriguez Herrera. CALL: A complexity analysis tool. Master's thesis, Centro Superior de Informática. Univ. La Laguna, 2002. <http://nereida.dei.uc3m.es/~call>.
7. Roger Hockney and Michael Berry. Public international benchmarks for parallel computers. *Scientific Programming*, 2(3):101–146, 1994.
8. S. Martello and P. Toth. *Knapsack Problems*. John Wiley & Sons, Chichester, 1990.
9. G. P. McKeown, V. J. Rayward-Smith, and S. A. Rush. *Parallel Branch-and-Bound*, chapter 5, pages 111–150. Advanced Topics in Computer Science. Blackwell Scientific Publications, Oxford, 1992.
10. David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.
11. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
12. A. Zavanella and A. Milazzo. Predictability of Bulk Synchronous programs using MPI. In *8th Euromicro PDP*, 2000.

The Implementation of ASSIST, an Environment for Parallel and Distributed Programming

Marco Aldinucci², Sonia Campa¹, Pierpaolo Ciullo¹, Massimo Coppola²,
Silvia Magini¹, Paolo Pesciullesi¹, Laura Potiti¹, Roberto Ravazzolo¹,
Massimo Torquati¹, Marco Vanneschi¹, and Corrado Zoccolo¹

¹ University of Pisa, Dip. di Informatica – Via Buonarroti 2, 56127 Pisa, Italy

² Ist. di Scienza e Tecnologie dell'Informazione, CNR – Via Moruzzi 1,
56124 Pisa, Italy, coppola@di.unipi.it

Abstract. We describe the implementation of ASSIST, a programming environment for parallel and distributed programs. Its coordination language is based of the parallel skeleton model, extended with new features to enhance expressiveness, parallel software reuse, software component integration and interfacing to external resources. The compilation process and the structure of the run-time support of ASSIST are discussed with respect to the issues introduced by the new characteristics, presenting an analysis of the first test results.

1 Introduction

The development of parallel programs to be used in the industrial and commercial fields is still a difficult and costly task, which is worsened by the current trend to exploit more heterogeneous and distributed computing platforms, e.g. large NOW and Computational Grids.

In our previous research [1,2] we focused on *structured parallel programming* approaches [3], and the ones based on parallel algorithmic skeletons in particular. We exploited skeletons as a parallel coordination layer of functional modules, eventually made up of conventional sequential code. Our experience is that this kind of skeleton model brings several advantages, but does not fulfill all the practical issues of efficient program development and software engineering.

In this paper we discuss the implementation of ASSIST [4], a general-purpose parallel programming environment. Through a coordination language approach we aim at supporting parallel, distributed and GRID applications. ASSIST eases software integration also by exploiting the software component paradigm.

In ASSIST, the skeleton approach is extended with new ideas to allow for (1) sufficient expressiveness to code more complex parallel solutions in a modular, portable way, (2) the option to write skeleton-parallel applications interacting with, or made up of components, and (3) the option to export parallel programs as high performance software components.

* This work has been supported by the *Italian Space Agency*: ASI-PQE2000 Programme on “Development of Earth Observation App.s by Means of Systems and Tools for HPC”, and by the *National Research Council*: Agenzia 2000 Programme.

Sect. 2 summarizes the motivation and main features of the ASSISTcl coordination language. We describe in Sect. 3 the structure and implementation of the first prototype compiler, and in Sect. 4 the compilation process and the language run-time, with respect to the constraints and the issues raised by the language definition. First results in verifying the implementation are shown in Sect. 5, Sect. 6 discusses related approaches in the literature, and Sect. 7 draws conclusions and outlines future development.

2 The ASSIST Approach, Motivation, and Features

Our previous research about structured parallelism and skeleton-based coordination has verified several advantages of the approach. However, our experience is that data-flow functional skeletons are not enough to express in a natural, efficient way all parallel applications, especially those that *(i)* involve irregular or data-driven computation patterns, *(ii)* are data-intensive in nature (I/O bound or memory constrained), *(iii)* are complex and multidisciplinary, hence require access to specific external resources and software modules by means of mandated protocols and interfaces. Stateless, isolated modules linked by simple and deterministic graphs can lead to inefficient skeleton compositions in order to emulate state-dependent or concurrent activities in a parallel program.

Moreover, the approach did not allow us the reuse of *parallel* programs as components of larger applications, nor to easily exploit dynamically varying computational resources. These are major issues in view of the affirmation of high performance, large-scale computing platforms (from huge clusters to Computational Grids), which require us both to promote cooperation of software written within different frameworks, and to exploit a more informed management of computational resources.

The ASSIST environment has been designed with the aim of providing

- high-level programmability and productivity of software development
- performance and performance portability of the resulting applications
- enhanced software reuse, and easier integration between ASSIST programs and other sequential/parallel applications and software layers.

These requirements have a visible impact on the structure of the coordination language. While pipeline, farm and map parallel skeletons are still supported by ASSISTcl, new, more expressive constructs are provided by the language. The language support has been designed to more easily interact with heterogeneous program modules and computational resources (e.g. different communication supports and execution architectures). Due to lack of space, we can only summarize here the distinguishing features of ASSIST, which are detailed in [5,4].

Coordination Structure. A module is either a unit of sequential code, or a skeleton construct coordinating more sequential or parallel modules. In addition to the older skeletons, a generic *graph* container is available, that specifies arbitrary connections among modules, allowing multiple input and

output channels per module. This implies a radical departure from the simpler and more restrictive notion of module coordination we adopted in past research.

Parallel Expressiveness. The *ParMod* skeleton [4] is introduced, which exploits increased programmability. A *ParMod* coordinates a mixed task/data parallel computation over a set of *virtual processors* (VP), to be mapped to real processing units. The set of VPs has a topology, data distribution policy, communications and synchronization support. The *ParMod* allows managing multiple, independent streams of data per VP, different computations within a VP being activated in a data-driven way. It is possible to manage *nondeterminism* over input streams, using a CSP-like semantics of communication, as well as to explicitly enforce synchronous operation of all the VPs. When needed, data structures from the streams can be broadcast, distributed on-demand, or spread across the VPs, according to one of the available *topologies*. With respect to the topology, fixed and variable stencil communications are allowed, as well as arbitrary, computation dependent communication patterns.

Module State. Program modules are no longer restricted to be purely functional, they can have a local state. A *shared* declaration allows *ASSIST* variables to be accessed from all the modules of a skeleton. Consistency of shared variables is not rigidly enforced in the language, it can be ensured (i) by algorithmic properties, (ii) by exploiting the synchronization capabilities of the *ParMod* skeleton, or (iii) by a lower-level implementation of the data types as *external objects*.

External Resources. The abstraction of *external objects* supports those cases when module state includes huge data, or unmovable resources (software layers or physical devices, databases, file systems, parallel libraries). External objects are black-box abstract data types that the *ASSIST* program can communicate with, sequentially and in parallel, by means of object method calls. As an essential requirement, external objects are implemented by a separate run-time tool, which must avoid hidden interactions with the execution of parallel programs. The approach is being used to develop a run-time support of module state which handles out-of-core data structures as objects in virtual shared memory, and is independent from *ASSIST* shared variables.

Software Components. *ASSIST* can both use and export component interfaces (e.g. CORBA). The new features of the language and the improvements in the run-time design make it possible to use external, possibly parallel software components as *ASSIST* program modules, and to export an *ASSIST* application as a component in a given standard.

The expressive power of the graph and *ParMod* constructs encompasses that of conventional skeletons¹ and allows the expression of combinations of task and data parallelism not found in other skeleton-based languages or in data-parallel models (e.g. systolic computations and large/irregular divide and conquer algorithms). The new features, and the management of dynamic, heterogeneous

¹ In principle we can exploit VPs to write low-level, unstructured parallel programs.

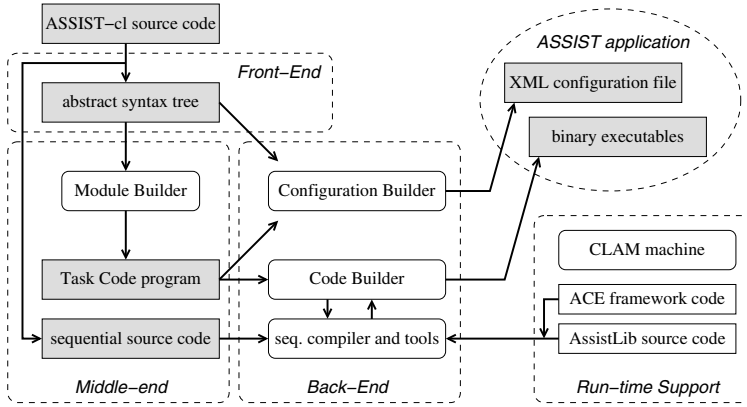


Fig. 1. Compilation steps in ASSIST

resources, lead to new open issues in the implementation, in particular w.r.t. compile-time optimizations and run-time performance models.

3 Implementation of the ASSIST Compiler

The ASSISTcl compiler (Fig. 1) has been designed to be modular and extendable. In particular, (i) the compilation process is divided into completely separate phases, (ii) portable file formats are used for the intermediate results, (iii) the compiler exploits object-oriented and design pattern techniques (e.g. visitor and builder patterns are consistently used from the Front-End to the Back-End of the compiler). The intermediate formalisms of all compilation phases have been designed with great care for the issues of portability and compatibility of the compilation process with heterogeneous hardware and software architectures.

Front end. The compiler front-end is based on a fairly standard design, in which the high-level source code is parsed, turned into an abstract syntax tree and type-checked. Host language source code (C, C++, Fortran) is extracted from ASSIST program modules and stored separately, gathering information about all sequential libraries, source files and compilation options used.

Middle End. The middle-end compilation (the *Module Builder* unit) employs an intermediate representation called *Task Code*, described in Sect. 3.1. Contrary to the high-level syntax, Task Code represents a materialized view of the process graph used to implement the program, with each skeleton compiled to a subgraph of processes that can be thought of as its implementation template (Fig. 2b). It is actually a mixed-type process network, as some of the graph nodes represent internally parallel, SPMD activities. Global optimizations can then be performed on the whole Task Code graph.

Back End. The compiler Back-End phase contains two main modules.

The *Code Builder* module compiles Task Code processes to binary executables, making use of other (sequential) compilers and tools as appropriate for each module. Most of the run-time support code is generated from the Task-Code specification and the sources of the C++ AssistLib library.

In the same compilation phase, the *Configuration Builder* module generates a program configuration file, encoded in XML, from the abstract syntax tree and Task Code information. This `assist.out.xml` file describes the whole program graph of executable modules and streams, including the information needed by the run-time for program set-up and execution (machine architecture, resources and a default process mapping specification).

3.1 Program Representation with Task Code and Its Run-Time Support

Task Code (Fig. 2b) is the intermediate parallel code between the compiler Front-End and its Middle-End. It is a memory resident data structure, with full support for serialization to, and recovery from a disk-resident form. This design choice enhances compiler portability, and allows intermediate code inspection and editing for the sake of compiler engineering and debugging. The Task Code definition supports sequential and SPMD parallel “processes”, each one having

- a (possibly empty) set of input and output stream definitions, along with their data distribution and collection policy
- sequential code information previously gathered
- its parallel or concurrent behaviour as defined by the high-level semantics (e.g. nondeterminism, shared state management information).

Task Code supports all of the features of ParMod semantics. Different parts of the abstract machine that supports these functions are implemented at compile time, at load-time and at run-time. The parallel primitives we have mentioned so far are mainly provided in the AssistLib library. The Task Code graph is also a target for performance modelling and optimization. While results about the classical skeletons can be applied to simple enough subgraphs, in the general case approximate or run-time adaptive solutions will be required to deal with the increased complexity of the ASSIST coordination model, and the more dynamic setting of heterogeneous resources.

AssistLib is a compile-time repository of object class definitions and method implementations. Heavy use is made of C++ templates and inline code in order to reduce the run-time overhead in the user code. AssistLib comprises:

- routines for communication and synchronization
- implementation of stream data distribution policies
- implementation of the SMU (state management unit)
- interface to shared data structures handled by the SMU
- interface to “external object” resources.

Most of the concurrency and communication handling is currently performed by Reactor objects of the ACE library (see Sect. 4). Template and inline code implements communications exploiting the information statically available about communication channels (data types, communication policy).

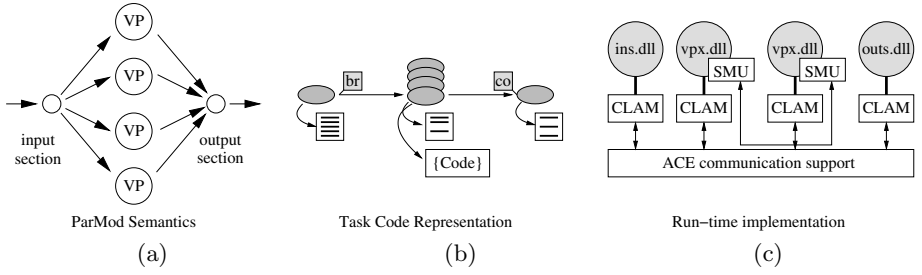


Fig. 2. ParMod compilation steps.

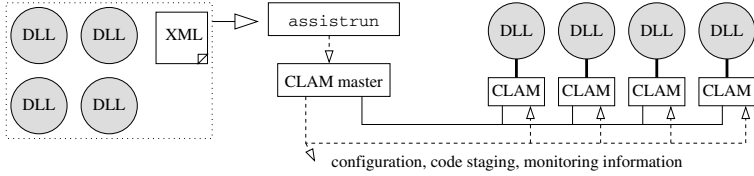


Fig. 3. ASSIST program execution scheme.

The SMU code encapsulates shared state management within the user code, mapping shared variables to data that is replicated or distributed across VPs according to the declarations in the ASSIST program. The SMU core essentially becomes part of the user code (Fig. 2c), providing the run-time methods to access shared data, and to ensure state coherence among the VPs when required by ParMod semantics. Optimizations are performed if multiple VPs run as threads in the same process, and owner-initiated communications are employed if the shared data access pattern is easily predictable (e.g. a fixed stencil).

The AssistLib library uses communication and resource management primitives provided by the run-time support. In the current implementation, some functionalities are provided by the ACE library, and their implementation is chosen at compile-time, while other ones can still be changed at run-time.

4 Run-Time Architecture

The current prototype of the ASSIST environment has been developed on a cluster of LINUX workstations, exploiting the communication and O.S. abstraction layers of the ACE library [6] (Fig. 2c). The use of the ACE object-oriented framework to interface to primary services (e.g. socket-based communications, threads), besides simplifying the run-time design, ensures a first degree of portability which includes all POSIX compliant systems.

The communication support provided by ACE relies on standard UNIX sockets and the TCP/IP stack, which are ubiquitous, but cannot fully exploit the performance of modern communication hardware. The language run-time is expandable either by extending the ACE classes to use new communication libraries, or by rewriting part of the communication primitives of AssistLib.

The standard ASSIST loader `assistrun` uses the XML configuration file to set-up the process net when running the program (Fig. 3). Manual, automatic and visual tool assisted editing of program configuration is also possible, enhancing the flexibility of the compiled application. The collected executable modules are designed to run with the help of CLAM, the coordination language abstract machine. The CLAM is indeed no complete virtual machine, it provides run-time services (resource location and management, computation monitoring) within heterogeneous and dynamically configurable architectures. To accomplish this goal, the CLAM either implements its own configuration services (current version) or interfaces to existing ones, e.g. [7].

A CLAM master process monitors the computation and manages the global configuration, interfacing to a set of CLAM slave processes that run on the available computing nodes. CLAM slaves can load code in the form of dynamically linked libraries (DLL), and they can act as intermediaries between the executable modules and the communication software layer. The CLAM master “executes” the `assist.out.xml` file by assigning the slaves a proper mapping of executable modules in the program. CLAM support processes react to changes in the program and in machine configuration. They communicate using a simple and portable internal protocol, while application communications are performed according to CLAM internal process mapping information. When several VPs are mapped to the same physical node, CLAM slaves execute their DLL either as separate processes, or as threads. The first method is more robust, as separation of the user and support code is complete, while the second introduces less run-time overheads. A third executable form, useful for testing and for statically configured programs and architectures, simply bypasses the CLAM and employs ACE processes and primitives.

Interoperability with standards like CORBA is currently supported in two ways.

- ASSIST programs can access CORBA functions from within the user code sections by gaining access to an external ORB. The ORB reference can be provided in the application configuration file.
- CORBA services can be implemented by ASSIST modules, for instance by activating a CORBA servant within a program module. Exporting CORBA services at the user code level is a blocking operation, so the module becomes unavailable as long as the ORB is alive, and the servant must actually be a sequential module.

A different, compiler-supported solution, which we are currently developing, allows using as a servant a generic parallel ASSIST subprogram f with unique input and output. The compiler inserts into the program a support process p (actually a ParMod with a single virtual processors) that runs ORB code as a thread. Process p forwards external CORBA requests to f and gets back answers on the ASSIST internal interfaces.

A variant of this scheme has been used to run Java bytecode inside VPs, by starting Java virtual machines as separate processes and forwarding them the tasks to compute. Building on these integration experiences, we plan to

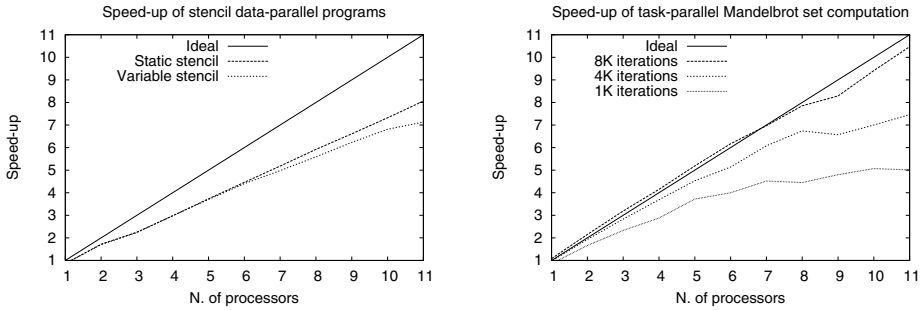


Fig. 4. Speed-up results. (left) Computation of a function with stencil access over an array of 1600x1600 points, computation is 470 ns per point. — (right) Farm load-balancing computation over the same array, Mandelbrot function at 1K, 4K and 8K iterations per point, computation grain is 1280 points per packet.

fully integrate high performance components in more effective ways in the next versions of the ASSIST run-time.

5 First Results

We are currently running simple tests on a small Beowulf cluster in order to test compiler and support functionalities. We have verified that the ParMod implementation can efficiently handle both geometric algorithms (data parallelism with stencil access to non-local data, Fig. 4-left) and dynamic load balancing farm computations, already at a small computation grain (470 ns is 30-300 machine instructions on the Pentium II platform used), Fig. 4-right. Although some support optimization are still missing, test results in Fig. 4-left show that the SMU support for dynamically computed stencil patterns is almost as efficient as that of static (unchanging) stencils.

These results are confirmed by preliminary tests on a computational kernel of molecular dynamic simulation, developed using the ASSIST technology (AssistLib) as part of the ASI-PQE2000 Research Program. More results are found in [8]. To verify the feasibility of mixing data parallelism and task parallelism within the same application kernel, we are currently implementing with ASSIST the C4.5 decision-tree classifier described in [4]. Decision tree classifiers are divide and conquer algorithms with an unbalanced, irregular and data-driven computation tree, and require using data and control-parallel techniques in different phases of the execution, while at the same time managing out-of-core data structures. This kind of applications can be written only with low-level, unstructured programming models (e.g. MPI), and clearly show the limits of pure data-parallel programming models. In [4] we propose a solution based on two ParMod instances and external object support to shared memory data.

6 Related Work

In [9] an extensive comparison of the characteristics of several parallel programming environments is reported, including *ASSIST*, and developing standards for High Performance and Grid components (*CCAFFEINE*, *XCAT*) are referenced. Another notable example reported is *CO₂P₃S*, a language based on parallel design patterns. *CO₂P₃S* is aimed at SMP multithreaded architectures, and its expressive power comes in part from exposing the implementation of patterns to the application programmer. The *Ligature* project [10] also aims at designing a component architecture for HPC, and an infrastructure to assist multicomponent program development. *Kuchen* [11] describes a skeleton-based C++ library which supports both data and task parallelism using a two tier approach, and the classical farm and map skeletons. As a closer approach, we finally mention the *GrADS* project [12]. It aims at smoothing Grid application design, deployment and performance tuning. In *GrADS* the emphasis is much stronger on the dynamic aspects of program configuration, including program recompilation and reoptimization at run-time, and thus performance models and contracts issues. Indeed, in the *GrADS* project the programming interface is a problem solving environment which combines components.

7 Conclusions

We have presented a parallel/distributed programming environment designed to be flexible, expandable and to produce efficient and portable applications. Primary features of language are the ease of integration with existing applications and resources, the ability to mix data and control parallelism in the same application, and the ability to produce application for heterogeneous architectures and Computing Grids. By design, it is possible to exploit different communication support methods by extending the *CLAM* or the *ACE* library. Key resources in this direction are libraries for faster communication on clusters (e.g. active messages) and Computational Grid support libraries like *Nexus* [7].

Future improvements will be the result of a multidisciplinary work, in collaboration with several other research groups. We already made efforts to integrate the *ASSIST* environment with existing parallel numerical libraries [13], and a tool has been developed to help run *ASSIST* applications over Grid Computing Environments [14].

The support of the language has been verified on *Beowulf* clusters. At the time of this writing, we are developing more complex applications to stress the environment and tune performance optimization in the compiler. Among the “heavyweight” applications in development there are data mining algorithms for association rule mining and supervised classification; earth observation applications using SAR interferometry algorithms; computational geometry kernels.

The parallel cooperation model of *ASSIST* allows consideration of a module, especially a *ParMod*, as a software component. It is already possible to use CORBA sequential objects, and to export a CORBA interface from an *ASSIST* program. In a more general setting, *ASSIST* modules can be seen as parallel

software components. Thus, future development of the ASSIST environment will interact with the development of parallel and high-performance component interfaces, exploiting them to connect program modules.

Acknowledgments. We wish to thank D. Laforenza, S. Orlando, R. Perego, R. Baraglia, P. Palmerini, M. Danelutto, D. Guerri, M. Lettere, L. Vaglini, E. Pistoletti, A. Petrocelli, A. Paternesi, P. Vitale.

References

1. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: A structured high level programming language and its structured support. *Concurrency Practice and Experience* **7** (1995) 225–255
2. Bacci, B., Danelutto, M., Pelagatti, S., Vanneschi, M.: SkIE : A heterogeneous environment for HPC applications. *Parallel Computing* **25** (1999) 1827–1852
3. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. *ACM Computing Surveys* **30** (1998) 123–169
4. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* **28** (2002) 1709–1732
5. Vanneschi, M.: ASSIST: an Environment for Parallel and Distributed Portable Applications. Technical Report TR-02-07, Dip. di Informatica, Università di Pisa (2002)
6. Schmidt, D.C., Harrison, T., Al-Shaer, E.: Object-oriented components for high-speed network programming. In: *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems (COOTS)*, Monterey, CA, USENIX (1995) extended version online at <http://www.cs.wustl.edu/~schmidt/ACE-papers.html>.
7. Foster, I., Kesselman, C.: The Globus toolkit. In Foster, I., Kesselman, C., eds.: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann (1998)
8. Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Danelutto, M., Pesciullesi, P., Ravazzolo, R., Torquati, M., Vanneschi, M., Zoccolo, C.: Assist demo: A high level, high performance, portable, structured parallel programming environment at work. In: *Proceedings of EuroPar'03*. (2003) to appear.
9. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Advanced environments for parallel and distributed applications: a view of current status and trends. *Parallel Computing* **28** (2002) 1637–1662
10. Keahey, K., Beckman, P., Ahrens, J.: Ligation: Component architecture for high performance applications. *The International Journal of High Performance Computing Applications* **14** (2000) 347–356
11. Kuchen, H.: A skeleton library. In Monien, B., Feldman, R., eds.: *Euro-Par 2002 Parallel Processing*. Number 2400 in LNCS (2002) 620–629
12. : The GrADS Project. <http://hipersoft.cs.rice.edu/grads/> (2003)
13. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Integrating MPI-based numerical software into an advanced parallel computing environment. In: *Proc. of 11th Euromicro Conf. PDP2003, IEEE* (2003) 283–291
14. Baraglia, R., Danelutto, M., Laforenza, D., Orlando, S., Palmerini, P., Perego, R., Pesciullesi, P., Vanneschi, M.: Assistconf: A grid configuration tool for the assist parallel programming environment. In: *Proc. of 11th Euromicro Conf. PDP2003, IEEE* (2003) 193–200

The Design of an API for Strict Multithreading in C++

Wolfgang Blochinger and Wolfgang Küchlin

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, Sand 14, D-72076 Tübingen, Germany
{blochinger,kuechlin}@informatik.uni-tuebingen.de
<http://www-sr.informatik.uni-tuebingen.de/>

Abstract. This paper deals with the design of an API for building distributed parallel applications in C++ which embody strict multithreaded computations. The API is enhanced with mechanisms to deal with highly irregular non-deterministic computations often occurring in the field of parallel symbolic computation. The API is part of the Distributed Object-Oriented Threads System DOTS. The DOTS environment provides support for strict multithreaded computations on highly heterogeneous networks of workstations.

1 Introduction

High level programming models typically exhibit a high degree of abstraction, thus shielding the programmer from many low-level details of the parallel execution process. In this paper we deal with the *multithreading* parallel programming model [8] (not to be confused with the shared memory model) which is located on a higher level of abstraction than other commonly employed parallel models (e.g. message passing). Within the multithreading parallel programming model, low-level synchronization, communication, and task mapping are carried out completely transparently. Many parallel algorithm models like master-slave, divide-and-conquer, branch-and-bound, and search with dynamic problem decomposition can be realized by multithreaded computations in a straight forward manner. This property makes the multithreading model also particularly attractive for the development of the now popular Internet-computing based parallel applications, provided it is available for highly heterogeneous distributed systems.

Parallel programs that employ the multithreading programming model are called *multithreaded programs*. A *multithreaded computation* results from the execution of a multithreaded program with a given input. A considerable amount of research has been carried out on (static) scheduling techniques for multithreaded computations in order to minimize time and/or space requirements [9,10]. In this paper we study the design of an API (application programmers interface) that supports the creation of multithreaded programs and multithreaded computations with dynamic thread creation. It is part of our parallel middleware DOTS (Distributed Object-Oriented Threads System). The overall design goal of the presented API was on the one hand to support as large a class of multithreaded computations, while on the other hand keeping the API as lean as possible in order to enable the rapid and easy creation as well as maintenance of parallel programs.

The rest of the paper is organized as follows. In Section 2 a graph-theoretic definition and a classification of multithreaded computations are given. Section 3 provides details

of the multithreading API of the DOTS parallel system environment. Section 4 gives a summary and comparison with related work.

2 Definition and Classification of Multithreaded Computations

In this section we summarize the graph-theoretic model for multithreaded computations introduced by *Blumofe and Leiserson* [8] which also provides a means to classify computations of this kind according to their expressiveness.

A multithreaded computation can be described by an *execution graph*. The main structural entities of multithreaded computations are *threads*. Each thread is composed of one or more unit-time *tasks* which represent the nodes of the execution graph. *Continue edges* between tasks indicate their sequential ordering within a thread and also define the extent of the thread.

Tasks of different threads can be executed concurrently, provided that two kinds of dependencies between the threads of a computation are considered:

- A thread can create new threads by executing special tasks, called *spawn tasks*. The resulting parent/child relationship among two threads is indicated by a *spawn edge* which goes from the spawn task of the parent to the first task of the child thread. The parent thread may pass an argument to its child thread along with its creation. The tree composed by all threads of a computation and the corresponding spawn edges is called *spawn tree*. It can be viewed as the parallel variant of a call tree.
- The second type of dependencies stem from producer/consumer relationships between two threads of a computation. A thread can produce one or more results which are later consumed by other threads. This relationship is indicated by *data-dependency edges* which go from the producing to the consuming task.

A multithreaded computation can thus be represented by a directed acyclic graph of unit-tasks connected by continue, spawn and data-dependency edges. Such computations are also called *general multithreaded computations*. In Figure 1 the execution graph of a general multithreaded computation is given. Threads are shaded gray, spawn edges are printed as solid arrows and data-dependency edges are printed as dotted arrows.

Subclasses of general multithreaded computations can be identified by restricting the kinds of data-dependencies that can occur between two threads of a computation. The simplest subclass is the class of *asynchronous procedure calls*. In such computations every thread produces exactly one result which is consumed by its parent. More formally, there is only one outgoing data-dependency edge per thread and it goes to its parent thread (see Figure 2). In *fully strict computations* a thread can produce several results, but the corresponding data-dependency edges also go to its parent thread without exceptions (see Figure 3). The more comprehensive class of *strict computations* is characterized by the property that all data-dependency edges of a thread go to an ancestor of the thread in the spawn tree (see Figure 4).

The class of strict computations plays an important role, since it is the most comprehensive class of multithreaded computations for which certain properties hold [9, 10]. In particular, it is considered to be the largest class of multithreaded computations which can be regarded as “well-structured”. We will take advantage of this property in the design of the API for multithreading.

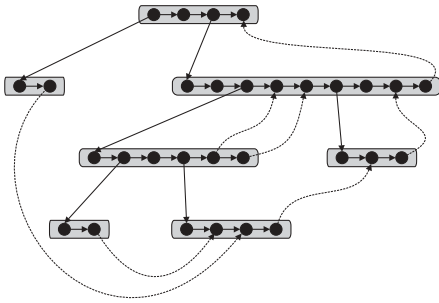


Fig. 1. Execution Graph of a General Multi-threaded Computation

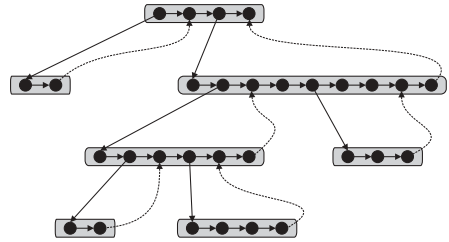


Fig. 2. Execution Graph of a Computation with Asynchronous Procedure Calls

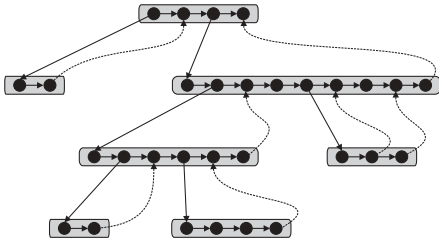


Fig. 3. Execution Graph of a Fully Strict Multi-threaded Computation

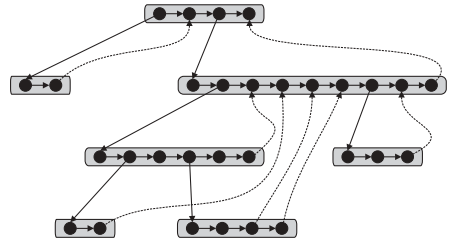


Fig. 4. Execution Graph of a Strict Multi-threaded Computation

3 Multithreaded Computations in DOTS

3.1 DOTS Overview

To provide an appropriate context for the subsequent discussion on multithreading in DOTS, we first give a brief overview of the system. DOTS is a system environment for building and executing parallel C++ programs that integrates a wide range of different computing platforms into a homogeneous parallel environment [3]. Up to now, it has been deployed on heterogeneous clusters composed of the following platforms: Microsoft Windows 95/98/NT/2000/XP, Sun Solaris, SGI IRIX, IBM AIX, FreeBSD, Linux, QNX Realtime Platform and IBM Parallel Sysplex Cluster (clusters of IBM S/390 mainframes running under OS /390) [4].

Although DOTS was originally designed for the parallelization of algorithms from the realm of symbolic computation [5,7,6,21], it has also been used in other application domains like parallel computer graphics [17]. The primary design goal of DOTS was to provide a flexible and handy tool for the rapid prototyping of algorithms, especially supporting highly irregular symbolic computations, e.g. in data-dependent divide-and-conquer algorithms.

The DOTS system environment is organized in several layers. The lowest layer is the OS adaptation layer which is implemented by the ACE (ADAPTIVE Communication Environment) toolkit [1]. It homogenizes operating system APIs for a wide range of

different platforms. On top of the OS adaptation layer, the DOTS run-time system is built. It provides several low-level services like object-serialization, message exchange, TCP connection caching and support for the internal component-architecture. Additionally, higher level services like task migration, node directory, logging, and load distribution are provided by the DOTS run-time.

DOTS applications can be based on several parallel programming models which are represented by separate APIs. It is possible to mix primitives from different APIs within an application.

- The *Task API* represents the basic API layer of DOTS on which all other APIs are based. It supports object-oriented task parallel programming at a basic level using directly the services of the DOTS run-time.
- The *Active Message API* provides support for object-oriented message passing.
- The *Autonomous Tasks API* can be employed to create task objects that operate as mobile agents in a heterogeneous distributed system.
- The *Thread API* provides support for strict multithreaded computations. For the rest of the paper we will focus on this API.

3.2 The DOTS API for Multithreading

The design of the multithreading API of DOTS was guided by three main goals:

- Providing high-level and easy-to-use primitives.
The API should provide a small number of powerful and orthogonal primitives facilitating the creation of distributed parallel programs. Also the parallelization of object-oriented programs should be supported.
- Support for a comprehensive class of multithreaded computations.
In order to support more complex multithreaded computations than asynchronous procedure calls and at the same time preserving the "well-structured" property of computations the class of strict multithreaded computations should be supported.
- Support for typical requirements of the field of symbolic computation.
Typical properties of parallel applications of this field are a high degree of irregularity and non-determinism. Also primitives for cancellation are required in order to efficiently support parallel search procedures.

The key concept of the DOTS API for multithreaded computations is the use of *thread group* objects as links between different primitives of the API. A thread group represents one or more active threads of a computation. Within a program, any number of thread group objects can be used. A thread can be placed *explicitly* or *implicitly* into a thread group. In the former case, the thread group object has to be supplied as argument to a primitive. In the latter case, the corresponding thread group is determined by the dynamic context in which a primitive is executed. For all primitives that are subsequently applied to a thread group no distinctions are made between threads that have been placed explicitly and threads that have been placed implicitly into the group.

The basic primitives provided by the Thread API are `dots_fork` and `dots_hyperfork` for thread creation, `dots_join` for synchronizing with the results computed by other threads

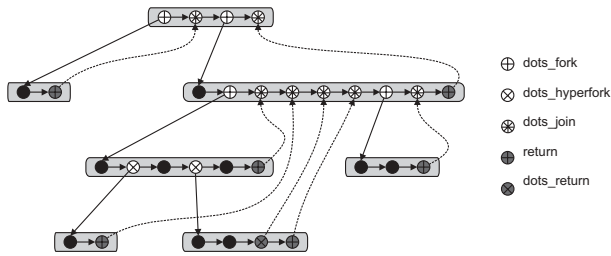


Fig. 5. A Strict Computation with DOTS Primitives

(which are returned using `dots_return`), and `dots_cancel` for thread cancellation. To facilitate the parallelization of existing C++ programs within the multithreading programming model, the DOTS Thread API is enhanced with object-oriented features, like argument and result objects for threads.

If a thread is created using the `dots_fork` primitive, it is placed explicitly into a specified thread group. Whereas, if a thread is created using `dots_hyperfork`, it is placed implicitly in the same thread group as its closest ancestor in the spawn tree which has been created using `dots_fork`. In both cases a procedure to be executed by the child thread and an argument-object for the child thread must be supplied.

Threads return result objects using the `dots_return` primitive. The last result of a thread is delivered by the final return statement of the procedure.

When `dots_join` is called on a thread group *join-any* semantics is applied: The first result which becomes available from any thread in the given group is delivered, regardless of whether the thread has been placed explicitly or implicitly into the group. If no result is available, the calling thread is blocked until one thread of the group delivers a result. If a thread has finished its execution and all result objects of the thread have been joined, it is removed from the thread group. In case `dots_join` is applied to an empty group (i.e. a thread group which doesn't hold explicit or implicit threads any more), 0 is returned by `dots_join`. By checking this return value, the end of a computation can be determined. (Note, that in general the joining thread cannot know how many results are available in a thread group.)

Figure 5 depicts the realization of the strict multithreaded computation previously shown in Figure 4 using DOTS primitives.

When applying the `dots_cancel` primitive to a thread group object, the execution of all threads within this group is aborted and results which have not yet been joined are deleted. All threads are removed from the thread group. A thread can abort its own execution process with `dots_abort`. Analogously it is removed from its thread group and unjoined results of the thread are deleted.

The described interaction of `dots_fork`, `dots_hyperfork` and thread groups enables the programmer to succinctly formulate strict multithreaded computations. Additionally, the join-any property of `dots_join` together with the `dots_cancel` primitive extend this strict multithreading model to a flexible tool for dealing with high degrees of non-determinism, for example occurring in highly irregular search problems. Moreover, since all primitives are completely orthogonal the API can be applied easily.

3.3 Example Program

In this Section we present an example program illustrating the DOTS multithreading primitives. Figure 6 shows the (simplified) code of a parallel search process with dynamic problem decomposition.

```
bool search(SearchDesc search_space) {
    while(!search_space.empty()) {
        // begin search step
        ...
        if (solution_found)
            return true;
        ...
        // end search step

        if (dots_task_queue_length() < SPLIT_THRESHOLD) {
            // split search space and spawn new thread
            SearchDesc new_search_space = split(search_space);
            dots_hyperfork(search, new_search_space);
        }
    }
    return false;
}

int main(int argc, char* argv[]) {
    dots_reg(search); dots_init(argc, argv);

    // read in description of search space
    SearchDesc search_space; read(search_space);

    DOTS_Thread_Group thread_group;
    dots_fork(thread_group, search, search_space);

    // dots_join returns 0 if no more threads
    // are available
    bool solution;
    while (dots_join(thread_group, solution) > 0)
        if (solution==true)
            break;

    if (solution) {
        printf("solution found");
        dots_cancel(thread_group);
    } else
        printf("no solution found");

    return 0;
}
```

Fig. 6. Parallel Search with Dynamic Problem Decomposition Employing Strict Multithreading

The parallel search starts with one search thread which is forked by the main thread. Initially, this search thread has the whole search space assigned. For load balancing, new search threads are (dynamically) spawned. Each additional search thread receives a part of the search space of its parent. New threads are created during the initial phase of the parallel search in order to assign work to all available processors and every time a processor runs out of work when it has completed the execution of a thread. Typically, newly created threads will be assigned to a processor by a receiver initiated strategy which is transparently provided by a built-in DOTS load distribution component. In order to provide enough work that can be transferred to other nodes, new search threads are created when the length of the local task queue falls below a predefined limit.

Since all additional search threads are spawned using `dots.hyperfork` their results need not to be joined by their parents but are directly passed to the main thread resulting in a strict computation. In many search problems, the size of the individual search threads cannot be predicted. The resulting non-determinism is handled by the `join-any` semantics of the `dots_join` primitive. The strict multithreading execution model decouples parent and child threads, leading to a smaller number of active threads. A search thread can complete its execution without prior synchronization with results of its descendants. Moreover, dynamic search space splitting can be carried out in a distributed manner. This enhances the scalability of the search process, compared to an alternative approach, where the master thread is also responsible for performing search space splits and creating corresponding search threads.

4 Related Work

In the last decade, many parallel system environments which are able to support (distributed) multithreaded computations in some way have been developed. In this section we carry out a classification of the more relevant of these approaches into three categories focusing on the programming models which they provide.

4.1 Shared Memory Multithreading Based on Distributed Shared Memory (DSM)

The central theme of approaches to multithreading that fall into this category is to support the shared memory multithreading programming model, e.g. provided by many modern operating systems, on parallel architectures with distributed memory as transparently as possible. Examples of DSM multithreading systems are DSM-Threads [18], Millipede [16] or DSM-PM2 [2].

Since there are no restrictions for carrying out communication between threads in the shared memory model, even general multithreaded computations can be realized with the system environments belonging into this category. But the APIs of these platforms have always to be located on a comparably low-level of abstraction, so the formulation of structured multithreaded computations, like the strict ones, in principle cannot lead to equally structured and compact multithreaded programs.

Additionally, due to the consistency problems caused by the replication of shared data objects, all approaches to DSM multithreading have to cope with the problem of combining efficiency and programmability (transparency) on large scale distributed (heterogeneous) parallel machines. Up to now, no sufficient solutions for this problem domain exist [22].

For applications that do not further profit from DSM, using DOTS can ensure the efficient applicability in larger scale heterogeneous distributed systems and at the same time enabling rapid application development using its high level API for multithreading.

4.2 HPC Middleware for Integrating Communication and Multithreading

The system platforms discussed in this section pursue the tight integration of communication and multithreading. However, the main intention of their parallel programming models is not to carry out communication transparently, resulting in a lower level of abstraction. Typical representatives are Nexus [15], Panda [20] or Athapascan-0 [11].

These system platforms are primarily designed to be used as compiler targets or as middleware for building higher-level parallel system platforms. Due to their general nature, general multithreading can be realized with all of these platforms but communication is not transparent on the application programming level. Using such low level programming models can lead to artificially complex programs of well structured computations like strict computations.

4.3 Platforms Supporting the Fork/Join Multithreading Programming Model

Systems in this category are the most similar to DOTS; they support distributed multithreading by providing a fork/join like API. This approach to distributed multithreading carries out communication completely transparently by using argument-result semantics. However, communication between the threads of a computation is restricted to specific points during the execution of a thread.

DTS [12] (which is the predecessor of DOTS) realizes asynchronous remote procedure calls in C and Fortran. No support for object-oriented programming is provided in DTS and its deployment is limited to distributed systems composed of UNIX nodes.

Cilk [19] is a language for multithreaded parallel programming that represents a superset of ANSI C. It uses pre-compilation techniques for static code instrumentation in order to support the Cilk runtime system. There exists a prototype implementation of a distributed version of Cilk, called distributed Cilk [14] that spans clusters of SMPs.

The current version of the Cilk language features a very compact and easy to use set of parallel keywords, but no support for strict multithreaded computations is included. DOTS is library based and therefore avoids typical problems of systems that extend standard languages like the lack of standard development tools (e.g. debuggers). Moreover, DOTS is based on C++ and supports object-oriented programming. Since distributed Cilk is currently available only on a few platforms, its usability in highly heterogeneous distributed environments is limited.

Virtual Data Space (VDS) [13] is a load balancing system for irregular applications also supporting strict multithreading. In contrast to DOTS the VDS API for strict computations is very complex. For example, the VDS Fibonacci program presented in [13] needs about double the number of lines of code than an equivalent DOTS program of the same functionality.

VDS is implemented in C and therefore provides no direct support for object-oriented programming and it is not available for a wider range of common platforms, resulting in limited support for heterogeneous high performance computing.

5 Conclusion

In this paper we presented an API for building parallel programs based on the multithreading parallel programming model. Our approach is distinguished from related work by supporting the comprehensive class of strict multithreaded computations while at the same time keeping the API compact and easy to use by completely orthogonal primitives. Additionally, the API includes primitives for efficiently dealing with non-determinism occurring in highly irregular computations.

References

- [1] The ADAPTIVE Communication Environment (ACE). <http://www.cs.wustl.edu/~schmidt/ACE.html>.

- [2] ANTONIU, G., AND BOUGÉ, L. DSM-PM2: A portable implementation platform for multi-threaded dsm consistency protocols. In *In Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)* (San Francisco, April 2001), vol. 2026 of *Lect. Notes in Comp. Science*, Springer-Verlag, pp. 55–70.
- [3] BLOCHINGER, W. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)* (San Francisco, CA, U.S.A., April 2001), IEEE Computer Society Press, p. 90.
- [4] BLOCHINGER, W., BÜNDGEN, R., AND HEINEMANN, A. Dependable high performance computing on a Parallel Sysplex cluster. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)* (Las Vegas, NV, U.S.A., June 2000), H. R. Arabnia, Ed., vol. 3, CSREA Press, pp. 1627–1633.
- [5] BLOCHINGER, W., KÜCHLIN, W., LUDWIG, C., AND WEBER, A. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation* 49 (1999), 161–178.
- [6] BLOCHINGER, W., SINZ, C., AND KÜCHLIN, W. Parallel consistency checking of automotive product data. In *Proc. of the Intl. Conf. ParCo 2001: Parallel Computing – Advances and Current Issues* (Naples, Italy, 2002), G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, Eds., Imperial College Press, pp. 50–57.
- [7] BLOCHINGER, W., SINZ, C., AND KÜCHLIN, W. Parallel propositional satisfiability checking with distributed dynamic learning. *J. Parallel Computing* (2003). To appear.
- [8] BLUMOF, R. D., AND LEISERSON, C. E. Space efficient scheduling of multithreaded computations. In *Proc. of the Twenty Fifth Annual ACM Symp. on Theory of Computing* (San Diego, CA, May 1993), pp. 362–371.
- [9] BLUMOF, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In *35th Annual Symp. on Foundations of Computer Science (FOCS '94)* (Mexico, November 1994), pp. 356–368.
- [10] BLUMOF, R. D., AND LEISERSON, C. E. Space-efficient scheduling of multithreaded computations. *SIAM Journal Computing* 27, 1 (Feb 1998), 202–229.
- [11] BRIAT, J., GINZBURG, I., PASIN, M., AND PLATEAU, B. Athapaskan runtime : Efficiency for irregular problems. In *Proc. of the Europar'97 Conference* (Passau, Germany, August 1997), Springer Verlag, pp. 590–599.
- [12] BUBECK, T., HILLER, M., KÜCHLIN, W., AND ROSENSTIEL, W. Distributed symbolic computation with DTS. In *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR'95* (Lyon, France, Sep 1995), A. Ferreira and J. Rolim, Eds., vol. 980 of *LNCS*, Springer-Verlag, pp. 231–248.
- [13] DECKER, T. Virtual data space - load balancing for irregular applications. *Parallel Computing* 26 (2000), 1825–1860.
- [14] distributed Cilk. <http://supertech.lcs.mit.edu/cilk/home/distcilk5.1.html>.
- [15] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The nexus approach to integrating multi-threading and communication. *Journal of Parallel and Distributed Computing* 37 (1996), 70–82.
- [16] FRIEDMAN, R., GOLDIN, M., ITZKOVITZ, A., AND SCHUSTER, A. Millipede: Easy parallel programming in available distributed environment. *Software: Practice and Experience* 27, 8 (August 1997), 929–965.
- [17] MEISSNER, M., HÜTTNER, T., BLOCHINGER, W., AND WEBER, A. Parallel direct volume rendering on PC networks. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)* (Las Vegas, NV, U.S.A., July 1998), H. R. Arabnia, Ed., CSREA Press.
- [18] MUELLER, F. On the design and implementation of DSM-threads. In *Int. Conference on Parallel and Distributed Processing Techniques and Applications* (June 1997), pp. 315–324.

- [19] RANDALL, K. H. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.
- [20] RÜHL, T., BAL, H. E., BENSON, G., BHOEDJANG, R. A. F., AND LANGENDOEN, K. Experience with a portability layer for implementing parallel programming systems. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)* (Sunnyvale, CA, 1996), pp. 1477–1488.
- [21] SINZ, C., BLOCHINGER, W., AND KÜCHLIN, W. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)* (Boston, MA, June 2001), H. Kautz and B. Selman, Eds., vol. 9 of *Electronic Notes in Discrete Mathematics*, Elsevier Science Publishers.
- [22] TANENBAUM, A. S., AND VAN STEEN, M. *Distributed Systems – Principles and Paradigms*. Prentice-Hall, 2002.

High-Level Process Control in Eden^{*}

Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,klusik,loogen,priebe,weskamp}@informatik.uni-marburg.de

Abstract. High-level control of parallel process behaviour simplifies the development of parallel software substantially by freeing the programmer from low-level process management and coordination details. The latter are handled by a sophisticated runtime system which controls program execution. In this paper we look behind the scenes and show how the enormous gap between high-level parallel language constructs and their low-level implementation has been bridged in the implementation of the parallel functional language Eden. The main idea has been to implement the process control in a functional language and to restrict the extensions of the low-level runtime system to a few selected primitive operations.

1 Introduction

A growing number of applications demand a large amount of computing power. This calls for the use of parallel hardware including clusters and wide-area grids of computers, and the development of software for these architectures. Parallel programming is however hard. The programmer usually has to care about process synchronisation, load balancing, and other low-level details, which makes parallel software development complex and expensive. Our approach, Eden [3], aims at simplifying the development of parallel software. Eden extends the lazy functional language Haskell [14] by syntactic constructs for *explicitly* defining processes. Eden's process model provides direct control over process granularity, data distribution and communication topology while process creation and placement of processes, necessary communication and synchronisation are automatically managed by the runtime system (RTS), i.e. the implementation of Eden's virtual machine. In his foreword to [6] Simon Peyton Jones writes: "*Parallel functional programs eliminate many of the most unpleasant burdens of parallel programming. . . . Parallelism without tears, perhaps? Definitely not. . . . The very high-level nature of a parallel functional program leaves a large gap for the implementation to bridge.*" The Eden implementation¹ is based on the Glasgow Haskell Compiler (GHC) [13]. By embedding Eden's syntax into Haskell, we could use the front-end of the compiler without any changes. The bulk of the

^{*} Work supported by the DAAD (German Academic Exchange Service)

¹ Freely available at www.mathematik.uni-marburg.de/inf/eden and via the GHC CVS repository (see www.haskell.org/ghc).

extensions lies in the back end of the compiler and the RTS [2,7]. Kernel parts of the parallel functional RTS, like thread and memory management, are shared with GUM, the implementation of GpH [19]. Eden's RTS is implemented on top of the message passing library PVM [15].

In this paper we describe how the Eden implementation has been re-organised in layers to achieve more flexibility and to improve the maintainability of this highly complex system. The main idea underlying Eden's new layered implementation shown in Fig. 1 is to lift aspects of the RTS to the level of the functional language,

i.e. defining basic workflows on a high level of abstraction and concentrating low-level RTS capabilities in a couple of primitive operations. In this way, part of the complexity has been eliminated from the imperative RTS level.

Eden programmers will typically develop parallel programs using the Eden language constructs, together with parallel skeletons provided in special libraries [9]. This is briefly described in Section 2. Every Eden program must import the Eden module, which contains Haskell defini-

tions of Eden's language constructs, as explained in Section 3. These Haskell definitions use primitive operations which are functions implemented in C that can be accessed from Haskell. The extension of GHC for Eden is mainly based on the implementation of these new primitive operations, which provide the elementary functionality for Eden. The paper ends with a discussion of related work and conclusions.

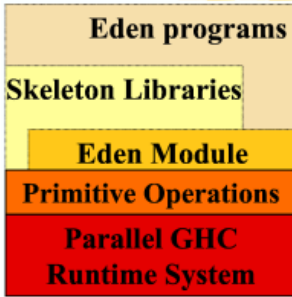


Fig. 1. Layer structure of the Eden system

2 Eden's Main Features

Eden gives the programmer high-level control over the parallel behaviour of a program by introducing the concept of processes into the functional language Haskell [14]. Evaluation of the expression **(process funct) # arg** leads to the creation of a new process for evaluating the application of the function **funct** to the argument **arg**. The argument **arg** is evaluated locally and sent to the newly created process. With the high-level Eden constructs:

- **process :: (Trans a, Trans b) => (a -> b) -> Process a b**
to transform a function into a process abstraction and
- **(#) :: (Trans a, Trans b) => Process a b -> a->b** to create a process,

the programmer can concentrate on partitioning the algorithm into parallel sub-tasks, thereby taking into account issues like task granularity, topology, and data distribution. Eden processes are eagerly instantiated, and instantiated processes produce their output even if it is not demanded. These deviations from lazy evaluation aim at increasing the parallelism degree and at speeding up the

distribution of the computation. In general, a process is implemented by several threads concurrently running in the same processor, so that different values can be produced independently. The concept of a virtually shared global graph is avoided, to save the administration costs while paying the price of possibly duplicating work.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. When trying to access input which is not available yet threads are temporarily suspended. The type class **Trans** (short for transmissible) comprises all types which can be communicated and provides overloaded, implicitly used functions for sending values. All primitive types like **Int**, **Bool**, **Char** etc., pre- and user-defined algebraic types² as well as function and process abstraction types belong to **Trans**.

Example: The following function is at the heart of a simple ray-tracer program. It computes an image with **y** lines and **x** columns of a scene consisting of spheres. The sequential function body of the **ray** function is simply the expression `map (traceLine x world) [0..y-1]`. The parallel version produces the image by several processes each computing a chunk of lines:

```
ray :: Int -> Int -> Int -> [Sphere] -> [[RGB]]
ray chunk x y world
  = concat ([process (map (traceLine x world)) # linenumbers
            | linenumbers <- splitAtN chunk [0..y-1]]
          'using' spine)
```

The function `concat` flattens a list of lists into a list, thus removing one level of nested lists — the one introduced by the list of processes. The addendum `'using' spine` is needed to produce early demand for the evaluation of the process instantiations. ◀

Many-to-one communication is supported by a predefined process abstraction **merge** which, when instantiated, does a fair merging of input streams into a single (non-deterministic) output stream. Moreover, an Eden process may explicitly generate a new *dynamic input channel* (of type **ChanName a**) and communicate the channel's name to another process, which can use the channel for answering directly. This enables the creation of arbitrary communication topologies despite the tree-like generation of process systems. The primitive operations for dynamic channel creation are also used to establish the channels between parent and child processes during process instantiation (see Section 3).

The task of parallel programming is further simplified by a library of predefined skeletons [9]. Skeletons are higher-order functions defining parallel interaction patterns which are shared in many parallel applications [16]. The programmer can simply use such known schemes to achieve an instant parallelisation of a program. The automatic selection of appropriate skeletons during compile-time using new Haskell meta-programming constructs is shown in [5]. The recent journal paper [8] shows that Eden achieves good performance in comparison with Glasgow parallel Haskell (GpH) [4] and Parallel ML with Skeletons (PMLS) [10].

² For user-defined algebraic types, instances of **Trans** must be derived.

3 A Layered Implementation of Coordination

This section considers the internals of Eden’s layered implementation. These are transparent for the ordinary programmer except that every Eden program must import the Eden module. This module contains Haskell definitions of the high-level Eden constructs, thereby making use of the eight primitive operations shown in Fig. 2. The primitive operations implement basic actions which have to be performed directly in the runtime system of the underlying sequential compiler GHC³. In the Eden module, every primitive operation is wrapped in

1. <code>createProcess#</code>	request process instantiation on another processor
2. <code>createDC#</code>	create communication channel
3. <code>setChan#</code>	connect communication channel in the proper way
4. <code>sendHead#</code>	send head element of a list on a communication channel
5. <code>sendVal#</code>	send single value on a communication channel
6. <code>noPE#</code>	determine number of processing elements in current setup
7. <code>selfPE#</code>	determine own processor identifier
8. <code>merge#</code>	nondeterministic merge of a list of outputs into a single input

Fig. 2. Primitive Operations for Eden

a function⁴, thereby providing additional type information, triggering execution and limiting actual use to the desired degree. Information provided by primitives like `selfPE#` or `noPE#` should e.g. only be used for process control. The misuse of such operations may introduce nondeterministic effects.

In the following, we abstract from low-level implementation details like graph reduction and thread management which are well-understood and explained elsewhere [12,19,2]. Instead we focus on coordination aspects, in particular process abstractions and process instantiations and their implementation in the Eden module. Communication channels are now explicitly created and installed to connect processes. Primitives provided for handling Eden’s dynamic input channels are used for that purpose.

3.1 Channels and Communication

As explained in Section 2, the type class `Trans` comprises all types which can be communicated. It provides an overloaded function `sendChan :: a -> ()` for sending values along communication channels where the channel is known from the context of its application. The additional functions `tupsize` and `writeDCs` shown in Fig. 3 will be explained later. The context `NFData` (normal form data)

³ Note that, in GHC, primitive operations and types are distinguished from common functions and types by `#` as the last sign in their names.

⁴ With the same name as the primitive operation, except for the `#`.

is needed to ensure that transmissible data can be fully evaluated (using the overloaded function `rnf` (reduce to normal form)) before sending it (using the primitive operation `sendVal#` wrapped by `sendVal`). Lists are transmitted in a *stream*-like fashion, i.e. element by element. For this, `sendChan` is specialized to `sendStream` which first evaluates each list element to normal form and transmits it using `sendHead` (see Fig. 3).

```

class NFData a => Trans a where
  sendChan :: a -> ();           sendChan x = rnf x 'seq' sendVal x
  tupsize  :: a -> Int;          tupsize _ = 1
  writeDCs :: ChanName a -> a -> (); writeDCs (cx:_) x = writeDC cx x
                                   -- default definitions, changed
                                   -- appropriately for tuples and lists

instance Trans a => Trans [a] where
  sendChan x = sendStream x

  sendStream :: Trans a => [a] -> ()
  sendStream [] = sendVal []
  sendStream (x:xs) = (rnf x) 'seq' ((sendHead x) 'seq' (sendStream xs))

```

Fig. 3. Type Class `Trans` with List Instance

Before any communication can take place, a channel must be created and installed. For this, the functions shown in Fig. 4 are provided. The RTS equivalent to channels is a structure *Port* which contains (1) a specific Port number, (2) the process id (pid), and (3) the processor number, forming a unique port address. At module level, these port addresses (connection points of a channel to a process) are represented by the type `ChanName`'. Objects of this type contain exactly the port identifier triple (see Fig. 4). Note that the higher level type `ChanName a` is a *list* of port identifiers; one for each component of type `a` in case `a` is a tuple.

The function `createDC :: Trans a => a -> (ChanName a, a)` creates a new (dynamic) input channel, i.e. a channel on which data can be received, using the corresponding primitive operation `createDC#`. It yields the channel name which can be sent to other processes and (a handle to) the input that will be received via the channel. If `createDC` is used for tuple types `a`, a list of port identifiers (type `ChanName`') and a tuple of input handles will be created. To ensure correct typing, `createDC` is always applied to its second output, but will only use it to determine the needed number of channels, using the overloaded function `tupsize` in class `Trans`.

Data transmission is done by the function `writeDC`. This function takes a port identifier and a value, connects the current thread to the given port (`setChan#`) and sends the value using the function `sendChan`. The connection by `setChan#` prior to evaluating and sending values guarantees that the receiver of data mes-

```

type ChanName a = [ChanName' a]
data ChanName' a = Chan Int# Int# Int#

createDC :: a -> (ChanName a, a)
createDC t = let (I# i#) = tupSize t
              in case createDC# i# of (# c,x #) -> (c,x)

writeDC  :: ChanName' a -> b -> ()
writeDC chan a = setChan chan 'seq' sendChan a

```

Fig. 4. Channel Creation and Communication

sages is always defined when `sendVal#` or `sendHead#` is called. While `writeDC` defines the behaviour of a single thread, the overloaded function `writeDCs` (see Fig. 3) handles tuples in a special way. It takes a list of port identifiers (length identical to the number of tuple components) and creates a thread for each tuple component. The instance declaration of `Trans` for pairs is e.g. as follows:

```

instance (Trans a, Trans b) => Trans (a,b) where
  tupsize _ = 2
  writeDCs (cx:cy:_) (x,y) = writeDC cx x 'fork' writeDC cy y

```

The Eden module contains corresponding instance declarations for tuples with up to eight components.

3.2 Process Handling

Subsequently, we will focus on the module definitions for process abstraction and instantiation shown in Fig. 5 and 6. Process creation can be defined on this level, using the internal functions to create channel names and to send data on them, plus the primitive operation `createProcess#` for forking a process on a remote processor.

A process abstraction of type `Process a b` is implemented by a function `f_remote` (see Fig. 5) which will be evaluated remotely by a corresponding child process. It takes two channel names: the first `outDCs` (of type `ChanName b`) is a channel for sending its output while the second `chanDC` (of type `ChanName (ChanName a)`) is an administrative channel to return the names of input channels to the parent process. The exact number of channels which are established between parent and child process does not matter in this context, because the operations on dynamic channels are overloaded. The definition of `process` shows that the remotely evaluated function, `f_remote`, creates its input channels via the function `createDC`. Moreover, `writeDCs` is used twice: the dynamically created input channels of the child, `inDCs`, are sent to the parent process via the channel

```

data (Trans a, Trans b) =>
  Process a b = Proc (ChanName b -> ChanName (ChanName a) -> ())
process :: (Trans a, Trans b)
  => (a -> b) -> Process a b
process f = Proc f_remote
  where f_remote outDCs chanDC
    = let (inDCs, invals) = createDC invals
      in writeDCs chanDC inDCs 'fork'
        (writeDCs outDCs (f invals))

```

Fig. 5. Haskell definitions of Eden process abstraction

```

( # ) :: (Trans a, Trans b) => Process a b -> a -> b
pabs # inps = case createProcess (-1#) pabs inps of Lift x -> x
data Lift a = Lift a

createProcess :: (Trans a, Trans b) =>
  Int# -> Process a b -> a -> Lift b
createProcess on# (Proc f_remote) inps
  = let (outDCs, outvals) = createDC outvals
      (chanDC, inDCs) = createDC inDCs
      pinst = f_remote outDCs chanDC
  in outDCs 'seq' chanDC 'seq'
    case createProcess# on# pinst of
      1# -> writeDCs inDCs inps 'fork' (Lift outvals)
      _ -> error "process creation failed"

```

Fig. 6. Haskell definitions of Eden process instantiation

`chanDC` and the results of the process determined by evaluating the expression `(f invals)` are sent via the channels `outDCs`⁵.

Process instantiation by the operator `( # )` defines process creation on the parent side. To cope with lazy evaluation and to get back control without waiting for the result of the child process, the process results are lifted to an immediately available weak head normal form using the constructor `Lift`. Before returning the result, the `Lift` is removed. The function `createProcess` takes the process abstraction and the input expression and yields the lifted process result. The placement parameter `on#` is a primitive integer (type `Int#`) which can be used to allocate newly created processes explicitly. The current system does not make use of this possibility, processes are allocated round-robin or randomly on the available processors. The channels are handled using `createDC` and `writeDCs` in the

⁵ The prefixes `in` and `out` in channel names in Fig. 5 and 6 reflect the point of view of a child process. Thus, in a process instantiation, the inputs `inps` for the child are written into the channels `inDCs`, which are outputs of the parent process.

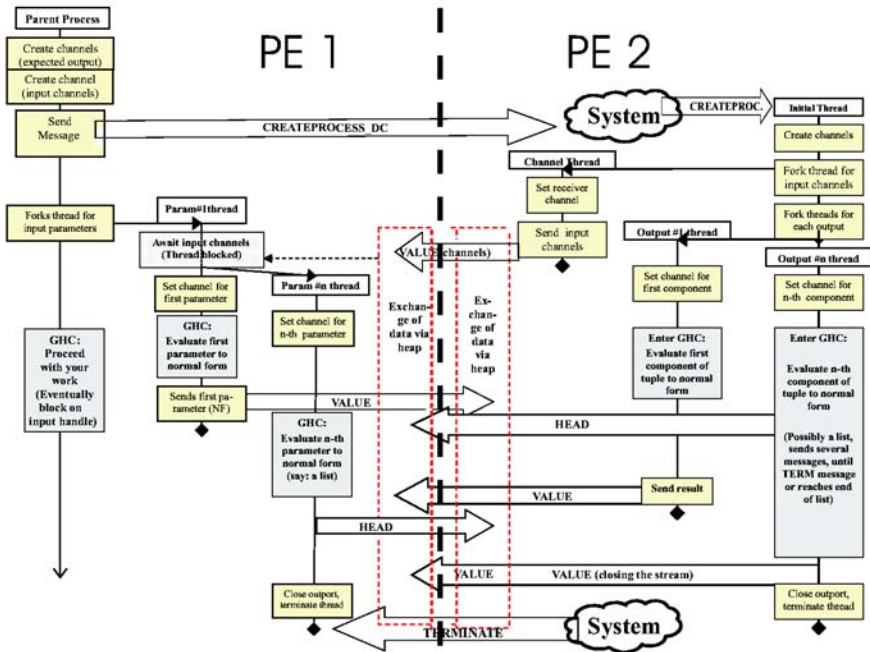


Fig. 7. Sequence Diagram for Process Instantiation

same way as on the child side (see the process abstraction). The remote creation of the child process is performed by the primitive operation `createProcess#`.

The whole sequence of actions is shown in Fig. 7, which illustrates the interplay of the codes in Fig. 5 and 6. Process creation is preceded by the creation of new channels (one for every result) plus one additional port to receive channels for input parameters upon creation. The primitive `createProcess#` sends a message `createProcessDC` to a remote processor, which contains these channels and the process abstraction (an unevaluated `Proc f.remote` packed as a subgraph).

The remote processor (PE 2 in Fig. 7) receives this message, unpacks the graph and starts a new process by creating an initial thread. As the first thread in a process, this thread plays the role of a *process driver*. Evaluating the code shown in Fig. 5, it forks a first thread to communicate channels and then evaluates the results, forking one thread for each tuple component but the last, which is evaluated and sent back by the initial thread itself. Thus, one thread is only used to communicate channels, the other threads evaluate the output expressions. Threads block on the created input handles if they need their arguments. As soon as the input arrives, these threads are reactivated by the communication handler when it writes the received values into the heap.

This concludes our discussion of fundamental mechanisms in the Eden system.

4 Related Work and Conclusions

Implementations of parallel functional languages are either based on a parallel abstract machine or on parallel libraries, linked to an ordinary sequential system. Eden is a typical example for the monolithic approach (parallel abstract machine). It is closely related to GpH [19], using the same framework and even sharing a lot of code. Although GpH, combined with evaluation strategies [18] provides comparable control, it generally follows the concept of implicit, or “potential” parallelism, whereas Eden passes parallelism control to the programmer. During the last decade, the extension of functional languages for parallelism moved from implicit to more and more explicit approaches, because it became clear that an effective parallelisation needs some support from the programmer. The *para-functional programming* approach [11], as well as *Caliban* [17], provide explicit control over subexpression evaluation and process placement, going essentially further than our process concept.

Providing a set of parallel skeletons [16] is another way of implementing (more or less explicit) parallelisation facilities. Parallel skeletal languages, as e.g. P³L [1], provide a fixed set of skeletons, sometimes combined with sophisticated program analysis, as e.g. in PMLS [10]. The skeleton implementation is usually out of reach for the programmer, whereas in Eden, programming a skeleton requires nothing but ordinary parallel functional programming.

By lifting the implementation of explicit low-level process control out of the RTS into the functional language itself, we reach two goals: As new techniques like Grid-Computing evolve, it becomes more and more important for a parallel programming system to provide not only good performance and speed-up behaviour, but also to make parallel programming as simple as possible. We achieve this not only on the highest level, i.e. for the ordinary application programmer, but also for the advanced programmer interested in the development of skeletons or even parallel extensions. An advantage of our layered implementation is that developers can use the high-level layers of module and skeleton library which we have introduced.

With rising demand for efficient compilers and better exploitation of parallel computers, compiler construction is getting much more complex. By the layer concept, we gain advantages on portability, code reuse, extensibility, maintenance, and abstraction on the implementation side. The implementation based on a few primitive operations leads to clean interfaces between the implementation layers, which makes it easier to follow version changes of the underlying sequential compiler. This is the first step to our long-term objective: the design of a generic parallel extension of sequential functional runtime systems, on top of which various parallel functional languages could be implemented.

Acknowledgements. We thank Hans-Wolfgang Loidl, Simon Marlow and Fernando Rubio for their support during the development of the Eden runtime system. Additionally, we are grateful to Phil Trinder and the anonymous referees for helpful comments on this paper.

References

1. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P³L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.
2. S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98*, LNCS 1490, pages 318–334. Springer, 1998.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical report, 1996. Available at <http://www.mathematik.uni-marburg.de/inf/eden>.
4. Glasgow Parallel Haskell. WWW page. <http://www.cee.hw.ac.uk/~dsg/gph/>.
5. K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. In *HLPP 2003. Paris, France*, June 2003.
6. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
7. U. Klusik, Y. Ortega-Mallén, and R. Peña Marí. Implementing Eden – or: Dreams Become Reality. In *IFL'98*, LNCS 1595, pages 103–119. Springer, 1999.
8. H.-W. Loidl, F. Rubio Diez, N. Scaife, K. Hammond, U. Klusik, R. Loogen, G. Michaelson, S. Horiguchi, R. Pena Mari, S. Priebe, A. R. Portillo, and P. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-order and Symbolic Computation*, 16(3), 2003.
9. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
10. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.*, 16:181–206, 2001.
11. R. Mirani and P. Hudak. First-Class Schedules and Virtual Maps. In *FPCA'95*, pages 78–85. ACM Press, June 1995.
12. S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. of Functional Programming*, 2(2):127–202, 1992.
13. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *JFIT'93*, pages 249–257, March 1993. <http://www.dcs.gla.ac.uk/fp/papers/grasp-jfit.ps.Z>.
14. S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at <http://www.haskell.org/>.
15. PVM: Parallel Virtual Machine. WWW page. <http://www.epm.ornl.gov/pvm/>.
16. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
17. F. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Department of Computing, Imperial College, London, 1997. <http://www.lieder.demon.co.uk/thesis/thesis.ps.gz>.
18. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, 1998.
19. P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*, pages 78–88. ACM Press, May 1996.

Using Skeletons in a Java-Based Grid System

Martin Alt and Sergei Gorlatch

Technische Universität Berlin, Germany
{mnalt|gorlatch}@cs.tu-berlin.de

Abstract. Grid systems connect high-performance servers via the Internet and make them available to application programmers. This paper addresses the challenge of software development for Grids, by means of reusable algorithmic patterns called *skeletons*. Skeletons are generic program components, which are customizable for a particular application and can be executed remotely on high-performance Grid servers. We present an exemplary repository of skeletons and show how a particular application, FFT (Fast Fourier Transform), can be expressed using skeletons and then executed using RMI (Remote Method Invocation). We describe a prototypical Java-based Grid system, present its optimized RMI mechanism, and report experimental results for the FFT example.

1 Introduction

Grid systems connect high-performance computational servers via the Internet and make them available to application programmers. While the enabling infrastructures for Grid computing are fairly well developed [1], initial experience has shown that entirely new approaches are required for Grid programming [2]. A particular challenge is the phase of algorithm design: since the type and configuration of the servers on which the program will be executed is not known in advance, it is difficult to make the right design decisions, to perform program optimizations and estimate their impact on performance.

We propose to address Grid programming by providing the application programmers with two kinds of software components on the server side: (1) traditional library functions, and (2) reusable, high-level patterns, called *skeletons*. Skeletons are generic algorithmic components, customizable for particular applications by means of their functional parameters. Time-intensive skeleton calls are executed remotely on high-performance Grid servers, where architecture-tuned, efficient parallel implementations of the skeletons are provided.

The contributions and organization of the paper are as follows: We present the structure of our Grid system based on Java and RMI and explain the advantages of skeletons on the Grid (Sect. 2). We introduce a repository of skeletons for expressing parallel and distributed aspects of Grid applications (Sect. 2.1), discuss the inefficiency of standard Java RMI on the Grid and propose using future-based RMI (Sect. 2.2). We show how a mathematical specification of FFT is expressed using our skeletons, develop a skeleton-based Java program and report experimental performance results for it (Sect. 3). We conclude by discussing our results in the context of related work.

2 Our Grid Prototype

Our prototypical Grid environment (Fig. 1) consists of two university LANs – one at the Technical University of Berlin and the other at the University of Erlangen. They are connected by the German academic internet backbone (WiN), covering a distance of approx. 500 km. There are three high-performance servers in our Grid: a shared-memory multiprocessor SunFire 6800, a MIMD supercomputer of type Cray T3E, and a Linux cluster with SCI network. Application programmers work from clients (PCs and workstations). A central entity called “lookup service” is used for resource discovery. The reader is referred to [3] for details of the system architecture and the issues of resource discovery and management.

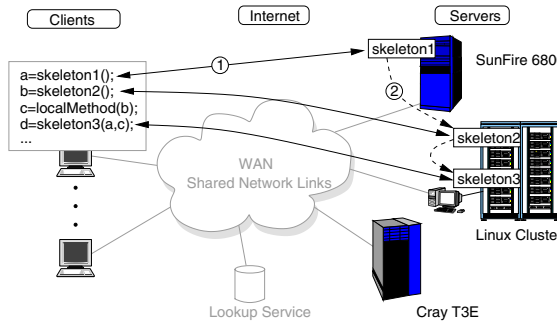


Fig. 1. System architecture and interaction of its parts

We propose developing application programs for such Grid systems using a set of reusable, generic components, called *skeletons*. As shown in the figure, a program on a client is expressed as a sequential composition of skeleton calls and local calls. The servers in the Grid provide architecture-tuned implementations of the skeletons: multithreaded, MPI-based, etc. Applications composed of skeletons can thus be assigned for execution to particular servers in the Grid with a view to achieving better performance.

Time-intensive skeleton calls are executed remotely on servers which provide implementations for the corresponding skeleton (arrow ① in the figure). If two subsequent skeleton calls are executed on different servers, then the result of the first call must be communicated as one of inputs for the second call (arrow ②). This situation is called *composition of skeletons*.

Using skeletons for programming on the Grid has the following advantages:

- Skeletons’ implementations on the server side are usually highly efficient because they can be carefully tuned to the particular server architecture.
- The once-developed, provably correct implementation of a skeleton on a particular server can be reused by different applications.
- Skeletons hide the details about the executing hardware and the server’s communication topology from the application programmer.
- Skeletons provide a reliable model for performance prediction, providing a sound information base for selecting servers.

2.1 A Repository of Skeletons

In the following, we describe a (by no means exhaustive) collection of skeletons. Since at least some of the skeletons' parameters are functions, skeletons can be formally viewed as higher-order functions. In practice, functional parameters are provided as program codes, in our system as Java bytecodes.

We begin our presentation with simple skeletons that express data parallelism:

Map: Apply a unary function f to all elements of a list:

$$\text{map}(f, [x_1, \dots, x_n]) = [f(x_1), \dots, f(x_n)]$$

Scan: Compute prefix sums of a list by traversing the list from left to right and applying a binary associative operator $\oplus$:

$$\text{scan}(\oplus, [x_1, \dots, x_n]) = [x_1, (x_1 \oplus x_2), \dots, (\dots(x_1 \oplus x_2) \oplus x_3) \oplus \dots \oplus x_n]$$

A more complex data-parallel skeleton, DH (Distributable Homomorphism) [4], expresses a divide-and-conquer pattern with parameter operators $\oplus$ and $\otimes$:

DH: Formally, $dh(\oplus, \otimes, x)$ transforms a list $x = [x_1, \dots, x_n]$ of size 2^l into a result list $y = [y_1, \dots, y_n]$, whose elements are computed as follows:

$$y_i = \begin{cases} u_i \oplus v_i, & \text{if } i \leq n/2 \\ u_{i-n/2} \otimes v_{i-n/2}, & \text{otherwise} \end{cases} \quad (1)$$

where $u = dh(\oplus, \otimes, [x_1, \dots, x_{n/2}])$ and $v = dh(\oplus, \otimes, [x_{n/2+1}, \dots, x_n])$.

In addition to these data-parallel skeletons, we provide two auxiliary skeletons, whose aim is efficient communication between client and server:

Replicate: Create a new list containing n times element x : $\text{repl}(x, n) = [x, \dots, x]$. The *repl* skeleton can be called remotely on a server to create there a list of n identical elements, instead of sending the whole list over the network.

Apply: Applies a unary function f to a parameter x : $\text{apply}(f, x) = f(x)$.

The *apply* skeleton is used to remotely execute a function f by shipping its code to the server, rather than moving the data to the client, executing the function locally and then sending the result to the server again.

Our skeleton-based Grid programming environment for the system shown in Fig. 1 is built on top of Java and RMI. We chose the Java platform mostly for reasons of portability (see [5] for “10 reasons to use Java in Grid computing”).

In the system, skeletons are offered as Java (remote) interfaces, which can be implemented in different ways on different servers. To be as flexible as possible, all skeletons operate on `Object`s or arrays of `Object`. For example, the interface for the *scan* skeleton contains a single method

```
public Object[] invoke(Object[], BinOp opplus);
```

To use the `scan` skeleton, the client first finds a server for execution, using the lookup service (see [3] for details). After obtaining an RMI reference to the `scan` implementation on the server, the skeleton is executed via RMI by calling the `invoke` method with appropriate parameters.

2.2 Future-Based RMI for the Grid

Using the RMI mechanism in Grid programs has the important advantage that the outsourcing of skeleton calls to remote servers is transparent for the programmer: remote calls are coded in exactly the same way as local calls. However, since the RMI mechanism was developed for client-server systems, it is not optimal for the Grid. We illustrate this using the following example: a composition of two skeleton calls, with the result of the first call being used as an argument of the second call (`skeleton1` and `skeleton2` are remote references):

```
result1 = skeleton1.invoke(...);
result2 = skeleton2.invoke(result1,...);
```

Executing such a composition of methods using standard RMI, the result of a remote method invocation is always sent back directly to the client. This is exemplified for the above example in Fig. 2 (left). When `skeleton1` is invoked (①), the result is sent back to the client (②), then to `skeleton2` (③). Finally, the result is sent back to the client (④). For typical applications consisting of many composed skeletons, this feature of RMI results in very high time overhead.



Fig. 2. Skeleton composition using plain RMI (left) and future-based RMI (right)

To eliminate this overhead, we have developed so-called *future-based RMI*: an invocation of a skeleton on a server initiates the skeleton's execution and then returns immediately, without waiting for the skeleton's completion (see Fig. 2, right). As a result of the skeleton call, a future reference is returned to the client (②) and can be used as a parameter for invoking the next skeleton (③). When the future reference is dereferenced (④), the dereferencing thread on the server is blocked until the result is available, i.e. the first skeleton actually completes. The result is then sent directly to the server dereferencing the future reference (⑤). After completion of `skeleton2`, the result is sent to the client (⑥).

Compared with plain RMI, our future-based mechanism substantially reduces the amount of data sent over the network, because only a reference to the data is sent to the client; the result itself is communicated directly between the servers. Moreover, communications and computations overlap, effectively hiding latencies of remote calls. We have implemented future-based RMI on top of SUN Java RMI and report experimental results in Sect. 3.3. (see [6] for further details).

Future references are available to the user through a special Java interface `RemoteReference`. There are only few differences when using future-based RMI compared with the use of plain RMI: (1) instead of `Objects`, all skeletons return values of type `RemoteReference`, and (2) skeletons' interfaces are extended by `invoke` methods, accepting `RemoteReferences` as parameters.

3 Case Study: Programming FFT Using Skeletons

By way of an example application, we consider the Fast Fourier Transformation (FFT). The FFT of a list $x = [x_0, \dots, x_{n-1}]$ of length $n = 2^l$ yields a list whose i -th element is defined as $(\text{FFT } x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$, where ω_n denotes the n -th complex root of unity, i.e. $\omega_n = e^{2\pi\sqrt{-1}/n}$.

3.1 Expressing FFT with Skeletons

We now outline how the FFT can be expressed as a composition of skeletons (see [4] for details). The FFT can be written in divide-and-conquer form as follows, where $u = [x_0, x_2, \dots, x_{n-2}]$ and $v = [x_1, x_3, \dots, x_{n-1}]$:

$$(\text{FFT } x)_i = \begin{cases} (\text{FFT } u)_i \hat{\oplus}_{i,n} (\text{FFT } v)_i & \text{if } i < n/2 \\ (\text{FFT } u)_{i-n/2} \hat{\otimes}_{i-n/2,n} (\text{FFT } v)_{i-n/2} & \text{else} \end{cases} \quad (2)$$

where $a \hat{\oplus}_{j,m} b = a + \omega_m^j b$, and $a \hat{\otimes}_{j,m} b = a - \omega_m^j b$.

The formulation (2) is close to the dh skeleton format from Sect. 2.1, except for $\hat{\oplus}$ and $\hat{\otimes}$ being parameterized with the position i of the list element and the length n of the input list. Therefore we express the FFT as instance of the dh skeleton, applied to a list of triples (x_i, i, n) , with operator $\oplus$ defined on triples as $(x_1, i_1, n_1) \oplus (x_2, i_2, n_2) = (x_1 \hat{\oplus}_{i_1, n_1} x_2, i_1, 2n_1)$. Operator $\otimes$ is defined similarly.

Computing FFT using skeletons: As skeletons are higher-order functions, we first provide a functional program for FFT, which is then transformed to Java in a straightforward manner. The FFT function on an input list x can be expressed using skeletons by transforming the input list into a list of triples, applying the dh skeleton and finally taking the first elements of the triples for the result list:

$$\text{FFT} = \text{map}(\pi_1) \circ dh(\oplus, \otimes) \circ \text{apply}(\text{triple})$$

where *triple* is a user-defined function that transforms a list $[x_1, \dots, x_n]$ to the list of triples $[(x_1, 1, 1), \dots, (x_i, i, 1), \dots, (x_n, n, 1)]$, and $\circ$ denotes function composition from right to left, i.e. $(f \circ g)(x) = f(g(x))$.

Both operators $\hat{\oplus}$ and $\hat{\otimes}$ in (2) repeatedly compute the roots of unity ω_n^i . Instead of computing these for every call, they can be computed once a priori and stored in a list $\Omega = [\omega_n^1, \dots, \omega_n^{n/2}]$, accessible by both operators, thus reducing computations. Using the relation $\omega_m = \omega_n^{n/m}$, the computation of ω_m^i in $\hat{\oplus}/\hat{\otimes}$ can be replaced with $\pi(ni/m, \Omega)$, where $\pi(k, \Omega)$ selects the k -th entry of Ω . Therefore, $\hat{\oplus}$ can be expressed as $a \hat{\oplus}_{j,m,\Omega} b = a + \pi(nj/m, \Omega)b$. Operator $\hat{\otimes}$ can be expressed using Ω analogously. Thus, $\oplus/\otimes$ are parameterized with Ω ; we express this by writing $\oplus(\Omega)/\otimes(\Omega)$ in the following.

Now, we can express the computation of Ω using the *repl* and the *scan* skeletons, and arrive at the following skeleton-based program for the FFT:

$$\begin{aligned} \Omega &= \text{scan}(\ast) \circ \text{repl}(n/2, \omega_n) \\ \text{FFT} &= \text{map}(\pi_1) \circ dh(\oplus(\Omega), \otimes(\Omega)) \circ \text{apply}(\text{triple}) \end{aligned} \quad (3)$$

where $\ast$ denotes complex number multiplication.

3.2 Skeleton-Based Java Program for FFT

The Java code for the FFT, obtained straightforwardly from (3), is as follows:

```
//repl,scan,map,dh are remote refs to skeletons on servers
//compute roots of unity
RemoteReference r = repl.invoke(length, omega_n);
RemoteReference omegas = scan.invoke(r, new ScanOp());
//instantiate operators for dh
oplus = new FFT0plus(omegas);
otimes = new FFT0times(omegas);
//fft
r = apply.invoke(inputList, new TripleOp());
r = dh.invoke(oplus, otimes, r);
r = map.invoke(r, new projTriple());
//get result
result = r.getValue();
```

At first, the roots of unity are computed, using the *repl* and *scan* skeletons. Both *repl* and *scan* are RMI references to the skeletons' implementation on a remote server, obtained from the lookup service. Execution of the skeletons is started using the *invoke* methods. Variable *OmegaN* passed to the *repl* skeleton corresponds to ω_n and *omegas* corresponds to Ω . As a binary operator for *scan*, complex multiplication is used, implemented in class *ComplexMult*. The operators $\oplus(\Omega)$ and $\otimes(\Omega)$ for the *dh*-skeleton are instantiated as objects of classes *FFT0plus* and *FFT0times* on the client side. The constructor for the parameters receives the list Ω as an argument. Each operator stores a reference to the list in a private variable in order to access it later for computations.

Next, the FFT itself is computed in three steps. First the input list is transformed to a list of triples, using the *apply* skeleton with a user-defined function. Then the *dh*-skeleton is called on the list of triples, using the two customizing operators defined earlier. Finally, the list of result values is retrieved from the list of triples using the *map* skeleton with an instance of the user-defined class *projTriple*.

In a preliminary step (omitted in the code presented above), the program obtains from the lookup service a remote reference for each skeleton used in the program (*repl*, *scan*, *map* and *dh*). The program is executed on the client side; all calls to the *invoke* method of the involved skeletons are executed remotely on servers via RMI.

3.3 Experimental Results

We measured the performance of the skeleton-based FFT program using the testbed of Fig. 1. We used a SunFire 6800 with 12 US-III+ 900 MHz processors in Berlin as our server and an UltraSPARC-IIi 360 MHz as client in Erlangen, both using SUN's JDK1.4.1 (HotSpot Client VM in mixed mode). Because there were several other applications running on the server machine during our experiments, a maximum of 8 processors was available for measurements.

Fig. 3 shows the runtimes for different problem sizes (ranging from 2^{15} to 2^{18}) and four different versions of the program: the first running locally on the client (“local FFT”), second using plain RMI, third version using future-based RMI, and the fourth version where the FFT is executed as a single server sided method called from the client (“ideal remote”). We consider the fourth version as ideal, as there is no overhead for remote composition of skeletons for that version: it corresponds to copying the whole program to the server and executing it there. For the plain RMI version, only the *scan* and *dh* skeletons are executed on the server, because all parameters and results are transmitted between client and server for each method call using plain RMI, so that executing the *repl*, *apply* and *map* skeleton remotely would slow down the program unnecessarily. For the future-based RMI version, all skeletons are executed on the server.

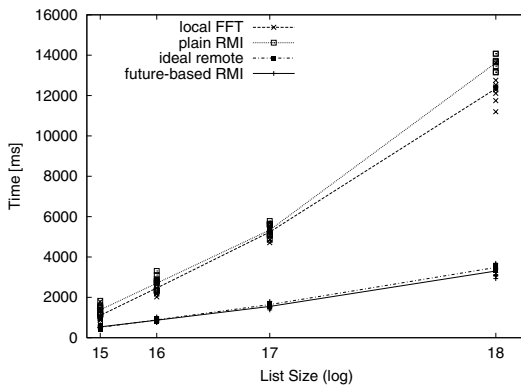


Fig. 3. Measured runtimes for the FFT programs

The figure shows ten measurements for each program version, with the average runtimes for each parameter size connected by lines. The plain RMI version is much (three to four times) slower than the future-based RMI version and unable to outperform the local, client sided FFT. Thus, the communication overhead outweighs the performance gain for execution on the server.

By contrast, the future-based RMI version eliminates most of the overhead and is three to four times faster than the local version. Compared with the “ideal remote” case the runtimes are almost identical. For large input lists (2^{17} and 2^{18}), the future-based version is even slightly faster than the remote version. This is due to the fact, that the future-based version invokes skeletons asynchronously, so the *apply* skeleton is already called while the *scan* skeleton is still running. Thus, using future-based RMI allows an efficient execution of programs with compositions of remote methods, in particular compositions of skeletons.

4 Conclusions and Related Work

In this paper, we have addressed the challenging problem of software design for heterogeneous Grids, using a repository of reusable algorithmic patterns called

skeletons, that are executed remotely on high-performance Grid servers. While the use of skeletons in the parallel setting is an active research area, their application for the Grid is a new, intriguing problem.

We have described our prototypical Grid system. Java and RMI were chosen to implement our system in order to obtain a highly portable solution. Other promising opportunities include, e.g. the Lithium system [7] for executing mainly task-parallel skeletons in Java. We have proposed a novel, future-based RMI mechanism, which substantially reduces communication overhead for compositions of skeleton calls. It differs from comparable approaches because it combines hiding network latencies using asynchronous methods (as in [8,9]) and reducing network dataflow by allowing server/server communication (e.g. found in [10]).

We have proposed an exemplary (and by no means exhaustive) repository of skeletons, which includes several elementary data-parallel functions, the divide-and-conquer skeleton DH, and two auxiliary skeletons which are helpful in a Grid environment. We have demonstrated how a mathematical description of FFT (Fast Fourier Transform) can be expressed using our skeletons, leading to an efficient Java program with remote calls for skeletons.

At present, each skeleton call is executed on a single Grid node. We plan to allow distribution of skeletons across several nodes in the future, at least for task-parallel and simple data-parallel skeletons.

References

1. Foster, I., Kesselmann, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (1998)
2. Kennedy, K., et al.: Toward a framework for preparing and executing adaptive grid programs. In: Proc. of NSF Next Generation Systems Program Workshop (2002)
3. Alt, M., et al.: Algorithm design and performance prediction in a Java-based Grid system with skeletons. In: Proc. of Euro-Par 2002. LNCS Vol. 2400, Springer (2002)
4. Gorlatch, S., Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters* **8** (1998)
5. Getov, V., et al.: Multiparadigm communications in Java for Grid computing. *Communications of the ACM* **44** (2001) 118–125
6. Alt, M., Gorlatch, S.: Optimizing the use of Java RMI for Grid application programming. Technical Report 2003/08, TU Berlin (2003) ISSN 1436-9915.
7. Danelutto, M., Teti, P.: Lithium: A structured parallel programming environment in Java. In: Proc. of Computational Science. ICCS. LNCS Vol. 2330, Springer (2002)
8. Raje, R., Williams, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism in Java. *Concurrency: Practice and Experience* **9** (1997)
9. Falkner, K.K., Coddington, P., Oudshoorn, M.: Implementing asynchronous remote method invocation in Java. In: Proc. of Parallel and Real Time Systems (1999)
10. Yeung, K.C., Kelly, P.H.J.: Optimising Java RMI programs by communication restructuring. In: *Middleware 2003*, Springer (2003)

Prototyping Application Models in Concurrent ML

David Johnston, Martin Fleury, and Andy Downton

Dept. Electronic Systems Engineering, University of Essex, Wivenhoe Park,
Colchester, Essex, CO4 3SQ, UK

{djjohn, fleum, acd}@essex.ac.uk

http://www.essex.ac.uk/ese/research/mma_lab/index.htm

Abstract. We report on the design of parallel applications using harnesses, which are abstractions of concurrent paradigms. A split/combine harness is prototyped using the functional language Concurrent ML (CML). The event model simplifies harness development by unifying the structure of serial and parallel algorithms. The portability of the harness between different hardware (both serial and parallel) is planned through the compact and elegant CML programming model.

1 Introduction

Concurrent Meta-Language (CML) is a version of the well-established functional language ML with extensions for concurrency by Reppy [16]. Concurrent processes cooperate through CSP [10] channels *i.e.* unbuffered communication. The instantiation of CSP formalism within CML is more powerful than that of the classical version of the parallel language occam2 [6], specifically:

1. dynamic recursion, process and channel creation
(*c.f.* no recursion, static processes and static channels in occam2)
2. selection between channels based on input and output events
(*c.f.* just input selection in occam2 for implementation reasons)
3. functions can be sent over channels
(side effect of ML treating code and data the same)
4. events and event functions
(*c.f.* only synchronised communication in occam2)

Modern occam [1] has absorbed some of these capabilities *e.g.* recursion and the ability to send processes over channels. It is worth also comparing CML with the occam2-like Handel-C [17], a silicon compiler language targeted at FPGAs, where 2 and 3 are also present. Modern occam and CML are general purpose in nature rather than static in the tradition of embedded languages. Parallel designs in both languages could map onto the fine-grained parallelism of Handel-C or onto systems composed of distributed processor nodes.

This paper demonstrates, through example, how the features of CML allow the rapid prototyping of software infrastructures (or harnesses) that support

generic application-level parallelism. Such harnesses have been available within `occam2` for some time, but following through the consequences of the CML event model and CML's functional nature results in a different and somewhat surprising outcome. Section 2 presents the context of the work within a larger development method. CML is introduced in Section 3 through some small illustrative examples. The main technical content within Section 4 builds upon these earlier examples to show how support for parallel execution is prototyped within CML. Section 5 presents related work, and the paper ends with conclusions and a description of the current status of the work.

2 Development Method

The key idea behind our project¹ is that the same paradigms of parallelism occur in many different applications. The execution of a single application is characterised by a dynamically changing set of such paradigms. Accordingly, a number of customised harnesses for each paradigm detected dynamically within the executing application can be transparently launched.

To illustrate, consider the simple split/combine application model for parallelism where the input can be split recursively; the parts processed in parallel; and the outputs combined. Figure 1 shows how this and other generic structures can be implemented as a harness containing the parallelism (H_{CML} or H_{C++}) in conjunction with user's application-specific code (A''_{ML} or A'_{C++} respectively) which does not. The Figure presents a variety of software engineering routes because there is a choice of working in CML or a more conventional language such as C++. The use of CML is not an end in itself (though there is useful software output as a side effect) but a means to rapidly prototype a C++ version of the system, which a C++ developer can use off-the-shelf. The splitting of an application into a harness and serial application code is shown both for CML and C++. However, the solid arrows represent which activities are intended exclusively for systems developers *i.e.* they are capturing the real parallelisations and then abstracting them within general harnesses so users need not be concerned with the parallelisation of specific applications. If the user is prototyping in ML, then the translation to C++ is possible either as sequential code or after fitting into a CML harness.

3 CML Examples

3.1 Example 1: Communication-Based Remote Procedure Call

To illustrate the use of CML, the following code performs a remote procedure call (RPC). The server applies a function (**f**) to the input argument (**arg**) and sends the result down a channel (**ch**) to the RPC client. For its part, the client

¹ Rapid Parallel Prototyping in the Image/Multimedia Domain, EPSRC Contract: GR/N20980

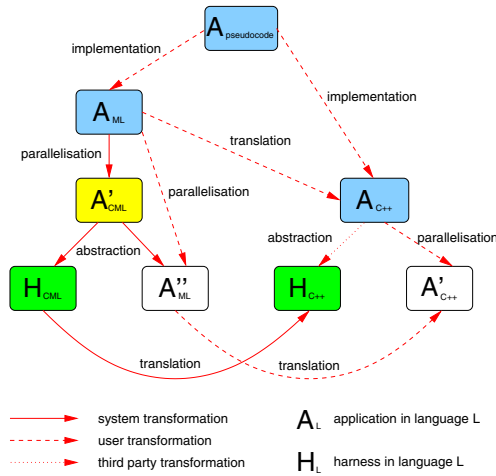


Fig. 1. Overall development methodology

creates the channel; spawns the server process with its arguments and receives a result from the channel. The client is placed within a `main` process so it can be run in the CML environment using `RunCML.doit`.

```

fun rpc_server ( ch, f, arg ) = send( ch, f(arg) );

fun rpc_client f arg =
let
  val ch = channel();
  val tid = spawnc rpc_server ( ch, f, arg );
in
  recv( ch )
end;

fun main () = print ( Real.toString( rpc_client Math.sqrt 2.0 ) ^ "\n" );

RunCML.doit ( main, NONE );

```

3.2 Example 2: Event-Based Remote Procedure Call

The second example performs the same actions as the first, only it is written in terms of the primitive event functions of CML: `sendEvt` and `recvEvt` instead of the derived `send` and `recv`. The communication only happens after the event returned by `sendEvt` and the corresponding event returned by `recvEvt` are both explicitly synchronised using the `sync` operator. In the server, the send event is directly synchronised; whereas the client actually returns a receive event which is synchronised at a higher level in the code. These useful client and server routines will be used later in Sections 4.3, 4.4 and 4.5.

```

fun rpc_server( ch, f, arg ) = sync( sendEvt( ch, f(arg) ) );

fun rpc_client f arg =
let
    val ch = channel();
    val tid = spawn rpc_server ( ch, f, arg );
in
    recvEvt( ch )
end;

fun main () = print ( Real.toString( sync ( rpc_client Math.sqrt 2.0 ) ) ^ "\n" );

RunCML.doit ( main, NONE );

```

3.3 Example 3

A **guard** event operator implements a delay in terms of the absolute temporal event primitive **atTimeEvt**. The **guard** operator performs pre-synchronisation actions or may be informally considered to “push” an action back [12]. The absolute time for the output event is only evaluated when the **sync** is called and not when **timeout** is called. The result is that synchronising on event **e** always causes a delay of exactly one second.

```

fun timeout t = guard ( fn () => atTimeEvt (Time.+ (t, Time.now())) );

fun main () =
let
    val e = timeout (Time.fromSeconds 1);
in
    sync e
end;

RunCML.doit ( main, NONE );

```

4 Split/Combine Application Harness

CML allows the rapid prototyping of a parallel harness which implements the split/combine model described in Section 2. This is shown below in five stages starting with a serial harness.

4.1 Serial Hierarchical Harness – SH_harness

The harness calls itself twice recursively if the problem is still to be split. Otherwise the input data is processed normally. Recursion is limited by simple recursion depth, though in a practical implementation this would be dynamic recursion through monitoring of performance as a function of granularity. Note the combine function is an output of the split function, so it can be a customised inverse operation.


```

(*)
* argument      description
* -----
* depth        current recursion depth
* max_depth    maximum recursion depth allowed
* f            function that does processing of input to output
* split        function that splits input AND provides matching
*             function which combines the corresponding outputs
* input        input data to application
*)

fun SH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
  let
    val ( combine, input1, input2 ) = split input;
    val daughter = SH_harness ( depth+1, max_depth ) ( f, split );
    val output1 = daughter input1;
    val output2 = daughter input2;
  in
    combine output1 output2
  end
else
  f input;

```

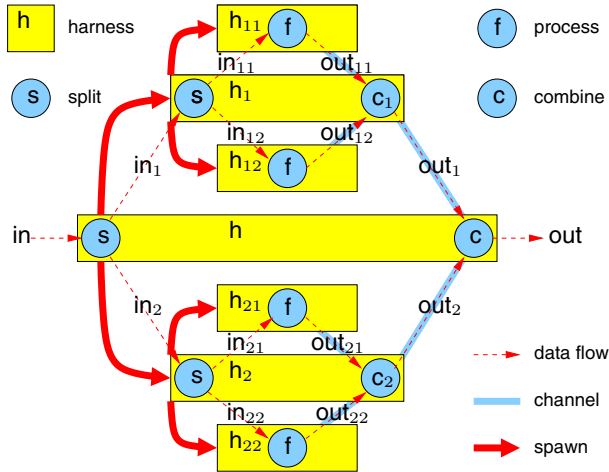


Fig. 2. Communicating parallel hierarchical split/combine harness

4.2 Communicating Parallel Hierarchical Harness – CPH_harness

An extra channel argument is supplied for the harness to send its result rather than returning it. Parallel execution is obtained by spawning two further versions of the harness instead of calling them sequentially in turn. To achieve parallelism

the results must be received *after* both processes have been spawned. Note that this code will have to naïvely wait for the first harness spawned to finish even though the second could have finished earlier. Corresponding sends and receives occur confusingly between different recursion levels of the harness. Figure 2 shows the process diagram. The subscript sequences indicate where each item is in the binary tree hierarchy. Each harness is spawned with its input parameters, while the output results are communicated by channel.

```

fun CPH_harness ( depth, max_depth ) ( f, split ) ch input =
if ( depth < max_depth )
then
  let
    val ch1 = channel();
    val ch2 = channel();

    val ( combine, input1, input2 ) = split input;
    val daughter = CPH_harness ( depth+1, max_depth ) ( f, split );

    val _ = spawnc (daughter ch1) input1;
    val _ = spawnc (daughter ch2) input2;

    val output1 = recv(ch1);
    val output2 = recv(ch2);

  in
    send( ch, combine output1 output2 )
  end
else
  send( ch, f input );

```

4.3 Event-Explicit Parallel Hierarchical Harness – EEPH_harness

Explicit communication is now done within the RPC functions, so the harness is abstracted from channel use. `recvEvt` and `sendEvt` appear together in the RPC calls using the same channel, so less channel management is required. If the `recvEvt` in the client were a `recv` the harness would spawn the daughter harnesses sequentially - not what is required!

```

fun EEPH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
  let
    val ( combine, input1, input2 ) = split input;
    val daughter = EEPH_harness ( depth+1, max_depth ) ( f, split );

    val event1 = rpc_client daughter input1;
    val event2 = rpc_client daughter input2;

    val output1 = sync(event1);
    val output2 = sync(event2);

  in
    combine output1 output2
  end
else
  f input;

```

4.4 Event-Implicit Parallel Hierarchical Harness – EIPH_harness

If the harness purely deals in events, synchronisation can be done at the last possible moment just before the data is about to be sent on to provide maximum decoupling. The `event_and` binary event operator “fires” only when both individual events would fire. The `alwaysEvt` operator is necessary to turn a value into an (always enabled) event of that value.

```
fun event_and combine event1 event2 =
let
  val event1' = rpc_client sync event1;
  val event2' = rpc_client sync event2;
in
  guard ( fn () => alwaysEvt ( combine ( sync event1' ) ( sync
                                                                    event2' ) ) )
end;

fun EIPH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
  let
    val ( combine, input1, input2 ) = split input;
    val daughter = EIPH_harness ( depth+1, max_depth ) ( f, split );
    val event1 = rpc_client daughter input1;
    val event2 = rpc_client daughter input2;
  in
    event_and combine event1 event2
  end
else
  alwaysEvt( f input );
```

4.5 Unified Harness – U_harness

The previous parallel harness is identical to the serial harness apart from the use of three higher order functions in three places to abstract the parallelism. Both the serial and parallel harnesses can thus be instantiated as parameterised versions of the same generic harness as below. It is difficult to visualise an event-based solution compared to a message-passing one, but the communication patterns are still those shown in Figure 2. However, the timing and software engineering characteristics are considerably better.

```
fun U_harness ( exec, merge_using, return ) ( depth, max_depth ) ( f, split )
                                                                    input =
if ( depth < max_depth )
then
  let
    val ( combine, input1, input2 ) = split input;
    val daughter = U_harness ( exec, merge_using, return )
                          ( depth+1, max_depth ) ( f, split );
    val output1 = exec daughter input1;
    val output2 = exec daughter input2;
  in
    merge_using combine output1 output2
  end
else
  return ( f input );
```

```

val id = fn a => a; (* identity operator *)

val SH_harness = U_harness ( id , id , id );
val EIPH_harness = U_harness ( rpc_client, event_and, alwaysEvt );

```

5 Related Work

The concept of parallel processing paradigms is almost as old as parallel processing itself. For example, in [11] the ‘divide-and-conquer’ paradigm is characterized as a suitable for architectures in which dynamic process creation is possible. The concept of a harness which encapsulates or abstracts the underlying parallel paradigm or communication pattern seems to have emerged from occam and the transputer, for example in the TINY communications harness [2, 9]. It also seems to have preceded the pattern movement in conventional software [7]. The parallel paradigm is expressed in its clearest form, sometimes called an ‘algorithmic skeleton’ [3], in functional languages. Arising from the algorithmic skeleton approach has come the claim that parallel processing can be executed by functional composition of a restricted number of paradigms [5], such as the pipeline and the task queue, which claim we remain neutral on.

Unfortunately, attempts to implement a parallel architecture suitable for processing functional languages seem to have floundered [15]. Therefore it has been suggested [4], that functional languages are best used as a way of expressing parallel decomposition in a non-parallel language. In [14], this approach was indeed taken, but the prototype was in the sequential (*i.e* Standard) version of ML, before transferring to occam. In this paper, the approach has been taken one stage further, in that a concurrent functional language, CML, has been used to express parallelism. Hammond & Michaelson [8] give an overview of recent work in the parallel functional field.

6 Conclusion and Future Work

CML has proved an effective tool for the development of parallel harnesses: the split/combine harness shown is only one of a number of harnesses that have been developed in this way. The CML model of parallelism is small, clean and elegant providing an ideal porting layer that should enable the harnesses to be moved to a variety of parallel architectures in future. Sadly, execution within the current CML environment is strictly serial. Modern occam in the form of kroc [1] was considered as an implementation layer, and this would have had the advantage of execution in a multi-process environment though prototyping would have been less rapid.

In the absence of a genuinely distributed implementation of CML, current research is looking at mixed language solutions (C++ & CML) to establish whether the harnesses can remain in CML or need to be ultimately implemented directly in C++. Some of the harnesses have been implemented in C++ for a distributed processor architecture and have been evaluated for performance on

pre-existing image processing applications [13] but this is outside the scope of this paper.

The developmental method shifts the ball-park of parallelisation significantly. Instead of writing parallel mechanism code to some external system specification, the user writes application-oriented sequential code, leaving the parallelisation to be described by a separate harness. By freeing the user from the concerns and difficulties of parallelisation, time is available to evolve existing software to match harness templates. The split/combine harness presented can clearly dynamically adapt to the appropriate granularity and scalability for the parallel hardware on which it finds itself running. All the user has to supply are suitable application-specific split and combine routines for the input and output datatypes respectively of a particular function, for that function to be transparently executed in parallel.

It was a surprise that the parallel split/combine harness developed under CML emerged homologous to the serial harness, and it is the expressive power of CML that permits such strong unifications. With the appropriate high level constructs it may be that supporting parallel execution is no more complex than writing serial code to the same pattern.

References

1. F. Barnes and P. Welch. Prioritised dynamic communicating processes – part 1. In *Communicating Process Architectures – 2002*, pages 331–362, 2002.
2. L. Clarke. TINY: Discussion and user guide, 1989. Newsletter 7, Edinburgh Concurrent Supercomputer Project.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT, Cambridge, MA, 1989.
4. M. Cole. Writing parallel programs in non-parallel languages. In R. Perrott, editor, *Software for Parallel Computers: Exploiting Parallelism through Software Environments, Tools, Algorithms and Application Libraries*, pages 315–325. Chapman & Hall, London, 1992.
5. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, and R. L. While. Parallel processing using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe*, 1993. LNCS 694.
6. Hoare C.A.R. Ed. *Occam 2 Reference Manual*. Prentice-Hall, 1988.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP’93*, pages 406–431, 1993.
8. K. Hammond and G. Michelson. *Research Directions in Parallel Functional Programming*. Springer Verlag, first edition, 1999. ISBN: 1852330929.
9. A. J. G. Hey. Experiments in MIMD parallelism. In *PARLE89, Parallel Architectures and Languages Europe*, pages 28–42, 1989. LNCS 366.
10. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, first edition, 1985. ASIN: 0131532715.
11. E. Horowitz and A. Zorat. Divide and conquer for parallel computing. *IEEE Transactions on Computers*, 32(6):582–585, 1983.
12. D Johnston. privatewww.essex.ac.uk/~djjohn/rappid A Concurrent ML Tutorial, 2002. Essex University.

13. D. J. Johnston, M. Fleury, and A. C. Downton. A functional methodology for parallel image processing development. In *Visual Information Engineering 2003*, 2003.
14. G. Michaelson and N. Scaife. Prototyping a parallel vision system in Standard ml. *Functional Programming, Concurrency, Simulation, and Automated Reasoning*, 5(3):345–382, 1995.
15. S. L. Peyton Jones and Lester D. *Implementing Functional Languages*. Prentice Hall, New York, 1992.
16. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, UK, 1999.
17. M. Spivey, I. Page, and W. Luk. How to program in Handel, 1995. Oxford University Hardware Compilation Unit.

THROOM – Supporting POSIX Multithreaded Binaries on a Cluster

Henrik Löf, Zoran Radović, and Erik Hagersten

Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
`henrik.lof@it.uu.se`

Abstract. Today, most software distributed shared memory systems (SW-DSMs) lack industry standard programming interfaces which limit their applicability to a small set of shared-memory applications. In order to gain general acceptance, SW-DSMs should support the same look-and-feel of shared memory as hardware DSMs. This paper presents a runtime system concept that enables unmodified POSIX (Pthreads) binaries to run transparently on clustered hardware. The key idea is to extend the single process model of multi-threading to a multi-process model where threads are distributed to processes executing in remote nodes. The distributed threads execute in a global shared address space made coherent by a fine-grain SW-DSM layer. We also present THROOM, a proof-of-concept implementation that runs unmodified Pthread binaries on a virtual cluster modeled as standard UNIX processes. THROOM runs on top of the DSZOOM fine-grain SW-DSM system with limited OS support.

1 Introduction

Clusters built from high-volume compute nodes, such as workstations, PCs, and small symmetric multiprocessors (SMPs), provide powerful platforms for executing large-scale parallel applications. Software distributed shared memory (SW-DSM) systems can create the illusion of a single shared memory across the entire cluster using a software run-time layer, attached between the application and the hardware. In spite of several successful implementation efforts [1], [2], [3], [4], [5], SW-DSM systems are still not widely used today. In most cases, this is due to the relatively poor and unpredictable performance demonstrated by the SW-DSM implementations. However, some recent SW-DSM systems have shown that this performance gap can be narrowed by removing the asynchronous protocol overhead [5], [6], and demonstrate a performance overhead of only 30-40 percent in comparison to hardware DSMs (HW-DSM) [5]. One obstacle for SW-DSMs is the fact that they often require special constructs and/or impose special programming restrictions in order to operate properly. Some SW-DSM systems further alienate themselves from HW-DSMs by relying heavily on very weak memory models in order to hide some of the false sharing created by their page-based coherence strategies. This often leads to large performance variations when comparing the performance of the same applications run on HW-DSMs.

We believe that, SW-DSMs should support the same look-and-feel of shared memory as the HW-DSMs. This includes support for POSIX [7] threads running on some standard memory model and a performance footprint similar to that of HW-DSMs, i.e., the performance gap should remain approximately the same for most applications. Our goal is that binaries that run on HW-DSMs could be run on SW-DSMs, without modifications.

In contrast to a HW-DSM system, where the whole address space of all processes are kept coherent by hardware, most SW-DSMs only keep coherence for specified segments in the user-level part of the virtual address space. This segment, which we call `G_MEM`, is mapped shared across the DSM nodes using the interconnect hardware. Furthermore, the text (program code), data and stack segments of the UNIX process abstraction are private to the parent process and its children on each node of the cluster. This creates a SW-DSM programming model where special constructs are needed to separate shared data, which must be allocated in `G_MEM`, from private data, which is allocated in the data and stack segments of the UNIX process at program loading. This is often done by creating a separate heap space in `G_MEM` with an associated primitive for doing allocation. In a standard multi-threaded world, there exist only one process and one address space which is shared among all threads. There is no distinction between shared and private data. Consider the following example: An application allocates a shared global array for its threads to operate on. This is often done by a single thread in an initialization phase. In a typical SW-DSM system such as TreadMarks [3], a special `malloc()`-type call has to be implemented to allocate the memory for the shared array inside the `G_MEM`. Also, the pointer variable holding the address, which is allocated in the static data segment of the process, has to be propagated to all remote nodes. This is often done by introducing a special propagation primitive.

In this paper, we present THROOM which is a runtime system concept that creates the illusion of a single process shared memory abstraction on a cluster. In essence, we want to make the static data and heap segments globally accessible by threads executing in remote nodes without introducing special DSM constructs in the application code. In the light of the example above, the application should use a standard `malloc()` call and the pointer variable should be replicated automatically. The rest of this paper is organized as follows: First, the THROOM concept is presented. Second, we give a brief presentation of the SW-DSM used. We also specify the requirements of THROOM on the SW-DSM. Third, we present a proof-of-concept implementation of THROOM on a single system image cluster and finally we discuss the performance of this implementation as well as the steps needed to take THROOM to a real cluster.

2 The THROOM Concept

In many implementations of SW-DSMs, the different nodes of the cluster all run some daemon process to maintain the `G_MEM` mappings and to deal with requests for coherency actions. In this paper we use the term *user node* to refer

to the cluster node in which the user executes the binary (the *user process*). All other nodes are called *remote nodes* and their daemon processes will be called *shadow processes*. In execution, THROOM consists of one process per node of the cluster.

The fundamental idea of THROOM is to distribute threads from the user process to shadow processes executing on remote nodes running different instances of a standard UNIX OS kernel. As discussed earlier, such systems exist, but they require non-standard programming models. To support a standard model such as POSIX, it is required that the whole address space of the user process can be accessed by all of the distributed threads. To accomplish this, we can simply place the text, data and stack segments inside a G_MEM-type segment made coherent by a SW-DSM. This will create the illusion of a large scale shared memory multiprocessor built out of standard software and hardware components.

3 DSZOOM – A Fine-Grained SW-DSM

Our prototype implementation is based on the sequentially consistent DSZOOM SW-DSM [5]. Each DSZOOM node can either be a uniprocessor, a SMP, or a CC-NUMA cluster. The node's hardware keeps coherence among its caches and its memory. The different cluster nodes run different kernel instances and do not share memory with each other in a hardware-coherent way. DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory in other nodes, similar to the remote `put/get` semantics found in the cluster version of the Scalable Coherent Interface (SCI), or the emerging InfiniBand standard that supports RDMA READ/WRITE as well as the atomic operations `CmpSwap` and `FetchAdd` [8].¹ Another example is the Sun Fire (TM) Link interconnect hardware [9].

While traditional page-based SW-DSMs rely on TLB traps to detect coherence “violations”, fine-grained SW-DSMs like Shasta [10], Blizzard-S [11], Sirocco-S [12] and DSZOOM [5] insert the coherence checks in-line. In DSZOOM, this is done by replacing each load and store that may reference shared data of the binary with a code *snippet* (short sequence of machine code). In terms of THROOM, the only requirement on the SW-DSM system is that it uses binary instrumentation. THROOM will also inherit the memory consistency model of the SW-DSM system.

4 Implementing THROOM

This section discusses how we can implement the THROOM concept using standard software components and the DSZOOM SW-DSM.

¹ Atomic operations are needed to support a *blocking directory* protocol [5].

4.1 Achieving Transparency

The most important aspect of THROOM is that it is totally transparent to the application code, no recompilation is allowed. To achieve this, we use a technique called *library interposition* or *library pre-loading* [13], which allow us to change the default behavior of a shared library call without recompiling the binary. Many operating systems implement the core system libraries such as `libc`, `libpthread` and `libm` as shared libraries. Using inter-positioning, we can catch a call, to any shared library and redirect it to our own implementations. In practice, this is done by redefining a symbol in a separate shared library to be pre-loaded at runtime. When an application calls the function represented by the symbol, the runtime linker searches its path for a match. Pre-loading simply means that we can insert an alternate implementation before the standard implementation in the search path of the linker. Pre-loading also allow us to reuse the native implementation. Original arguments can be modified in the interposer before the call to the native implementation is made.²

4.2 Distributing Threads

To distribute threads, the `pthread_create()` call is redefined in a pre-loaded library. The interposed implementation, first schedules the thread for execution in a remote shadow process. Second, the chosen shadow process is told to create a new thread, by calling the native `pthread_create()` from within the interposing library. The new distributed thread will start to execute in the shadow process, with arguments pointing to the software context of its original user process.

4.3 Creating a Global Shared Address Space

A minimal requirement for a distributed thread to execute correctly in a shadow process is that it must share the whole address space of the user process. To accomplish this, the `malloc()` call is redefined in a pre-loaded library to allocate memory from G-MEM instead of the original data segment of the user process. This will make all dynamically allocated data accessible from the shadow processes. Code and static data are made globally accessible by copying the segments containing code and static data from the user process to the G-MEM. The application code is then modified, using binary instrumentation, to access the G-MEM copy instead of the original segments. This will make the application execute entirely in the global shared memory segment. Hence, no special programming constructs are needed to propagate writes to static data. The whole process is also transparent in the sense that a user does not need access to the application source code, as binary instrumentation modifies the binary itself.

All references to the G-MEM must also be made coherent as the hardware only support remote reads and writes. This is taken care of by a fine-grain SW-DSM. If the SW-DSM use binary instrumentation to insert snippets for access

² To our knowledge, Linux, Solaris, HP-UX, IRIX and Tru64 all support library pre-loading.

control, we can simply add instructions needed for the static data access diversion to these snippets. In all cases, a maximum of four instructions were added to the existing snippets of DSZOOM. To lower the overheads associated with binary instrumentation, the present implementation does not instrument accesses to the stack. Hence stacks are considered thread private. Although this is not in full compliance with the POSIX model of multi-threading, it is sufficient to support a large set of pthread applications.

4.4 Cluster-Enabled Library Calls

Most applications use system calls and/or calls to standard shared libraries such as `libc`. If the arguments refer to static data, the accesses must be modified to use the G.MEM in order for memory operations to be coherent across the cluster. This can be done in at least two ways. We either instrument all library code or we overload the library calls to copy any changes from the user process original data segments to the G.MEM copies at each library call. Remember that un-instrumented code referencing static data of the application will operate in the original data segments of the user process. Hence, copying is needed to make any modifications visible to other nodes.

Instrumenting all library code is in principle, the best way to cluster-enable library calls. However, our instrumentation tool, EEL [14], was not able to instrument all of the libraries. Instead, we had to use the library interposition method for our prototype implementation. An obvious disadvantage of this method is that we have to redefine a large amount of library calls, especially if we want complete POSIX support. Another disadvantage is the runtime overhead associated with data copying, especially for I/O operations. A better solution would be to generate the coherence actions on the original arguments before the call is made in the application binary, see Scales et. al. [1]. This requires a very sophisticated instrumentation tool, which is outside the scope of this work.

5 Implementation Details

We have implemented the THROOM system on a 2-node Sun WildFire prototype SMP cluster [15], [16]. The cluster is running a single-system image version of Solaris 2.6 and the hardware is configured as a standard CC-NUMA architecture. Although, this system already supports a global shared address space, we can still use it to emulate a future THROOM architecture.

The runtime system is implemented as a shared library. A user simply sets the `LD_PRELOAD` environment variable to the path of the THROOM runtime library, and then executes the instrumented binary. As the system is a single system image we can use standard Inter Process Communication (IPC) primitives to emulate a real distributed cluster. The DSZOOM address space is set up during initialization using the `.init` section. This makes the whole initialization transparent. Control is then given to the application. The user process issues

a `fork(2)` call to create a shadow process, which will inherit its parents mappings by the copy-on-write semantics of Solaris. The two processes are bound to the two nodes using the WildFire first-touch memory initialization and the `pset.bind()` call. The home process then reads its own `/proc` file system to locate the `.text`, `.data`, and `.bss` segments and copies them to the `G_MEM`.

The shadow process waits on a process shared POSIX conditional variable to create remote threads for execution in the `G_MEM`. Parameters are passed through a shared memory mapping separated from the `G_MEM`. Since the remote thread is created in another process, thread IDs are no longer unique. To fix this, the remote node ID is copied into the most significant eight bits of the thread type, which in the Solaris 2.6 implementation is an unsigned integer. Similar techniques are used for other pthread calls. Also, the synchronization primitives of the application were overloaded using pre-loading to pre-prepared `PROCESS_SHARED` POSIX primitives to allow for multi-process synchronization. More details on the implementation are available in L f et al. [17].

Table 1. Problem sizes and replacement ratios for the 10 SPLASH-2 applications studied. Instrumented loads and stores are showed as a percentage of the total amount of load or store instructions. The number in parenthesis shows the replacement ratio for the DSZOOM SW-DSM without THROOM.

Program	Problem size, Iterations	Replaced Loads (%)	Replaced Stores (%)
FFT	1 048 576 points (48.1 MByte)	44.6(19.0)	32.8(16.5)
LU-C	1024x1024, block 16 (8.0 MByte)	48.3(15.5)	23.0(9.4)
LU-NC	1024x1024, block 16 (8.0 MByte)	49.2(16.7)	27.7(11.1)
RADIX	4 194 304 items (36.5 MByte)	54.4(15.6)	31.4(11.6)
Barnes	16 384 bodies (8.1 MByte)	56.6(23.8)	55.4(31.1)
Ocean-C	514x514 (57.5 MByte)	50.6(27.0)	31.2(23.9)
Ocean-NC	258x258 (22.9 MByte)	51.0(11.6)	39.0(28.0)
Radiosity	room (29.4 MByte)	41.1(26.3)	35.1(27.1)
Water-NSQ	2197 mol, 2 steps (2.0 Mbyte)	50.4(13.4)	38.0(16.2)
Water-SQ	2197 mol, 2 steps (1.5 Mbyte)	48.5(15.7)	32.5(13.9)

6 Performance Study

First a set a test pthread programs were run to verify the correctness of the implementation. To produce a set of pthread programs to be used as a comparison to DSZOOM, ten SPLASH-2 applications [18] were compiled using the GCC v2.95.2 compiler without optimization (`-O0`)³ and a standard Pthread PARMACS macro implementation (`c.m4.pthreads.condvar_barrier`) was employed. No modifications was made to the PARMACS run-time system or the applications. To exclude the initialization time for the THROOM runtime system, timings are started at the beginning of the parallel phase. All timings have been performed on the 2-node Sun WildFire [15] configured as a traditional CC-NUMA architecture. Each node has 16 UltraSPARCII processors running at 250

³ The code is compiled without optimization to eliminate any delay slots, which EEL cannot handle correctly.

MHz. The access time to node-local memory is about 330ns. Remote memory is accessed in about 1800ns. In Table 1, we see that more instructions are replaced in the case of THROOM since *all* references to static data have to be instrumented. This large difference in replacement ratio compared to DSZOOM is explained by the fact that DSZOOM can exploit the PARMACS programming model and use *program slicing* to remove accesses to static data that are not shared. Figure 1 and 2 shows execution times in seconds for 8- and 16-processor runs for the following THROOM configurations:

THROOM_RR. THROOM runtime system using library pre-loading. Round-robin scheduling of threads between the two nodes. All references to static data are instrumented.

DSZOOM. Used as reference. Aggressive slicing and snippet optimizations. Optimized for a two-node fork-exec native PARMACS environment, see [5].

CC-NUMA. Uses the same runtime system as DSZOOM but without any instrumentations. Coherence is kept by the WildFire hardware [5]

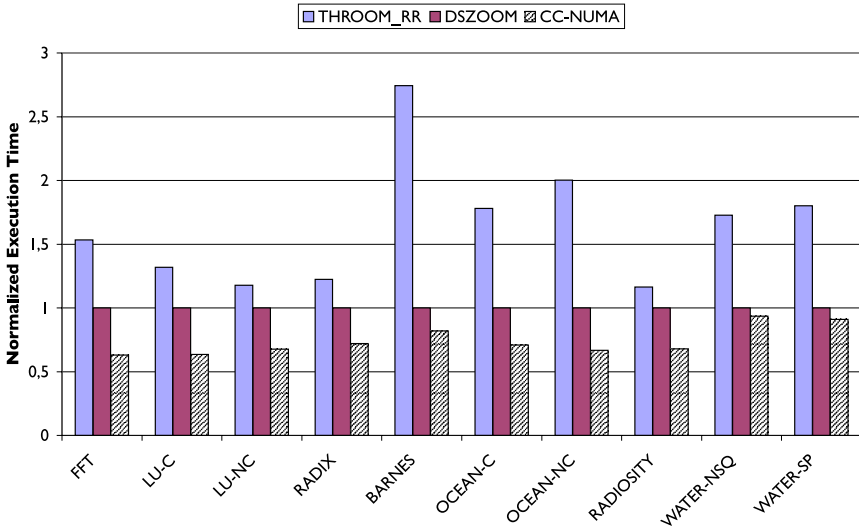


Fig. 1. Runtime performance of the THROOM runtime system. Two nodes with 4 CPUs each.

A study of Figures 1 and 2 reveals that the present implementation is slower than a state-of-the-art SW-DSM such as DSZOOM. The average runtime overhead compared to DSZOOM for THROOM_RR is 65% on 8 CPUs and 78% on 16 CPUs. In order to put these numbers into the context of total SW overhead compared to a HW-DSM, the average slowdown comparing the CC-NUMA and the DSZOOM cases is only 26%. The most significant contribution to the

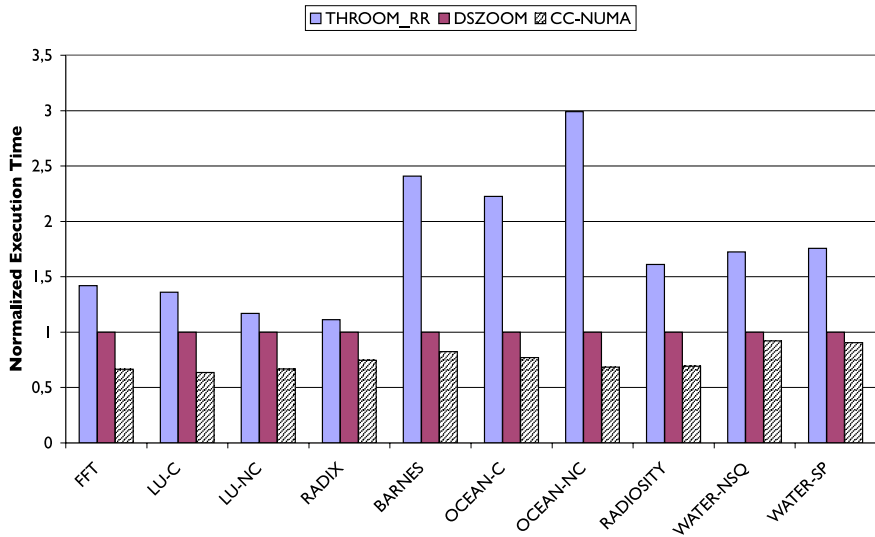


Fig. 2. Runtime performance of the THROOM runtime system. Two nodes with 8 CPUs each.

high overhead when comparing DSZOOM to THROOM is the increased number of instrumentations needed to support the POSIX thread model. Another source of overhead is the inefficient implementation of locks and barriers. This can be observed by comparing the performance of Barnes, Ocean-C, Ocean-NC and Radiosity from Figures 1 and 2. The performance of these four applications drops when increasing the number of threads as they spend a significant amount of time executing in synchronization primitives. The DSZOOM runtime system uses its own implementations of spin-locks and barriers which are more scalable.

7 Related Work

To our knowledge, no SW-DSM system has yet been built that enables transparent execution of an unmodified POSIX binary. The Shasta system [1], [2] come closest to our work and this system has showed that it is possible to run an Oracle database system on a cluster using a fine-grain SW-DSM. Shasta has solved the OS functionality issues in a similar way as is done in THROOM although they support a larger set of system calls and process distribution. THROOM differs from Shasta in that it supports sharing of static data. THROOM also supports thread distribution. Shasta motivates the lack of multi-threading support by claiming that the overhead associated with access checks lead to lower performance [1].

Another system announced recently is the CableS system [19] built on the GeNIMA page-based DSM [6]. This system support a large set of system calls,

but they have not been able to achieve binary transparency. Some source code modifications must be made and the code must be recompiled for the system to operate. Another work related to THROOM is the OpenMP interface to the TreadMarks page-based DSM [20] [3], where a compiler front-end translates the OpenMP pragmas into TreadMark fork-join style primitives. The DSM-Threads system [21] provide a page-based DSM interface similar to the Pthreads standard without binary transparency.

8 Conclusions

We have showed that it is *possible* to extend a single process address space to a multi-process model. Even though the current THROOM implementation relies on some of the WildFire's single system image properties, we are convinced that the THROOM concept can be implemented on a real cluster. In a pure distributed setting, additional issues need to be addressed. One way of initializing the system could be to use a standard MPI runtime system for process creation and handshaking. The address space mappings must also be set up using the RDMA features of the interconnect hardware. Also, synchronization needs to be handled more efficiently (see Radović et. al. [22]), and we need to create more complete and more efficient support for I/O and other library calls. For complete POSIX compliance, we also need to address the problem of threads sharing data on the stack.

References

1. Scales, D.J., Gharachorloo, K.: Towards Transparent and Efficient Software Distributed Shared Memory. In: Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France. (1997)
2. Dwarkadas, S., Gharachorloo, K., Kontothanassis, L., Scales, D.J., Scott, M.L., Stets, R.: Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In: Proceedings of the 5th International Symposium on High-Performance Computer Architecture. (1999) 260–269
3. Keleher, P., Cox, A.L., Dwarkadas, S., Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In: Proceedings of the Winter 1994 USENIX Conference. (1994) 115–131
4. Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S., Scott, M.: Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In: Proceedings of the 16th ACM Symposium on Operating System Principle. (1997)
5. Radović, Z., Hagersten, E.: Removing the Overhead from Software-Based Shared Memory. In: Proceedings of Supercomputing 2001. (2001)
6. Bilas, A., Liao, C., Singh, J.P.: Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In: Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99). (1999)

7. IEEE Std 1003.1-1996, ISO/IEC 9945-1: Portable Operating System Interface (POSIX)–Part1: System Application Programming Interface (API) [C Language]. (1996)
8. InfiniBand(SM) Trade Association: InfiniBand Architecture Specification, Release 1.0. (2000) Available from: <http://www.infinibandta.org>.
9. Sistare, S., Jackson, C.J.: Ultra-High Performance Communication with MPI and the Sun Fire(TM) Link Interconnect. In: Proceedings of the IEEE/ACM SC2002 Conference. (2002)
10. Scales, D.J., Gharachorloo, K., Thekkath, C.A.: Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII). (1996) 174–185
11. Schoinas, I., Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Larus, J.R., Wood, D.A.: Fine-grain Access Control for Distributed Shared Memory. In: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI). (1994) 297–306
12. Schoinas, I., Falsafi, B., Hill, M., Larus, J.R., Wood, D.A.: Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In: Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques. (1998)
13. Thain, D., Livny, M.: Multiple Bypass, Interposition Agents for Distributed Computing. In: Cluster Computing. (2001)
14. Larus, J.R., Schnarr, E.: EEL: Machine-Independent Executable Editing. In: Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation. (1995) 291–300
15. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantative Approach 3:rd edition. Morgan Kaufman (2003)
16. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. In: Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture. (1999) 172–181
17. Löf, H., Radović, Z., Hagersten, E.: THROOM — Running POSIX Multithreaded Binaries on a Cluster. Technical Report 2003-026, Department of Information Technology, Uppsala University (2003)
18. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95). (1995) 24–36
19. Jamieson, P., Bilas, A.: CableS: Thread Control and Memory Mangement Extensions for Shared Virtual Memory Clusters. In: 8th International Symposium on High-Performance Computer Architecture, HPCA-8. (2002)
20. A. Scherer, H. Lu, T.G., Zwaenepoel, W.: Transparent Adaptive Parallelism on NOWs using OpenMP. In: Principles Practice of Parallel Programming. (1999)
21. Mueller, F.: Distributed Shared-Memory Threads:DSM-Threads. In: Proc. of the Workshop on Run-Time Systems for Parallel Programming. (1997)
22. Radović, Z., Hagersten, E.: Efficient Synchronization for Nonuniform Communication Architectures. In: Proceedings of Supercomputing 2002. (2002)

An Inter-entry Invocation Selection Mechanism for Concurrent Programming Languages

Aaron W. Keen¹ and Ronald A. Olsson²

¹ Computer Science Department,
California Polytechnic State University,
San Luis Obispo, CA 93407 USA
`akeen@csc.calpoly.edu`

² Department of Computer Science,
University of California, Davis,
Davis, CA 95616 USA
`olsson@cs.ucdavis.edu`

Abstract. Application-level message passing is supported by many concurrent programming languages. Such languages allow messages to be generated by invoking an entry. Messages are removed from an entry by an invocation selection mechanism. Such mechanisms may allow selection from a set of entries (multi-way receive). Many concurrent languages provide support for multi-way receives, but they are limited in their expressive power. This paper presents three proposed inter-entry selection mechanisms. The proposed mechanisms overcome the limitations of the existing mechanisms. Each of these mechanisms allows an invocation selection algorithm to examine the entire set of pending invocations (and their parameters) as part of the invocation selection process. These mechanisms are analyzed and compared both qualitatively and quantitatively.

1 Introduction

Many concurrent programming languages support application-level message passing as a form of thread synchronization. A message, henceforth called an *invocation*, is generated when a process invokes an operation (i.e., an entry or port), optionally passing it parameters (such message passing includes synchronous and asynchronous invocation of an operation). Once generated, the invocation is added to the set of pending invocations associated with the invoked operation. An invocation is removed from an operation's set by a servicing process using an invocation servicing mechanism. Such a mechanism, depending on the language, may allow selection of an invocation from a single operation (a receive) or from one of potentially many operations (a multi-way receive).

This paper focuses on multi-way receives and the invocation selection support provided. Ada [12], Concurrent C [9], CSP [11,15,10], Erlang [3], Hermes [17], JR [14], Limbo [8], occam [5,16], Orca [4], SR [2,1], and SRR [7] each provide support for multi-way receives, but with differing expressive power. None of these languages provides general support for inter-entry invocation selection.

Specifically, these languages do not provide simultaneous access to all pending invocations (both within an operation's set and over the set of operations) during selection. Inter-entry selection facilitates the implementation of, for example, lottery scheduling for resource management [18] (in which a manager process picks a request randomly) and preferential job scheduling [7] (in which preference is given to normal interactive jobs unless there is a superuser batch job).

This paper presents three proposed inter-entry invocation selection mechanisms that address the limitations discussed above. The designs combine aspects of functional, object-oriented, and concurrent programming languages. The result of this study is the addition of the most balanced (in terms of abstraction and performance) mechanism to the JR concurrent programming language [14], which provides threads and message passing.

The rest of this paper is organized as follows. Section 2 discusses the limitations of multi-way receives provided by other languages. Section 3 describes the details of the proposed selection mechanisms. Section 4 analyzes the proposed mechanisms. Section 5 compares the proposed mechanisms. Section 6 concludes. Further discussion and details appear in Reference [13].

2 Background

Many concurrent languages provide support for multi-way receives, but with limited expressive power. For example, Ada provides the **select** statement, which allows multiple **accept** statements, each servicing an operation. When a **select** statement is executed, an operation is nondeterministically selected from those with pending invocations, and an invocation from the selected operation is serviced. Figure 1 demonstrates, using JR syntax (where an **inni** statement specifies a multi-way receive), a simple two-way receive that services invocations from two operations. Each **accept** statement may be preceded by a boolean guard to restrict the operations considered for servicing. A guard, however, cannot access an invocation's arguments and, thus, cannot use the values of these arguments in deciding which invocation to select. These guards provide a very coarse-grained control over selection.

Other languages have similar limitations. For example, occam supports boolean guards to restrict the candidate invocations, but does not allow selection based on an invocation's parameters or on the entire set of pending invocations. Erlang allows a boolean guard that specifies which messages are acceptable for servicing, but does not allow selection based on all pending invocations.

SR provides a more flexible invocation selection statement called the input (**in**) statement. A guard on an arm of an input statement can contain a *synchronization* expression and a *scheduling* expression. The former specifies which invocations are acceptable for servicing; the latter specifies the order in which to service acceptable invocations. Figure 2 shows a modification of the two-way receive from the earlier example, lines 7–8 in Figure 1. This modified two-way receive uses scheduling expressions (specified via **by** clauses) to service each operation's invocations in order of highest priority (lowest integer value). Unlike

```

1 public class Server {
2     public op void entryOne(int priority, int b);
3     public op void entryTwo(int priority, float b);
4     protected void server() {
5         /* repeatedly service an invocation from entryOne or entryTwo */
6         while (true) {
7             inni void entryOne(int priority, int b) { /* service */ }
8             [] void entryTwo(int priority, float b) { /* service */ }
9         }
10    }
11 }

```

Fig. 1. Simple Two-way Receive.

the guards in Ada, SR's synchronization and scheduling expressions can access an invocation's arguments and use their values to determine an invocation's acceptability and to order invocations. Such expressions, however, cannot simultaneously access the arguments of multiple invocations, and cannot be used to order invocations between multiple operations.

```

7 inni void entryOne(int priority, int b) by -priority { /* service */ }
8 [] void entryTwo(int priority, float b) by -priority { /* service */ }

```

Fig. 2. Two-way Receive with Scheduling Expression.

SRR extends SR to provide the **rd** statement, which allows the examination of all pending invocations of a set of operations. The **rd** statement, however, cannot directly service an invocation. An invocation is marked for service by a **mark** statement and is serviced by a subsequent **take** statement. Though this approach allows selection based on the pending invocations, the separation of selection into three non-atomic stages (**rd**, **mark**, and **take**) can complicate solutions. For example, one thread might **mark** an invocation and then attempt to **take** it, only to find that it was serviced by another thread in the intervening time between marking and taking.

None of the aforementioned mechanisms provides support for implementing selection algorithms that require atomic access to the entire set of pending invocations. This class of selection algorithms includes debugging, visualization, and scheduling (e.g., lottery scheduling). Extending the example in Figure 2, these mechanisms cannot directly enforce the priority ordering between operations; an invocation in **entryTwo** may be serviced even if **entryOne** has a higher priority invocation pending.

3 Proposed Invocation Selection Mechanisms

An acceptable inter-entry invocation selection mechanism must:

- allow atomic selection of any pending invocation in the set of operations serviced.
- provide access to each invocation's actual arguments.

- disallow removal of multiple invocations from the set of pending invocations.
- disallow insertion into the set of pending invocations as a side-effect.

We devised three approaches that satisfy these criteria: Invocation Enumeration, Functional Reduction, and Hybrid. Invocation Enumeration and Functional Reduction are discussed below. The Hybrid approach combines aspects of the other approaches, but mixes programmer abstractions and performs poorly, so it is not discussed further (though the experiments do include Hybrid).

3.1 Invocation Enumeration

The Invocation Enumeration approach provides an enumeration of the currently pending invocations. This enumeration is passed to a programmer-specified method that selects an individual invocation. The `inni` statement services the invocation returned from this method.

We designed two approaches to Invocation Enumeration: View Enumeration and Named Enumeration. These variants differ in how they name invocations and invocation parameters. View Enumeration names each invocation’s parameters as invocations are extracted from an enumeration. Named Enumeration names invocations as they are placed into an enumeration.

```

1  Invocation select_method(ArmEnumeration arm) { ... }
2  ...
3  inni with select_method over
4    void entryOne(int i, float f)          { ... }
5    [] void entryTwo(int i, float f, float g) { ... }
```

Fig. 3. Invocation Enumeration: Specification of selection method.

Both approaches use a `with-over` clause to specify the selection method (line 3 of Figure 3). The method must have type signature $ArmEnumeration \rightarrow Invocation$. An `ArmEnumeration` enumerates `InvocationEnumerations`, each corresponding to the operation serviced by an arm of the `inni` statement. Each `InvocationEnumeration` is an enumeration of the respective operation’s pending invocations. Figure 4 gives a pictorial example of the enumeration structure.

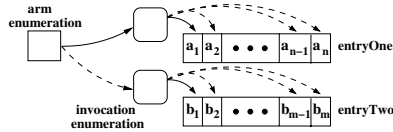


Fig. 4. Arm and Invocation Enumeration Structure.

View Enumeration. An abridged implementation of the selection method used in Figure 3 is given in Figure 5. This example demonstrates accesses to individual invocations and their arguments. Line 3 shows the extraction of

an `InvocationEnumeration` from the `ArmEnumeration`. An `Invocation` is extracted from an `InvocationEnumeration` on line 6. Under View Enumeration, the class `Invocation` is viewed as a variant type [6] of all invocation types and each invocation as a value of this variant type. A `view` statement is used to determine the underlying type of a specific invocation and to provide access to the arguments of the invocation. The `view` statement on lines 7–8 compares invocation `invoc` to the underlying invocation types $int \times float$ and $int \times float \times float$. The statement associated with the matching `as` clause is executed with the invocation’s arguments bound to the respective parameters.

```

1  Invocation select_method(ArmEnumeration arm) {
2      while (arm.hasMoreElements()) {
3          InvocationEnumeration invoc_enum = arm.nextElement();
4          if (invoc_enum == null) continue;
5          while (invoc_enum.hasMoreElements()) {
6              Invocation invoc = invoc_enum.nextElement();
7              view invoc as (int i, float f) { ... } // block using i and f
8              as (int i, float f, float g) { ... } // block using i, f, and g
9          }
10     }
11 }
```

Fig. 5. View Enumeration: Implementation of selection method.

Named Enumeration. Named Enumeration allows the programmer to “name” the invocation type used within an invocation enumeration. These types are specified via `as` clauses (Figure 6). Each “named” type (e.g., `nameOne` and `nameTwo`) is a class that extends the `Invocation` class and provides a constructor with type signature matching the operation being serviced. The “named” types are used within the selection method to access each invocation’s parameters.

```

1  inni with select_method over
2      void entryOne(int i, float f) as nameOne { ... }
3      [] void entryTwo(int i, float f, float g) as nameTwo { ... }
```

Fig. 6. Named Enumeration: Specification of selection method and invocation types.

3.2 Functional Reduction

The Functional Reduction approach splits invocation selection into two phases to provide a simple means for accessing the invocation parameters. Figure 7 gives a pictorial example of the two phases. The first phase (depicted by the horizontal arrows), which has direct access to the parameters of the invocations, selects a single invocation from each arm of the `inni` statement; this invocation is called the representative for its respective arm. The second phase (depicted by the vertical arrow), which can access invocation parameters through user-defined invocation types, selects the actual invocation to service from the representatives. Each phase selects an invocation through a reduction, using a programmer-specified method, over the appropriate set of invocations.

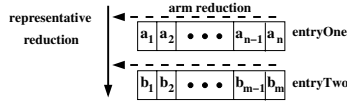


Fig. 7. Arm Reduction and Representative Reduction Structure.

4 Analysis of Mechanisms

Though the proposed mechanisms all satisfy the criteria for acceptable invocation selection mechanisms, they differ in terms of abstraction and performance. This section discusses solutions to representative examples from three problem domains. In the interest of space, and since it is ultimately adopted, only the View Enumeration solutions are shown. The problem domains are:

- Selection Independent of Arguments: Does not require access to the arguments of the invocations. The representative, RANDOM, selects a random invocation from the pending invocations.
- Single Comparison Required: A comparison between two invocations is sufficient to eliminate one from consideration. The representative, PRIORITY, selects an invocation based on a “priority” argument.
- Multiple Comparisons Required: Simultaneous examination and comparison of multiple invocations is required. The representative, MEDIAN, selects the invocation with the median first argument of all pending invocations.

```

1 Invocation select_random(ArmEnumeration arm) {
2   int num = 0;
3   while (arm.hasMoreElements()) { // tally invocations
4     InvocationEnumeration invoc_enum = arm.nextElement();
5     if (invoc_enum != null) num += invoc_enum.length();
6   }
7   int rand = RandomInvoc.rand.nextInt(num);
8   arm.reset(); // return to beginning of enumeration
9   while (arm.hasMoreElements()) { // find invocation
10    InvocationEnumeration invoc_enum = arm.nextElement();
11    if (invoc_enum != null) {
12      if (rand >= invoc_enum.length) rand -= invoc_enum.length;
13      else {
14        while (rand > 0) { // find invocation in this "arm"
15          invoc_enum.nextElement(); rand--;
16        }
17        return (Invoc) (invoc_enum.nextElement());
18      } } }
19   return null; // Shouldn't get here
20 }
```

Fig. 8. View Enumeration: RANDOM.

4.1 Selection Independent of Arguments

Invocation Enumeration. Figure 8 gives the selection method used in both Invocation Enumeration solutions to RANDOM. The `select_random` method

calculates the total number of pending invocations in the enumeration, generates a random number in the range, and, finally, returns the “random” invocation.

Functional Reduction. A Functional Reduction solution to RANDOM first selects a representative invocation for each arm by randomly determining if the “current” invocation should be the representative based on the number of invocations examined thus far (i.e., the “previous” invocation carries a weight based on the number of invocations that it has beaten out). Finally, the invocation to service is selected from the representatives in the same manner.

4.2 Single Comparison Required

Invocation Enumeration. Figure 9 gives the selection method used in a View Enumeration solution to PRIORITY. The algorithm used is very simple: loop through the invocations for each arm and record the invocation with the highest priority. A `view` statement (line 9) is used to access each invocation’s arguments.

```

1 Invocation prio_select(ArmEnumeration arm_enum) {
2   Invocation cur = null;
3   int best_prio = 0, cur_prio;
4   while (arm_enum.hasMoreElements()) {
5     InvocationEnumeration invoc_enum = arm_enum.nextElement();
6     if (invoc_enum == null) continue;
7     while (invoc_enum.hasMoreElements()) {
8       Invocation invoc = invoc_enum.nextElement();
9       view invoc as (int prio_tmp, int i) //params of invoc
10      cur_prio = prio_tmp;
11      ... // for each invocation type
12      if ((cur == null) || (cur_prio < best_prio)) {
13        cur = invoc;   best_prio = cur_prio;
14      } } }
15   return cur;
16 }
```

Fig. 9. View Enumeration: PRIORITY.

Functional Reduction. The Functional Reduction solution to PRIORITY compares the “current” invocation with the “previous” invocation and returns the one with highest priority. This general reduction is used to first select the representative invocations and then to select the invocation to service.

4.3 Multiple Comparisons Required

Invocation Enumeration. Figure 10 outlines the selection method used in a View Enumeration solution to MEDIAN. The algorithm used gathers the invocations into a `Vector`, converts the `Vector` into an array, sorts the array, and, finally, selects the median invocation.

Functional Reduction. The Functional Reduction solution to MEDIAN highlights the drawbacks of the approach. The arm reduction method gathers each invocation into a `Vector` stored within a user-defined object. It is necessary to gather the invocations in this manner because the Functional Reduction approach does not provide direct simultaneous access to all of the invocations.

```

1 Invocation median_select(ArmEnumeration arm) {
2   Vector v = new Vector();
3   while (arm.hasMoreElements()) {
4     InvocationEnumeration invoc_enum = arm.nextElement();
5     if (invoc_enum == null) continue;
6     view invoc as (int firstarg, int i) // params of invoc
7     v.add(new Element(invoc, firstarg));
8     ... // for each invocation type
9   }
10  ... // convert vector, sort, and return median invocation
11 }

```

Fig. 10. View Enumeration: MEDIAN.

With the invocations for each arm gathered into objects, a reduction over the arms selects the final invocation to service. Unfortunately, since a specific call of the reduction method cannot determine if it is the last, each call of the reduction method must select the “final” invocation to service (i.e., the median invocation of those examined thus far). As such, each call of the representative reduction method performs the costly median invocation selection.

4.4 Performance

A number of experiments were run to evaluate the performance of each of the proposed mechanisms. The experiments were conducted on a 850 MHz Intel Pentium III with 128 MB of RAM running Linux kernel 2.2.19 and IBM’s JRE 1.3.0. The experiments apply the solutions to an `inni` statement servicing 10 distinct operations with pending invocations initially evenly distributed. The results report, for each solution, the time to select and service all pending invocations. (Other experiment sizes were also run with similar results.)

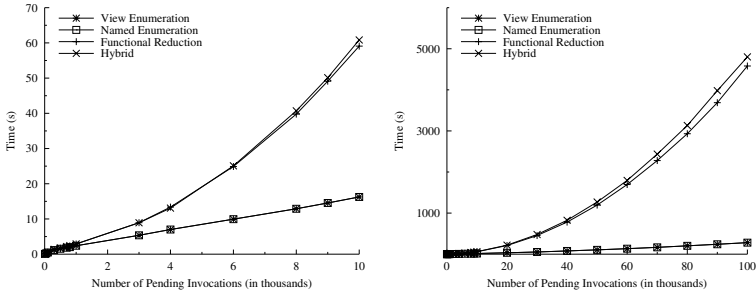


Fig. 11. RANDOM: Time to service all pending invocations.

RANDOM. Figure 11 plots the number of initially pending invocations versus time for each solution to RANDOM. In this experiment, Functional Reduction does not compare favorably to Invocation Enumeration. It must be noted, however, that the Invocation Enumeration solution to RANDOM implements a more efficient algorithm than does Functional Reduction. The Invocation Enumeration

solutions use a divide-and-conquer approach to examine a fraction of the pending invocations. An operation’s entire set of pending invocations is skipped if the random invocation is not a member of the set (see lines 12-16 in Figure 8). The same divide-and-conquer algorithm cannot be implemented using the Functional Reduction approach because the reduction examines every invocation.

PRIORITY. Figure 12 plots the number of initially pending invocations versus time for each solution to PRIORITY. This problem requires access to invocation arguments, which is the reason that the Named Enumeration solution performs poorly. To support the abstraction provided by Named Enumeration, an object (of the programmer “named” type) must be created for each pending invocation. This accounts for the performance difference between View and Named Enumeration.

MEDIAN. Figure 13 plots the number of initially pending invocations versus time for each solution to MEDIAN. Functional Reduction performs poorly because the expensive selection computation (selecting the median element) is executed for each call of the arm reduction method (equal to the number of arms).

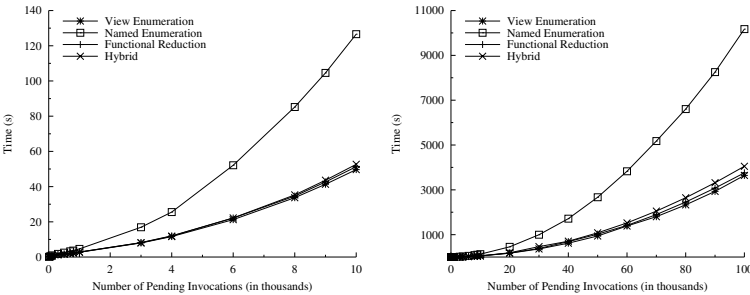


Fig. 12. PRIORITY: Time to service all pending invocations.

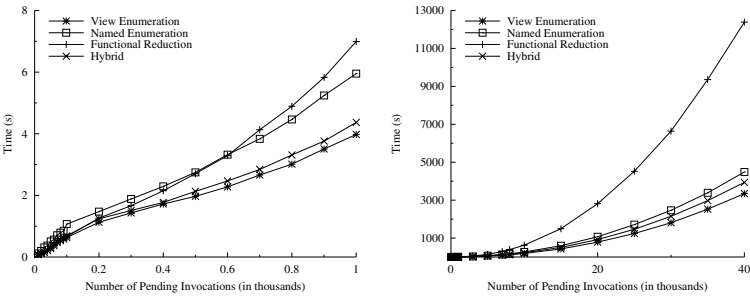


Fig. 13. MEDIAN: Time to service all pending invocations.

5 Discussion

View Enumeration performs the best. The `view` statement incurs little overhead over direct access to the underlying operation implementation, while providing sufficient abstraction of that implementation. Unfortunately, the structure of a `view` statement must closely match that of the associated `inni` statement. This structural dependence potentially limits selection method reuse. Named Enumeration reduces this dependence, but incurs object creation overhead.

Functional Reduction is elegant, but suffers greatly in terms of performance. This strategy's performance penalties can be categorized into method call overhead, repeated selection, and state maintenance. A method call overhead is incurred for each invocation, even when the invocation to service has already been found. In such a case, the number of method calls could be minimized if it were possible to abort reduction. Repeated selection is the repeated execution of selection code to satisfy an invariant (as in MEDIAN). A predicate indicating the last method call of a reduction could reduce the repeated selection penalty. State maintenance is the need to explicitly carry extra state through a reduction (as in MEDIAN). This, unfortunately, makes it necessary to create extra objects.

6 Conclusion

This paper discussed different candidate selection mechanisms considered for addition to the JR programming language. For consideration, a candidate had to satisfy criteria requiring selection of any invocation from the set of pending invocations, access to each invocation's actual arguments, and the prevention of selection side-effects. We have extended the JR programming language with support for the View Enumeration approach because of its high-level of abstraction and performance. Because View Enumeration does not rely on subclassing, this approach can also be used in concurrent languages that are not object-oriented.

References

1. G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.
2. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
3. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
4. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
5. A. Burns. *Programming in Occam*. Addison Wesley, 1988.
6. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

7. M. Chung and R. A. Olsson. New mechanisms for invocation handling in concurrent programming languages. *Computer Languages*, 24:254–270, December 1998.
8. S. Dorward and R. Pike. Programming in Limbo. In *Proceedings of the IEEE Comcon 97 Conference*, pages 245–250, 1997.
9. N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.
10. G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *WoTUG 20*, pages 48–76, 1997.
11. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
12. Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995.
13. A. W. Keen. *Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study*. PhD dissertation, University of California, Davis, Department of Computer Science, June 2002.
<http://www.csc.calpoly.edu/~akeen/papers/thesis.ps>.
14. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 575–584, April 2001.
15. University of Kent. Communicating sequential processes for Java.
<http://www.cs.kent.ac.uk/projects/ofa/jcsp/>.
16. University of Kent. Kent retargetable occam compiler.
<http://www.cs.kent.ac.uk/projects/ofa/kroc/>.
17. R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
18. C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, Monterey, CA, November 1994.

Parallel Juxtaposition for Bulk Synchronous Parallel ML

Frédéric Loulergue*

Laboratory of Algorithms, Complexity and Logic
61, avenue du général de Gaulle – 94010 Créteil cedex – France
loulergue@univ-paris12.fr

Abstract. The `BSMLlib` library is a library for Bulk Synchronous Parallel (BSP) programming with the functional language Objective Caml. It is based on an extension of the λ -calculus by parallel operations on a parallel data structure named parallel vector. An attempt to add a parallel composition to this approach led to a non-confluent calculus and to a restricted form of parallel composition. This paper presents a new, simpler and more general semantics for parallel composition.

1 Introduction

Declarative parallel languages are one possible way that may ease the programming of massively parallel architectures. Those languages do not enforce in the syntax itself an order of evaluation, and thus appear more suitable to automatic parallelization. Functional languages are often considered. Nevertheless, even if some problems encountered in the parallelization of sequential imperative languages are avoided, some still remain (for example two different but denotationally equivalent programs may lead to very different parallel programs) and some are added, for example the fact that in those languages data-structures are always dynamic ones. It makes the amount and/or the grain of parallelism often too low or difficult to control in case of speculation. An opposite direction of research is to give the programmer the entire control over parallelism. Message passing facilities are added to functional languages. But in this case, the obtained parallel languages are either non-deterministic, [10] or non-functional (i.e. referential transparency is lost) [3].

The design of parallel programming languages is, therefore, a tradeoff between the possibility of expressing parallel features necessary for predictable efficiency, but which make programs more difficult to write, to prove and to port ; the abstraction of such features that are necessary to make parallel programming easier, but which must not hinder efficiency and performance prediction.

An intermediate approach is to offer only a set of algorithmic skeletons [1, 11] that are implemented in parallel. Among researchers interested in declarative parallel programming, there is a growing interest in execution cost models

* This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org).

taking into account global hardware parameters like the number of processors and bandwidth. The Bulk Synchronous Parallel [12] execution and cost model offers such possibilities and with similar motivations we have designed BSP extensions of the λ -calculus [8] and a library for the Objective Caml language, called **BSMLlib**, implementing those extensions.

This framework is a good tradeoff for parallel programming for two main reasons. First, we defined a *confluent calculus* so we can design purely functional parallel languages from it. Without side-effects, programs are easier to prove, and to re-use (the semantics is compositional) and we can choose any evaluation strategy for the language. An eager language allows good performances. Secondly, this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

A BSP algorithm is said to be in *direct mode* [5] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the *confluent* BSL calculus [8]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional. Those operations are *flat*: it is impossible to express directly parallel divide-and-conquer algorithms. Nevertheless many algorithms are expressed as parallel divide-and-conquer algorithms [13] and it is difficult to transform them into flat algorithms. In a previous work, we proposed an operation called parallel composition [7], but it was limited to the composition of two terms whose evaluations require the same number of BSP super-steps.

In this paper we present a new version of this operation and a new operation which allow to juxtapose (this is the reason why we renamed this operation parallel *juxtaposition*) on subsets of the parallel machine the evaluations of two terms even if they require a different number of BSP super-steps. We first present the flat **BSMLlib** library (section 2). Section 3 addresses the semantics of the parallel juxtaposition. We also give a small example in section 4, discuss related and future work in section 5 and conclude in section 6.

2 Functional Bulk Synchronous Parallelism

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called **BSMLlib** library is based on the following elements.

It gives access to the BSP parameters of the underlying architecture. In particular, it offers the function: `bsp_p:unit->int` such as the value of `bsp_p()` is p , the static number of processes of the parallel machine. This value does not change during execution, for “flat” programming only. It is no longer true when parallel juxtaposition is added to the language.

There is also an abstract polymorphic type `'a par` which represents the type of p -wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. We have a type system [4] which enforces this restriction.

The BSML parallel constructs operates on parallel vectors. Those parallel vectors are created by: `mkpar: (int -> 'a) -> 'a par` so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p - 1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations and phases of global communication with global synchronization. Asynchronous phases are programmed with `mkpar` and with:

`apply: ('a -> 'b) par -> 'a par -> 'b par`

`apply (mkpar f) (mkpar e)` stores `(f i)` `(e i)` on process i .

The communication and synchronization phases are expressed by:

`put: (int->'a option) par -> (int->'a option) par`

where `'a option` is defined by: `type 'a option = None | Some of 'a`.

Consider the expression: `put(mkpar(fun i->fsi))(*)`. To send a value v from process j to process i , the function `fsj` at process j must be such that `(fsj i)` evaluates to `Some v`. To send no value from process j to process i , `(fsj i)` must evaluate to `None`. Expression `(*)` evaluates to a parallel vector containing a function `fdi` of delivered messages on every process. At process i , `(fdi j)` evaluates to `None` if process j sent no message to process i or evaluates to `Some v` if process j sent the value v to the process i .

A global conditional operation: `ifat: (bool par) * int * 'a * 'a -> 'a` would also be contained in the full language. It is such that `ifat (v, i, v1, v2)` will evaluate to `v1` or `v2` depending on the value of v at process i . But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core BSMLlib contains the function: `at: bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec: bool par)` and `(pid: int)`. `if at` expresses communication and synchronization phases. Without it, the global control cannot take into account data computed locally. Global conditional is necessary of express algorithms like:

Repeat Parallel Iteration Until Max of local errors $< \epsilon$

Due to lack of space, in the following sections we will omit it.

3 Semantics of Parallel Juxtaposition

Parallel juxtaposition allows the network to be divided in two subsets. If we add a notion of parallel juxtaposition to the BS λ -calculus, the confluence is lost [8]. Parallel juxtaposition has to preserve the BSP execution structure. Moreover the number of processors is no longer a constant and depends on the context of the term.

“Flat” terms. The local terms (denoted by lowercase characters) represents “sequential” values held locally by each processor. They are classical λ -expressions. They are given by the following grammar:

$e ::= n$ integer constant	$\lambda \dot{x}. e$ lambda abstraction	$(e \rightarrow e, e)$ conditional
b boolean constant	$e e$ application	$\dot{x}$ local variable
$\oplus$ operations		

The set of processor names $\mathcal{N}$ is a set of p closed local terms in normal form. In the following we will consider that $\mathcal{N} = \{0, \dots, p-1\}$.

The “flat” global terms (denoted by uppercase characters) represent parallel values. They are given by the grammar:

$$E_f ::= \bar{x} \mid E_f E_f \mid E_f e \mid \lambda \bar{x}. E_f \mid \lambda \dot{x}. E_f \mid \pi e \mid E_f \# E_f \mid E_f ? E_f$$

πe is the intentional parallel vector which contains the local term e at processor i . In the $\text{BS}\lambda_p$ -calculus, the number of processors is constant. In this case, πe is syntactic sugar for $\langle e\ 0, \dots, e\ (p-1) \rangle$, where the width of the vector is p . In the practical programming language we will no longer use the enumerated form of parallel vectors. Nevertheless in the evaluation semantics, we will use enumerated vectors as values. Their widths can range from 1 to p .

$E_1 \# E_2$ is the point-wise parallel application. E_1 must be a vector of functions and E_2 a vector of values. On each process, the function is applied to the corresponding value.

$E_1 ? E_2$ is the get operation used to communicate values between processors. E_2 is a parallel vector of processor names. If at processor i the value held by E_2 is j then the result of the get operation (a parallel vector) will contain the value held by parallel vector E_1 at processor j . The execution of this operation corresponds to a communication and synchronization phase of a BSP super-step. With the get operation, a processor can only obtain data from one other processor. There exists a more general communication operation *put*, which allows any BSP communication schema using only one super-step. With this operation a processor can send at most p messages to the p processors of the parallel machine, each messages can have a different content. The *put* operation is of course a bit more complicated and its interaction with parallel juxtaposition is the same than the get operation. For the sake of simplicity we will only consider the get operation in this section.

Terms with parallel juxtaposition. $E_1 \parallel_m E_2$ is the parallel juxtaposition, which means that the m first processors will evaluate E_1 and the others will evaluate E_2 . From the point of view of E_1 the network will have m processors named $0, \dots, m-1$. From the point of view of E_2 the network will have $p-m$ processors (where p is the number of processors of the current network) named $0, \dots, (p-m-1)$ (processor m is renamed 0, *etc.*). We want to preserve the flat BSP super-step structure and avoid subset synchronization [6].

The proposition of this paper relies on the **sync** operation. Consider the term **parjux** $m\ E_1\ E_2$. The evaluation of E_1 and E_2 will be as described below for flat terms but respectively on a network of m and $p-m$ processors except that the synchronization barriers of **put** and **if at** operations will concern the whole network. If for example E_2 has less super-steps and thus less synchronization barriers than E_1 then we need additional, useless in the point of view of E_2 ,

synchronization barriers to preserve the BSP execution model. This is why we have added the **sync** operation. Each operation using synchronization barrier labels its calls to the synchronization barrier. With this label each processor knows what kind of operation the other processors have called to end the super-step. The **sync** operation repeatedly requests synchronization barriers. When each processor requests a synchronization barrier labeled by **sync**, it means all the processors reached the **sync** and the evaluation of **sync** ends. The **sync** operation introduce an additional synchronization barrier, but it allows a more flexible use of parallel juxtaposition.

The complete syntax for global terms E is $E ::= E_f \mid \mathbf{sync}(E_s)$ where

$$E_s ::= \bar{x} \mid E_s E_s \mid E_s e \mid \lambda \bar{x}. E_s \mid \lambda \dot{x}. E_s \mid E_s \# E_s \mid E_s ? E_s \mid \pi e \mid E_s \parallel_e E_s$$

It is important to notice that this syntax does not allow nesting of parallel objects: a local term cannot contain a global term. Thus a parallel vector can not contain a parallel vector.

Semantics. The semantics starts from a *term*, and returns a *value*. The local values are the constants and the local λ -abstractions. The global values are the enumerated vectors of local values and the λ -abstractions.

The evaluation semantics for local terms $e \triangleright_{\text{loc}} v$ is a classical call-by-value evaluation (omitted here for the sake of conciseness). The evaluation of global terms with parallel juxtaposition $\triangleright_s$ uses an environment for global variables: a parallel vector can be bound to a variable in a subnetwork with p processors and this variable can be used in a subnetwork with p' processors ($p' \leq p$). In this case we are only interest in p' of the p values of the vector. So the global evaluation depends on the number of processors in the current subnetwork and the name (in the whole network) of the first processor of this subnetwork.

Judgments of this semantics have the form $\mathcal{E}_s \vdash_f^{p'} E \triangleright_s V$ which means “in global environment $\mathcal{E}_s$ on a subnetwork with p' processors whose first processor has name f in the whole network, the term E_s evaluates to value V ”.

The environment for the evaluation of global terms with parallel juxtaposition $\mathcal{E}_s$ can be seen as an association list which associates global variables to pairs composed of an integer and a global value. In the following we will use the notation $[v_0; \dots; v_n]$ for lists and in particular environments. We will also note $\mathcal{E}(\bar{x})$ for the first value associated with the global variable $\bar{x}$ in $\mathcal{E}$.

The two first rules are used to retrieve a value in the global environment. It is to notice that there are two different rules: one for parallel vectors and one for other values, i.e. global λ -abstractions. In this environment, a variable is not only bound to its value but to a pair whose first component is the name of the first processor of the network on which the value was produced. This information is only useful for the first rule. This rule says that if a parallel vector was bound to a variable in a subnetwork with p_1 processors and with first processor f_1 , then this variable can be used in a subnetwork with p_2 processors ($p_2 \leq p_1$) and with first processor f_2 ($f_2 \geq f_1$). In this case we are only interested by p_2 components of the vector, starting at component numbered $f_2 - f_1$:

$$\frac{\mathcal{E}_s(\bar{x}) = (f_1, \langle v_0, \dots, v_{p_1-1} \rangle)}{\mathcal{E}_s \vdash_{f_2}^{p_2} \bar{x} \triangleright_s \langle v_{f_2-f_1}, \dots, v_{f_2-f_1+p_2-1} \rangle} \quad \frac{\mathcal{E}_s(\bar{x}) = (f_1, \lambda x.E_s)}{\mathcal{E}_s \vdash_{f_2}^{p_2} \bar{x} \triangleright_s \lambda x.E_s}$$

The next rule says that to evaluate an intentional parallel vector term πf , one has to evaluate the expression f i on each processor i of the subnetwork. This evaluation corresponds to a pure computation phase of a BSP super-step:

$$\frac{\forall i \in 0 \dots p' - 1, e \ i \triangleright_{\text{loc}} v_i}{\mathcal{E}_s \vdash_f^{p'} \pi e \triangleright_s \langle v_0, \dots, v_{p'-1} \rangle}$$

The three next rules are for application and point-wise parallel application. Observe that in the rule for application, the pair (f, V_2) , rather than a single value, is bound to the variable $\bar{x}$:

$$\frac{\mathcal{E}_s \vdash_f^{p'} E_s^2 \triangleright_s V_2 \quad (\bar{x} \mapsto (f, V_2)) :: \mathcal{E}_s \vdash_f^{p'} E_s^1 \triangleright_s V_1}{\mathcal{E}_s \vdash_f^{p'} (\lambda \bar{x}. E_s^1) E_s^2 \triangleright_s V_1} \quad \frac{e \triangleright_{\text{loc}} v \quad \mathcal{E}_s \vdash_f^{p'} E_s[x \leftarrow v] \triangleright_s V}{\mathcal{E}_s \vdash_f^{p'} (\lambda \dot{x}. E_s) e \triangleright_s V}$$

$$\frac{\mathcal{E}_s \vdash_f^{p'} E_s^1 \triangleright_s \langle \lambda x.e_0, \dots, \lambda x.e_{p'-1} \rangle \quad \mathcal{E}_s \vdash_f^{p'} E_s^2 \triangleright_s \langle v_0, \dots, v_{p'-1} \rangle \quad \forall i \in 0 \dots p' - 1, (\lambda x.e_i) v_i \triangleright_{\text{loc}} w_i}{\mathcal{E}_s \vdash_f^{p'} E_s^1 \# E_s^2 \triangleright_s \langle w_0, \dots, w_{p'-1} \rangle}$$

The next rule is for the operation that need communications:

$$\frac{\mathcal{E}_s \vdash_f^{p'} E_s^1 \triangleright_s \langle v_0, \dots, v_{p'-1} \rangle \quad \mathcal{E}_s \vdash_f^{p'} E_s^2 \triangleright_s \langle n_0, \dots, n_{p'-1} \rangle}{\mathcal{E}_s \vdash_f^{p'} E_s^1 ? E_s^2 \triangleright_s \langle v_{n_0}, \dots, v_{n_{p'-1}} \rangle}$$

The last important rule is the one for parallel juxtaposition. The left side of parallel juxtaposition is evaluated on a subnetwork with m processors, the right side is evaluated on the remaining subnetwork containing $p' - m$ processors.

$$\frac{\mathcal{E}_s \vdash_f^m E_s^1 \triangleright_s \langle v_0, \dots, v_{m-1} \rangle \quad \mathcal{E}_s \vdash_{f+m}^{p'-m} E_s^2 \triangleright_s \langle v_m, \dots, v_{p'-1} \rangle \quad e \triangleright_{\text{loc}} m \text{ and } 0 < m < p'}{\mathcal{E}_s \vdash_f^{p'} E_s^1 \parallel_e E_s^2 \triangleright_s \langle v_0, \dots, v_{p'-1} \rangle} \quad (1)$$

The remaining rule is a classical one: $\mathcal{E}_s \vdash_f^{p'} \lambda x.E_s \triangleright_s \lambda x.E_s$, where x is either $\dot{x}$ or $\bar{x}$.

We now need to define the evaluation of global terms E (with parallel juxtaposition enclosed in a **sync** operation). The environment $\mathcal{E}$ for the evaluation of global terms E can be seen as an association list which associates global variables to global values. This evaluation has judgments of the form $\mathcal{E} \vdash E \triangleright V$ which means “in global environment $\mathcal{E}$ on the whole network the term E evaluates to value V ”. The first rule is the rule used to evaluate a parallel composition in the scope of a **sync**:

$$\frac{\mathcal{E}_s \vdash_0^p E \triangleright_s V}{\mathcal{E} \vdash \text{sync}(E) \triangleright V}$$

where $\mathcal{E}_s(\bar{x}) = (0, V)$ if and only if $\mathcal{E}(\bar{x}) = V$. The remaining rules are similar to the previous ones except that the number of processes and the name of the first processor of the sub-network are not longer useful. Due to the lack of space we omit these rules.

4 Example

Objective Caml is an eager language. To express parallel juxtaposition as a function we have to “freeze” the evaluation of its parallel arguments. Thus parallel juxtaposition must have the following type:

`parjux: int -> (unit->'a par) -> (unit->'a par) -> 'a par`

The `sync` operation has in the `BSMLlib` library the type `'a -> 'a` even if it is only useful for global terms.

The following example is a divide-and-conquer version of the `scan` program which is defined by $\text{scan} \oplus \langle v_0, \dots, v_{p-1} \rangle = \langle v_0, \dots, v_0 \oplus v_1 \oplus \dots \oplus v_{p-1} \rangle$:

```
let rec scan op vec =
  if bsp_p()=1 then vec
  else let mid = bsp_p()/2 in
    let vec'=parjux mid (fun ()->scan op vec) (fun ()->scan op vec) in
    let msg vec=apply(mkpar(fun i v->
      if i=mid-1
      then fun dst-> if dst>=mid then Some v else None
      else fun dst-> None)) vec
      and parop=parfun2(fun x y->match x with None->y|Some v->op v y) in
    parop (apply(put(msg vec'))(mkpar(fun i->mid-1))) vec'
```

The network is divided into two parts and the scan is recursively applied to those two parts. The value held by the last processor of the first part is broadcast to all the processors of the second part, then this value and the value held locally are combined together by the operator `op` on each processor of the second part. To use this function at top-level, it must be put into a `sync` operation. For example: `let this = mkpar (fun pid->pid) in sync (scan (+) this)`.

5 Related Work

[14] presents another way to divide-and-conquer in the framework of an object-oriented language. There is no formal semantics and no implementation from now on. Furthermore, the proposed operation is rather a parallel *superposition*, several BSP threads use the whole network, than a parallel *juxtaposition*. The same author advocates in [9] a new extension of the BSP model in order to ease the programming of divide-and-conquer BSP algorithms. It adds another level to the BSP model with new parameters to describe the parallel machine.

[15] is an algorithmic skeletons language based on the BSP model and offers divide-and-conquer skeletons. Nevertheless, the cost model is not really the BSP model but the D-BSP model [2] which allows subset synchronization. We follow [6] to reject such a possibility.

6 Conclusions and Future Work

Compared to a previous attempt [7], the parallel juxtaposition has not the two main drawbacks of its predecessor : (a) the two sides of spatial parallel composi-

tion may not have the same number of synchronization barriers. This enhancement comes to the price of an additional synchronization barrier, but only one barrier even in the case of nested parallel compositions ; (b) the cost model is a compositional one. The remaining drawback is that the semantics is strategy-dependant. The use of the spatial parallel composition changes the number of processors: it is an imperative feature. Thus there is no hope in this way to obtain a confluent calculus with a parallel juxtaposition.

The next released implementation of the BSMLlib library will include parallel juxtaposition. Its ease of use will be experimented by implementing BSP algorithms described as divide-and-conquer algorithms in the literature.

References

1. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
2. P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Euro-Par'96*, number 1123–1124. Springer Verlag, 1996.
3. C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, 1995.
4. F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, LNCS. Springer Verlag, 2003. to appear.
5. A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
6. G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98*, pages 242–246. CSREA Press, 1998.
7. F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.
8. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
9. J. Martin and A. Tiskin. BSP Algorithms Design for Hierarchical Supercomputers. *submitted for publication*, 2002.
10. P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.
11. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.
12. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
13. A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.
14. A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, (4), 2001.
15. A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Università degli studi di Pisa, 1999.

Parallelization with Tree Skeletons

Kiminori Matsuzaki¹, Zhenjiang Hu^{1,2}, and Masato Takeichi¹

¹ Graduate School of Information Science and Technology,
University of Tokyo

kmatsu@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

² PRESTO21, Japan Science and Technology Corporation.

Abstract. Trees are useful data structures, but to design efficient parallel programs over trees is known to be more difficult than to do over lists. Although several important tree skeletons have been proposed to simplify parallel programming on trees, few studies have been reported on how to systematically use them in solving practical problems; it is neither clear how to make a good *combination* of skeletons to solve a given problem, nor obvious even how to find suitable operators used in a single skeleton. In this paper, we report our first attempt to resolve these problems, proposing two important transformations, the *tree diffusion transformation* and the *tree context preservation transformation*. The tree diffusion transformation allows one to use familiar recursive definitions to develop his parallel programs, while the tree context preservation transformation shows how to derive associative operators that are required when using tree skeletons. We illustrate our approach by deriving an efficient parallel program for solving a nontrivial problem called the *party planning problem*, the tree version of the famous maximum-weight-sum problem.

Keywords: Parallel Skeletons, Tree Algorithms, Parallelization, Program Transformation, Algorithm Derivation.

1 Introduction

Trees are useful data types, widely used for representing hierarchical structures such as mathematical expressions or structured documents like XML. Due to irregularity (imbalance) of tree structures, developing efficient parallel programs manipulating trees is much more difficult than developing efficient parallel programs manipulating lists. Although several important tree skeletons have been proposed to simplify parallel programming on trees [4,5,13], few studies have been reported on how to systematically use them in solving practical problems.

Although many researchers have devoted themselves to constructing systematic parallel programming methodology using *list* skeletons [1,2,6,8], few have reported the methodology with *tree* skeletons. Unlike linear structure of lists, trees do not have a linear structure, and hence the recursive functions over trees are not linear either (in the sense that there are more than one recursive call in

the definition body). It is this nonlinearity that makes the parallel programming on trees complex and difficult to solve.

In this paper, we aim at a systematic method for parallel programming using tree skeletons, by proposing two important transformations, the *tree diffusion transformation* and the *tree context preservation transformation*.

- The tree diffusion transformation is an extension of the list version [8]. It shows how to decompose familiar recursive programs into equivalent parallel ones in terms of tree skeletons.
- The tree context preservation transformation is an extension of the list version [1]. It shows how to derive associative operators that are required when using tree skeletons.

In addition, to show the usefulness of these theorems, we demonstrate a derivation of an efficient parallel program for solving the *party planning problem*, using tree skeletons defined in Section 2. The party planning problem is an interesting tree version of the well-known maximum-weight-sum problem [2], which appeared as an exercise in [3].

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical *tree* structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend. The problem is to design an algorithm making the guest list, and the goal is to maximize the sum of the conviviality rating of the guest.

It is not easy to decide which tree skeletons to use and how to combine them properly so as to solve this problem. Moreover, skeletons impose restriction (such as associativity) on the functions and operations, and it is not straightforward to find such ones.

The rest of the paper is as follows. After reviewing the tree skeletons in Section 2, we explain our two parallelization transformations for trees: the diffusion transformation in Section 3, and the context preservation transformation in Section 4. We show the experimental results in Section 5, and give conclusion in Section 6.

2 Parallel Skeletons on Trees

To simplify our presentation, we consider binary trees in this paper. The primitive parallel skeletons on binary trees are *map*, *zip*, *reduce*, *upwards accumulate* and *downwards accumulate* [13,14], and their formal definitions using the notation of the Haskell language [9] are described in Fig 1. We will use the Haskell notation for the rest of this paper.

```

data BTree  $\alpha$   $\beta$  = Leaf  $\alpha$ 
               | Node (BTree  $\alpha$   $\beta$ )  $\beta$  (BTree  $\alpha$   $\beta$ )

map :: ( $\alpha \rightarrow \gamma, \beta \rightarrow \delta$ )  $\rightarrow$  BTree  $\alpha$   $\beta \rightarrow$  BTree  $\gamma$   $\delta$ 
map ( $f_L, f_N$ ) (Leaf  $n$ )      = Leaf ( $f_L$   $n$ )
map ( $f_L, f_N$ ) (Node  $l$   $n$   $r$ ) = Node (map ( $f_L, f_N$ )  $l$ ) ( $f_N$   $n$ ) (map ( $f_L, f_N$ )  $r$ )

zip :: BTree  $\alpha$   $\beta \rightarrow$  BTree  $\gamma$   $\delta \rightarrow$  BTree ( $\alpha, \gamma$ ) ( $\beta, \delta$ )
zip (Leaf  $n$ ) (Leaf  $n'$ )      = Leaf ( $n, n'$ )
zip (Node  $l$   $n$   $r$ ) (Node  $l'$   $n'$   $r'$ ) = Node (zip  $l$   $l'$ ) ( $n, n'$ ) (zip  $r$   $r'$ )

reduce :: ( $\alpha \rightarrow \gamma, \gamma \rightarrow \beta \rightarrow \gamma \rightarrow \gamma$ )  $\rightarrow$  BTree  $\alpha$   $\beta \rightarrow \gamma$ 
reduce ( $f_L, f_N$ ) (Leaf  $n$ )      =  $f_L$   $n$ 
reduce ( $f_L, f_N$ ) (Node  $l$   $n$   $r$ ) =  $f_N$  (reduce ( $f_L, f_N$ )  $l$ )  $n$  (reduce ( $f_L, f_N$ )  $r$ )

uAcc :: ( $\alpha \rightarrow \gamma, \gamma \rightarrow \beta \rightarrow \gamma \rightarrow \gamma$ )  $\rightarrow$  BTree  $\alpha$   $\beta \rightarrow$  BTree  $\gamma$   $\gamma$ 
uAcc ( $f_L, f_N$ ) (Leaf  $n$ )      = Leaf ( $f_L$   $n$ )
uAcc ( $f_L, f_N$ ) (Node  $l$   $n$   $r$ ) = let  $l' = uAcc$  ( $f_L, f_N$ )  $l$ 
                                    $r' = uAcc$  ( $f_L, f_N$ )  $r$ 
                                   in Node  $l'$  ( $f_N$  (root  $l'$ )  $n$  (root  $r'$ ))  $r'$ 

dAcc :: ( $\gamma \rightarrow \gamma \rightarrow \gamma$ )  $\rightarrow$  ( $\beta \rightarrow \gamma, \beta \rightarrow \gamma$ )  $\rightarrow$  BTree  $\alpha$   $\beta \rightarrow \gamma \rightarrow$  BTree  $\gamma$   $\gamma$ 
dAcc ( $\oplus$ ) ( $f_L, f_R$ ) (Leaf  $n$ )  $c$       = Leaf  $c$ 
dAcc ( $\oplus$ ) ( $f_L, f_R$ ) (Node  $l$   $n$   $r$ )  $c$  = Node (dAcc ( $\oplus$ ) ( $f_L, f_R$ )  $l$  ( $c \oplus f_L$   $n$ ))  $c$ 
                                   (dAcc ( $\oplus$ ) ( $f_L, f_R$ )  $r$  ( $c \oplus f_R$   $n$ ))

```

Fig. 1. Definitions of five primitive skeletons

The *map* skeleton $map(f_L, f_N)$ applies function f_L to each leaf and function f_N to each internal node. The *zip* skeleton accepts two trees of the same shape and returns a tree whose nodes are pairs of corresponding two nodes of the original two trees. The *reduce* skeleton $reduce(f_L, f_N)$ reduces a tree into a value by applying f_L to each leaf, and f_N to each internal node upwards. Similar to *reduce*, the *upwards accumulate* skeleton $uAcc(f_L, f_N)$ applies f_L to each leaf and f_N to each internal node in a bottom-up manner, and returns a tree of the same shape as the original tree. The *downwards accumulate* skeleton $dAcc(\oplus)(f_L, f_R)$ c computes by propagating accumulation parameter c downwards, and the accumulation parameter is updated by $\oplus$ and f_L when propagated to left child, or updated by $\oplus$ and f_R when propagated to right child.

To guarantee the existence of efficient implementation for the parallel skeletons, we have requirement on the operators and functions used in the above skeletons.

Definition 1 (Semi-Associative). A binary operator $\otimes$ is said to be *semi-associative* if there is an associative operator $\oplus$ such that for any a, b, c , $(a \otimes b) \otimes c = a \otimes (b \oplus c)$. $\square$

Definition 2 (Quasi-Associative). A binary operator $\oplus$ is said to be *quasi-associative* if there is a semi-associative operator $\otimes$ and a function f such that for any a, b , $a \oplus b = a \otimes f b$. $\square$

Definition 3 (Bi-Quasi-Associative). A ternary operator f is said to be *bi-quasi-associative* if there is a semi-associative operator $\otimes$ and two functions f'_L, f'_R such that for any l, n, r , $f \ l \ n \ r = l \otimes f'_L \ n \ r = r \otimes f'_R \ n \ l$. We can fix a bi-quasi-associative operator f by providing $\otimes, \oplus$ (associative operator for $\otimes$), f'_L and f'_R , therefore, we will write f with 4-tuple as $f \equiv [[\otimes, \oplus, f'_L, f'_R]]$. $\square$

Based on the tree contraction technique [12], we require the f_N used in the *reduce* and *upwards accumulate* be bi-quasi-associative, and $\oplus$ in *downwards accumulate* be associative. We omit the detailed description of the cost for each skeleton. Informally, if all the operators used in the skeletons use constant time, all skeletons can be implemented in at most $O(\log N)$ parallel time using enough processors, where N denotes the number of nodes in the tree.

3 Tree Diffusion Theorem

Hu et al. proposed the diffusion theorem (on lists) [8], with which we can directly derive efficient combinations of skeletons from recursive programs. In this section, we start by formalizing a very general tree diffusion theorem, then discuss three practical cases, and finally derive a combination of skeletons for the party planning problem.

Theorem 1 (Tree Diffusion). Let f be defined in the following recursive way over binary trees:

$$\begin{aligned} f \ (Leaf \ n) \ c &= g_L \ (n, c) \\ f \ (Node \ l \ n \ r) \ c &= g_N \ (f \ l \ (c \otimes h_L \ n)) \ (n, c) \ (f \ r \ (c \otimes h_R \ n)) \end{aligned}$$

where g_N is a bi-quasi-associative operator, $\otimes$ is an associative operator, and g_L, h_L, h_R are user-defined functions. Then f can be equivalently defined in terms of the tree skeletons as follows.

$$\begin{aligned} f \ xt \ c &= \mathbf{let} \ ct = dAcc \ (\otimes) \ (h_L, h_R) \ xt \ c \\ &\quad \mathbf{in} \ reduce \ (g_L, g_N) \ (zip \ xt \ ct) \end{aligned}$$

Proof Sketch: This can be proved by induction on the structure of xt . Due to the limitation of space, the proof is given in the technical report [11]. $\square$

This theorem is very general. Practically, It is often the case that the function f returns a tree with the same shape as the input. If we naively apply this diffusion theorem, we will have a costly *reduce* skeleton for combining all sub-trees. To remedy this situation, we propose the following two useful specializations, in which we use appropriate skeletons rather than *reduce*.

The first specialization deals with the function whose computation of new values for each node depends on the original value and the accumulation parameter. For each internal node, such a function f can be defined as $f \ (Node \ l \ n \ r) = Node \ (f \ l \ (c \otimes h_L \ n)) \ (g_N \ (n, c)) \ (f \ r \ (c \otimes h_R \ n))$, and this function can be efficiently computed by *map* rather than *reduce*.

```

ppp xt = ppp' xt True
ppp' (Leaf n) c      = Leaf c
ppp' (Node l n r) c = let (lm, lu) = mis l
                        (rm, ru) = mis r
                        in Node (ppp' l (if c then False else (lm > lu))) c
                        (ppp' r (if c then False else (rm > ru)))
mis (Leaf n)      = (n, 0)
mis (Node l n r) = let (lm, lu) = mis l
                        (rm, ru) = mis r
                        in (lu + n + ru, (lm ↑ lu) + (rm ↑ ru))

```

Fig. 2. A sequential program for party planning program

The second specialization deals with the function whose computation of new values for each node depends on the original value, the accumulation parameter and the new value of its children. For each internal node, such a function f can be defined as $f \text{ (Node } l \text{ n } r) \text{ c} = \text{Node } l' \text{ (} g_N \text{ (root } l') \text{ (n, c) (root } r')) \text{ r'}$ where $l' = f \text{ l } (c \otimes h_L \text{ n})$ and $r' = f \text{ l } (c \otimes h_R \text{ n})$. This function can be efficiently computed by *upwards accumulate* rather than *reduce*.

Let us discuss another practical matter for the case where the function f calls an auxiliary function k to compute over the sub-trees. Such a function can be defined as follows.

$$\begin{aligned}
 f \text{ (Leaf } n) \text{ c} &= \text{Leaf } (g_L \text{ ((-, n, -), c)}) \\
 f \text{ (Node } n \text{ l } r) \text{ c} &= \text{let } n' = (k \text{ l } n, k \text{ r}) \\
 &\quad \text{in Node } (f \text{ l } (c \otimes h_L \text{ n}')) \text{ (} g_N \text{ (n', c)) (f r (c \otimes h_R \text{ n}')) \\
 k \text{ (Leaf } n) &= k_L \text{ n} \\
 k \text{ (Node } l \text{ n } r) &= k_N \text{ (k l) n (k r)}
 \end{aligned}$$

It is a little difficult to efficiently parallelize this recursive function into the combination of primitive skeletons, because there are multiple traversals over the trees, and naive computation of f will make redundant function calls of k . By making use of the tupling transformation and the fusion transformation [7], we can parallelize the function efficiently. In the following, we use a function *gather_ch*, which accepts two trees of the same shape and makes a triple for each node. The triple consists of a node of the first tree and two immediate children of the second tree. Detailed discussions are referred to [11].

Corollary 1 (Paramorphic Diffusion). The function f defined above can be diffused into the following combination of skeletons if k_N is a bi-quasi-associative operator, and $\otimes$ is associative.

$$\begin{aligned}
 f \text{ xt } c &= \text{let } yt = \text{gather\_ch } xt \text{ (uAcc (k}_L, k_N) \text{ xt)} \\
 &\quad \text{in dAcc } (\otimes) \text{ (h}_L, h_R) \text{ yt } c
 \end{aligned}$$

□

Having shown the diffusion theorem and its corollaries, we now try to derive a parallel program for the party planning problem. By making use of dynamic

programming technique, we can obtain an efficient sequential program as shown in Fig 2. Here, the function *mis* accepts a tree, and returns a pair of values which are the maximum independent sums when the root of the input is marked or unmarked. The recursive function *ppp'* is defined with an accumulation parameter, which represents a node to be marked or unmarked. The recursive function *ppp'* is a paramorphic function because it calls an auxiliary function *mis* on each sub-tree, therefore, let us use paramorphic diffusion theorem to derive the following program in terms of skeletons.

$$\begin{aligned} ppp \ x t &= ppp' \ x t \ True \\ ppp' \ x t \ c &= \text{let } yt = \text{gather\_ch } x t \ (uAcc \ (\underline{mis_L}, \underline{mis_N}) \ x t) \\ &\quad \text{in } dAcc \ (\underline{\otimes}) \ (\underline{h_L}, \underline{h_R}) \ yt \ c \end{aligned}$$

However, we have not yet parallelized the underlined parts successfully. First, from the definition of the sequential program, we can derive $\underline{mis_L} \ n = (n, 0)$ and $\underline{mis_N} \ (l_m, l_u) \ n \ (r_m, r_u) = (l_u + n + r_u, (l_m \uparrow l_u) + (r_m \uparrow r_u))$, however, we have still to show the bi-quasi-associativity of $\underline{mis_N}$. Second, we have to derive an associative operator $\otimes$ and two functions h_L and h_R such that $c \otimes h_L \ ((l_m, l_u), n, (r_m, r_u)) = \text{if } c \text{ then } False \text{ else } (l_m > l_u)$ and almost the same equation for h_R hold. In the following section, we will see how to derive those operators.

4 Tree Context Preservation

The parallel skeletons require the operators used in them to be (bi-quasi)-associative, however, it is not straightforward to find such ones for many practical problems. For linear self-recursive programs, Chin et al. proposed the *context preservation transformation* [1], with which one can systematically derive such operators based on the associativity of function composition. In this section, we will extend the transformation theorem for tree skeletons. Our main idea is to resolve the non-linear functions over trees into two linear recursive functions, and then we can consider the context preservation on these two linear functions. We start by introducing the basic notations and concepts about contexts.

Definition 4 (Context Extraction [1]). Given an expression E and sub-terms $\langle e_1, \dots, e_n \rangle$, we shall express its extraction by: $E \Longrightarrow E' \langle e_1, \dots, e_n \rangle$. The context E' has a form of $\lambda \langle \_1, \dots, \_n \rangle. [e_i \mapsto \_i]_{i=1}^n E$, where $\_i$ denotes a new hole and $[e_i \mapsto \_i]_{i=1}^n E$ denotes a substitution notation of e_i in E to $\_i$. $\square$

Definition 5 (Skeletal Context [1]). A context E is said to be a *skeletal context* if every sub-term in E contains at least one hole. Given a context E , we can make it into a skeletal one E_S by extracting all sub-terms that do not contain holes. This process shall be denoted by $E \Longrightarrow_S E_S \langle e_i \rangle_{i \in N}$ $\square$

Definition 6 (Context Transformation [1]). A context may be transformed (or simplified) by either applying laws or unfolding. We shall denote this process as $E \Longrightarrow_T E'$. $\square$

Definition 7 (Context Preservation Modulo Replication [1]). A context E with one hole is said to be preserved modulo replication if there is a skeletal context E_S , $E \Rightarrow_S E_S \langle t_i \rangle$ and $E_S \langle \alpha_i \rangle \circ E_S \langle \beta_i \rangle = E_S \langle \gamma_i \rangle$ hold, where α_i and β_i are variables, and γ_i are sub-terms without holes. $\square$

Now, we will discuss about the functions which can be transformed into a program with *reduce*. We can show the case of *uAcc* in the same manner, however, due to the limitation of space, we give it in the technical report [11].

Definition 8 (Simple Upwards Recursive Function). A function is said to be a *simple upwards recursive function* (*SUR-function* for short) if it has the following form.

$$\begin{aligned} f \text{ (Leaf } n) &= f_L n \\ f \text{ (Node } l \ n \ r) &= f_N (f \ l) \ n \ (f \ r) \end{aligned} \quad \square$$

The inductive case of an SUR-function has two recursive calls, $f \ l$ and $f \ r$, therefore, we cannot apply the Chin's theorem. To resolve this non-linearity, we define the extraction of two linear recurring contexts from an SUR-function, and extended context preservation for these two contexts as shown in the following.

Definition 9 (Left(Right)-Recurring Context). For the inductive case of an SUR-function, we can extract the *left(right)-recurring context* E^L (E^R) by abstracting either of the recurring terms: $f \text{ (Node } l \ n \ r) = E^L \langle f \ l \rangle = E^R \langle f \ r \rangle$. $\square$

Definition 10 (Mutually Preserved Contexts). Two linear recurring contexts E^L, E^R are said to be *mutually preserved* if there exists a skeletal context E_S such that $E^L \Rightarrow_S E_S \langle g^l \ n \ r \rangle$, $E^R \Rightarrow_S E_S \langle g^r \ n \ l \rangle$ and $E_S \langle \alpha \rangle \circ E_S \langle \beta \rangle = E_S \langle \gamma \rangle$ hold. Here, γ is a sub-terms computed only with variables α and β . $\square$

Based on the idea of tree contraction algorithm, we can parallelize the SUR-function as shown in the following theorem. Due to the limitation of space we omit the proof, which is given in the technical report [11].

Theorem 2 (Context Preservation for SUR-function). The SUR-function function f can be parallelized to $f = \text{reduce } (f_L, f_N)$ if there exist a skeletal context E_S such that $E^L \Rightarrow_S E_S \langle g^l \ n \ r \rangle$, $E^R \Rightarrow_S E_S \langle g^r \ n \ l \rangle$ and $E_S \langle \alpha \rangle \circ E_S \langle \beta \rangle = E_S \langle \gamma \rangle$ hold. Here, f_N is a bi-quasi-associative operator such as $f_N \equiv [[\oplus, \otimes, g^l, g^r]]$ where $x \oplus \alpha = E_S \langle \alpha \rangle \langle x \rangle$ and $\beta \otimes \alpha = \gamma$. $\square$

Next, we discuss about the functions which can be transformed into a program with *dAcc*. As is the case of *reduce*, based on the tree contraction algorithm, we can parallelize a non-linear function by extracting two linear contexts and showing these contexts to be mutually preserved. Due to the limitation of space, we only show the definitions and theorem for this.

Definition 11 (Simple Downwards Recursive Function). A function is said to be a *simple downwards recursive function* (*SDR-function* for short) if it has the following form.

$$\begin{aligned} f \text{ (Leaf } n) \ c &= \text{Leaf } c \\ f \text{ (Node } l \ n \ r) \ c &= \text{Node } (f \ l \ (f_L \ c \ n)) \ c \ (f \ r \ (f_R \ c \ n)) \end{aligned} \quad \square$$

Definition 12 (Recurring Contexts for SDR-function). For the inductive case of an SDR-function f , we can obtain two *recurring contexts* D^L, D^R by abstracting the recursive calls on the accumulative parameter respectively, $f \text{ (Node } l \text{ } n \text{ } r) \text{ } c = \text{Node } (f \text{ } l \text{ } D^L \langle c \rangle) \text{ } c \text{ (} f \text{ } r \text{ } D^R \langle c \rangle)$. $\square$

Theorem 3 (Context Preservation for SDR-function). The SDR-function f can be parallelized to $f \text{ } xt \text{ } c = \text{map } ((c \otimes), (c \otimes)) \text{ (} dAcc \text{ } (\oplus) \text{ (} g^l, g^r \text{) } \iota_{\oplus})$ if there exist a skeletal context E_S such that $D^L \Rightarrow_S D_S \langle g^l \text{ } n \rangle$, $D^R \Rightarrow_S D_S \langle g^r \text{ } n \rangle$ and $D_S \langle \alpha \rangle \circ D_S \langle \beta \rangle = D_S \langle \gamma \rangle$ hold. Here, the operators are defined as $\beta \oplus \alpha = \gamma$ and $c \otimes \alpha = D_S \langle \alpha \rangle \langle c \rangle$, and $\iota_{\oplus}$ is the unit of $\oplus$. $\square$

Having shown the context preservation theorems for trees, we now demonstrate how these theorems work by deriving an associative operator $\otimes$ and functions h_R, h_L in the diffused program in Section 3. The corresponding part is defined recursively as follows.

$$\begin{aligned} ppp' \text{ (Node } l \text{ ((} l_m, l_u \text{)), } n, (r_m, r_u) \text{) } r) \text{ } c \\ = \text{Node } (ppp' \text{ } l \text{ (if } c \text{ then } False \text{ else } (l_m > l_u))) \text{ } c \\ (ppp' \text{ } r \text{ (if } c \text{ then } False \text{ else } (r_m > r_u))) \end{aligned}$$

From this definition, we can obtain the following two linear recurring contexts by abstracting recursive calls.

$$\begin{aligned} D^L &= \lambda \langle c \rangle. \text{if } c \text{ then } False \text{ else } (l_m > l_u) \\ D^R &= \lambda \langle c \rangle. \text{if } c \text{ then } False \text{ else } (r_m > r_u) \end{aligned}$$

We can show that these two contexts are mutually recursive because the skeletal context $D_S = \lambda \langle \_1, \_2 \rangle. \lambda \langle c \rangle. \text{if } c \text{ then } \_1 \text{ else } \_2$ satisfies our requirement.

$$\begin{aligned} D^L &= D_S \langle g^l \text{ ((} l_m, l_u \text{)), } n, (r_m, r_u) \rangle, \quad D^R = D_S \langle g^r \text{ ((} l_m, l_u \text{)), } n, (r_m, r_u) \rangle \\ \textbf{where } g^l \text{ ((} l_m, l_u \text{)), } n, (r_m, r_u) \rangle &= (false, (l_m > l_u)) \\ g^r \text{ ((} l_m, l_u \text{)), } n, (r_m, r_u) \rangle &= (false, (r_m > r_u)) \end{aligned}$$

$$\begin{aligned} D_S \langle \alpha_1, \alpha_2 \rangle \circ D_S \langle \beta_1, \beta_2 \rangle \\ = \lambda \langle c \rangle. \text{if } c \text{ then (if } \beta_1 \text{ then } \alpha_1 \text{ else } \alpha_2) \text{ else (if } \beta_2 \text{ then } \alpha_1 \text{ else } \alpha_2) \\ = D_S \langle \text{if } \beta_1 \text{ then } \alpha_1 \text{ else } \alpha_2, \text{if } \beta_2 \text{ then } \alpha_1 \text{ else } \alpha_2 \rangle \end{aligned}$$

From the derivations above, we can apply theorem 3 to obtain an efficient parallel program with *map* and *downwards accumulate*. The whole parallel program for the party planning problem is shown in Fig 3. Detailed derivations are referred to [11].

5 An Experiment

We have conducted an experiment on the party planning problem. We have coded our algorithm using C++, the MPI library and our implementation of tree skeletons [10]. We have used a tree of 999,999 nodes for our experiment.

Fig 4 shows the result of the program executed on our PC-Cluster using 1 to 12 processors. This result is shown in the speedup excluding partitioning and flattening of the tree. The almost linear speedup shows the effectiveness of the program derived by our theorems.

```

ppp xt = let yt = gather_ch xt (uAcc (mis_L, mis_N) xt)
         in map (fst, fst) (dAcc (⊙) (h_L, h_R) yt ι_⊙)

where
mis_L = (n, 0)
mis_N ≡ [[⊕, ⊗, f^L, f^R]]
(β1, β2, β3, β4) ⊕ (α1, α2, α3, α4) = ((β1 + α1) ↑ (β3 + α2),
  (β2 + α1) ↑ (β4 + α2), (β1 + α3) ↑ (β3 + α4), (β2 + α3) ↑ (β4 + α4))
(xm, xu) ⊗ (α1, α2, α3, α4) = ((xm + α1) ↑ (xu + α2), (xm + α3) ↑ (xu + α4))
f^L n (rm, ru) = (−∞, n + ru, rm ↑ ru, rm ↑ ru)
f^R n (lm, lu) = (−∞, n + lu, lm ↑ lu, lm ↑ lu)

(β1, β2) ⊙ (α1, α2) = (if β1 then α1 else α2, if β2 then α1 else α2)
ι_⊙ = (True, False)
h_L ((lm, lu), n, (rm, ru)) = (False, (lm > lu))
h_R ((lm, lu), n, (rm, ru)) = (False, (rm > ru))

```

Fig. 3. Parallel program for party planning problem

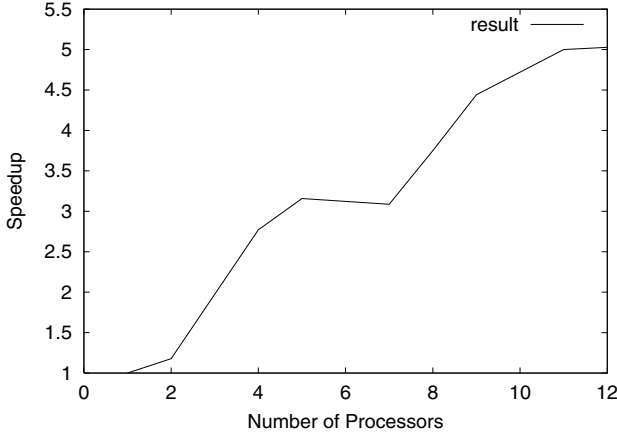


Fig. 4. Experiment result

6 Conclusion

In this paper, we have proposed two parallelization transformations, the *tree diffusion transformation* and the *context preservation transformation*, for helping programmers to systematically derive efficient parallel programs in terms of tree skeletons from the recursive programs. The list versions of these two theorems have been proposed and shown important in skeletal parallel programming, which once in fact motivated us to see if we could generalize them for trees. Due to the non-linearity of the tree structures, it turns out to be more difficult than we had expected. Although the usefulness of our theorems await more evidence, our successful derivation of the *first* skeletal parallel program for solving the party planning problem and the good experiment result have indicated that this is a good start and is worth further investigation.

We are currently working on generalizing the context preservation theorem so that we can relax conditions of the skeletons. In addition, we are figuring out whether we can automatically parallelize the recursive programs on trees.

References

1. W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pages 153–162, May 1998.
2. M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
4. J. Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.
5. J. Gibbons. Computing downwards accumulations on trees quickly. In G. Gupta, G. Mohay, and R. Topor, editors, *Proceedings of 16th Australian Computer Science Conference*, volume 15 (1), pages 685–691. Australian Computer Science Communications, February 1993.
6. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.
7. Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.
8. Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.
9. S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. Available online: <http://www.haskell.org>, February 1999.
10. K. Matsuzaki, Z. Hu, and M. Takeichi. Implementation of parallel tree skeletons on distributed systems. In *Proceedings of The Third Asian Workshop on Programming Languages And Systems*, pages 258–271, Shanghai, China, 2002.
11. K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. Technical Report METR 2003-21, Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2003.
12. M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann Publishers, 1996.
13. D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
14. D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

Topic 11

Numerical Algorithms and Scientific Engineering Problems

Iain S. Duff, Luc Giraud, Henk A. van der Vorst, and Peter Zinterhof
Topic Chairs

Following the traditions of previous Euro-Par Conferences, Euro-Par 2003 includes the topic “Numerical Algorithms”. This topic has been scheduled this year for a morning and afternoon sessions on 2003, 29 August.

Current research and its applications in nearly all areas of natural sciences involves, with increasing importance, mathematical modelling and numerical simulation. Indeed, such a basis for computation has recently been extended to other disciplines like engineering and economics. At the same time, there is a continuously increasing number of requests to solve problems with growing complexity. It is thus nearly superfluous to say that the efficient design of parallel numerical algorithms becomes more and more essential.

As for previous Euro-Par Conferences, we have accepted only papers in which fundamental numerical algorithms are addressed and which are therefore of a wide range of interest, including parallel numerical algebra with new developments for Sylvester-type matrix equations and the one-sided Jacobi method, parallel work in solving ODEs and PDEs including applications to global ocean circulation. Out of 14 submitted papers, we accepted only 5 as regular presentations and 2 as short.

According to this selection, we have divided the presentations into two sessions. One is mainly devoted to methods of parallel linear algebra and the second to parallel methods for PDEs.

We hope and we believe again that the sessions contain a highly interesting mix of parallel numerical algorithms. We would like to take this opportunity of thanking all contributing authors as well as all reviewers for their work.

Parallel ScaLAPACK-Style Algorithms for Solving Continuous-Time Sylvester Matrix Equations

Robert Granat, Bo Kågström, and Peter Poromaa

Department of Computing Science and HPC2N, Umeå University,
SE-901 87 Umeå, Sweden.

`{granat,bokg,peterp}@cs.umu.se`

Abstract. An implementation of a parallel ScaLAPACK-style solver for the general Sylvester equation, $\text{op}(A)X - X\text{op}(B) = C$, where $\text{op}(A)$ denotes A or its transpose A^T , is presented. The parallel algorithm is based on explicit blocking of the Bartels-Stewart method. An initial transformation of the coefficient matrices A and B to Schur form leads to a reduced triangular matrix equation. We use different matrix traversing strategies to handle the transposes in the problem to solve, leading to different new parallel wave-front algorithms. We also present a strategy to handle the problem when 2×2 diagonal blocks of the matrices in Schur form, corresponding to complex conjugate pairs of eigenvalues, are split between several blocks in the block partitioned matrices. Finally, the solution of the reduced matrix equation is transformed back to the originally coordinate system. The implementation acts in a ScaLAPACK environment using 2-dimensional block cyclic mapping of the matrices onto a rectangular grid of processes. Real performance results are presented which verify that our parallel algorithms are reliable and scalable.

Keywords: Sylvester matrix equation, continuous-time, Bartels-Stewart method, blocking, GEMM-based, level 3 BLAS, SLICOT, ScaLAPACK-style algorithms.

1 Introduction

We present a parallel ScaLAPACK-style solver for the Sylvester equation (SYCT)

$$\text{op}(A)X - X\text{op}(B) = C, \quad (1)$$

where $\text{op}(A)$ denotes A or its transpose A^T . Here A of size $M \times M$, B of size $N \times N$ and C of size $M \times N$ are arbitrary matrices with real entries. Equation (1) has a unique solution X of size $M \times N$ if and only if $\text{op}(A)$ and $\text{op}(B)$ have disjoint spectra. The Sylvester equation appears naturally in several applications. Examples include block-diagonalizing of a matrix in Schur form and condition estimation of eigenvalue problems (e.g., see [15,10,16]).

Our method for solving SYCT (1) is based on the Bartels-Stewart method [1]:

1. Transform A and B to upper (quasi)triangular form T_A and T_B , respectively, using orthogonal similarity transformations:

$$Q^T A Q = T_A, \quad P^T B P = T_B.$$

2. Update the matrix C with respect to the transformations done on A and B :

$$\tilde{C} = Q^T C P.$$

3. Solve the reduced (quasi)triangular matrix equation:

$$\text{op}(T_A) \tilde{X} - \tilde{X} \text{op}(T_B) = \tilde{C}.$$

4. Transform the solution $\tilde{X}$ back to the original coordinate system:

$$X = Q \tilde{X} P^T.$$

The quasitriangular form mentioned in Step 1 is also called the real Schur form, which means that the matrix is upper block triangular with 1×1 and 2×2 diagonal blocks, corresponding to real and complex conjugate pairs of eigenvalues, respectively. To carry out Step 1 we use the QR-algorithm [2]. The updates in Step 2 and the back-transformation in Step 4 are carried out using ordinary GEMM-operations $C \leftarrow \beta C + \alpha \text{op}(A) \text{op}(B)$, where α and β are scalars [5,13,14].

Our focus is on Step 3. Using the Kronecker product notation, $\otimes$, we can rewrite the triangular Sylvester equation as a linear system of equations

$$Zx = y, \tag{2}$$

where $Z = I_N \otimes \text{op}(A) - \text{op}(B)^T \otimes I_M$ is a matrix of size $MN \times MN$, $x = \text{vec}(X)$ and $y = \text{vec}(C)$. As usual, $\text{vec}(X)$ denotes an ordered stack of the columns of the matrix X from left to right starting with the first column. The linear system (2) can be solved to the cost of $O(M^3 N^3)$ using ordinary LU factorization with pivoting. This is a very expensive operation, even for moderate-sized problems. Since A and B are (quasi)triangular, the triangular Sylvester equation can indeed be solved to the cost $O(M^2 N + MN^2)$ using a combined backward/forward substitution process [1]. In blocked algorithms, the explicit Kronecker matrix representation $Zx = y$ is used in kernels for solving small-sized matrix equations (e.g., see [11,12,15]).

The rest of the paper is organized as follows: In Section 2, we give a brief overview of blocked algorithms for solving the triangular SYCT equation. Section 3 is devoted to parallel algorithms focusing on the solution of the reduced triangular matrix equations. Finally, in Section 4, we present experimental results and discuss the performance of our general ScaLAPACK-style solver.

Our parallel implementations mainly adopt to the ScaLAPACK software conventions [3]. The P processors (or virtual processes) are viewed as a rectangular processor grid $P_r \times P_c$, with $P_r \geq 1$ processor rows and $P_c \geq 1$ processor columns such that $P = P_r \cdot P_c$. The data layout of dense matrices on a rectangular grid is assumed to be done by the two-dimensional (2D) block-cyclic distribution scheme.

2 Blocked Algorithms

Blocking is a powerful tool in Numerical Linear Algebra to restructure well-known standard algorithms in level 3 operations with the potential to reuse data already stored in cache or registers. This will make things faster and more efficient on one processor or in a shared memory environment. Blocking is also useful for parallelizing tasks in distributed memory environments.

2.1 The Non-transposed Case

We start by reviewing the serial block algorithm proposed in [15] for the non-transposed triangular Sylvester equation

$$AX - XB = C. \quad (3)$$

Here A and B have already been transformed to real Schur form. Let MB and NB be the block sizes used in the partitioning of A and B , respectively. Then MB is the row-block size and NB is the column-block size of C and X (which overwrites C). Now, the number of diagonal blocks of A and B can be expressed as $D_a = \lceil M/MB \rceil$ and $D_b = \lceil N/NB \rceil$, respectively. Then Equation (3) can be rewritten in block-partitioned form:

$$A_{ii}X_{ij} - X_{ij}B_{jj} = C_{ij} - \left(\sum_{k=i+1}^{D_a} A_{ik}X_{kj} - \sum_{k=1}^{j-1} X_{ik}B_{kj} \right), \quad (4)$$

where $i = 1, 2, \dots, D_a$ and $j = 1, 2, \dots, D_b$. Based on this summation formula, a serial blocked algorithm can be formulated, see Figure 1.

```

for j=1, Db
  for i=Da, 1, -1
    {Solve the (i, j)th subsystem}
    AiiXij - XijBjj = Cij
    for k=1, i - 1
      {Update block column j of C}
      Ckj = Ckj - AkiXij
    end
    for k=j + 1, Db
      {Update block row i of C}
      Cik = Cik + XijBjk
    end
  end
end

```

Fig. 1. Block algorithm for solving $AX - XB = C$, A and B in upper Schur form.

2.2 The Transposed Cases

The three other cases, namely,

$$AX - XB^T = C, \quad A^T X - XB = C, \quad \text{and} \quad A^T X - XB^T = C,$$

can be treated in the same way. Each of these matrix equations correspond to a summation formula based on the same block partitioning of the matrices:

$$\begin{aligned} A_{ii}X_{ij} - X_{ij}B_{jj}^T &= C_{ij} - \left(\sum_{k=i+1}^{D_a} A_{ik}X_{kj} - \sum_{k=j+1}^{D_b} X_{ik}B_{jk}^T \right), \\ A_{ii}^T X_{ij} - X_{ij}B_{jj} &= C_{ij} - \left(\sum_{k=1}^{i-1} A_{ki}^T X_{kj} - \sum_{k=1}^{j-1} X_{ik}B_{kj} \right), \\ A_{ii}^T X_{ij} - X_{ij}B_{jj}^T &= C_{ij} - \left(\sum_{k=1}^{i-1} A_{ki}^T X_{kj} - \sum_{k=j+1}^{D_b} X_{ik}B_{jk}^T \right). \end{aligned}$$

For each of these summation formulas a serial block algorithm is formulated. In Figure 2, we present the one corresponding to the reduced triangular equation $A^T X - XB^T = C$.

```

for j= Db, 1, -1
  for i=1, Da
    {Solve the (i, j)th subsystem}
    AiiTXij - XijBjjT = Cij
    for k=i+1, Da
      {Update block column j of C}
      Ckj = Ckj - AikTXij
    end
    for k=1, j-1
      {Update block row i of C}
      Cik = Cik + XijBkjT
    end
  end
end
end

```

Fig. 2. Block algorithm for solving $A^T X - XB^T = C$, A and B in upper Schur form.

Notice that each summation formula sets a starting point in the matrix C/X where we start to compute the solution. For example, while solving subsystems and updating C/X with respect to these subsolutions in Figure 1, we traverse the matrix C/X along its block diagonals from South-East to North-West (or vice versa). This “wavefront” starts in the South-West corner of C/X , as depicted in Figure 3, and moves in the North-Eastern direction. Along the way, each computed X_{ij} will be used to update block-row i and block-column j of C .

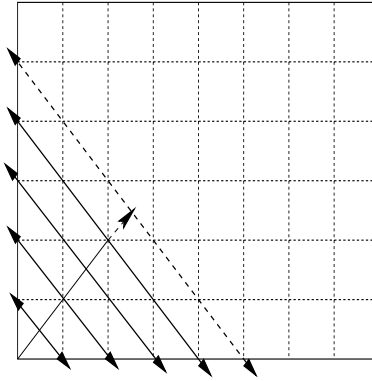


Fig. 3. Traversing the matrix C/X when solving $AX - XB = C$.

3 Parallel Block Algorithms

We assume that the matrices A , B and C are distributed using 2D block-cyclic mapping across a $P_r \times P_c$ processor grid. To carry out Steps 1, 2 and 4 of the Bartels–Stewart method in parallel we use the ScaLAPACK library-routines PDGEHRD, PDLAQR and PDGEMM [3]. The first two routines are used in Step 1 to compute the Schur decompositions of A and B (reduction to upper Hessenberg form followed by the parallel QR algorithm [9,8]). PDGEMM is the parallel implementation of the level 3 BLAS DGEMM operation and is used in Steps 2 and 4 for doing the two-sided matrix multiply updates.

To carry out Step 3 in parallel, we traverse the matrix C/X along its block diagonals, starting in the corner that is decided by the data dependencies. To be able to compute X_{ij} for certain values of i and j , we need A_{ii} and B_{jj} to be owned by the same process that owns C_{ij} . We also need to have the blocks used in the updates in the right place at the right time. The situation is illustrated in Figure 4, where all the data lie on the right processors. This will however not be the general case.

In general, we have to communicate for some blocks during the solves and updates. For example, while traversing C/X for the triangular $AX - XB^T = C$, we solve the small subsystems

$$A_{ii}X_{ij} - X_{ij}B_{jj}^T = C_{ij}, \quad (5)$$

associated with the current block diagonal in parallel. Then we do the GEMM-updates,

$$\begin{cases} C_{kj} = C_{kj} - A_{ki}X_{ij}, & k = 1, \dots, j-1 \\ C_{ik} = C_{ik} + X_{ij}B_{kj}^T, & k = 1, \dots, i-1, \end{cases} \quad (6)$$

of block-row i and block-column j , which can also be done in parallel. In equation (6), the submatrix X_{ij} has been computed in the preceeding step and broadcasted (see Figure 4) to the processors involved in the GEMM-updates. It can be

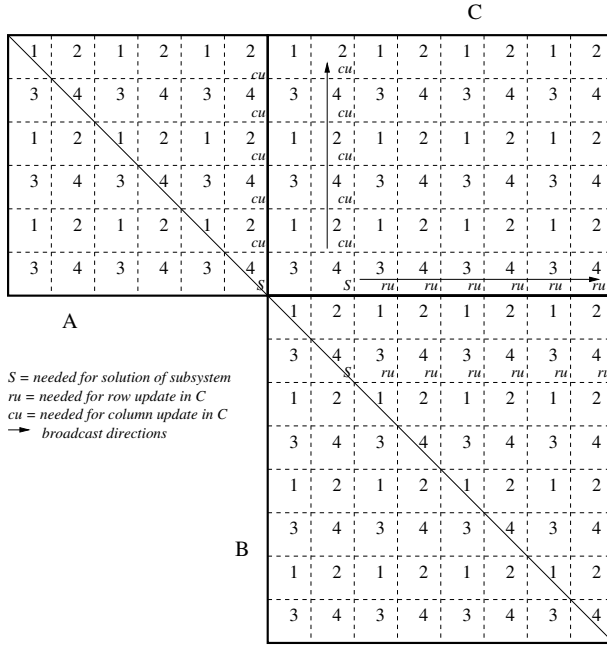


Fig. 4. Data dependencies and mapping for $AX - XB = C$ on a 2×2 processor grid.

shown that this gives a theoretical limit for the speedup of the triangular solver as $\max(P_r, P_c)$ for solving the subsystems, and as $P_r \cdot P_c$ for the GEMM-updates [7,16]. A high-level parallel block algorithm for the solving the general triangular SYCT equation (1) is presented in Figure 5.

3.1 The 2×2 Diagonal Block Split Problem

When entering the triangular solver we already have transformed A , B and C distributed across the processor grid (2D block cyclic mapping). Now, we have to assure that no 2×2 diagonal block of A or B in Schur form, corresponding to conjugate pairs of complex eigenvalues is being split between two blocks (processors). This happens when any element on the first subdiagonal of the real Schur form is not equal to zero and does not belong to the same block (submatrix) as the closest elements to the North, East and North-East. We solve this problem by extending one block in the real Schur form of the matrix such that the lost element is included in that block. At the same time we have to diminish some other neighboring blocks. An explicit redistribution of the matrices would cause to much overhead. Instead, we do the redistribution *implicitly*, that is, we only exchange elements in one row and one column, which are stored in local workarrays. Somehow we must keep track of the extensions/reductions done. As we can see, they are completely determined by the looks of A_{ii} and B_{jj} . Therefore, we can use two 1D-arrays, call them INFO_ARRAY_A and INFO_ARRAY_B

```

for  $k=1$ , # block diagonals in  $C$ 
  {Solve subsystems on current block diagonal in parallel}
  if(mynode holds  $C_{ij}$ )
    if(mynode does not hold  $A_{ii}$  and/or  $B_{jj}$ )
      Communicate for  $A_{ii}$  and/or  $B_{jj}$ 
    Solve for  $X_{ij}$  in  $op(A_{ii})X_{ij} - X_{ij}op(B_{ij}) = C_{ij}$ 
    Broadcast  $X_{ij}$  to processors that need  $X_{ij}$  for updates
  elseif(mynode needs  $X_{ij}$ )
    Receive  $X_{ij}$ 
    if(mynode does not hold needed block in  $A$  for updating block column  $j$ )
      Communicate for requested block in  $A$ 
    Update block column  $j$  of  $C$  in parallel
    if(mynode does not hold needed block in  $B$  for updating block row  $i$ )
      Communicate for requested block in  $B$ 
    Update block row  $i$  of  $C$  in parallel
  endif
end

```

Fig. 5. Parallel block algorithm for $op(A)X - Xop(B) = C$, A and B in Schur form.

of length $\lceil M/MB \rceil$ and $\lceil N/NB \rceil$, respectively, which store information of the extensions as integer values as follows:

$$INFO\_ARRAY\_A(i) = \begin{cases} 0 & \text{if } A_{ii} \text{ is unchanged} \\ 1 & \text{if } A_{ii} \text{ is extended} \\ 2 & \text{if } A_{ii} \text{ is diminished} \\ 3 & \text{if } A_{ii} \text{ is extended and diminished} \end{cases}$$

The first thing to do in our triangular solver is traversing the first subdiagonal of A and B and assigning values to their `INFO_ARRAY`s. Here, we are forced to do some broadcasts since the information must be global, but since this is an $O(\lceil M/MB \rceil)$ or $O(\lceil N/NB \rceil)$ operation they only effect the overall performance marginally. Then, using the data in the global arrays, we can carry out an implicit redistribution by exchanging data between the processors and build up local arrays of double precision numbers holding the extra rows and/or columns for each block of A , B and C . These local arrays can then be used to form the “correct” submatrices for our solves and updates in the parallel triangular solver.

4 Performance Results and Analysis

We present measured performance results of our ScaLAPACK-style algorithms using up to 64 processors on the IBM Scalable POWERparallel (SP) system at High Performance Computing Center North (HPC2N). A theoretical scalability analysis ongoing work and is not included in this paper.

We vary $P = P_r \cdot P_c$ between 1 and 64 in multiples of 2. Speedup S_p and efficiency E_p are computed with respect to the run for the current problem size

Table 1. Performance of PDTRSY solving $AX - XB = C$ and $AX - XB^T = C$.

$M = N$	MB	P_r	P_l	Time (sec.)		S_p		E_p	
				A, B	A, B^T	A, B	A, B^T	A, B	A, B^T
1024	64	1	1	58	46	1.0	1.0	1.00	1.00
1024	64	2	1	14	13	4.1	3.4	2.06	1.68
1024	64	2	2	7.8	9.4	7.5	4.9	1.87	1.22
1024	64	2	4	7.2	8.2	8.1	5.6	1.01	0.70
1024	64	4	4	7.3	6.0	8.0	7.7	0.50	0.48
2048	64	2	2	133	143	1.0	1.0	1.00	1.00
2048	64	4	2	39	59	3.4	2.4	1.69	1.21
2048	64	4	4	30	32	4.4	4.5	1.11	1.12
2048	64	8	4	25	28	5.3	5.1	0.67	0.64
2048	64	8	8	22	20	6.0	7.2	0.38	0.45
4096	64	4	4	281	301	1.0	1.0	1.00	1.00
4096	64	8	4	168	188	1.4	1.6	0.84	0.80
4096	64	8	8	117	97	2.4	3.1	0.60	0.78

Table 2. Performance results of PDGESY solving $AX - XB^T = C$.

$M = N$	MB	P_r	P_l	Time (sec.)	S_p	E_p	#Ext	Abs. residual
1024	64	1	1	696	1.0	1.00	179	$0.10E - 10$
1024	64	2	1	397	1.7	0.85	160	$0.98E - 11$
1024	64	2	2	260	2.7	0.68	160	$0.96E - 11$
1024	64	4	2	183	3.8	0.48	148	$0.89E - 11$
1024	64	4	4	140	5.0	0.31	146	$0.89E - 11$
2048	64	2	2	2057	1.0	1.00	663	$0.27E - 10$
2048	64	4	2	1061	1.9	0.97	664	$0.26E - 10$
2048	64	4	4	553	3.7	0.93	704	$0.26E - 10$
2048	64	4	8	384	5.4	0.67	604	$0.24E - 10$
2048	64	8	8	364	5.7	0.35	663	$0.24E - 10$
4096	64	4	4	5158	1.0	1.00	3400	$0.72E - 10$
4096	64	8	4	2407	2.1	1.10	3376	$0.67E - 10$
4096	64	8	8	1478	3.5	0.87	3360	$0.68E - 10$

that we were able to solve with as few processors as possible. Therefore, the results for the speedup and efficiency must be understood from the context. All timings are performed on random generated problems which typically are pretty ill-conditioned and have large-normed solutions X .

In Table 1, we present performance results for the triangular solver PDTRSY when solving $AX - XB = C$ and $AX - XB^T = C$, and A and B are in upper real Schur form.

In Table 2, we present performance results for the general solver PDGESY when solving $AX - XB^T = C$. Here, the timings include all four steps. Moreover, we display the number of 2×2 diagonal split problems that were involved and the absolute residual of the solutions. The sizes of the residuals are due to the fact that the random problems are rather ill-conditioned, i.e., the separation between

Table 3. Execution time profile of PDGESY solving $AX - XB^T = C$.

$M = N$	MB	P_r	P_c	Step 1 (%)	Steps 2+4 (%)	Step 3 (%)	Total time (sec.)
1024	64	1	1	83	12	4	696
1024	64	2	1	90	6	3	397
1024	64	2	2	92	4	4	260
1024	64	4	2	89	5	4	183
1024	64	4	4	89	2	4	140
2048	64	2	2	81	12	7	2057
2048	64	4	2	85	6	9	1061
2048	64	4	4	90	3	6	553
2048	64	4	8	87	4	7	384
2048	64	8	8	86	3	5	364
4096	64	4	4	75	17	8	5158
4096	64	8	4	79	12	8	2407
4096	64	8	8	89	4	7	1478

A and B are quite small resulting in near to singular systems to solve. Finally, in Table 3, we present the execution profile of the results of Table 2. As expected, it is the transformations to Schur form in Step 1 that dominate the execution time. However, it is still important to have a scalable and efficient solver for the triangular SYCT equations, since in condition estimation we typically have to call PDTRSY several (about five) times [15,11,12].

Our software is designed for integration in state-of-the-art software libraries such as ScaLAPACK [3] and SLICOT [17,6].

Acknowledgements. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

Financial support has been provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

- 1. R.H. Bartels and G.W. Stewart Algorithm 432: Solution of the Equation $AX + XB = C$, *Comm. ACM*, 15(9):820–826.
- 2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov and D. Sorensen. *LAPACK User's Guide*. Third Edition. SIAM Publications, 1999.
- 3. S. Blackford, J. Choi, A. Clearly, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitot, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM Publications, Philadelphia, 1997.

4. K. Dackland and B. Kågström. An Hierarchical Approach for Performance Analysis of ScaLAPACK-based Routines Using the Distributed Linear Algebra Machine. In Wasniewski et al., editors, *Applied Parallel Computing in Industrial Computation and Optimization, PARA96*, Lecture Notes in Computer Science, Springer, Vol. 1184, pages 187–195, 1996.
5. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.
6. E. Elmroth, P. Johansson, B. Kågström, and D. Kreissner. A Web Computing Environment for the SLICOT Library, In P. Van Dooren and S. Van Huffel, *The Third NICONET Workshop on Numerical Control Software*, pp 53–61, 2001.
7. R. Granat, A Parallel ScaLAPACK-style Sylvester Solver, *Master Thesis*, UMNAD 435/03, Dept. Computing Science, Umeå University, Sweden, January, 2003.
8. G. Henry and R. Van de Geijn. Parallelizing the *QR* Algorithm for the Unsymmetric Algebraic Eigenvalue Problem: Myths and Reality. *SIAM J. Sci. Comput.* 17:870–883, 1997.
9. G. Henry, D. Watkins, and J. Dongarra, J. A Parallel Implementation of the Nonsymmetric *QR* Algorithm for Distributed Memory Architectures. Technical Report CS-97-352 and Lapack Working Note 121, University of Tennessee, 1997.
10. N.J. Higham. Perturbation Theory and Backward Error for $AX - XB = C$, *BIT*, 33:124–136, 1993.
11. I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part I: One-Sided and Coupled Sylvester-Type Equations, *ACM Trans. Math. Software*, Vol. 28, No. 4, pp 393–415, 2002.
12. I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part II: Two-Sided and Generalized Sylvester and Lyapunov Equations, *ACM Trans. Math. Software*, Vol. 28, No. 4, pp 416–435, 2002.
13. B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.
14. B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: Portability and optimization issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.
15. B. Kågström and P. Poromaa. Distributed and shared memory block algorithms for the triangular Sylvester equation with Sep^{-1} estimators, *SIAM J. Matrix Anal. Appl.*, 13 (1992), pp. 99–101.
16. P. Poromaa. Parallel Algorithms for Triangular Sylvester Equations: Design, Scheduling and Scalability Issues. In Kågström et al. (eds), *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, Vol. 1541, pp 438–446, Springer-Verlag, 1998.
17. SLICOT library in the Numerics in Control Network (NICONET) website: www.win.tue.nl/niconet/index.html

RECSY — A High Performance Library for Sylvester-Type Matrix Equations

Isak Jonsson and Bo Kågström

Department of Computing Science and HPC2N, Umeå University,
SE-901 87 Umeå, Sweden.
{isak,bokg}@cs.umu.se

Abstract. RECSY is a library for solving triangular Sylvester-type matrix equations. Its objectives are both speed and reliability. In order to achieve these goals, RECSY is based on novel recursive blocked algorithms, which call high-performance kernels for solving small-sized leaf problems of the recursion tree. In contrast to explicit standard blocking techniques, our recursive approach leads to an automatic variable blocking that has the potential of matching the memory hierarchies of today's HPC systems. The RECSY library comprises a set of Fortran 90 routines, which uses recursion and OpenMP for shared memory parallelism to solve eight different matrix equations, including continuous-time as well as discrete-time standard and generalized Sylvester and Lyapunov equations. Uniprocessor and SMP parallel performance results of our recursive blocked algorithms and corresponding routines in state-of-the-art libraries LAPACK and SLICOT are presented. The performance improvements of our recursive algorithms are remarkable, including 10-fold speedups compared to standard algorithms.

Keywords: Sylvester-type matrix equations, recursion, automatic blocking, superscalar, GEMM-based, level 3 BLAS, LAPACK, SLICOT, RECSY

1 Introduction

In [8,9], we describe recursive blocked algorithms for solving different Sylvester-type matrix equations. We differentiate between one-sided and two-sided matrix equations. The notation one-sided matrix equations is used when the solution is only involved in matrix products of two matrices, e.g., $\text{op}(A)X$ or $X\text{op}(A)$, where $\text{op}(A)$ can be A or A^T . In two-sided matrix equations, the solution is involved in matrix products of three matrices, both to the left and to the right, e.g., $\text{op}(A)X\text{op}(B)$. Table 1 lists eight different types of matrix equations considered together with the acronyms used (one-sided equations in the top and two-sided equations in the bottom part).

Table 1. One-sided (top) and two-sided (bottom) matrix equations.

Name	Matrix equation	Acronym
Standard Sylvester (CT)	$AX - XB = C$	SYCT
Standard Lyapunov (CT)	$AX + XA^T = C$	LYCT
Generalized Coupled Sylvester	$(AX - YB, DX - YE) = (C, F)$	GCSY
Standard Sylvester (DT)	$AXB^T - X = C$	SYDT
Standard Lyapunov (DT)	$AXA^T - X = C$	LYDT
Generalized Sylvester	$AXB^T - CXD^T = E$	GSYL
Generalized Lyapunov (CT)	$AXE^T + EXA^T = C$	GLYCT
Generalized Lyapunov (DT)	$AXA^T - EXE^T = C$	GLYDT

1.1 Triangular Matrix Equations

The classical method of solution of the Sylvester-type matrix equations is based on the Bartels–Stewart method [2], which includes three major steps. First, the matrix (or matrix pair) is transformed to a Schur (or generalized Schur) form. This leads to a reduced triangular matrix equation. For example, the coefficient matrices A and B in the Sylvester equation $AX - XB = C$ are in upper triangular or upper quasi-triangular form. Finally, the solution of the reduced matrix equation is transformed back to the originally coordinate system.

Reliable and efficient algorithms for the reduction step can be found in LAPACK [1] for the standard case, and in [3] for the generalized case, where a blocked variant of the QZ method is presented.

Triangular matrix equations also appear naturally in estimating the condition numbers of matrix equations and different eigenspace computations, including block-diagonalization of matrices and matrix pairs and computation of functions of matrices. Related applications include the direct reordering of eigenvalues in the real (generalized) Schur form and the computation of additive decompositions of a (generalized) transfer function (see [8,9] for more information and references).

1.2 Motivation

Our goal is to produce a state-of-the-art library which solves the triangular Sylvester-type matrix equations listed in Table 1. The library should be easy to use, provide excellent performance, and it should be expandable and tunable for new routines and different platforms.

In this contribution, we present a new library, RECSY, which contains sequential and parallel implementations of all triangular matrix equations listed in Table 1. Also, the routines feature several different transpose and sign options, so all in all the RECSY routines are able to solve 42 different variants.

Before we go into details of the library, we outline the rest of the paper. In Section 2, we describe the routines for solving one-sided triangular equations. We do this by reviewing our recursive approach, and how it gives better performance. Furthermore, we describe the fast and highly optimized kernels used

Algorithm 1: SYCT function $[X] = \text{recsyct}(A, B, C, \text{uplo}, \text{blks})$

```

if  $1 \leq M, N \leq 4$  then
     $X = \text{trsytct}(A, B, C, \text{uplo})$ ;
else
    if  $1 \leq N \leq M/2$  % Case 1: Split  $A$  (by rows and columns),  $C$  (by rows only)
         $X_2 = \text{recsyct}(A_{22}, B, C_2, 1, \text{blks})$ ;
         $C_1 = \text{gemm}(-A_{12}, X_2, C_1)$ ;
         $X_1 = \text{recsyct}(A_{11}, B, C_1, 1, \text{blks})$ ;
         $X = [X_1; X_2]$ ;
    elseif  $1 \leq M \leq N/2$  % Case 2: Split  $B$  (by rows and columns),  $C$  (by columns only)
         $X_1 = \text{recsyct}(A, B_{11}, C_1, 1, \text{blks})$ ;
         $C_2 = \text{gemm}(X_1, B_{12}, C_2)$ ;
         $X_2 = \text{recsyct}(A, B_{22}, C_2, 1, \text{blks})$ ;
         $X = [X_1, X_2]$ ;
    else %  $M, N \geq \text{blks}$ , Case 3: Split  $A, B$  and  $C$  (all by rows and columns)
         $X_{21} = \text{recsyct}(A_{22}, B_{11}, C_{21}, 1, \text{blks})$ ;
         $C_{22} = \text{gemm}(X_{21}, B_{12}, C_{22})$ ;  $C_{11} = \text{gemm}(-A_{12}, X_{21}, C_{11})$ ;
         $X_{22} = \text{recsyct}(A_{22}, B_{22}, C_{22}, 1, \text{blks})$ ;  $X_{11} = \text{recsyct}(A_{11}, B_{11}, C_{11}, 1, \text{blks})$ ;
         $C_{12} = \text{gemm}(-A_{12}, X_{22}, C_{12})$ ;
         $C_{12} = \text{gemm}(X_{11}, B_{12}, C_{12})$ ;
         $X_{12} = \text{recsyct}(A_{11}, B_{22}, C_{12}, 1, \text{blks})$ ;
         $X = [X_{11}, X_{12}; X_{21}, X_{22}]$ ;
    end
end

```

Algorithm 1: Recursive blocked algorithm for solving the triangular continuous-time Sylvester equation (SYCT).

to solve small problems. We also show how we easily provide parallel routines for SMP/OpenMP-aware systems. The routines for solving two-sided triangular equations are listed in Section 3. The differences between the one-sided and the two-sided equations are explained, and we show how the library takes care of these issues. In Section 4, we give a short guide on how to use the library. Some performance results are given in Section 5.

2 One-Sided Triangular Matrix Equations

In [6], we present pseudo-code algorithms for three one-sided triangular matrix equations. These are the continuous-time standard Sylvester (SYCT), the standard Lyapunov (LYCT), and the generalized coupled Sylvester (GCSY) equations, together with the mathematical derivation of the algorithms. For reference, we show a compressed version of one of the algorithms, SYCT, in Algorithm 1. The equation is solved by recursion, where the problem is split into two or four new, smaller, problems of approximately the same size. The aim is to split the matrix exactly as close to the middle as possible. If the splitting point appears at a 2×2 diagonal block, the matrices are split just below the splitting point.

2.1 Performance Due to Recursive Level 3 Blocking and Updates

The recursive approach provides automatic cache blocking through great temporal locality. Furthermore, it reveals level 3 BLAS updates, shown as **gemm** calls

in Algorithm 1. The same observations hold for the LYCT and GCSY equations. For LYCT, the updates are done by symmetric rank- k and symmetric rank- $2k$ operations instead, due to its symmetry characteristics. However, most of the work in LYCT is done in SYCT, as LYCT uses SYCT to solve off-diagonal blocks in the solution X .

It has been observed that for some platforms, the vendor's DGEMM does not perform very well for small matrices. This is due to overhead from parameter checking and temporary buffer setup. Therefore, the library supplies its own matrix-matrix multiply which uses 4×4 outer loop unrolling, but without blocking for memory cache. It shows good performance on different types of processors. This routine is used instead of the regular DGEMM when the problem fits into level 1 cache. Note that recursion itself provides good temporal locality.

2.2 Performance Due to Fast Kernels

At the bottom of the recursion tree for the Sylvester equations (SYCT, GCSY), where the problem size is very small ($\leq 4 \times 4$), the recursion is terminated and the problem is instead solved by a very fast kernel. Now, the small matrix equations are represented as a linear system of equations $Zx = y$, where Z is a Kronecker product representation of the Sylvester-type operator, and x and the right hand side y are vectors representing the solution and the the right hands side(s). For example, $Z_{\text{SYCT}} = I_n \otimes A - B^T \otimes I_m$ (see also [8,9]). This kernel features unrolling for the updates, and uses partial pivoting for solving small matrix equations. These equations lead to Kronecker product matrix representations Z of size $1 \times 1 - 4 \times 4$ for SYCT and $2 \times 2 - 8 \times 8$ for GCSY. If a good pivot candidate cannot be found (problem is nearly singular), or if the solution procedure is close to produce an overflow, the fast kernel is aborted. On abortion, the recursive procedure backtracks and we construct a new Kronecker product representation, now leading to a matrix Z of size 16×16 for SYCT and 32×32 for GCSY. The system is then solved using complete pivoting with perturbed pivot elements if necessary and right hand scaling to avoid overflow.

For LYCT, the recursion continues until subproblems of size 1×1 or 2×2 . Such a problem, with Kronecker product matrix Z of size 1×1 or 3×3 , is solved using the fast kernel solver (partial pivoting), or with complete pivoting for very ill-conditioned cases. The reason for not optimizing LYCT as much as SYCT and GCSY is that most of the solving takes part in SYCT. The number of small problems solved in LYCT by the Lyapunov small problem solver is $O(n)$, whereas the number of problems solved in LYCT by the SYCT solver is $O(n^2)$.

2.3 Performance Due to Parallelism

First of all, it is possible to get a fair amount of speedup on systems using only SMP versions of level 3 BLAS routines. This is due to fact that the recursive algorithms call the level 3 BLAS routines with large, squarish blocks. The routines can thus do each update in parallel with good efficiency.

As can be observed in Case 3 of Algorithm 1, there are two recursive calls that can be executed simultaneously. This is used in our implementation of SYCT and GCSY to create OpenMP versions of our algorithms for SMP systems, where not only the updates, but also the solves are done in parallel. The OpenMP versions take an additional argument, PROC, the number of processors available. Whenever $\text{PROC} > 1$, and the problem is a Case 3 problem, the second and third recursive calls (and corresponding updates) are done in parallel, giving a diamond-shaped execution graph. For more details, see [6]. For LYCT, there is no explicit parallelism in the Lyapunov solver. However, as most of the work takes place in the Sylvester solver SYCT, the LYCT routine gains performance from the OpenMP parallelism as well.

In order for this scheme to work well, two conditions must hold. First, there should not be any degradation in the total performance when the SMP BLAS 3 routines are called simultaneously (multiple parallelism). This has been a problem on a small number of machines, where OpenMP and SMP BLAS did not work together. Second, if there are more than two processors in the system, the OpenMP compiler must support nested parallelism in order to fully utilize the system, which most modern OpenMP compilers do.

The overhead of SMP BLAS routines can be pretty large, especially when solving too small problems. Fortunately, the library’s own matrix-matrix multiply routine alleviates this problem. As it is not parallelized, it is perfectly suited for small problems, where the parallel divide-and-conquer splitting already has taken place at higher levels of the recursion tree.

2.4 Fortran Interfaces

The routines are implemented in Fortran 90, which greatly simplifies the recursive calls. Also, this makes the routines’ signatures easier, as some arguments can be declared optional. Below, the six routines for solving one-sided triangular matrix equations are listed.

- RECSYCT (UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, INFO, MACHINE)
 - RECSYCT.P (PROCS, UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, INFO, MACHINE)
- Solves the triangular continuous-time Sylvester equation. A and B are upper quasitriangular of size $M \times M$ and $N \times N$, C is rectangular $M \times N$.

UPLOSIGN	SYCT equation
0	$AX + XB = \text{scale } C, C \leftarrow X$
1	$AX - XB = \text{scale } C, C \leftarrow X$
2	$AX + XB^T = \text{scale } C, C \leftarrow X$
3	$AX - XB^T = \text{scale } C, C \leftarrow X$
4	$A^T X + XB = \text{scale } C, C \leftarrow X$
5	$A^T X - XB = \text{scale } C, C \leftarrow X$
6	$A^T X + XB^T = \text{scale } C, C \leftarrow X$
7	$A^T X - XB^T = \text{scale } C, C \leftarrow X$

- RECLYCT (UPLO, SCALE, M, A, LDA, C, LDC, INFO, MACHINE)

- RECLYCT_P (PROCS, UPLO, SCALE, M, A, LDA, C, LDC, INFO, MACHINE)

Solves the symmetric triangular continuous-time Lyapunov equation. A is upper quasitriangular $M \times M$, C is symmetric $M \times M$, only upper part of C is accessed.

UPLOSIGN	LYCT equation
0	$AX + XA^T = \text{scale } C, C \leftarrow X$
1	$A^T X + XA = \text{scale } C, C \leftarrow X$

- RECGCSY (UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, D, LDD, E, LDE, F, LDF, INFO, MACHINE)
- RECGCSY_P (PROCS, UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, D, LDD, E, LDE, F, LDF, INFO, MACHINE)

Solves the generalized coupled triangular Sylvester equation. A is upper quasitriangular $M \times M$, D is upper triangular $M \times M$, B is upper quasitriangular $N \times N$, E is upper triangular $N \times N$, C and F are rectangular $M \times N$.

UPLOSIGN	GCSY equation
0	$(AX + YB = \text{scale } C, DX + YE = \text{scale } F), (C, F) \leftarrow (X, Y)$
1	$(AX - YB = \text{scale } C, DX - YE = \text{scale } F), (C, F) \leftarrow (X, Y)$
2	$(AX + YB^T = \text{scale } C, DX + YE^T = \text{scale } F), (C, F) \leftarrow (X, Y)$
3	$(AX - YB^T = \text{scale } C, DX - YE^T = \text{scale } F), (C, F) \leftarrow (X, Y)$
4	$(A^T X + YB = \text{scale } C, D^T X + YE = \text{scale } F), (C, F) \leftarrow (X, Y)$
5	$(A^T X - YB = \text{scale } C, D^T X - YE = \text{scale } F), (C, F) \leftarrow (X, Y)$
6	$(A^T X + YB^T = \text{scale } C, D^T X + YE^T = \text{scale } F), (C, F) \leftarrow (X, Y)$
7	$(A^T X - YB^T = \text{scale } C, D^T X - YE^T = \text{scale } F), (C, F) \leftarrow (X, Y)$

3 Two-Sided Triangular Matrix Equations

The recursive templates for solving two-sided triangular matrix equations are similar to the templates for solving one-sided triangular matrix equations. However, the two-sidedness of the matrix updates leads to several different implementation choices. In the RECSY library, two internal routines are used: RECSY_MULT_LEFT and RECSY_MULT_RIGHT. These routines compute $C = \beta C + \alpha \text{op}(A)X$ and $C = \beta C + \alpha X \text{op}(A)$, respectively, where $\text{op}(A)$ is A or A^T , and A is rectangular (DGEMM), upper triangular, upper quasitriangular, upper trapezoidal, or upper quasitrapezoidal (i.e., trapezoidal with 1×1 and 2×2 diagonal blocks). C and X are rectangular matrices. The actual computation is carried out by calls to DGEMM, DTRMM, the internal small matrix-matrix multiply, and level 1 BLAS routines for subdiagonal elements. Note that these routines require extra workspace when DTRMM is used, as DTRMM only computes the product ($C = \text{op}(A)X$) and not the multiply-and-add ($C = C + \text{op}(A)X$).

Combining RECSY_MULT_LEFT and RECSY_MULT_RIGHT give the routine RECSY_AXB. It computes the update $C = \beta C + \alpha \text{op}(A)X \text{op}(B)$, where A and B can independently be any of the five matrix structures listed above. There are two ways of computing the update, either by doing the left multiplication first or by doing the right multiplication first. RECSY_AXB chooses the ordering which uses the least amount of floating point operations. RECSY_AXB also requires workspace to store the intermediate product.

However, for symmetric two-sided updates $C = \beta C + \alpha \text{op}(A)X\text{op}(A^T)$, where C and X are symmetric, the SLICOT [12] routine MB01RD is used. MB01RD uses a series of level 3 updates to calculate C without requiring workspace and with fewer floating point operations than RECSY_AXB.

3.1 Fortran Interfaces

As shown above, the two-sided matrix equation solvers require extra workspace. As Fortran 90 allows dynamic memory allocation, this facilitates the subroutine calls. The user now has the option of either supplying workspace to the routine, or, preferably, leave this task to the subroutine itself. Below, the ten routines for solving two-sided triangular matrix equations are shown.

- RECSYDT (UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E
 - RECSYDT.P (PROCS, UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E
- Solves the triangular discrete-time Sylvester equation. A is upper quasitriangular $M \times M$, B is upper quasitriangular $N \times N$, C is rectangular $M \times N$.

UPLOSIGN	SYDT equation
0	$AXB + X = \text{scale } C, C \leftarrow X$
1	$AXB - X = \text{scale } C, C \leftarrow X$
2	$AXB^T + X = \text{scale } C, C \leftarrow X$
3	$AXB^T - X = \text{scale } C, C \leftarrow X$
4	$A^T X B + X = \text{scale } C, C \leftarrow X$
5	$A^T X B - X = \text{scale } C, C \leftarrow X$
6	$A^T X B^T + X = \text{scale } C, C \leftarrow X$
7	$A^T X B^T - X = \text{scale } C, C \leftarrow X$

- RECLYDT (UPLO, SCALE, M, A, LDA, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E
 - RECLYDT.P (PROCS, UPLO, SCALE, M, A, LDA, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E
- Solves the symmetric triangular discrete-time Lyapunov equation. A is upper quasitriangular $M \times M$, C is symmetric $M \times M$, only upper part of C is accessed.

UPLOSIGN	LYDT equation
0	$AXA^T - X = \text{scale } C, C \leftarrow X$
1	$A^T X A - X = \text{scale } C, C \leftarrow X$

- RECGSYL (UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, D, LDD, E, LDE, INFO, MACHINE, WORKSPACE, WKSIZ)E
 - RECGSYL.P (UPLOSIGN, SCALE, M, N, A, LDA, B, LDB, C, LDC, D, LDD, E, LDE, INFO, MACHINE, WORKSPACE, WKSIZ)E
- Solves the triangular generalized Sylvester equation. A is upper quasitriangular $M \times M$, C is upper triangular $M \times M$, B is upper quasitriangular $N \times N$, D is upper triangular $N \times N$, C and F are rectangular $M \times N$.

UPLOSIGN	GSYL equation
0	$AXB + CXD = \text{scale } E, E \leftarrow X$
1	$AXB - CXD = \text{scale } E, E \leftarrow X$
2	$AXB^T + CXD^T = \text{scale } E, E \leftarrow X$
3	$AXB^T - CXD^T = \text{scale } E, E \leftarrow X$
4	$A^T XB + C^T XD = \text{scale } E, E \leftarrow X$
5	$A^T XB - C^T XD = \text{scale } E, E \leftarrow X$
6	$A^T XB^T + C^T XD^T = \text{scale } E, E \leftarrow X$
7	$A^T XB^T - C^T XD^T = \text{scale } E, E \leftarrow X$
10	$AXB^T + CXD^T = \text{scale } E, E \leftarrow X, B \text{ is upper triangular } N \times N, D \text{ is upper quasitriangular } N \times N \text{ (used by RECGLYCT)}$
12	$A^T XB + C^T XD = \text{scale } E, E \leftarrow X, B \text{ is upper triangular } N \times N, D \text{ is upper quasitriangular } N \times N \text{ (used by RECGLYCT)}$

- RECGLYDT (UPLO, SCALE, M, A, LDA, E, LDE, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E)
- RECGLYDT_P (PROCS, UPLO, SCALE, M, A, LDA, E, LDE, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E)
- RECGLYCT (UPLO, SCALE, M, A, LDA, E, LDE, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E)
- RECGLYCT_P (PROCS, UPLO, SCALE, M, A, LDA, E, LDE, C, LDC, INFO, MACHINE, WORKSPACE, WKSIZ)E)

Solves the triangular generalized discrete-time and continuous-time Lyapunov equation, respectively. A is upper quasitriangular $M \times M$, E is upper triangular $M \times M$, C is symmetric $M \times M$, only upper part of C is accessed.

UPLOSIGN	GLYDT equation	GLYCT equation
0	$AXA^T - EXE^T = \text{scale } C, C \leftarrow X$	$AXE^T + EXA^T = \text{scale } C, C \leftarrow X$
1	$A^T XA - E^T XE = \text{scale } C, C \leftarrow X$	$A^T XE - E^T XA = \text{scale } C, C \leftarrow X$

4 Library Usage

The library, together with building instructions, can be downloaded from <http://www.cs.umu.se/~isak/recsy>. The library is easily built with any Fortran 90 compiler, both UNIX and Windows compilers are supported. By using the routines listed in Sections 2 and 3, the user gets access to all of the different options and tuning parameters of the library. The types of the parameters to the routines are listed in Table 2. The parameter MACHINE, which can be omitted – as the authors recommend, sets different tuning parameters. For details, see the library homepage.

4.1 SLICOT/LAPACK Wrapper Routines

Instead of using the native routines listed, the library also provides wrapper routines which overload SLICOT [12] and LAPACK [1] routines, see Table 3. By linking with the library, calls to SLICOT and LAPACK Sylvester-type matrix equations solvers will be replaced to a call to the optimized RECSY equivalent. We remark that neither LAPACK nor SLICOT provide a GSYL solver.

Note that not only the SLICOT and LAPACK routines listed above are affected. Any routine which uses any of these routines will also be using the RECSY library. In particular, this is true for the SLICOT routines which solve

Table 2. The parameters to the RECSY library routines, their type and meaning.

Parameter	Type	Meaning
UPLOSIGN, UPLO	INTEGER	Defines different options (see above).
A, B, C, D, E, F	DOUBLE PRECISION(*)	Coefficient/right hand side matrices (see above).
SCALE	DOUBLE PRECISION	Scaling factor (output argument), to avoid overflow in the computed solution.
LDA, LDB, LDC, LDD, LDE, LDF	INTEGER	Leading dimension of matrices A to F.
INFO	INTEGER	Output parameter. INFO= -100: Out of memory, INFO= 0: No error, INFO> 0: The equation is (nearly) singular to working precision; perturbed values were used to solve the equation.
MACHINE	DOUBLE PRECISION(0:9)	Optional. Specify only if you want different parameters than the default parameters, which are set by RECSY_MACHINE.
WORKSPACE	DOUBLE PRECISION(*)	Optional. Specify only if you provide your own workspace.
WKSIZE	INTEGER	Optional. Specify only if you provide your own workspace.

Table 3. SLICOT and LAPACK routines and their RECSY equivalents.

Routine	Native routine
SLICOT SB03MX	RECLYDT
SLICOT SB03MY	RECLYCT
SLICOT SB04PY	RECSYDT
SLICOT SG03AX	RECGLYDT
SLICOT SG03AY	RECGLYCT
LAPACK DTRSYL	RECSYCT
LAPACK DTGSYL [10]	RECSCSY

Table 4. a) Performance results for the triangular Lyapunov equation—IBM Power3, 200 MHz (left) and SGI Onyx2 MIPS R10000, 195 MHz (right). b) Performance results for the triangular discrete-time Sylvester equation—IBM Power3, 4×375 MHz (left) and Intel Pentium III, 2×550 MHz (right).

$(M=N)$	a) $AX + XA^T = C$							b) $AXB - X = C$						
	IBM Power3			MIPS R10000				IBM Power3			Intel Pentium III			
	Mflops/s		Speedup	Mflops/s		Speedup		Time (sec)		Speedup	Time (sec)		Speedup	
	A	B	B/A	A	B	B/A		C	D	D/C E/D	C	D	D/C E/D	
100	77.0	166.5	2.16	82.0	123.5	1.51	1.73e-2	7.41e-3	2.33	1.16	2.20e-2	2.20e-2	1.00	1.22
250	85.3	344.5	4.04	88.7	224.5	2.53	6.20e-1	6.93e-2	8.95	0.98	6.83e-1	2.31e-1	2.96	1.33
500	10.6	465.0	43.85	42.2	277.8	6.58	2.32e+1	4.60e-1	50.50	1.48	7.82e+0	1.56e+0	5.00	1.35
1000	7.7	554.7	72.20	14.5	254.0	17.57	2.44e+2	3.26e+0	74.65	1.94	7.04e+1	1.10e+1	6.40	1.48
1500	7.0	580.5	83.19	9.7	251.0	25.81	9.37e+2	1.08e+1	86.66	2.04	2.78e+2	3.58e+1	7.76	1.53
2000							2.84e+3	2.41e+1	117.71	2.14	9.07e+2	8.03e+1	11.29	1.57
	A – SLICOT SB03MY B – RECLYCT							C – SLICOT SB04PY D – RECSYDT E – RECSYDT.P						

unreduced problems, e.g., SB04MD (SYCT solver) or SM03MD (LYDT/LYCT solver). SB04MD and SB03MD call DTRSYL and SB03MX/SB03MY, respectively, for solving the reduced equation. With the RECSY library, performance for SB04MD and SB03MD are also improved [7,9].

5 Performance Results

We list a few results obtained with the RECSY library. In Table 4 a), results obtained for the continuous-time Lyapunov equation (LYCT) are shown. Here, the speedup is remarkable. The RECSY library is up to 83 times faster than the original SLICOT library. This is both due to faster kernels and multi-level blocking from recursion. The same results can be observed for LAPACK routines.

For example, the RECSY routine RECSYCT is more than 20 times faster than the LAPACK routine DTRSYL for $M = N > 500$ on the IBM PowerPC 604e.

Timings for two-sided matrix equation examples and given in Table 4 b). For the largest example, the SLICOT library requires more than 45 minutes to solve the problem. The RECSY library solves the same problem in less than a half minute. The extra speedup from the OpenMP version of the library is also given. For further results, we refer to [8,9] and the library homepage.

Acknowledgements. Financial support for this project has been provided by the *Swedish Research Council* under grants TFR 98-604, VR 621-2001-3284 and the *Swedish Foundation for Strategic Research* under grant A3 02:128.

References

1. E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen. *LAPACK Users Guide, Third Edition*. SIAM Publications, 1999.
2. R.H. Bartels and G.W. Stewart. Algorithm 432: Solution of the Equation $AX + XB = C$, *Comm. ACM*, 15(9):820–826, 1972.
3. K. Dackland and B. Kågström. Blocked Algorithms and Software for Reduction of a Regular Matrix Pair to Generalized Schur Form. *ACM Trans. Math. Software*, 24(4):425–454, December 1999.
4. J.D. Gardiner, A.J. Laub, J.J. Amato, and C.B. Moler. Solution of the Sylvester Matrix Equation $AXB^T + CXD^T = E$, *ACM Trans. Math. Software*, 18(2):223–231, June 1992.
5. J.D. Gardiner, M.R. Wette, A.J. Laub, J.J. Amato, and C.B. Moler. A Fortran 77 Software Package for Solving the Sylvester Matrix Equation $AXB^T + CXD^T = E$, *ACM Trans. Math. Software*, 18(2):232–238, June 1992.
6. I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part I: One-Sided and Coupled Sylvester-Type Equations, *SLICOT Working Note* 2001-4.
7. I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part II: Two-Sided and Generalized Sylvester and Lyapunov Equations, *SLICOT Working Note* 2001-5.
8. I. Jonsson, and B. Kågström. Recursive blocked algorithms for solving triangular systems — Part I: One-sided and coupled Sylvester-type matrix equations. *ACM Trans. Math. Softw.*, 28(4):392–415, December 2002.
9. I. Jonsson, and B. Kågström. Recursive blocked algorithms for solving triangular systems — Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations *ACM Trans. Math. Softw.*, 28(4):416–435, December 2002.
10. B. Kågström and P. Poromaa. LAPACK-Style Algorithms and Software for Solving the Generalized Sylvester Equation and Estimating the Separation between Regular Matrix Pairs. *ACM Trans. Math. Software*, 22(1):78–103, March 1996.
11. B. Kågström and L. Westin. Generalized Schur methods with condition estimators for solving the generalized Sylvester equation. *IEEE Trans. Autom. Contr.*, 34(7):745–751, July 1989.
12. SLICOT library and the Numerics in Control Network (NICONET) website: www.win.tue.nl/niconet/index.html

Two Level Parallelism in a Stream-Function Model for Global Ocean Circulation

Martin van Gijzen

CERFACS,
42, avenue Gaspard Coriolis,
31057 Toulouse CEDEX 1, France
gijzen@cerfacs.fr
<http://www.cerfacs.fr/algor>

Abstract. In this paper, we show how a stream-function model for global ocean circulation can be parallelised, and in particular how the continent boundary conditions can be treated in a parallel setting. Our iterative solution technique for the linear system that results after discretisation, combines loop-level parallelism with a domain-decomposition approach. This two-level parallelism can be exploited in a natural way on a cluster of SMP's.

1 Introduction

It is customary (e.g. [1]) to split the 3D-ocean flow into a depth-averaged barotropic part, and the 3D-baroclinic deviations from it. The 2D-barotropic part is often formulated in terms of a stream function, which simplifies the governing partial differential equations. The disadvantage of this formulation is that the boundary conditions on continents and islands are more complicated. In [10] a Finite-Element model is described that incorporates these boundary conditions in a systematic and natural way.

The stream-function model described in [10] can be vectorised by exploiting the regularity of the grid, see [9]. A similar loop-level parallelisation approach can be used on so called Symmetric Multi Processors (SMP's) where all processors share the same memory space. For computers with a disjoint memory space, however, one has to use a coarser-grain parallelism, which is usually done by a domain-decomposition approach, as in [8].

In recent years, new computer architectures have appeared that combine disjoint memory address space between groups of processors and a global memory address space within each group of processors. This kind of computer is usually called a "Cluster of SMPs". The organisation of the memory of a cluster of SMP's perfectly matches the requirements of parallel algorithms that can exploit two levels of parallelism. The outer/coarser level is implemented between the SMPs and the inner/finer within each SMP. The corresponding parallel programming paradigms are message passing at the coarser level and loop-level parallelism at the finer. This two-level parallelism has received considerable attention lately, see [4] and its references.

In this paper we show how a two-level parallelisation method can be implemented in a (simplified) stream function model by combining loop-level parallelisation with a domain decomposition. Special attention will be given to the treatment of the continent boundary conditions, which are literally global by nature and therefore complicate parallelisation. The parallel performance of the resulting method is examined by means of numerical experiments on a cluster of bi-processor PC's.

2 A Simple Ocean Circulation Model

2.1 Analytical Formulation

Steady barotropic flow in a homogeneous ocean with constant depth, zero lateral viscosity and in nearly equilibrium can be described by the following partial differential equation:

$$-r\nabla^2\psi - \beta\frac{\partial\psi}{\partial x} = \nabla \times \mathbf{F} \quad \text{in } \Omega, \quad (1)$$

in which ψ is the stream function and Ω the part of the world that is covered by sea. The external force field $\mathbf{F}$ is equal to the wind stress τ divided by the average depth of the ocean H times the water density ρ . The other parameters in this equation are the bottom friction r and the Coriolis parameter β .

The no-normal-flow boundary condition implies that the stream function is constant on each continent,

$$\psi = C_k \quad \text{on } \Gamma_k, \quad k = 1, \dots, n_k, \quad (2)$$

with n_k the number of continents. Additional conditions to determine these unknown constants are obtained by requiring continuity of the water-level around each continent boundary Γ_k . This yields for each continent an integral condition:

$$\oint_{\Gamma_k} r \frac{\partial\psi}{\partial n} ds = - \oint_{\Gamma_k} \mathbf{F} \cdot \mathbf{s} ds. \quad (3)$$

The equations are commonly expressed in spherical coordinates, which maps the physical domain onto a rectangular domain with periodic boundary conditions

$$\psi(-180^\circ, \theta) = \psi(+180^\circ, \theta). \quad (4)$$

Here θ is the latitude. The coordinate transformation introduces a singularity on the North pole (the South pole is land). This singularity is solved by imposing the additional boundary condition $\psi = 0$ if $\theta = 90^\circ$.

2.2 Discretisation

The main problem of discretising (1-4) is posed by the boundary conditions for the continents. As explained in [10], this problem can be solved in a natural way

within the context of the Finite Element Method. The procedure is as follows:
Step 1: Define a structured grid on the rectangular domain, disregarding islands and continents.

Step 2: Discretise (1) everywhere, again disregarding islands and continents. This yields a right-hand-side vector $\mathbf{f}$ and a structured matrix $\mathbf{K}$ with only a few diagonals with nonzero elements. Discretisation with the Finite Element Method allows us to satisfy the integral conditions in a natural way.

Step 3: Add together all the rows in $\mathbf{f}$ and the rows and columns in $\mathbf{K}$ that correspond to the same island or continent. We denote the resulting right-hand-side vector and system matrix by $\bar{\mathbf{f}}$ and $\bar{\mathbf{K}}$, respectively. Note that $\bar{\mathbf{K}}$ is *not* structured.

The above procedure yields the following system of linear equations:

$$\bar{\mathbf{K}}\bar{\mathbf{x}} = \bar{\mathbf{f}}. \quad (5)$$

In this equation, $\mathbf{x}$ contains the values of the stream function in the sea gridpoints and the n_k values of the stream function C_k (see equation (2)) on the continents.

The parallelisation of the discretisation of (1-4) is straightforward. All the element matrices and vectors can be computed in parallel. Moreover, the time for the discretisation is almost negligible (less than 10 %) compared to the time it takes to solve (5). The parallel solution of the linear system of equations is the subject of the next Section.

3 The Parallel Solution of the Linear System of Equations

3.1 Operations with the System Matrix

The diagonal structure of the matrix $\mathbf{K}$ is ruined by the inclusion of the conditions for the continents (Step 3 in the discretisation). Operations with the resulting unstructured matrix require indirect addressing, which may be inefficient on sequential computers and may also hamper parallelisation.

To preserve as much as possible the structure in operations with $\bar{\mathbf{K}}$ we define the matrix $\mathbf{P}$ that maps the structured-grid variables to the set that comprises the sea-point variables and one variable per continent only. This matrix $\mathbf{P}$ is defined as follows:

- The column of $\mathbf{P}$ that corresponds to a sea-point j is just the corresponding j -th basis-vector:

$$\mathbf{P}_j = \mathbf{e}_j \quad \forall j \text{ in } \Omega. \quad (6)$$

- And a column that corresponds to a continent k is the sum of all the basis vectors corresponding to the gridpoints j on this continent (including the interior):

$$\mathbf{P}_k = \sum_{j \in \Gamma_k} \mathbf{e}_j. \quad (7)$$

System (5) can now be written as

$$\mathbf{P}^T \bar{\mathbf{K}} \mathbf{P} \bar{\mathbf{x}} = \mathbf{P}^T \bar{\mathbf{f}}. \quad (8)$$

The above form is of no advantage if a direct solution method is used since these methods make a decomposition of $\bar{\mathbf{K}}$ in triangular factors. However, for iterative solution methods there is an advantage since the only operation with $\bar{\mathbf{K}}$ is the matrix-vector multiplication. The operation $\bar{\mathbf{v}} = \mathbf{P}^T \mathbf{K} \mathbf{P} \bar{\mathbf{u}}$ can be performed in three steps:

- 1 Give all the land nodes their continent value, sea nodal values remain unchanged: $\mathbf{u} = \mathbf{P} \bar{\mathbf{u}}$
- 2 Multiply: $\mathbf{v} = \mathbf{K} \mathbf{u}$ (Structured matrix-vector product).
- 3 Add up the values on each continent: $\bar{\mathbf{v}} = \mathbf{P}^T \mathbf{v}$.

The main part of this operation is the structured matrix-vector multiplication, which can be performed diagonalwise, without indirect addressing.

3.2 Iterative Solution Method

To fully exploit the diagonal structure of $\mathbf{K}$ we solve (8) with an iterative solution method.

Our iterative method of choice is GMRESR [7]. GMRESR is a nested Krylov subspace method which uses GCR [3] for the outer iterations and GMRES [6] for the inner iterations. Both GCR and GMRES explicitly store a set of basis vectors for the Krylov subspace (GCR even two). Nesting the methods has the advantage that the number of outer iterations can be reduced so that a smaller number of basis vectors has to be computed and stored. This normally outweighs the cost (both in storage and in computations) of the inner iterative process. In practice the number of outer GCR iterations can still be too big to store all basis vectors. The usual remedy is to truncate the process after k iterations, so that only the last k iterations are taken into account when the solution is updated.

The main building blocks of GMRESR are: matrix-vector products, preconditioning operations, vector updates and inner products. For the preconditioning operation we simply use diagonal scaling, i.e. multiplication with the inverse of the main diagonal of $\bar{\mathbf{K}}$. However, note that the GMRES-inner iterations can be seen as a preconditioning method for the outer-iterative method GCR.

Multiplication with a diagonal matrix is an inexpensive and easy to parallelise operation. In the remainder of this Section we will therefore concentrate on the parallelisation of the other three operations: the matrix-vector product, the vector update, and the inner product.

3.3 Parallelisation

Fine-Grain Parallelism

A fine grain parallel implementation of the inner product and vector update can be made in a straightforward and portable way by adding appropriate OpenMP-directives [2] around the respective loops.

The first step in the matrix-vector multiplication, giving all the gridpoints on the continents their continent value, can be performed in parallel for the

different continents. However, this operation is too inexpensive to be parallelised. The overhead of the parallelisation is bigger than the gain. For this reason the operation is performed sequentially. The second step in the matrix vector-multiplication, the diagonalwise multiplication with $\mathbf{K}$, is composed of a double loop, the outer over the nonzero diagonals, the inner over the number of gridpoints. The inner loop, the multiplication with a diagonal of $\mathbf{K}$, can be parallelised by adding the appropriate directives. The third step, adding all the contributions per continent together, can be parallelised by performing the additions for the different continents in parallel. However, as in the first step, this operation is too inexpensive for parallelisation, and is therefore performed sequentially.

Coarse-Grain Parallelism

Coarse grain parallelism is introduced by a domain-decomposition technique [8], in combination with message passing to exchange information between subdomains. The original domain is split into equally-sized subdomains. Each subdomain is mapped onto a processor. All finite elements are uniquely assigned to a subdomain, but the subdomains share the (sea) gridpoints at the interfaces. Continents give global connections between subdomains. Hence continent values can be shared between subdomains that are not direct neighbours. We have used MPI routines [11] to implement the communication between subdomains.

The vector update is trivially parallelised, it is performed per subdomain. To compute an inner product, all contributions per subdomain can be computed locally. The results must be sent to the other subdomains (global communication) and added together. We have implemented this with the `MPI_ALLREDUCE` routine. Special attention must be paid to the values corresponding to interface nodes and to continents to assure that they are taken into account only once.

In the first step of the matrix-vector multiplication all the land points in a subdomain are given their continent value. This value is always locally available, therefore no communication is necessary. The second step is the diagonalwise matrix-vector multiplication. This operation can be performed by locally multiplying with the part of the system matrix that corresponds to the subdomain. At the end of this step, communication has to be performed with neighbouring subdomains to add up the interface values. We have implemented this with the `MPI_SENDRECV` routine. The third step is the addition of all the continent values. A specific continent may be in many, even in all domains. To exchange the local contribution of the subdomains to a continent value we therefore perform a *global* communication operation, so that all local continent values are sent to all subdomains after which they are added together. We have implemented this with the `MPI_ALLREDUCE` routine.

Combining Finer- Grain and Coarse. Grain Parallelism

The domain-decomposition approach can be combined in a straightforward way with fine-grain parallelisation by exploiting the loop parallelism (using OpenMP) per subdomain. This is a natural idea on clusters of SMP's, where each subdomain can be mapped onto an SMP. MPI can be used for the communication between SMP's and OpenMP on an SMP [4].

4 Numerical Experiments

4.1 The Test Problems

The ocean model has been discretised with linear triangular elements on two different grids, the first of 1° resolution (361×181 gridpoints) and the second of 0.5° resolution (721×361 gridpoints). The values for the physical parameters have been taken from [10].

The resulting two linear systems have been solved with GMRESR, with 30 GMRES inner-iterations per outer iteration, and with truncation after 20 outer iterations.

4.2 The Cluster of SMP's

The numerical experiments have been performed on a cluster of 8 bi-processor Pentium PC's running under LINUX. The main features of this cluster of SMP's are tabulated in Table 1. The Portland Group Fortran 90 compiler, which supports the OpenMP directives [2], has been used to compile the program. The message-passing library is based on MPI-CH [5].

Table 1. Main features of the cluster of Pentium PC's

SMP	Processor	Cache	Network
2 proc.	Pentium III	16 KB L1	Myrinet
1 GB RAM	933 Mhz	256 KB L2	250 MB/s bandwidth
8 SMP's	933 MFlops		each way in full-duplex

4.3 Numerical Results: 1° -Resolution Problem

The total number of gridpoints for the 1° -resolution problem (including all land points) is about 65,000. The number of GMRESR iterations to solve this problem is 51, which corresponds to 1581 matrix-vector multiplications (MATVEC), a little more than 26,000 inner products (INPROD), and about 27,000 vector updates (UPDATE).

Table 2. Elapsed time [s], combined OpenMP/MPI, 1° -resolution problem

# Processors	SOLVER	MATVEC	INPROD	UPDATE
$(1*1)*1 = 1$	181.8	64.9	48.2	50.8
$(1*1)*2 = 2$	90.5	45.3	22.4	12.8
$(2*1)*2 = 4$	30.0	9.6	12.3	3.8
$(2*2)*2 = 8$	15.6	4.7	7.1	1.9
$(4*2)*2 = 16$	9.2	2.5	4.9	0.9

The time measurements (elapsed time) for the combination of OpenMP and MPI are given in Table 2. The first column of this Table gives the number of subdomains in longitudinal and latitudinal direction (the two numbers between paranthesis), the number of threads, and the total number of processors used. The second column, denoted SOLVER, gives the total time spent in GMRESR. The third, fourth and fifth columns give the timings (including communication time) for the three operations MATVEC, INPROD and UPDATE. At a first glance, the speed-up for pure OpenMP (first two lines of Table 2) seems to be quite satisfactory. The apparently linear speed-up, however, is largely due to the super-linear speed-up for the UPDATE operation. This super-linear speed-up can be explained by a better use of the L2 cache in the procedure to orthogonalise a new basis vector for the Krylov subspace in GMRES. The orthogonalisation of a vector $\mathbf{w}$ with respect to another vector $\mathbf{v}$ is performed in two steps, an inner product followed by a vector update. If the vectors $\mathbf{v}$ and $\mathbf{w}$ both fit in the L2 cache, they have to be loaded only for the inner-product operation and they can remain in cache for the update. The amount of memory needed to store one vector is $361 \times 181 \times 4$ (number of gridpoints times storage for a single precision number), which is 255,2 KB, just about the size of the L2 cache. If the operations are performed in parallel by two processors, the effective vector length per processor is halved, which means that two vectors should fit in the cache. The fact that we also see a super-linear speed-up for the update-operation from 2 to 4 processors may be an indication that this is not exactly the case.

A similar cache effect can be observed for the matrix-vector product. The structured matrix-vector multiplication $\mathbf{v} = \mathbf{K}\mathbf{u}$ is composed of a sequence of multiplications with a diagonal matrix and vector updates. During these operations, the vectors $\mathbf{u}$ and $\mathbf{v}$ can remain in the L2 cache if there is enough space for *three* vectors, namely $\mathbf{u}$ and $\mathbf{v}$ and also for one diagonal of $\mathbf{K}$. This is the case if the vectors can be distributed over four L2-caches. This explains the super-linear speed-up for the MATVEC, going from 2 to 4 processors.

The adverse effects of communication are rather small. Some effect is visible in the results on 16 processors, in particular for the inner product.

Table 3. Elapsed time [s], MPI only, 1°-resolution degree problem

# Processors	SOLVER	MATVEC	INPROD	UPDATE
(1*1)*1 = 1	181.8	64.9	48.2	50.8
(2*1)*1 = 2	109.7	27.1	35.3	35.2
(2*2)*1 = 4	38.9	13.2	12.7	9.0
(4*2)*1 = 8	14.9	4.1	7.4	1.8
(4*4)*1 = 16	9.4	2.1	5.7	0.9

Table 3 gives the time measurements if only MPI is used. The most striking feature in this Table is the poor parallel performance on 2 processors, i.e. two MPI processes share one SMP. Disregarding the cache effect in the update,

this was also the case for the OpenMP results given in Table 2. This problem has also been observed and analysed in [4] where it was explained by memory contention. In Table 3 we see several instances of super-linear speed-up. These cases, however, are harder to relate directly to a specific cache effect than in the combined OpenMP/MPI case.

The speed-up of the time for the solution phase (with respect to the single processor time) displayed in Figure 1, shows that, for this problem, the combination of OpenMP and MPI is at least competitive with pure MPI.

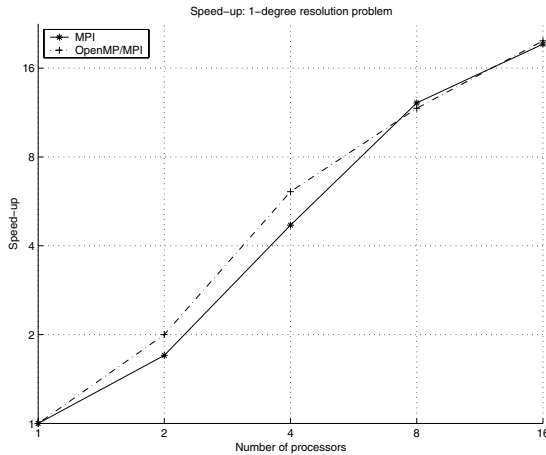


Fig. 1. Speed-up, 1° -resolution problem

4.4 Numerical Results: 0.5-Degree Resolution Problem

In this case, the total number gridpoints is about 260,000. The number of GM-RESR iterations to solve this problem is 120, which corresponds to 3720 matrix-vector multiplications, and a little more than 62,000 inner products and 64,000 vector updates.

The time measurements for the combination of OpenMP and MPI are presented in Table 4. The 0.5° -resolution problem is four times as large as the 1° -resolution problem. For this reason we see cache effects for the update and for the matrix-vector multiplication that are similar to the 1° -resolution problem, but now on four times the number of processors.

Table 5 gives the time measurements if only MPI is used. Noticeable is again, as for the 1° -resolution problem, the poor parallel scaling on a single SMP, with a speed-up of 1,6 going from one to two processors. The parallel scaling of OpenMP (see first two lines of Table 4) is even worse, with a speed-up of only 1,4.

Table 4. Elapsed time [s], combined OpenMP/MPI, 0.5°-resolution problem

# Processors	SOLVER	MATVEC	INPROD	UPDATE
(1*1)*1 = 1	1818	643	421	560
(1*1)*2 = 2	1310	477	303	379
(2*1)*2 = 4	593	167	178	180
(2*2)*2 = 8	216	81	67	44
(4*2)*2 = 16	80	26	34	9

Table 5. Elapsed time [s], MPI only, 0.5°-resolution problem

# Processors	SOLVER	MATVEC	INPROD	UPDATE
(1*1)*1 = 1	1818	643	421	560
(2*1)*1 = 2	1130	321	288	376
(2*2)*1 = 4	574	155	172	180
(4*2)*1 = 8	286	70	110	79
(4*4)*1 = 16	109	32	50	18

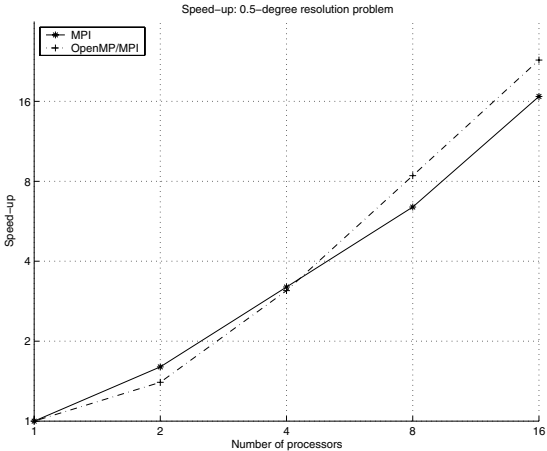


Fig. 2. Speed-up, 0.5°-resolution problem.

The speed-up of the time for the solution phase (with respect to the single processor time) is shown in Figure 2. This Figure shows a significantly better performance of the combination of OpenMP and MPI on 8 and 16 processors. Note that we saw the same improved performance for the combination of OpenMP and MPI for the 1°-resolution problem on 2 and 4 processors. The timing results given in Tables 4 and 5 indicate that, as for the 1°-resolution problem, the improved performance of the combined method can be explained by a better usage of the L2-cache in the update operation and in the matrix-vector multiplication.

5 Concluding Remarks

We have discussed the parallelisation of a stream-function model for global ocean circulation by a combination of loop-level parallelisation using OpenMP and a domain decomposition method in which communication between subdomains is implemented with MPI. Experiments on a cluster of bi-processor PC's show good parallel scaling properties. In our examples, the combined approach is competitive with, and often more efficient than a pure MPI-approach.

Acknowledgments. The author thanks Marielba Rojas, Bruno Carpentieri, Luc Giraud and Shane Mulligan for their valuable comments on an earlier version of this paper.

References

1. Bryan, K., and Cox, M.D.: The circulation of the world ocean: a numerical study. Part I, A homogeneous model. *J. Phys. Oceanogr.* **2** (1972) 319–335
2. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D.J and McDonald, J.: *Parallel Programming in OpenMP* Morgan Kaufmann Publishers (2000)
3. Eisenstat, S.C., Elman, H.C., and Schultz, M.H.: Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.* **20** (1983) 345–357
4. Giraud, L.: Combining Shared and Distributed Memory Programming Models on Clusters of Symmetric Multiprocessors: Some Basic Promising Experiments. *Int. J. High Perf. Comput. Appl.* **16:4** (2002) 425–430
5. Gropp, W.D., Lusk, E., Doss, N., and Skjellum, A.: A high-performance, portable implementation of MPI message passing interface standard. *Parallel Computing* **22(6)** (1996) 789–828
6. Saad Y., and Schultz, M.H.: GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Statist. Comput.* **7** (1986) 856–869
7. Van der Vorst, H.A., and Vuik C.: GMRESR: A family of nested GMRES methods, *Num. Lin. Alg. Appl.* **1** (1993) 1–7
8. Van Gijzen, M.B.: Parallel ocean flow computations on a regular and on an irregular grid. In: H.Liddell et al (eds.) *Proceedings HPCN 96, Lecture Notes in Computer Science*, Vol. 1067. Springer-Verlag, Berlin Heidelberg New York (1996) 207–212
9. Van Gijzen, M.B., Sleijpen, G.L.G., and Van der Steen, A.J.: The data-parallel iterative solution of the finite element discretization of stream-function models for global ocean circulation. In: A. Sydow, et al (eds.): *Proceedings 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, August 24–29, 1997, Berlin, Part III, Computational Physics, Biology, and Chemistry. Wissenschaft & Technik Verlag, Berlin, Germany (1997) 479–484
10. Van Gijzen, M.B., Vreugdenhil, C.B., and Oksuzoglu, H.: A finite element discretization for stream-function problems on multiply connected domains, *J. Comp. Phys.* **140** (1998) 30–46
11. Message Passing Interface Forum: MPI: A message-passing interface standard. *Int. J. Supercomputer Applications and High Performance Computing*, **8(3/4)**, 1994. (Special issue on MPI)

Scalable Parallel RK Solvers for ODEs Derived by the Method of Lines

Matthias Korch and Thomas Rauber

University of Bayreuth, Faculty of Mathematics and Physics
`{matthias.korch, rauber}@uni-bayreuth.de`

Abstract. This paper describes how the specific access structure of the Brusselator equation, a typical example for ordinary differential equations (ODEs) derived by the method of lines, can be exploited to obtain scalable distributed-memory implementations of explicit Runge-Kutta (RK) solvers. These implementations need less communication and therefore achieve better speed-ups than general explicit RK implementations. Particularly, we consider implementations based on a pipelining computation scheme leading to an improved locality behavior.

1 Introduction

Several approaches towards the parallel solution of ODEs exist. These include extrapolation methods [8], relaxation techniques [2], multiple shooting [6], and iterated Runge-Kutta methods, which are predictor-corrector methods based on implicit Runge-Kutta methods [9,14]. Two-step RK methods based on the computation of s stage approximations are proposed in [10]. The approach exploits parallelism across the method and yields good speed-ups on a shared address space machine. A good overview of approaches for the parallel execution of ODE solution methods can be found in [2,3,4]. Most of these approaches are based on the development of new numerical algorithms with a larger potential for a parallel execution, but with different numerical properties than the classical embedded RK methods.

In this paper, we consider the solution of initial value problems (IVPs)

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad \mathbf{y} : \mathbb{R} \rightarrow \mathbb{R}^n, \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n, \quad (1)$$

by explicit RK methods. In particular, we concentrate on ODEs derived from partial differential equations (PDEs) with initial conditions by discretizing the spatial domain using the method of lines. We choose the 2D-Brusselator equation [5] that describes the reaction of two chemical substances as a representative example for such ODEs. Two unknown functions u and v represent the concentration of the substances. A standard five-point-star discretization of the spatial derivatives on a uniform $N \times N$ grid with mesh size $1/(N-1)$ leads to an ODE system of dimension $2N^2$ for the discretized solution $\{U_{ij}\}_{i,j=1,\dots,N}$ and $\{V_{ij}\}_{i,j=1,\dots,N}$

$$\begin{aligned} \frac{dU_{ij}}{dt} &= 1 + U_{ij}^2 V_{ij} - 4.4 U_{ij} + \alpha(N-1)^2 (U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{ij}), \\ \frac{dV_{ij}}{dt} &= 3.4 U_{ij} - U_{ij}^2 V_{ij} + \alpha(N-1)^2 (V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{ij}), \end{aligned} \quad (2)$$

which is a non-stiff ODE system for $\alpha = 2 \cdot 10^{-3}$ and appropriate values of N . Non-stiff ODE systems can be solved efficiently by explicit RK methods with stepsize control using embedded solutions. An embedded RK method with s stages which uses the argument vectors $\mathbf{w}_1, \dots, \mathbf{w}_s$ to compute the two new approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ from the two previous approximations η_{κ} and $\hat{\eta}_{\kappa}$ is represented by the computation scheme

$$\begin{aligned} \mathbf{w}_l &= \eta_{\kappa} + h_{\kappa} \sum_{i=1}^{l-1} a_{li} \mathbf{f}(x_{\kappa} + c_i h_{\kappa}, \mathbf{w}_i), \quad l = 1, \dots, s, \\ \eta_{\kappa+1} &= \eta_{\kappa} + h_{\kappa} \sum_{l=1}^s b_l \mathbf{f}(x_{\kappa} + c_l h_{\kappa}, \mathbf{w}_l), \quad \hat{\eta}_{\kappa+1} = \eta_{\kappa} + h_{\kappa} \sum_{l=1}^s \hat{b}_l \mathbf{f}(x_{\kappa} + c_l h_{\kappa}, \mathbf{w}_l). \end{aligned} \quad (3)$$

The coefficients a_{ij} , c_i , b_i , and $\hat{b}_i$ are determined by the RK method used.

If the right hand side function $\mathbf{f}$ of the ODE system is assumed to have an arbitrary dependence structure, in general, the process of computing the argument vectors is sequential, and only minor program modifications are possible to increase locality [12]. The discretized Brusselator equation—similar to other ODEs derived from PDEs—has a loosely coupled dependence structure that is induced by the five-point-star discretization. Thus, the evaluation of component U_{ij} only accesses the components V_{ij} and U_{ij} , $U_{i+1,j}$, $U_{i,j+1}$, $U_{i-1,j}$, $U_{i,j-1}$, if available. Similarly, the evaluation of component V_{ij} uses the components U_{ij} and V_{ij} , $V_{i+1,j}$, $V_{i,j+1}$, $V_{i-1,j}$, $V_{i,j-1}$, if available. When such special properties of $\mathbf{f}$ are known, they can be exploited to obtain faster solvers for this particular class of ODEs. In [7], a sequential pipelining computation scheme has been proposed that leads to an improved locality behavior resulting in a significant reduction of the execution time for different processors. In this paper, we develop parallel implementations for a distributed address space that exploit special properties of the right hand side function $\mathbf{f}$. In contrast to a straightforward implementation that uses global communication operations, these implementations are based on single transfer operations only. The new implementations try to hide the communication costs as far as possible by overlapping communication and computation. As a result, these parallel implementations show good scalability even for large numbers of processors. We demonstrate this for an UltraSPARC II SMP, a Cray T3E-1200 and a Beowulf cluster.

2 Exploiting Specific Access Structure for Locality Improvement

Though it is possible to improve the locality of RK methods by modifications of the loop structure and other rearrangements of the code [12], we can maximize locality only by exploiting special properties of the problem to be solved. In order to store the components $\{U_{ij}\}_{i,j=1,\dots,N}$ and $\{V_{ij}\}_{i,j=1,\dots,N}$ of the Brusselator equation in a linear vector of dimension $n = 2N^2$, different linearizations can be used. In our implementations, we use the *mixed row-oriented storage scheme* that reduces the maximum distance of the components accessed in the function evaluation of a single component:

$$U_{11}, V_{11}, U_{12}, V_{12}, \dots, U_{ij}, V_{ij}, \dots, U_{NN}, V_{NN}. \quad (4)$$

Here, the evaluation of function component f_l accesses the argument components $l - 2N, l - 2, l, l + 1, l + 2, l + 2N$ (if available) for $l = 1, 3, \dots, 2N^2 - 1$ and $l - 2N, l - 2, l - 1, l, l + 2, l + 2N$ (if available) for $l = 2, 4, \dots, N^2$. For this access structure, the most distant components of the argument vector to be accessed for the computation of one component of $\mathbf{f}$ have a distance equal to $2N$.

We consider the division of the mixed row-oriented storage scheme (4) into N blocks of size $2N$. Analyzing the data dependences occurring during the function evaluations, we can derive a pipelining computation scheme that computes all argument vectors during a single diagonal sweep [7]. Figure 1(a) illustrates the computation order of the pipelining computation scheme, and Fig. 1(b) shows the working space and the dependences of one pipelining step executed to compute block J of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. In Fig. 1(b), apart from the blocks completed during the pipelining step, all blocks are highlighted that are accessed during this step. Blocks required for function evaluations are marked by crosses and blocks updated using results of function evaluations are marked by squares. The working space of one pipelining step consists of $\Theta(s^2)$ blocks of size $2N$. This corresponds to a total of $\Theta(s^2N)$ components. In contrast, at each stage a general implementation potentially accesses one complete argument vector during the function evaluations and all succeeding argument vectors in order to update the partial sums they hold. This corresponds to a working space of $\Theta(sn)$, which is equal to $\Theta(s \cdot 2N^2)$ in the case of the Brusselator function. The reduction of the working space by the pipelining computation scheme leads to increased locality and thus to better execution times on different sequential machines [7].

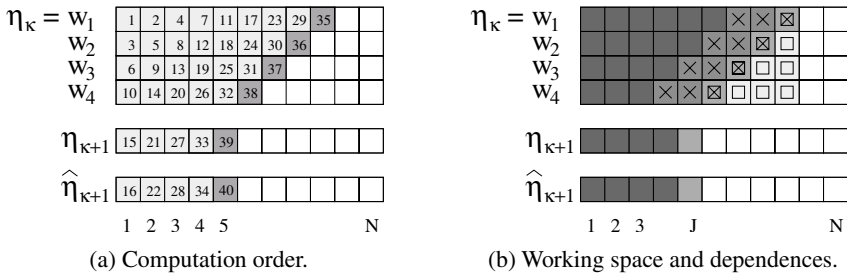


Fig. 1. Illustration of one pipelining step.

3 Exploiting Specific Access Structure for Parallelization

3.1 General Parallel Implementation

The general parallel implementation (Fig. 2) based on computation scheme (3) computes the argument vectors $\mathbf{w}_1, \dots, \mathbf{w}_s$ sequentially one after another. To exploit parallelism, the computation of the components of the argument vectors is distributed equally among the processors. But since the access structure is not known in advance, it must be assumed that each component of $\mathbf{f}$ accesses all components of the argument vector. Therefore, the argument vectors must be made available by multibroadcast operations.

```

multibroadcast( $\eta_\kappa$ );
for ( $j = \text{first\_component}; j \leq \text{last\_component}; j++$ ) {
  compute  $F = h f_j(t + c_1 h, \eta_\kappa)$ ;
  set  $\mathbf{w}_i[j] = \eta_\kappa[j] + a_{i1} F$  for  $i = 2, \dots, s$ 
   $\eta_{\kappa+1}[j] = \eta_\kappa[j] + b_0 F$ ;  $\hat{\eta}_{\kappa+1}[j] = \eta_\kappa[j] + \hat{b}_0 F$ ; }
for ( $l = 2; l \leq s; l++$ ) {
  multibroadcast( $\mathbf{w}_l$ );
  for ( $j = \text{first\_component}; j \leq \text{last\_component}; j++$ ) {
    compute  $F = h f_j(t + c_l h, \mathbf{w}_l)$ ;
    update  $\mathbf{w}_i[j] += a_{il} F$  for  $i = l + 1, \dots, s$ 
     $\eta_{\kappa+1}[j] += b_l F$ ;  $\hat{\eta}_{\kappa+1}[j] += \hat{b}_l F$ ; }
  perform error control and stepsize selection
}

```

Fig. 2. Pseudocode of one time step of a general parallel RK implementation for arbitrary right hand side functions $\mathbf{f}$.

Usually, the execution time of multibroadcast operations increases linearly with the number of participating processors [15]. Therefore, this implementation is not scalable to a large number of processors. Previous experiments have shown that for ODEs resulting from the spatial discretization of PDEs only a limited speed-up can be expected (e.g., DOPRI8(7) on Intel Paragon obtained 5.1 on 8 processors, 4.6 on 32 processors).

3.2 Parallel Blockwise Implementation

The special structure of Brusselator-like functions allows the derivation of more efficient implementations by applying the division of argument vectors into blocks of size $2N$. Thus, N/p adjacent blocks of each argument vector are assigned to each processor for computation.

One possible parallel blockwise implementation is illustrated in Fig. 3. As the general implementation (Fig. 2), this implementation processes the stages successively. But in contrast to the general implementation, no global communication is necessary. Since only the blocks $J - 1$, J , and $J + 1$ of the previous argument vector are required to compute block J of one argument vector, it is sufficient to send the first and the last block of a processor's part of each argument vector to its predecessor and its successor, respectively, to satisfy the dependence structure of the function. This can be done using single transfer operations like `MPI_Isend()` and `MPI_Irecv()`. These operations usually have execution times consisting of a constant startup time and a transfer time increasing linearly with the data size to be transferred. Consequently, the blockwise implementation has communication costs invariant to the number of processors.

Using the order $\{\text{first\_block} + 1, \dots, \text{last\_block} - 1, \text{last\_block}, \text{first\_block}\}$ to compute the blocks of a stage, the communication time can be hidden completely if the neighboring processors send off their first block and their last block of the preceding argument vector by a non-blocking send operation directly after their computation is finished and if the time to compute the $N/p - 2$ inner blocks is longer than the time needed to send two blocks through the network. The overall communication overhead in this ideal case consist of $2s$ times the startup time of the send operation and $2s$ times the startup time of the receive operation. If it is not possible to overlap communication and computation completely, the communication time increases by a fraction of $2s$ times the transfer time of $2N$ floating point values.

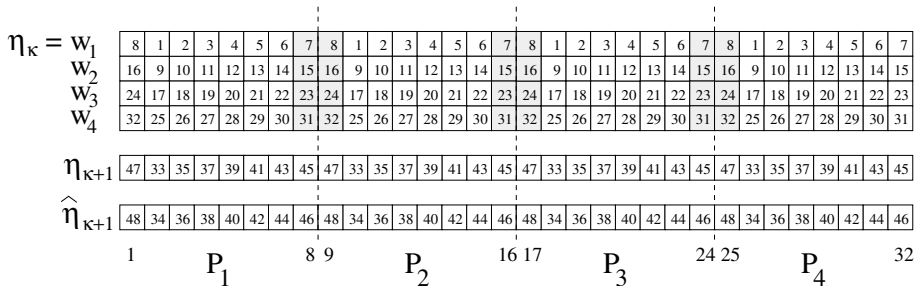


Fig. 3. Illustration of the parallel blockwise implementation.

3.3 Parallel Pipelining

Though the blockwise implementation has reduced communication costs, it does not exploit locality optimally. The pipelining computation scheme [7] allows to derive parallel implementations with similar communication costs but higher locality.

At each time step, the processors start the computation of the blocks (*first_block* + $i - 1$), ..., (*last_block* - $i + 1$) of the argument vectors w_i , $i = 1, \dots, s - 1$, in pipelining order. To ensure that each processor computes at least one block of w_s , we assume that at least $2s$ blocks are assigned to each processor. In the second phase, the processors finalize their pipelines at both ends of their data area. To provide efficient communication, neighboring processors use a different order to finalize the two ends of their pipelines. Thus, when processor j finalizes the side of its pipeline with higher index, processor $j + 1$ finalizes the side of its pipeline with lower index simultaneously, and vice versa. Figure 4 illustrates the parallel pipelining scheme. The numbers determine the computation order of the blocks. Blocks accessed during the finalization phase and blocks transmitted between processors are highlighted.

Communication is first required during the finalization of the pipelines, because function evaluations of the first and the last blocks access blocks stored in neighboring processors. When processor j computes its last block of argument vector w_i , $i = 2, \dots, s$, it needs the first block of argument vector w_{i-1} of processor $j + 1$, and vice versa. Again, we use non-blocking single send and receive operations in this transfer.

If the processors work simultaneously, the computation of blocks stored in the neighboring processor that are required to perform function evaluations is finished before they are needed. In fact, one diagonal across the argument vectors is computed between the time when processor $j + 1$ finishes its first block of w_{i-1} and the time when processor j needs this block to compute its last block of w_i , and vice versa. This time can be used to transfer the data required between the processors, thus overlapping communication and computation. But as the finalization phase proceeds, the length of the diagonals decreases from $s - 1$ to 1. Therefore, it is usually not possible to hide the transfer times at the end of the finalization phase completely. As a result, the communication costs of the parallel pipelining implementation are similar to the blockwise implementation, but the fraction of the transfer times that cannot be overlapped by computations is usually larger.

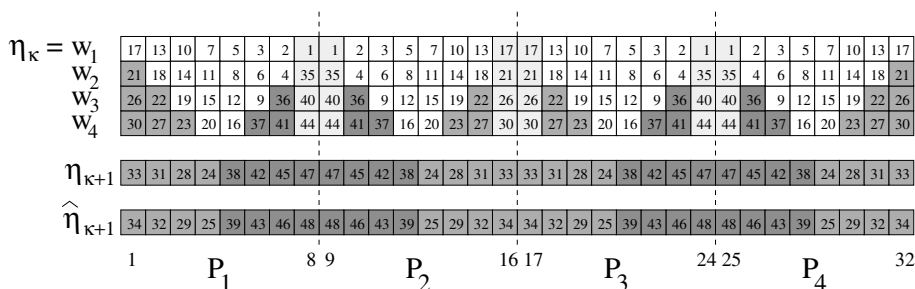


Fig. 4. Illustration of the parallel pipelining implementation (I).

3.4 Parallel Pipelining with Alternative Finalization

The start-up times, which largely determine the communication costs of the specialized parallel implementations previously described, can be quite expensive on some machines. Therefore, it is reasonable to investigate different implementations of parallel pipelining which execute fewer communication operations but transfer more data.

Such an implementation can be obtained by letting each processor finalize the higher end of its own pipeline and the lower end of the pipeline of its cyclic successor. Thus, to compute the stages, only one pair of communication operations is necessary to transfer data from a processor to its cyclic predecessor. This communication involves $2s + \sum_{i=2}^s i - 1$ blocks of argument vectors, s blocks of $\eta_{\kappa+1}$, s blocks of $\hat{\eta}_{\kappa+1}$, and s additional blocks (information for stepsize control). All in all, $s(s+9)/2$ blocks have to be transferred. The transfer can be started when the pipeline has been initialized, but the data are not needed before the finalization starts. Hence, the diagonal sweep across the argument vectors can be executed in parallel to the data transfer. During this sweep, $(N/p - 2s) \cdot s$ blocks of argument vectors are computed.

During the finalization of the pipelines, the processors compute parts of $\eta_{\kappa+1}$ that are required by their successors to start the next time step. Thus, another communication operation is needed to send s blocks of $\eta_{\kappa+1}$ to this processor. This is only necessary if the step is accepted by the error control, because otherwise the step is repeated using η_{κ} and, therefore, $\eta_{\kappa+1}$ is not needed (implementation (II)). But by exchanging the appropriate parts of $\eta_{\kappa+1}$ in every step, it is possible to hide part of the transfer time by computation. This presumes that the finalization starts at the lower end of the pipeline of the succeeding processor and proceeds toward the higher end of the pipeline of the processor considered (implementation (III)). Figure 5 illustrates the computation order of both implementations and the data transmitted.

4 Runtime Experiments and Analysis

We have executed several runtime experiments on three different parallel systems. The first machine is a shared-memory multiprocessor equipped with four Sun UltraSPARC II processors at 400 MHz. The second machine is a Cray T3E-1200 equipped with DEC Alpha 21164 processors at 600 MHz. Also, we have performed measurements on the Chemnitz Linux Cluster (CLiC)—a Beowulf cluster consisting of 528 Pentium III machines at

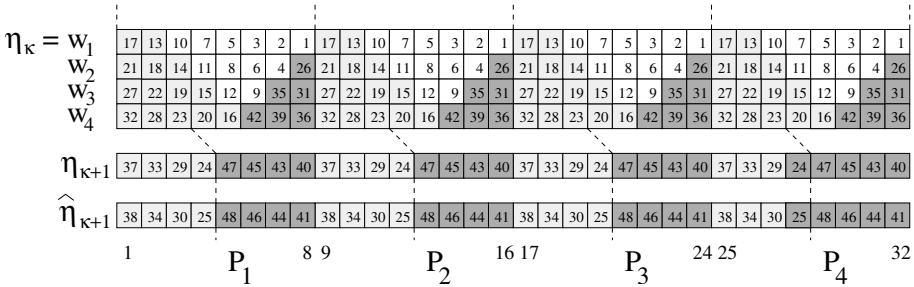


Fig. 5. Illustration of the parallel pipelining implementations (II) and (III).

800 MHz connected by a Fast Ethernet network. All programs have been implemented in C and use double precision. To perform communication, the MPI library [13] has been used, which provides message passing operations for distributed-memory machines but can also be used as a means of communication between processes on a shared-memory system. As RK methods we use the DOPRI5(4) and DOPRI8(7) methods [11]. The speed-up values presented compare the parallel execution times of the different parallel implementations with the execution time of the fastest sequential implementation on the respective machine.

All experiments we performed on the UltraSPARC II machine use the gridsize $N = 384$. The execution times of the general and the pipelining implementations measured for the two RK methods and the integration interval $H = 4.0$ are displayed in Tab. 1. With these parameters, the specialized implementations obtain better speed-ups than the general implementation. For example, using DOPRI5(4) and $H = 4.0$, speed-ups between 3.21 and 3.46 have been measured for the specialized implementations while the general implementation only obtains a speed-up of 1.27. Because of improved locality, the pipelining implementations are faster than the blockwise implementation, particularly for small numbers of processors. But the difference in the speed-ups decreases when the number of processors is increased. The pipelining implementations (II) and (III) both obtain smaller speed-ups than the pipelining implementation (I). The difference increases with the number of processors. Using four processors, the pipelining implementations (II) and (III) are slower than the blockwise implementation for the DOPRI5(4) method.

Table 1. Execution time (in seconds) of the parallel implementations on an UltraSPARC II SMP.

Parameters	DOPRI5(4), H=4.0, N=384				DOPRI8(7), H=4.0, N=384			
Processors	1	2	3	4	1	2	3	4
general	536.07	323.77	307.23	339.90	863.77	511.33	447.13	476.85
blockwise	521.37	264.16	174.50	130.00	832.07	428.89	295.53	218.79
pipelining (I)	445.40	231.57	154.53	124.24	676.48	346.46	234.15	184.53
pipelining (II)	447.15	230.92	162.48	131.73	678.37	350.64	241.82	198.46
pipelining (III)	450.98	232.00	161.99	133.87	674.99	351.78	242.97	197.48

Table 2 shows the execution times measured for the gridsizes $N = 384$ and $N = 896$ and the integration intervals $H = 0.5$ and $H = 4.0$ using DOPRI5(4) and DOPRI8(7) on the Cray T3E. In all these experiments, the specialized implementations obtain much better speed-ups than the general implementation. Particularly, the experiment using DOPRI5(4) with $H = 0.5$ and $N = 896$ where many processors can be used, shows very good scalability for these implementations. Because of the precondition that at least $2s$ blocks must be assigned to every processor, the pipelining implementations are limited to 64 processors in this experiment. Their maximum speed-ups are in the range of 54.15 to 55.39. Using 64 processors, the blockwise implementation obtains a speed-up of 55.28. But since the limitation to at most $N/(2s)$ processors does not apply to it, higher speed-ups can be reached with larger numbers of processors. For example, a speed-up of 109.93 has been measured using 128 processors. The scalability of the general implementation is limited to about 32 processors in this experiment. Its maximum speed-up measured is 12.15.

Using the PCL library [1], we have measured that it takes about 370 cycles to perform the evaluation of one component of the right hand side function $\mathbf{f}$ on the Cray T3E. Since the cycle length of the processors of this machine is 1.67 ns and the maximum network bandwidth is 500 MB/s, at least 24.7 blocks of argument vectors consisting of $2N$ double values must be computed to hide the transfer time of one such block completely. Thus, considering the experiment using $N = 896$, the parallel blockwise implementation can overlap $(N/p - 2)/24.7 \approx 49\%$ of the communication in the case of 64 processors and 20 % when 128 processors are used. To hide all transfer times completely, less than 33 processors should be used. For the parallel pipelining implementations, the fraction of the communication time that can be overlapped by computations is smaller. The first data transfer concerning blocks of η_κ can usually be performed in the background since it can be started at the beginning of the time step. In the case of DOPRI5(4) ($s = 7$), $N = 896$ and $p = 64$, 56 blocks are computed in the meantime. But the first diagonal of the finalization phase only consists of $s - 1$ blocks. Thus, using $s = 7$, only 24 % of the block transfer time is covered. At each stage of the finalization phase, the length of the diagonal computed is reduced by one block. At the end of the finalization phase, only 4 % of the transfer time is overlapped. In the pipelining implementations (II) and (III), $s(s + 9)/2$ blocks are transferred in parallel to the computation of $sN/p - 2s^2$ blocks. That means that 98 blocks are computed during the transfer of 56 blocks in our example with $p = 32$. This corresponds to an overlap of 7 %. If the current step is accepted, the parallel pipelining implementations (II) and (III) additionally need to transfer s blocks of $\eta_{\kappa+1}$ to the succeeding processor before the next time step can be started. In implementation (II), this transfer is not overlapped by computations. But implementation (III) tries to hide part of that transfer time by sending those s blocks in every step. This allows the computation of $\lceil s/2 \rceil (\lfloor s/2 \rfloor + 1)$ blocks in parallel to this transfer. Using $s = 7$, this leads to an overlap of 9 %.

The results of the experiments with the parallel implementations on the CLiC are shown in Tab. 3. Due to the poor performance of global communication operations caused by the slow interconnection network, the execution time of the general implementation cannot be improved by parallel execution. Its execution time increases when two processors are used. The pipelining implementation (I) and the blockwise imple-

Table 2. Execution time (in seconds) of the parallel implementations on a Cray T3E-1200.

Parameters	DOP5(4), H=0.5, N=896			DOP5(4), H=4.0, N=384				DOP8(7), H=0.5, N=896		
Processors	1	64	128	1	24	64	128	1	32	128
general	4359.27	375.12		1135.75	91.80	94.27		11629.54	651.80	
blockwise	4598.50	74.19	37.31	1269.78	54.74	20.85	10.61	12375.57	405.15	101.30
pipelining (I)	4782.31	74.05	n/a	1629.33	54.67	n/a	n/a	12702.03	400.09	n/a
pipelining (II)	4874.49	75.62	n/a	1559.61	55.32	n/a	n/a	12790.38	399.60	n/a
pipelining (III)	5401.25	75.74	n/a	1578.41	55.76	n/a	n/a	12889.71	402.05	n/a

Table 3. Execution time (in seconds) of the parallel implementations on the CLiC.

Parameters	DOP5(5), H=0.5, N=896				DOP5(4), H=4.0, N=384				DOP8(7), H=0.5, N=896		
Processors	1	32	64	128	1	2	24	128	1	32	128
general	1141.49				316.05	666.68			1936.29		
blockwise	986.45	35.15	18.69	10.74	274.50	141.26	13.66	4.26	1724.24	59.35	17.62
pipelining (I)	1005.00	35.71	18.99	n/a	252.66	128.27	12.36	n/a	1747.24	60.29	n/a
pipelining (II)	1343.48	55.42	39.80	n/a	345.06	140.67	25.93	n/a	2164.65	93.86	n/a
pipelining (III)	991.33	54.75	39.20	n/a	247.73	137.61	25.14	n/a	1721.89	92.70	n/a

mentation obtain similar speed-ups as on the Cray T3E. The pipelining implementations (II) and (III) are slower. In the experiment with DOPRI5(4), $H = 0.5$ and $N = 896$, their speed-ups are 52.75 and 53.58, respectively, on 64 processors. The other pipelining implementations only reach 25.17 and 25.55. With 128 processors, the blockwise implementation even obtains a speed-up of 93.30. The part of the communication time that can be overlapped by computations is smaller than on the Cray T3E, because the processors are faster (cycle length 1.25 ns) and the network is slower (100 Mbit/s). Assuming that the evaluation of one component of $\mathbf{f}$ also takes about 370 cycles, 1349 blocks must be computed to hide the transfer time of one single block. Therefore, even in our experiment with the blockwise implementation using $N = 896$ and 64 processors where 49 % of the transfer times could be covered on the Cray T3E only an overlap of 0.9 % is possible on the CLiC.

5 Conclusions

We have derived parallel implementations of embedded RK methods for the Brusselator equation, a typical example for ODEs resulting from the spatial discretization of PDEs. These implementations require less communication than general implementations supporting arbitrary right hand side functions $\mathbf{f}$. While the new parallel blockwise implementation already reduces the communication remarkably, the parallel pipelining implementations also exploit locality by reducing the working space of the algorithm.

Runtime experiments confirm that the scalability of the new implementations is better than the scalability of the general implementation. Using 64 processors, speed-ups of about 55 have been measured on the Cray T3E and about 53 on the Beowulf cluster. The results on the Beowulf cluster are particularly interesting as the slow interconnection network (Fast Ethernet) makes it difficult to obtain good speed-ups. Whether the

locality optimizations of the pipelining implementations are successful, depends on the architecture of the machine. If large numbers of processors are used, communication issues usually outweigh the acceleration achieved by locality improvements. Overlapping communication and computations showed improvements on the Cray T3E. But on the Beowulf cluster the slow interconnection network prevented significant improvements.

Acknowledgment. We thank the NIC Jülich for providing access to the Cray T3E and the TU Chemnitz for providing access to the CLiC.

References

1. R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2)*. Research Centre Jülich, January 2003.
2. K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford Science Publications, 1995.
3. C. W. Gear. Massive Parallelism across Space in ODEs. *Applied Numerical Mathematics*, 11:27–43, 1993.
4. C. W. Gear and Xu Hai. Parallelism in Time for ODEs. *Applied Numerical Mathematics*, 11:45–68, 1993.
5. E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin, 1993.
6. B. M. S. Khalaf and D. Hutchinson. Parallel Algorithms for Initial Value Problems: Parallel Shooting. *Parallel Computing*, 18:661–673, 1992.
7. M. Korch, T. Rauber, and G. Rünger. Pipelining for locality improvement in RK methods. In *Proc. of 8th Int. Euro-Par Conf. (Euro-Par 2002)*, pages 724–733. Springer (LNCS 2400), 2002.
8. L. Lustman, B. Neta, and W. Gragg. Solution of ordinary differential initial value problems on an Intel Hypercube. *Computer and Math. with Applications*, 23(10):65–72, 1992.
9. S. P. Nørsett and H. H. Simonsen. Aspects of Parallel Runge–Kutta methods. In *Numerical Methods for Ordinary Differential Equations*, volume 1386 of *Lecture Notes in Mathematics*, pages 103–117, 1989.
10. H. Podhaisky and R. Weiner. A class of explicit two-step Runge-Kutta methods with enlarged stability regions for parallel computers. *Lecture Notes in Computer Science*, 1557:68–77, 1999.
11. P. J. Prince and J. R. Dormand. High order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7(1):67–75, 1981.
12. T. Rauber and G. Rünger. Optimizing locality for ODE solvers. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 123–132. ACM Press, 2001.
13. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. MIT Press, Cambridge, Mass., second edition, 1998.
14. P. J. van der Houwen and B. P. Sommeijer. Parallel iteration of high-order Runge–Kutta methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.
15. Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.

Hierarchical Hybrid Grids as Basis for Parallel Numerical Solution of PDE^{*}

Frank Hülsemann¹, Benjamin Bergen¹, and Ulrich Rüde¹

System Simulation Group of the Computer Science Department,
Friedrich-Alexander University Erlangen-Nuremberg, Germany,
`frank.huelsemann@cs.fau.de`

Abstract. This paper describes the hierarchical hybrid grid framework, which is a storage scheme for the efficient representation and application of discrete operators and variables on globally unstructured but patch-wise regular grids. By exploiting patch-wise regularity it overcomes the performance penalty commonly associated with unstructured grids. Furthermore, it is well suited for distributed memory parallel architectures.

1 The Project Goal

The aim of the hierarchical hybrid grid approach is to perform multilevel algorithms efficiently on unstructured grids. However, it is well known that operations on unstructured grids achieve a lower run-time performance as measured in GFLOP/s as operations on structured grids. The main advantage of the structured operations is that regular patterns in the problem are known at compile time and can therefore be exploited for optimisation purposes. Hence, our strategy to reduce the performance penalty associated with unstructured grids is to introduce regularity in the computations. The importance of regularity as a key to high performance is also investigated by the FEAST [1] and ExPDE [5] projects, amongst others. Current versions of the hierarchical hybrid grid library have been used in projects such as natural attenuation simulations in environmental sciences, see [2].

2 The Hierarchical Hybrid Grid Concept

To achieve our aim, we combine and extend techniques from geometric multigrid (generation of nested grids) and computational fluid dynamics (multiblock grids) to construct grids that exhibit hierarchical, patch-wise regular structures in a globally unstructured, hybrid grid. The construction of a hierarchical hybrid grid is illustrated in figure 1. Thus, by construction, our approach combines the performance of operations on structured regions with the main advantage of

^{*} This project is funded by the KONWIHR grant gridlib of the Bavarian High Performance Computing Initiative.

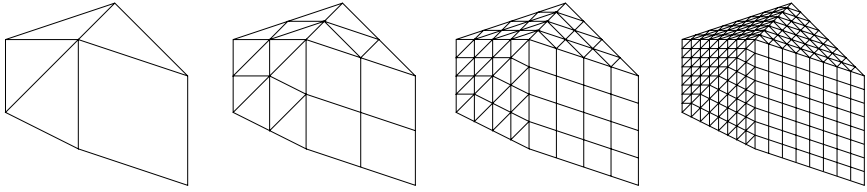


Fig. 1. From left to right: The hybrid input grid and the first three levels of refinement. After two levels of refinement (second from right) each patch has a structured interior which can be exploited to increase performance.

unstructured grids, namely the ability to represent complex problem domains. For further background information see [4].

It has to be pointed out that the patch-wise regularity of the refined grid levels alone does not produce an automatic performance improvement. The algorithms and the storage schemes have to adapt to reflect the structure. The standard way to represent a differential operator discretized by linear finite elements on a hierarchical hybrid grid is to store the stencil weights for each point in some sparse matrix format. Inside a volume cell, a cube say, the shape of the stencil is constant for all interior points. If, furthermore, the problem is linear with constant problem parameters inside the volume cell, then the entries of the stencil are also constant for all points in the cube. Hence, in this ideal case, the application of the operator to a variable can be reduced to the application of one stencil to all points inside the cell. Similar considerations apply to other cell types (prisms, tetrahedra) and faces (triangles, quadrilaterals) in 3D grids. Vertices and edges have to be treated in an unstructured fashion, given that the unstructured nature of the base grid manifests itself at these objects.

The efficient way of representing the operator is complemented by a storage scheme for the unknowns that allows index arithmetic to access neighbouring values, thus doing away with the need to store neighbourhood information explicitly.

3 Performance Results

In this section, we present performance results on different current architectures to show that the approach performs well on PCs and on supercomputers alike. Given that the aim is to make better use of the individual processors in a parallel application, we concentrate on MFLOP rates and not on the total run time of the applications, thus neglecting network influences on the parallel performance. The PCs are equipped with Intel PentiumIV processors, running at 2.4 GHz and a theoretical peak performance of 4.8 GFLOP/s. The processors of the Hitachi SR8000 supercomputer run at 375 MHz with a theoretical peak performance of 1.5 GFLOP/s.

Table 1. Performance of Gauß-Seidel iterations on a hexahedral element using a 27 point stencil. The results for the different implementations are given in MFLOP/s.

Refinement levels		4	5	6	7
#unknowns		3375	29791	250047	2048383
PentiumIV	CRS	328	352	347	352
	HHG	1278	1036	1106	980
SR8000	CRS	42	43	44	45
	HHG	146	333	530	443

The grids in the computations consist of hexahedra and quadrilaterals. In order to illustrate the maximum performance that can be achieved by exploiting regular structures, we concentrate on the case where the operator indeed reduces to a single stencil per object. The MFLOP rate is computed by dividing the number of floating point operations on that partition by the process time. The process time in turn is computed as the quotient of the number of all cycles used by the section of the process being assessed over the number of cycles per second. On the Intel platform, the PAPI library [3] was used to count the number of floating point operations and the number of cycles, while on the Hitachi the profiler of the compiler suite was employed.

We start with a comparison of the performance of Gauß-Seidel iterations on a cube, once implemented with the common compressed row storage format, the other time using stencil operations. As table 1 shows, the regular implementation is significantly faster on both tested platforms. More complex algorithms also achieve good performance, as can be seen from table 2 which gives the results for the conjugate gradient algorithm and a V(2,2) multigrid cycle. The example of the conjugate gradient method demonstrates that even algorithms with few matrix-vector operations per iteration gain in speed. To support our claim that the increase in the overall performance is due to the matrix-vector product, we include the performance of the daxpy-operation, which is the main other component in the algorithms and which has not been optimised yet.

4 Conclusions

We have presented a framework for the generation and exploitation of patch-wise regular structures from an unstructured input grid, thus preserving geometric flexibility while allowing efficient computations. Our experiments show that a substantial gain can be expected on all platforms over standard unstructured implementations. While architectures like the Intel PentiumIV permit a speed up factor of up to 3, the results on the Hitachi SR8000 show an improvement of a factor close to 10.

The implementation does not yet realize the full scope of the concept. The extension to different cell types, to problems with variable coefficients and the inclusion of grid adaptivity is ongoing work.

Table 2. Performance of conjugate gradient (CG) and geometric multigrid (GMG) algorithms, again using a 27 point stencil. These results were obtained on a single node in a MPI-parallel computation. The partition consisted of one cube with six remote faces. By daxpy we denote the operation of adding one vector, scaled by a scalar, to another vector. M-V product denotes the matrix-vector multiplication. All results are again given in MFLOP/s.

Refinement levels		4	5	6	7
#unknowns		3375	29791	250047	2048383
PentiumIV	daxpy	324	141	152	159
	M-V product	1345	1372	1216	1183
	CG	280	441	549	571
	V(2,2) GMG	190	441	632	749
SR8000	daxpy	58	102	158	209
	M-V product	186	430	672	847
	CG	22	113	247	391
	V(2,2) GMG	24	97	243	392

Acknowledgements. The authors acknowledge the financial support through KONWIHR project gridlib of the Bavarian High Performance Computing Initiative, which also provided the access to the Hitachi SR8000.

References

1. Becker, C., Kilian, S., Turek,S.: Some concepts of the software package FEAST, In: Palma, J., Dongarra, J., Hernandez, V. (Eds.): Proceedings Vector and Parallel Processing – VECPAR98. Lecture Notes in Computer Science **1573**, 271–284 Springer-Verlag, 1999

2. Bergen, B.: Hierarchical Hybrid Grids, Technical Report 03-5, System Simulation Group, University Erlangen, 2003

3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors, The International Journal of High Performance Computing Applications **14:3** (2000), 189–204

4. Hülsemann, F., Kipfer, P., Rüde, U., Greiner, G.: gridlib: Flexible and efficient grid management for Simulation and Visualization, In: Sloot, P., Tan, C.J.K., Dongarra, J., Hoekstra, A. (Eds.): Computational Science – ICCS 2002. Lecture Notes in Computer Science **2331**, 652–661 Springer-Verlag, 2002

5. Pflaum, C.: Semi-unstructured grids, Computing, **67** (2001), 141–166

Overlapping Computation/Communication in the Parallel One-Sided Jacobi Method*

El Mostafa Daoudi, Abdelhak Lakhouaja, and Halima Outada

University of Mohammed First, Faculty of Sciences

Department of Mathematics and Computer Science

LaRI Laboratory, 60 000 Oujda, Morocco

e-mail : {mdaoudi,lakhouaja,outada}@sciences.univ-oujda.ac.ma

Abstract. In this work, we propose some techniques for overlapping the communication by the computation in the parallelization of the one-sided Jacobi method for computing the eigenvalues and the eigenvectors of a real and symmetric matrix. The proposed techniques are experimented on a cluster of PCs and on the parallel system TN310.

1 Introduction

In this paper, we study, on distributed memory architecture, the parallelization of the one-sided Jacobi method for computing the eigenvalues and the eigenvectors of a real symmetric matrix A of size n . The one-sided algorithm is better suited to parallel computers [2,6], since it needs less communications than the classical Jacobi method ("*two-sided Jacobi method*" [3]). It only needs one-to-one communications type (column translation) in contrast to the two-sided Jacobi method which needs, in addition, global communications (rotations broadcast) [8,4]. On the other hand, the computation time complexity of the one-sided and the two sided (without exploiting the symmetry) algorithms are equivalent for computing the eigenvalues and the eigenvectors. Our objective in this study is to propose some techniques for overlapping the communications by the computations during the translation phase by carrying out simultaneously the update and the translation of one column [5].

We consider a distributed memory architecture composed of p processors, denoted by P_m , $0 \leq m \leq p-1$. The communication cost of L data between two neighbor processors is modeled by $comm(L)$ and the computation time of one floating point operation is designed by ω .

2 Sequential Algorithm

The basic idea of the Jacobi method [3] consists in constructing the sequences of matrices $\{A^{(k+1)} = J^{(k)\top} A^{(k)} J^{(k)}\}$ and $\{U^{(k+1)} = U^{(k)} J^{(k)}\}$ which converge respectively to a diagonal matrix that contains the eigenvalues and

* Supported by the "Comité mixte Franco-Marocain", AI n° : MA 01/19, and by the European INCO-DC Program, "DAPPI" Project

to a matrix that contains the eigenvectors. Where $A^{(1)} = A$, $U^{(1)} = I_n$ and $J^{(k)} = J(i, j)$, for $1 \leq j < i \leq n$, is a Jacobi rotation, in the (i, j) plane, chosen in order to annihilate $a_{i,j}^{(k)} = a_{j,i}^{(k)}$. It is completely determined by $a_{i,i}^{(k)}$, $a_{j,j}^{(k)}$ and $a_{i,j}^{(k)}$.

The one-sided Jacobi method: The application of the rotation $J(i, j)$ to the matrix, requires to update the columns i and j of the matrix and by symmetry the rows i and j . This update needs to exchange the rotations, which leads to an all-to-all communication type (global synchronization). In order to avoid the global synchronizations, one constructs the sequence $\{\bar{A}^{(k+1)} = \bar{A}^{(k)} J^{(k)}\}$ instead of the sequence $\{A^{(k)}\}$, where $\bar{A}^{(1)} = A$. At each step k of the algorithm, one computes one rotation (determination of $J^{(k)}$) and one updates two columns of $\bar{A}^{(k)}$ and two columns of $U^{(k)}$. Since the determination of $J^{(k)} = J(i, j)$, needs the knowledge of the coefficients $a_{i,i}^{(k)}$, $a_{j,j}^{(k)}$ and $a_{i,j}^{(k)}$, and since, we only know $\bar{A}^{(k)} = (\bar{A}_1^{(k)}, \dots, \bar{A}_n^{(k)})$ and $U^{(k)} = (U_1^{(k)}, \dots, U_n^{(k)})$, these coefficients can be computed by using the following relation [6]: $a_{m,l}^{(k)} = (U_m^{(k)})^\top \bar{A}_l^{(k)}$ for all $1 \leq m, l \leq n$ which requires $6n\omega$.

3 Study of the Parallelization of the One-Sided Jacobi

We consider a column distribution of $\bar{A}$ and U . Each processor P_m , for $0 \leq m \leq p-1$, holds $\frac{n}{p}$ columns of $\bar{A}$, denoted by $\bar{A}_{m_1(j)}$, $\bar{A}_{m_2(j)}$ and $\frac{n}{p}$ columns of U denoted by $U_{m_1(j)}$ and $U_{m_2(j)}$, for $1 \leq j \leq \frac{n}{2p}$. Initially, we have $m_1(i) = m \frac{n}{p} + i$, for $1 \leq i \leq \frac{n}{2p}$ and $m_2(i) = m \frac{n}{p} + i$, for $\frac{n}{2p} + 1 \leq i \leq \frac{n}{p}$, which corresponds to $\frac{n}{p}$ consecutive columns of A and $\frac{n}{p}$ consecutive columns of U .

The elements are annihilated according to the block odd-even ordering presented in [6]. During each step, all possible rotations are performed as shown on figure 1 for $n = 12$ and $p = 3$. For example, at the first step, P_1 holds the columns 5, 6, 7 and 8, then it performs all possible rotations (5, 6), (5, 7), (5, 8), (6, 7), (6, 8) and (7, 8). At the second step, it holds the columns 1, 2, 7 and 8, then it performs all possible rotations (1, 7), (1, 8), (2, 7) and (2, 8). The rotations (1, 2) and (7, 8) are done during the first step.

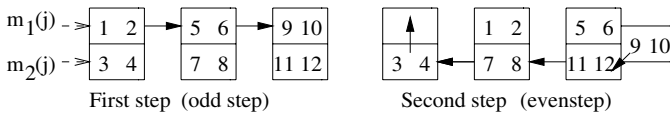


Fig. 1. The first and second steps of the block ordering for $n = 12$ and $p = 3$. The arrows show the sense of the column translation.

There are two phases for the parallelization. At each step, each processor:

- **Phase 1:** computes all possible local rotations and updates its local columns:

- in the first step, it computes all possible rotations $J(m_1(i), m_2(j))$, $J(m_1(i), m_1(j))$ and $J(m_2(i), m_2(j))$ for $1 \leq i < j \leq \frac{n}{2p}$, which corresponds to $\frac{\frac{n}{p}(\frac{n}{p}-1)}{2}$ rotations.
 - during the other steps, for $1 \leq i, j \leq \frac{n}{2p}$, it computes the rotations $J(m_1(i), m_2(j))$, which corresponds to $(\frac{n}{2p})^2$ rotations.
- **Phase 2:** communicates only with its two neighbors in order to send and receive $\frac{n}{2p}$ columns of $\bar{A}$ and $\frac{n}{2p}$ columns of U , in order to complete a sweep.

4 Overlapping the Communications

The basic idea to overlap the communication with the computation is to delay the computation of some rotations in order to be computed simultaneously during the translation phase. We assume that at one given step, the processor P_m , for $0 \leq m \leq p-1$, holds the columns $\bar{A}_{m_1(j)}$, $\bar{A}_{m_2(j)}$, $U_{m_1(j)}$ and $U_{m_2(j)}$, for $1 \leq j \leq \frac{n}{2p}$:

- it computes all possible rotations (illustrated by black squares in the figures 2) except the rotations $J(m_1(j), m_2(j))$, for $1 \leq j \leq \frac{n}{2p}$ (rotations illustrated by white squares in the figures 2).
- it updates the corresponding columns.

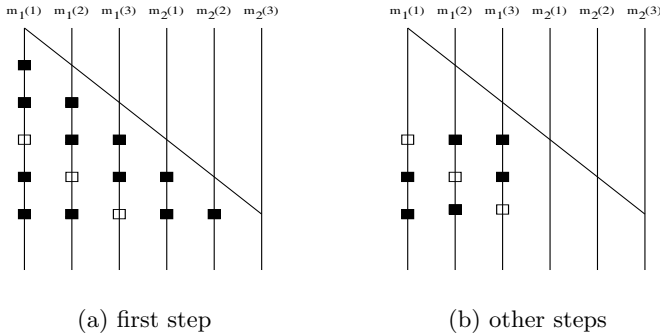


Fig. 2. Computation of the rotations during the overlapping phase.

The translation will be performed simultaneously with the computation of the remaining rotations $J(m_1(j), m_2(j))$, for $1 \leq j \leq \frac{n}{2p}$.

4.1 First Strategy

The theoretical study of this strategy is developed in [5]. Each processor computes $J(m_1(j), m_2(j))$ and updates $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$, then, simultaneously:

- updates $\bar{A}_{m_1(j)}$ and $U_{m_1(j)}$
- sends $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$ according to the odd-even ordering.

The update of one column needs $3n\omega$ and the computation of one rotation needs $6n\omega$, while the communication of two columns needs $comm(2n)$

Lemma: *If $6n\omega \geq comm(2n)$, all the communications can be overlapped by the computations.*

4.2 Second Strategy

Before translating columns, we divide each column of $\bar{A}$ and U into q blocks each of size $\frac{n}{q}$. The translation of each column is done in pipeline, as follow: Each processor:

- computes $J(m_1(1), m_2(1))$ and updates the first block of the columns $\bar{A}_{m_1(1)}$, $U_{m_1(1)}$, $\bar{A}_{m_2(1)}$ and $U_{m_2(1)}$.
- For $1 \leq j \leq \frac{n}{2p} - 1$:
 - * for $2 \leq i \leq q$, simultaneously:
 - updates the i^{th} blocks of $\bar{A}_{m_1(j)}$, $U_{m_1(j)}$, $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.
 - sends the $(i-1)^{th}$ blocks of $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.
 - * simultaneously:
 - computes $J(m_1(j+1), m_2(j+1))$ and updates the first block of the columns $\bar{A}_{m_1(j+1)}$, $U_{m_1(j+1)}$, $\bar{A}_{m_2(j+1)}$ and $U_{m_2(j+1)}$.
 - sends the last blocks of $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.
- for $j = \frac{n}{2p}$
 - * for $2 \leq i \leq q$, simultaneously:
 - updates the i^{th} blocks of $\bar{A}_{m_1(j)}$, $U_{m_1(j)}$, $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.
 - sends the $(i-1)^{th}$ blocks of $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.
 - * sends the last blocks of $\bar{A}_{m_2(j)}$ and $U_{m_2(j)}$.

The update of one block of a column needs $3\frac{n}{q}\omega$, while the communication of two blocks needs $comm(2\frac{n}{q})$. If we assume that the communication cost of L data between two neighbor processors is modeled by $comm(L) = \beta + L\tau$, where β is the start-up time and τ is the time to transmit one data, then the parallel time which necessitates this part of the parallel algorithm is modeled by:

$$T(q) = 6n\omega + \frac{12n\omega}{q} + \left(\frac{n}{2p} - 1\right) \left[(q-1) \max\left(\beta + \frac{2n\tau}{q}, \frac{12n\omega}{q}\right) + \max\left(\beta + \frac{2n\tau}{q}, 6n\omega + \frac{12n\omega}{q}\right) \right] + (q-1) \max\left(\beta + \frac{2n\tau}{q}, \frac{12n\omega}{q}\right) + \left(\beta + \frac{2n\tau}{q}\right)$$

Lemma: *The optimal size which minimize the parallel time is given by:*

1. $q^* = \min\left(\left\lceil \frac{2n(6\omega - \tau)}{\beta} \right\rceil, n\right)$ if $\beta + \frac{2n\tau}{q} \leq \frac{12n\omega}{q}$
2. $q^* = \min\left(\left\lceil 2\sqrt{\frac{6p\omega}{\beta}} \right\rceil, n\right)$ if $6n\omega + \frac{12n\omega}{q} \leq \beta + \frac{2n\tau}{q}$
3. if $\frac{12n\omega}{q} \leq \beta + \frac{2n\tau}{q} \leq 6n\omega + \frac{12n\omega}{q}$:
 - a) $q^* = \min\left(\left\lceil \sqrt{\frac{12n\omega + 4p\tau - 2n\tau}{\beta}} \right\rceil, n\right)$ if $n\tau \leq 6n\omega + 2p\tau$
 - b) $q^* = \min\left(\left\lceil 2n\frac{6\omega - \tau}{\beta - 6n\omega} \right\rceil, n\right)$ if $\beta - 6n\omega \geq 0$ and $6\omega - \tau \geq 0$

The proof of this lemma is straightforward by derivating the function $T(q)$.

5 Experimental Results

The proposed techniques have been implemented on our local parallel machine TN310 based on Transputers and on the cluster of PCs installed at IMAG-Grenoble-France [1]. Our objective is not to compare the obtained performances on the two different parallel systems, but to compare the performances obtained of the proposed techniques on each system. For the numerical tests, we have used the Frank matrix defined by: $A = (a_{i,j})$, where $a_{ij} = n - \max(i, j) + 1$, for $i, j = 1, 2, \dots, n$, which the eigenvalues are given by: $\lambda_k = \frac{1}{2(1 - \cos(\frac{2k-1}{2n+1}\pi))}$ for $k = 1, 2, \dots, n$.

In table 1 (resp. table 2), we compare the execution times for one sweep on the cluster (resp. TN310) without overlapping (n.ov) and the first strategie of the overlapping (str1). The obtained results on the cluster show that the use of the techniques of overlapping have weakly improved the execution time. On the TN310, the results show that good improvements are obtained, but the communications are not completely overlapped. This corroborate with the theoretical study. Note that the estimated machine parameters, for the TN310, under PVM environment, are $\beta = 0.002s$, $\tau = 22\mu s$ and $\omega = 0.25\mu s$.

Table 1. Execution times in seconds for one iteration on the cluster

n	128		256		512		768		1024		2048	
p	n.ov	str1	n.ov	str1	n.ov	str1	n.ov	str1	n.ov	str1	n.ov	str1
2	0.06	0.05	0.64	0.66	6.62	5.81	20.32	19.81	48.57	47.07	391.51	377.25
4	0.04	0.04	0.25	0.26	3.08	2.97	10.39	10.1	24.11	23.62	191.81	190.51
8	0.03	0.03	0.16	0.16	1.56	1.37	5.65	5.25	13.11	12.31	102.45	97.52
16	0.04	0.04	0.13	0.12	0.68	0.69	2.9	2.72	7.14	6.66	53.7	51.56
32	0.03	0.03	0.12	0.11	0.49	0.46	1.39	1.4	3.61	3.51	28.83	27.3

Table 2. Execution times in seconds for one iteration on the TN310

n	128		256		512		768		1024	
p	n.ov	str1	n.ov	str1	n.ov	str1	n.ov	str1	n.ov	str1
2	14.42	10.35	109.05	72.32	941.17	672.46				
4	6.65	5.47	45.42	36.66	471.77	374.81	1562.94	1267.31		
8	3.94	3.55	23.53	20.38	172.34	140.71	791.51	472.02	1857.85	1576.82
16	3.9	2.8	13.58	12.52	89.76	77.03	286.9	249.29	669.22	576.8

The table 3, shows that the lower execution time, on 16 processors, for the second strategie, is obtained for $q = 1$. This means that it is not necessary to subdivide the columns to be translated. This result corroborates with the theoretical study. Note that the algorithm for the case $q = 1$ is not identical to the algorithm proposed for the first strategie.

Table 3. Execution times on the TN310, with pipeline for 16 processors, using different values of q

	n				
q	128	256	512	768	1024
1	2.7	11.49	74.57	236.48	554.17
2	3.23	12.94	76.94	240.21	554.82
4	4.33	14.91	81.96	252.3	564.41
8	6.43	19.43	89.81	264.92	572.7

6 Conclusion

In this paper, we have proposed two algorithms, which enable the overlapping of the communications by the computations in the parallelization of the one-sided Jacobi method. These techniques are experimented under MPI on a cluster of PCs and under PVM on the parallel system TN310, since MPI is not implemented on our system.

The experimental execution times obtained on the TN310 corroborate with the theoretical results. The results obtained on the cluster are not those expected. We think that the overlap is not available with this platform. The proposed algorithms can be easily extended to the block one-sided Jacobi version.

Acknowledgment. The authors thanks the anonymous reviewers for there valuable comments.

References

1. <http://icluster.imag.fr>, "The icluster project".
2. J. Cuenca, D. Giménez, "Implementation of parallel one-sided block Jacobi methods for the symmetric eigenvalue problem", ParCo'99, Imperial College Press (2000), 291-298.
3. G.H. Golub, C.F. Van Loan, "Matrix computation", Johns Hopkins University Press, 1989, 2nd edition.
4. E.M. Daoudi, A. Lakhouaja, "Exploiting the symmetry in the parallelization of the Jacobi method", Parallel Computing 23 (1997), 137-151.
5. E.M. Daoudi, A. Lakhouaja, H. Outada, "Study of the parallel block one-sided Jacobi method", HPCN'2001, LNCS 2110 Springer Velag, 2001, 454-463.
6. P.J. Eberlein, H. Park, "Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures", Journal of Parallel and Distributed Computing 8 (1990), 358-366.
7. F.T. Luck, H. Park, "On parallel Jacobi orderings", SIAM J. Sci Stat. Comp. 10 (1989), 18-26.
8. M. Pourzandi, B. Tourancheau, "A parallel performance study of Jacobi-like eigenvalue solution", In Proceeding of First International Meeting on Vector and Parallel Processing, Porto, Portugal, September 1993.

Topic 12

Architectures and Algorithms for Multimedia Applications

Ishfaq Ahmad, Pieter Jonker Bertil Schmidt, and Andreas Uhl
Topic Chairs

In the recent years multimedia technology has emerged as a key technology, because of its ability to represent information in disparate forms as a bit-stream. This enables data from text to video and sound to be streamed, stored, processed, and delivered in digital form. A great part of the current research community effort has emphasized the delivery of the data as an important issue of multimedia technology. However, in the long run, also the creation, processing, and management of multimedia will most likely dominate the scientific interest. The aim to deal with information coming from video, text, and sound may result in a data explosion. The requirement to store, process, and manage large data sets naturally leads to the consideration of programmable parallel processing systems as strong candidates for supporting and enabling of multimedia technology. This makes, together with the inherent data parallelism in the data types, multimedia computing a natural application area for parallel processing. Concepts developed for parallel and distributed algorithms in general are quite useful for the implementation of distributed multimedia systems and applications. And hence, also the adaptation of these general methods to distributed multimedia systems is an interesting topic worth while studying.

This year, 9 papers discussing multimedia technology topics were submitted. Each paper was reviewed by at three reviewers and, finally, we selected 3 regular and 2 short papers, on the topics of: The design aspects of video mapping and network traffic balancing improving Double P-Tee architectures; An investigation on transmission schedulers that reduce the traffic burstiness in a Server-less Video-on-Demand System on Internet-like network topologies; A study on distributing multimedia content over peer-to-peer network by using a scheduling scheme that results in minimum buffering delay. A dynamic multicasting policy based on a phased proxy caching to reduce network bandwidth from a server to its clients. An enhanced register file architecture that performs either matrix multiplications or can be used to enlarge register bandwidth.

We would like to thank all authors who submitted a contribution, the Euro-Par Organizing Committee and the referees whose efforts have made this conference and this Topic possible.

Distributed Multimedia Streaming over Peer-to-Peer Networks

Jin B. Kwon¹ and Heon Y. Yeom²

¹ Sunmoon University, Dept. of Computer Science,
Asan, Chungnam, 336-708, South Korea
`jbkwon@sunmoon.ac.kr`

² Seoul National University, Dept. of Computer Science,
Seoul, 151-742, South Korea
`yeom@dcslab.snu.ac.kr`

Abstract. A peer-to-peer model is very useful in solving the server link bottleneck problem of a client-server model. In this work, we discuss the problems of distributing multimedia content over peer-to-peer network. We focus on two problems in peer-to-peer media content distribution systems. The first is the transmission scheduling of the media data for a multi-source streaming session. We present a sophisticated scheduling scheme, which results in minimum buffering delay. The second problem is on the fast distribution of media content in the peer-to-peer system that is self-growing. We propose a mechanism accelerating the speed at which the system's streaming capacity increases.

1 Introduction

It is generally believed that the streaming media will constitute a significant fraction of the Internet traffic in the near future. Almost all of the existing works on multimedia streaming is based on client-server models. Since multimedia streaming requires high bandwidth, server network bandwidth runs out rapidly if unicast client-server model is used. Single source multicast is one of the solutions that use a single stream to feed all the clients. The deployment of IP multicast has been slowed by difficult issues related to scalability, and support for higher layer functionality like congestion control and reliability. A peer-to-peer(P2P) model is ideal as a model to solve the server link bottleneck problem. In the P2P model, multimedia contents are distributed by using the bandwidth of the clients themselves.

The clients in P2P systems contribute resources to the community and in turn use the resources provided by other clients. More specifically, supplying peers holding a certain media file may stream it to requesting peers. Thus, data traffic is not localized on a specific site since the peers cooperate for sharing contents. It is typical that there is no central server that holds the contents, and peers work on an equal footing. How the contents exist at first in the P2P system is another question. We assume that there are seed peers. Examples of P2P content distribution systems include Napster[2], Gnutella[1], and so on.

There has been much work on P2P systems in recent years[3,6,7,8]. Those works dealt mainly with data lookup and storage management in a general P2P system. The problems on distributing *streaming media* over P2P network have also been studied in [4,5,9]. However, we dare say that the work on P2P media streaming systems is still in the early stage, and there is still some room for improvement of performance and generalization of model.

In this paper, we focus on two problems in P2P media content distribution systems. We first concentrate on the transmission scheduling of the media data for a multi-supplier P2P streaming session. More specifically, given a requesting peer and a set of supplying peers with heterogeneous out-bound bandwidth, the problem is how to schedule the segments of the media data using multiple channels respectively established with each supplying peer. A buffering delay is determined by the streaming schedule of each channel. The transmission scheduling has already been studied in [9], where Xu et al. presented OTS as its solution, which is optimal when the length of segments, the scheduling units, is identical. We present another scheduling scheme called *fixed-length slotted scheduling*(FSS), which further reduces the buffering delay. Unlike OTS, FSS employs variable length segments whose size is determined based on the bandwidth with which each segment is transmitted. The second problem is the fast distribution of media contents. Initially, there are only a few seed peers holding a content, and non-seed peers request the content as requesting peers. The P2P system is self-growing since the requesting peers can become supplying peers after they receive all the data. However, in the beginning, since there is only a small number of peers holding the content, the system can accommodate only a limited request arrival rate. The number of supplying peers grows as the content spreads out, and the system would eventually be able to service all the requests from other peers. Therefore, for a given arrival rate, it is very important to convert the requesting peers to the supplying peers in a short time so that all the incoming requests can be serviced. We have come up with a mechanism accelerating the speed at which the P2P system capacity increases, called FAST. It accelerates the distribution speed by allowing requesting peers that satisfy a condition to supply contents to other peers.

At first, we define a P2P media streaming model and state our assumptions. Our model is more practical and more general than the models of the previous works. A set of peers that can supply a media data is defined as the *candidate* set of the media content. A requesting peer selects its supplying peers from the set, opens a channel with each selected supplying peer, and requests the data segments from them according to a scheduling mechanism. The requesting peer receives the data from multiple channels and store them in its local storage, and then the peer may become a candidate of the media content. Note that a supplying peer can supply the media data to multiple requesting peers. Since the content searching problem in P2P network is another research issue, we assume that a requesting peer can get the candidate peer set and the information about resource usage of each peer using an appropriate searching mechanism. Let γ denote the playback rate of the media data. Each requesting peer P_r has an

in-bound bandwidth of $R_{in}(r)$ and an out-bound bandwidth of $R_{out}(r)$. Peers are heterogeneous in their in-bound and out-bound bandwidth. We assume that $0 < R_{in}(r) \leq \gamma$ and $R_{out}(r) > 0$.

2 Transmission Schedule

In this section, we study the problem of media data transmission scheduling. The problem is stated as follows: For a requesting peer P_r and n channels, to determine the data segments to be transmitted over each channel and the transmission order of the segments. The goal is to minimize buffering delay while ensuring a *continuous playback* at P_r , with *minimum* buffering delay. The buffering delay is defined as the time interval between the start of streaming and the start of playback at P_r . How to select the supplying peers of P_r is dealt with in Section 3. If the data is transmitted over multiple channels of various bandwidth lower than γ , the buffering delay is inevitable. That is because the transmission order of data is not the same as the playback one. Therefore, a well-devised data transmission schedule is essential in reducing the buffering delay.

A function $p(t)$ is defined as the amount of data being played for t seconds since the beginning of the playback. And a function $d(t)$ is defined as the amount of *consecutive data* from the beginning of the media file received for t seconds since the beginning of the streaming, at P_r . Since the data is assumed to be encoded in CBR, we express the amount of data in seconds. The data amount of k means the amount to be played for k seconds. To ensure the continuous playback, the following condition must be satisfied:

$$\forall t \geq 0, \quad d(t) \geq p(t). \quad (1)$$

Fig. 1(a) illustrates the increasing shape of $p(t)$ and $d(t)$ for the example of OTS[9]. In the figure, there are four channels with bandwidth of $\frac{7}{2}$, $\frac{7}{4}$, $\frac{7}{8}$ and $\frac{7}{8}$, respectively, and the requesting peer schedules the transmission of eight $\frac{7}{8}$ -second segments over the channels, according to OTS. As shown in the figure, the buffering delay δ is required to satisfy the condition of Eq.(1). δ is $\frac{3L}{8}$ in this example. Hence, $p(t)$ can be expressed as follows:

$$p(t) = \min\{\max\{t - \delta, 0\}, L\}, \quad t \geq 0, \quad (2)$$

where δ is the maximum of the difference between the dashed line($y = t$) and $d(t)$. That is, $\delta = \max_{t \geq 0}\{t - d(t)\}$. The closer $d(t)$ is to the straight dashed line, the smaller the buffering delay. A well-designed scheduling can make $d(t)$ close to the straight line. We propose the *fixed-length slotted scheduling*(FSS) as such a scheduling scheme.

To increase $d(t)$ linearly, the requesting peer P_r should receive the data in a sequential order. Based on this idea, FSS assigns the data to fixed-length slots of each channel, and the one-slot data chunks are defined as the data segments of FSS. Since the bandwidth of each channel is variable, the segment length also varies according to the channel bandwidth to which the data segment is assigned. And, the variable-length segments are assigned to the channels in a

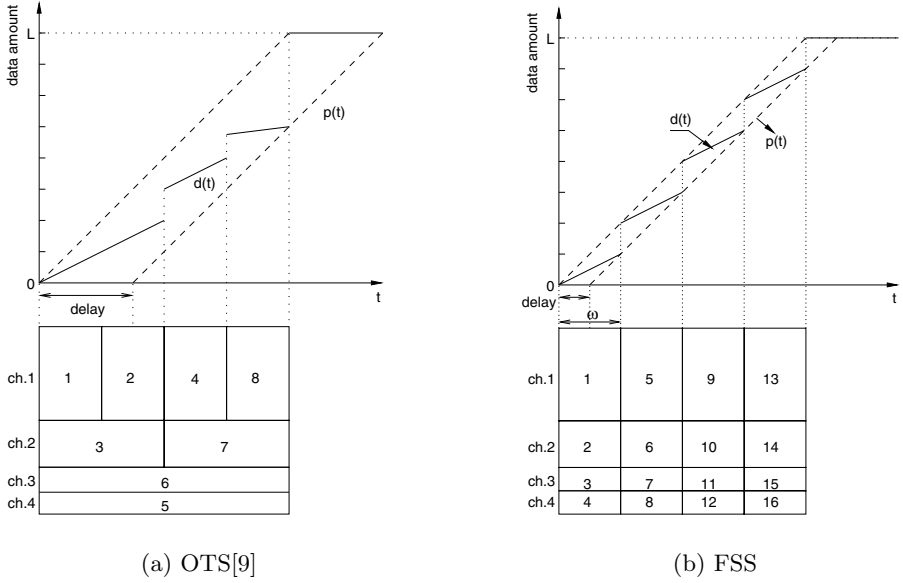


Fig. 1. Transmission Schedules

round-robin fashion. When the slot length is ω and i -th channel bandwidth is B_i , the segment length of channel i is ωB_i . Fig. 1(b) illustrates the concept of FSS, where the number of channels and the bandwidth of each channel are the same as those of Fig. 1(a). The slot length ω is $\frac{L}{4}$. In this example, the buffering delay of FSS is $\frac{L}{8}$, which is only a third of that of OTS.

Let us find the buffering delay, δ , of FSS. When the number of channels is n , the aggregated in-bound bandwidth $B^* = \sum_{i=1}^n B_i$. Then, $d(t)$ is:

$$d(t) = \min\left\{\frac{B_1}{\gamma}t + \frac{B^* - B_1}{\gamma} \left\lfloor \frac{t}{\omega} \right\rfloor \omega, L\right\}. \quad (3)$$

By expanding Eq.(1) with the $d(t)$ of Eq.(3) and the $p(t)$ of Eq.(2) and solving for δ , we can obtain the minimum δ . The detailed derivation is omitted here. The minimum buffering delay δ is:

$$\delta = \left(\frac{\gamma}{B^*} - 1\right)L + \left(\frac{B^* - B_1}{\gamma}\right)\omega, \quad \text{if } B^* \leq \gamma. \quad (4)$$

From the above equation, δ depends on B_1 and ω . This relationship is such that δ becomes smaller as B_1 gets greater and ω gets smaller. Thus, FSS chooses the channel with the maximum bandwidth as the first channel to minimize the buffering delay. When the aggregated bandwidth is equal to the playback rate, the shorter the slot length is, the shorter the buffering delay gets. However, the slot length is required to be long enough to cover the fluctuation of bandwidth and the overhead of transmission, delivery processing, and so on. Also, if it is too smaller than the maximum packet size of the underlying physical network, the network utilization would be low and the bandwidth would be wasted. Therefore, the slot length is a system parameter that should be carefully determined.

3 Fast Distribution

In [9], the candidate set consists of only the peers holding the whole media file. However, some of the peers while downloading the media data may be able to supply the data to requesting peers. Since the system capacity is proportional to the size of the candidate set, it would be beneficial to make the candidate set as large as possible. If it has a large candidate set, it would be easier for a requesting peer, P_r , to acquire sufficient bandwidth, and P_r would become a candidate peer within a short time. Thus, our goal is to find the peers that have enough data to act as supplying peers, in order to make the candidate set as large as possible.

A peer holding the whole media file is defined as a *mature* peer, and a peer being downloading the media data is defined as an *immature* peer. Fig. 2 shows P_r and its supplying peers. The shaded ones indicate mature peers, and the white ones indicate immature peers. Let $d_i(t)$ be $d(t)$ of a peer P_i , which we defined above, and we define another function as follows.

- $x_i(t, r)$: when P_i is assumed to be selected as a supplying peer of a requesting peer P_r , the position within the media file of the data to be requested to transmit at time t .

According to FSS, a supplying peer does not transmit the data continuously in a consecutive order, but transmits some data segments and skips some data segments in a predetermined pattern. Considering the instance of Fig. 1(b). The second channel transmits four segments of $\frac{\gamma}{4}\omega$ with the distance of $\frac{3\gamma}{4}\omega$. Therefore, as shown Fig. 2(b), $x_i(t, r)$ is a slanted staircase-shaped function, and the rate of increase of the solid line is $\frac{B_i(r)}{\gamma}$, where $B_i(r)$ is the i -th channel bandwidth of P_r . Also, the width of a step in the function corresponds to the slot length. $d_i(t)$ is L in case that P_i is a mature peer, and is determined by Eq.(3) in case that P_i is an immature peer. Fig. 2(b) shows $d_i(t)$ when P_i is an immature peer. That $x_i(t, r)$ crosses $d_i(t)$ as shown in the figure means that P_r would request the data which P_i would not have already received. Thus, P_i cannot be a supplying peer of P_r in this case. However, for an immature peer P_i to be a supplying peer of P_r , the following conditions must be satisfied:

$$\forall t \geq t_0, d_i(t) \geq x_i(t, r). \quad (5)$$

The immature peers satisfying this condition are called *semi-mature* peers of P_r . Consequently, the candidate set of P_r can consist of mature peers and semi-mature peers of P_r .

Although $d_i(t)$ has already been determined at the present time t_0 , $x_i(t, r)$ has not. That is because $x_i(t, r)$ cannot be determined until P_r finishes selecting its supplying peers. For this reason, we use an upper bound function of $x_i(t, r)$, $\bar{x}_r(t)$, instead. The upper bound function $\bar{x}_r(t) = R_{in}(r) \cdot \lfloor (t - t_0)/\omega \rfloor \omega$. In Fig. 2(b), the staircase-shaped dashed line indicates $\bar{x}_r(t)$. Therefore, a sufficient condition of Eq.(5) is: $\forall t \geq t_0, d_i(t) \geq \bar{x}_r(t)$. It can be used for a criterion to determine whether an immature peer P_i can be a semi-mature peer. However, since it is a sufficient condition, not satisfying it does not mean that P_i is not

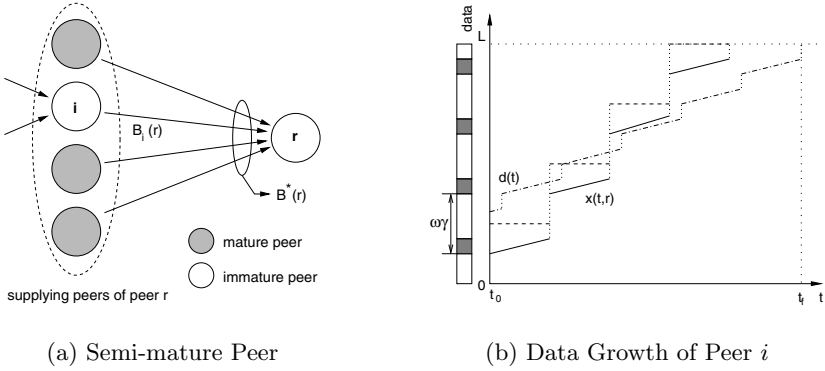


Fig. 2. Candidate Peers

a semi-mature peer. But, if it is satisfied, we only ensure it is a semi-mature peer. The procedure to select supplying peers is as follows: P_r determines its candidate set $\mathbb{C}_r$ such that each peer in $\mathbb{C}_r$ is a mature peer or a semi-mature peer and it has some available out-bound bandwidth¹. To determine $\mathbb{C}_r$, P_r should know whether or not each immature peer is a semi-mature peer, by testing $\forall t \geq t_0, d_i(t) \geq \bar{x}_r(t)$. Since the higher the first channel bandwidth is, the shorter the buffering delay in FSS(Eq.(4)), the peer with the maximal $v_{out}(i)$ among $\mathbb{C}_r$ is chosen as the first supplying peer of P_r . This procedure is repeated until the aggregated bandwidth $B^*(r)$ is equal to $R_{in}(r)$. The formal algorithm is omitted here due to the space limitation.

In case that there are so many requesting peers that the P2P system is beyond its capacity, it may not acquire enough bandwidth. However, in the latter case, P_r can have a choice among three policies. The first is to start downloading with the acquired channels(FAST1), the second is to withdraw the request and retry after σ (FAST2), and the third is to start downloading with the acquired channels and retry to acquire the remainder after T minutes(FAST3). In FAST3, unlike the others, the number of in-bound channels of P_r may be changed during a session. This means the change of $d_r(t)$, which accordingly affects the transmission schedule and the condition for a semi-mature peer. The details of this dynamic session is omitted here due to the space limitation.

4 Performance Study

First, we compare the buffering delay of FSS with that of OTS. However, since OTS has the restrictions on channel bandwidth and FSS has the concept of slots, it is not simple to compare them directly. For a fair comparison, we set a criterion on the slot length and evaluate the two schemes under the condition satisfying the restrictions of OTS. The restrictions of OTS is the following: 1) A single

¹ P_r is assumed to be able to know all the information needed to determine $d_i(t)$ for each immature peer P_i

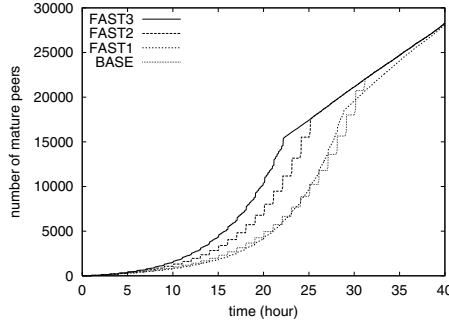


Fig. 3. Distribution Speed

channel bandwidth has one of $\frac{\gamma}{2}, \frac{\gamma}{4}, \frac{\gamma}{8}, \dots, \frac{\gamma}{2^N}$. 2) The aggregate bandwidth of a session $B^*(r) = \gamma$. Let m be the number of segments. Given a set of n supplying peers and a requesting peer P_r , OTS achieves $\delta_{OTS} = (n-1)L/m[9]$, if we assume that the content can be played while being downloaded. Under the same condition, the buffering delay of FSS is: $\delta_{FSS} = (1 - B_1(r)/\gamma) \cdot \omega$, according to Eq.(4). Since δ_{OTS} depends on m and δ_{FSS} on ω , the relation between m and ω should be drawn for the comparison. In OTS, the time taken for a segment to be transmitted over the channel with the highest bandwidth is corresponding to the slot length. That is, $\frac{\gamma}{B_1(r)} \cdot \frac{L}{m} = \omega$, and then, by applying it to δ_{OTS} : $\delta_{OTS} = (n-1) \frac{B_1(r)}{\gamma} \cdot \omega$. Here, let $c = B_1(r)/\gamma$. Then, since $B_1(r)$ is the highest bandwidth, n must be greater than or equal to $1/c$ to satisfy that $B^*(r) = \gamma$. Hence, $n \geq 1/c$. For example, if $B_1(r) = \gamma/4$, $n \geq 4$. Therefore,

$$\delta_{OTS} = (n-1)c \cdot \omega \geq (1/c - 1)c \cdot \omega = \delta_{FSS}.$$

Finally, the buffering delay of FSS is smaller than or equal to that of OTS. In the example used in Fig. 1(a) and 1(b), $\delta_{OTS} = 3\omega/2$ and $\delta_{FSS} = \omega/2$. FSS has only a third the buffering delay of OTS in that case.

Second, we study the performance of FAST using a simulation. We simulate a P2P system with total of 50,100 peers. Initially, there are only 100 'seed' peers, while the other 50,000 peers request the media data according to a given request arrival rate. The request arrival follows a Poisson distribution with mean $1/\theta$. Each seed peer possesses a copy of a popular video file. The running time of the video is 60 minutes. The in-bound bandwidth of all the peers is γ , equal to the playback rate. The out-bound bandwidth of the seed peers is $\frac{\gamma}{2}$, and that of the others is $\frac{\gamma}{2}, \frac{\gamma}{4}, \frac{\gamma}{8}$, or $\frac{\gamma}{16}$ with the distribution of 10%, 10%, 40%, and 40%, respectively. The time interval T for retrying to acquire bandwidth has a uniform distribution with a mean of 10 minutes. And, the slot length ω is set also to be 10 minutes. A BASE scheme for comparison to FAST considers only mature peers as supplying peers, and it allows a requesting peer to start downloading when it acquires a sufficient bandwidth like FAST2.

As the content is distributed to more peers, the P2P system's capacity increases. Thus, when requests arrive as Poisson process with a rate of $1/\theta$, not all of the requests may be serviced at the beginning stage when there are only

a small number of mature peers and semi-mature peers. This congestion due to the high arrival rate lasts for a while, until the number of mature peers grows sufficiently. We define θ -capacity time as the time to reach a capacity being able to service all the requests arriving at a rate of $1/\theta$. The smaller θ -capacity time means the faster distribution of the media content. Fig. 3 shows the number of mature peers of BASE, FAST1, FAST2, and FAST3 when $\theta = 5$ seconds. If the system reaches θ -capacity, the number of mature peers increases linearly at a rate of $1/\theta$. However, as shown in the figure, the lines increase in a concave fashion in the early stage, and then increase in a straight line. The concave curves indicate that the system is congested. The 5-capacity time of FAST3, about 21 hours, is shorter than those of FAST2, FAST1, and BASE, which are about 24, 28, and 30 hours, respectively. And, FAST3 has almost more than twice the number of mature peers than that of BASE.

5 Conclusion

We have discussed the problems of distributing multimedia content over P2P network. The first problem is the transmission scheduling of the media data for a multi-source streaming session. We have presented a sophisticated scheduling scheme, which results in minimum buffering delay. The second one is on the fast diffusion of media content in the P2P system that is self-growing. We have also proposed a mechanism accelerating the speed at which the system's streaming capacity increases.

References

1. Gnutella. <http://gnutella.wego.com>.
2. Napster. <http://www.napster.com>.
3. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. of Workshop on Design Issues in Anonymous and Unobservability*, July 2000.
4. T. Nguyen and A. Zakhor. Distributed Video Streaming Over Internet. In *Proc. of Multimedia Computing and Systems*, San Jose, California, January 2002.
5. Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, Miami Beach, FL, May 2002.
6. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *ACM SIGCOMM*, August 2001.
7. A. Rowstron and P. Druschel. Pastry: Scalable Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM Middleware*, November 2001.
8. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, August 2001.
9. Dongyan Xu, Mohamed Hefeeda, Susanne Hambrusch, and Bharat Bhargava. On Peer-to-Peer Media Streaming. In *Proc. of Int. Conf. on Distributed Computing Systems 2002*, Austria, July 2002.

Exploiting Traffic Balancing and Multicast Efficiency in Distributed Video-on-Demand Architectures *

Fernando Cores, Ana Ripoll, Bahjat Qazzaz, Remo Suppi, Xiaoyuan Yang,
Porfidio Hernández, and Emilio Luque

Computer Science Department – University Autònoma of Barcelona – Spain
{Fernando.Cores, Ana.Ripoll, Remo.Suppi, Porfidio.Hernandez,
Emilio.Luque}@uab.es
{Bahjat, Xiaoyuan}@aows10.uab.es

Abstract. Distributed Video-on-Demand (DVoD) systems are proposed as a solution to the limited streaming capacity and null scalability of centralized systems. In a previous work, we proposed a fully distributed large-scale VoD architecture, called Double P-Tree, which has shown itself to be a good approach to the design of flexible and scalable DVoD systems. In this paper, we present relevant design aspects related to video mapping and traffic balancing in order to improve Double P-Tree architecture performance. Our simulation results demonstrate that these techniques yield a more efficient system and considerably increase its streaming capacity. The results also show the crucial importance of topology connectivity in improving multicasting performance in DVoD systems. Finally, a comparison among several DVoD architectures was performed using simulation, and the results show that the Double P-Tree architecture incorporating mapping and load balancing policies outperforms similar DVoD architectures.

1 Introduction

Video on Demand (VoD) has been gaining popularity over recent years with the proliferation of high-speed networks. Distributed continuous media applications, are expected to provide service to a large number of clients often geographically dispersed over a metropolitan, country-wide or even global area. Employing only one large centralized continuous media server to support these distributed clients results in a high cost and non-scalable system with inefficient resource allocations.

To address this problem, researchers have proposed distribution of the service in order to manage client dispersal. Systems based on this approach are termed Distributed VoD systems and they have demonstrated the ability to provide minimum communication-storage cost for distributed continuous media streaming applications [1].

A DVoD system requires the arrangement of those servers that offer the video retrieval and playback services in a distributed system, in order to support a large num-

* This work was supported by the MCyT-Spain under contract TIC 2001-2592 and partially supported by the Generalitat de Catalunya- Grup de Recerca Consolidat 2001SGR-00218.

ber of concurrent streams. In the literature, these approaches range from: 1) the use of Independent servers, 2) one level proxies, to 3) hierarchical distributed systems. The initial approach is based on replicating VoD servers close to clients' networks so that these users do not need to access the main server [2]. One-level proxies try to reduce the size of local servers in such a way that they only store those videos with a higher access frequency; these servers are managed as main-server caches and are called proxies [9]. Hierarchical DVoD systems are based on a network with a hierarchical topology, with individual servers on the nodes and network links on the edge of the hierarchy. The nodes at the leaves of the hierarchy, termed head-ends, are points of access to the system where clients are usually connected [3][9][12][14].

In [6] we proposed an architecture for a fully distributed VoD system (called Double P-Tree) which, in addition to supporting a large number of concurrent streams, allows for the distribution of network traffic in order to minimize the network's bandwidth requirements. This is achieved by distributing both the servers as well as the clients throughout the topology, avoiding the concentration of communication traffic on the last level of the hierarchy (head-end). It is demonstrated through an analytical study that this distributed architecture is fault-tolerant and guarantees unlimited and low-cost growth for a large-scale VoD system.

In this paper, we focus on the design aspects of the Double P-Tree architecture with the view to optimizing its performance and to supporting a greater streaming capacity. We concentrate particularly on two aspects: incorporating a video-mapping mechanism to minimize service distance, and the proposal of traffic-balancing policies that allow a reduction in network bandwidth requirements for the system. These proposed policies have been evaluated through several simulation experiments and the results have shown significant improvement in the Double P-Tree architecture's performance. In addition, on the one hand we study the influence of the architecture's connectivity in improving the efficiency of multicast policies in distributed systems, and on the other hand we analyze the proxy storage capacity.

The remainder of this paper is organized as follows: in section 2, we first give an overview of the Double P-Tree architecture and we describe some topics related to its implementation. In section 3, we propose some techniques related to video placement and traffic balancing. Performance evaluation is shown in section 4 and, finally, in the last section, we indicate the main conclusions to be drawn from our results.

2 Distributed VoD Architecture

Fig.1a depicts the architecture of the proposed DVoD system. This architecture is designed as a network with a tree topology, with individual small servers (proxies) as the nodes, and network links as the edges of the hierarchy. Nodes are assumed to be able to store a limited number of videos and stream a finite number of videos. Meanwhile, networks links are expected to guaranteed the specific QoS requirements of video communications. A brief description of the system architecture is given below.

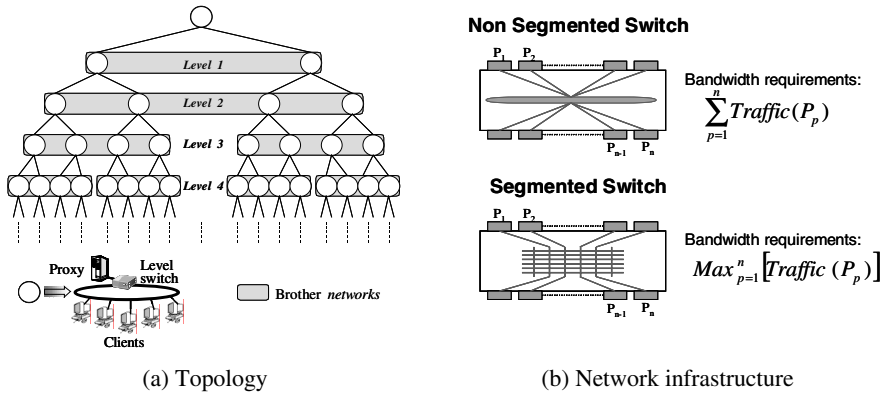


Fig. 1. Double P-Tree Architecture

2.1 Network Topology

For the network topology we have selected a fully distributed topology based on proxies. The structure of this topology consists of a series of levels, in accordance with the number of local networks and the order of the tree. Each hierarchy level is made up of a series of local networks with its local-proxy and clients that forms the following tree level. To improve topology connectivity, several local networks (named brother networks) from the same level are interconnected, increasing the number of adjacent local networks without changing the topology size or last level width. This new architecture is named Double P-Tree because the brother networks are joined in a second tree within the original tree topology [6].

In order to reduce network bandwidth requirements the network infrastructure is designed using segmented switches in local networks. Fig 1b, shows network bandwidth requirements for non-segmented and segmented switches. This selection is based on the fact that in segmented switches, every port (P_i) has an Independent-bandwidth, and therefore, it is only necessary to have enough bandwidth in order to support the maximum traffic from all ports. Segmented switches allow the reduction of switch-bandwidth requirements if traffic is distributed among different ports.

Double P-Tree architecture can make better use of segmented switches due to its network traffic being distributed among different sources. A possible drawback of this utilization is that topology-port traffic (ports used to implement the topology) and server-port (port used to connect proxy-server) could be unbalanced when the proxy load is centralized in one server-port. In order to solve this unbalance (increasing the bandwidth requirements), the architecture connects the proxies to local-networks using several ports.

2.2 Proxy Server

The simple inclusion of a hierarchical system with proxies does not, in itself, obtain improvements in the system's scalability or efficiency: since, as all the proxies are

caching the same videos, if a proxy cannot serve a request from its client, then it is also very probable that none of the other proxies will be able to serve this request, and the solution will then require accessing the main server. We therefore need to use a new proxy organization and functionality to increase the hit probability as the request climbs the various levels on the tree. This proposal lies in dividing the storage space available within proxies into two parts: one of these will continue functioning as a cache, storing the most requested videos; the other proxy space will be used for making a distributed mirror of system videos [6].

In order to provide True VoD we have concentrated on multicast transmission techniques [7][10]. These techniques can greatly reduce server I/O and network bandwidth. But with them, it is difficult to implement VCR functions since a dedicated channel is not allocated to each user. Whenever a user tries to play the VCR functions he will disjoin the multicast channel, and some new resources that have not been planned before must then be reserved and assigned to him. These resources will be used to meet the VCR functions and/or to provide a unicast channel so that the user can continue with the normal playback. Several ideas have been proposed to solve this problem basically reserving some channels for these specific actions [5][11].

Our implementation for the VCR functions does not reserve specific channels and is based on the observation that there are periods of times during which network bandwidth is under-utilized. During these periods, the proxy server sends more video in advance (pre-fetching) to an appropriate client's buffer. Whenever a user invokes a VCR action, the resources that have been assigned before the peak time are recovered in favor of this VCR request.

Another important element that affects the proxy performance is the proxy file system. Proxy servers that implement conventional file systems have been designed to reduce load on servers as well as client access time [4][13]. Nevertheless for continuous media with soft real-time constraints, typical file systems based on best-effort paradigm are inadequate for achieving this new requirement. Proxy servers can provide performance guarantees to applications only in conjunction with an operating system that can allocate resources in a predictable manner.

In our case, the most representative workload is the updates and removes in the caching and mirroring subsystem; consequently, the disk broker must employ placement policies that minimize fragmentation of disk space resulting from frequent writes and deletes. In order to obtain the best performance from the disk driver, track-buffer techniques are used. These techniques eliminate rotational delays in reads and obtain maximum performance on write operations.

3 Architecture Design Issues

In order to implement an efficient DVoD architecture several challenging research problems have to be solved to allow an efficient management of network and the services. Some of these problems are related with the subjects of reducing service distance and balancing communication traffic. In this section we propose some policies to accomplish these goals.

Table 1. Performance of Video-Mapping Heuristic for Double P-Tree

Mirror Distribution		Effective bandwidth	Mean service distance
Unicast	sequential	11.001 Mb/s	1,80
	heuristic	12.849 Mb/s	1,747
Multicast	sequential	14.913 Mb/s	1,80
	heuristic	15.951 Mb/s	1,747

3.1 Videos Mapping on Distributed Mirror

The main factor that penalizes DVoD architectures performance, measured as the number of concurrent clients supported by the system (effective bandwidth), is the over-bandwidth required due to requests that cannot be served locally. In this case, a remote service requires: local bandwidth in the remote server, bandwidth in the server-port in the switch, bandwidth in the remote switch-topology port, bandwidth in the local switch-topology port and finally bandwidth in the client switch-port. A good approximation to evaluate this over-bandwidth is mean service distance (the distance needed to reach all movies from every node in the system).

In particular, Double P-Tree mean service distance is affected by diverse factors, such as topology connectivity, proxy storage distribution between caching and mirroring, and mirror-videos mapping in proxies. The first and the second issues were analyzed in a previous paper [6], and the last one is studied below.

Given that it is too complex achieve a optimal video distribution, we have developed a heuristic to choose which videos need to be mapped in every proxy-mirror. This heuristic consists of calculating, for each proxy within the architecture, the minimum distance where we can find each of the movies of the system-repository (taking into account videos already mapped in the previous proxies). Then, in order to minimize the mean service-distance, we always choose those videos that are stored in the proxy-mirrors furthest from this proxy. In the case of there being various videos at the same distance, the most popular are then selected.

Table 1 shows the mean service distance and effective bandwidth obtained by the heuristic and a sequential distribution of videos based simply on assigning a group of videos to each one of the proxies in a sequential manner. These results use the simulation parameters given in section 4, taking unicast and multicast policies into account. As we can see, the heuristic reduces mean-service distance from 1.80 to 1.74, which allows for an improvement in system performance ("Effective Bandwidth" column) of 7% and 14% for unicast and multicast, respectively.

3.2 Traffic Balancing Policies

In architectures using segmented switches, the bandwidth requirements for a network depends on the maximum traffic supported by any of its ports. Therefore, it is very

Table 2. Performance of Traffic Balancing Policies for Double P-Tree

Traffic Balancing Policy		Effective bandwidth	Mean Distance	Imbalance
Unicast	Unbalance	12.849 Mb/s	1,747	56,44%
	Mirror Balanced	12.240 Mb/s	1,769 ⁽³⁾	55,61% ⁽¹⁾
	Traffic Balancing	15.653 Mb/s	1,747	30,45% ⁽⁵⁾
Multicast	Unbalance	15.951 Mb/s	1,732	78,27%
	Mirror Balanced	15.630 Mb/s	1,769 ⁽⁴⁾	71,70% ⁽²⁾
	Traffic Balancing	19.700 Mb/s	1,747	48,07% ⁽⁶⁾

important to balance port traffic in order to reduce bandwidth requirements and to increase system performance.

In Double P-Tree architecture, most loaded ports are of the server and topology ports. Server ports can easily be balanced because the proxy server can choose at any time through which port a request is attended.

On the other hand, balancing topology ports is more difficult because their load depends on video placement on distributed mirrors. For example, the links connecting proxy servers that map the most popular videos would be more overloaded than others, producing a traffic imbalance.

To achieve traffic-load balancing, we have studied two different approximations:

- Mirror Balanced.

An initial approximation to avoid imbalance is by building the distributed mirror in more balanced way. This objective can be achieved tuning the previous mapping heuristic because we do not always select the most popular videos; rather, we choose a mixture of highly and less popular videos.

- Dynamic traffic balancing.

However, balancing through mirror distribution has a very limited maneuverability and cannot easily adapt to changes in traffic patterns or video access frequency. Therefore we have proposed another more dynamic policy for traffic load balancing. As the imbalance problem only appears with remote requests (which are the only ones that use topology ports), when a request cannot be attended, the local proxy also receives information about traffic from all alternative sources (i.e. proxies that have a copy of requested video and enough resources to attend the request). Using this information, and if there are two or more alternative paths (meaning some video replication), our balancing policy always chooses the least-loaded path in order to balance traffic.

Using the simulation parameters given in section 4, in Table 2, we show the results obtained with these balance policies. As we can see, compared with imbalance, the mirror balanced policy is successful in reducing imbalance^{1,2}, but this reduction is not enough to compensate for the rise in service distance^{3,4}. In contrast, the results obtained with the dynamic traffic load balancing policy are much better, decreasing imbalance significantly^{5,6} (without affecting service distance) and increasing performance by around 24% (15.951Mbs and 19.700Mbs as against 12.849Mbs and 15.653Mbs, for unicast and multicast respectively).

4 Performance Evaluation

In this section, we show the simulation results for Double P-Tree architecture and contrast these with other distributed architectures. We conducted several experiments to 1) evaluate the effect of multicast, and 2) study the effect of proxy storage capacity on system behavior.

Table 3. Simulation Parameters

Parameter	Value	Parameter	Value
• Number of videos	100	• Multicast technique	Patching
• Video length	90 minutes	• Client buffer size	5 minutes
• Local networks	63	• Request rate (λ)	10 req/min by net
• Network bandwidth	100 Mb/s	• Poisson distribution	$p_i = \frac{\lambda_i^k}{k!} e^{-\lambda_i} \quad (1)$
• Server bandwidth 1server port (1Sp)	100 Mb/s		
• Server bandwidth 3server port (3Sp)	300 Mb/s	• Zipf distribution	$p_x = \frac{1}{x^z \cdot \sum_{i=1}^{S_v} 1/i^z} \quad (2)$
• Proxy capacity	20 videos		

4.1 Simulation Environment

To guide this objective, we have designed and implemented an object-oriented VoD simulator. The main parameters of the simulation environment are summarized in Table 3. In all studies, we use architectures with 63 local networks (6 levels in Double P-Tree topology) using 100 Mbps segmented switches. The request inter-arrival time is generated by the simulation of a Poisson distribution¹ with a mean of $1/\lambda$, where λ is the request arrival rate in every local network in the VoD system. The selection of the video is modeled with a Zipf distribution², with a skew factor of 0.7 (z), which models the popularity of rental movies [1].

4.2 Effect of Multicast on Distributed VoD Architectures

In this section, we evaluate DVoD system performance, using effective bandwidth (number of users attended * 1.5Mbs) as the main metric. This study allows us to evaluate the maximum streaming capacity for different architectures: Independent servers, one level proxies and Double P-Tree (using heuristic mapping and dynamic traffic balancing policy), for both unicast and multicast techniques.

In order to obtain the results, plotted in Fig 2, we have assigned an aggregate network bandwidth of 6.300Mbs and an aggregate sever bandwidth of 6.300Mbs (with 1 server port) or 18.900Mbs (with 3 Server ports). To obtain the maximum system streaming capacity, we saturated the system using a high request ratio (10 req/min by network) and simulated the system behavior until the aggregate bandwidth is ex-

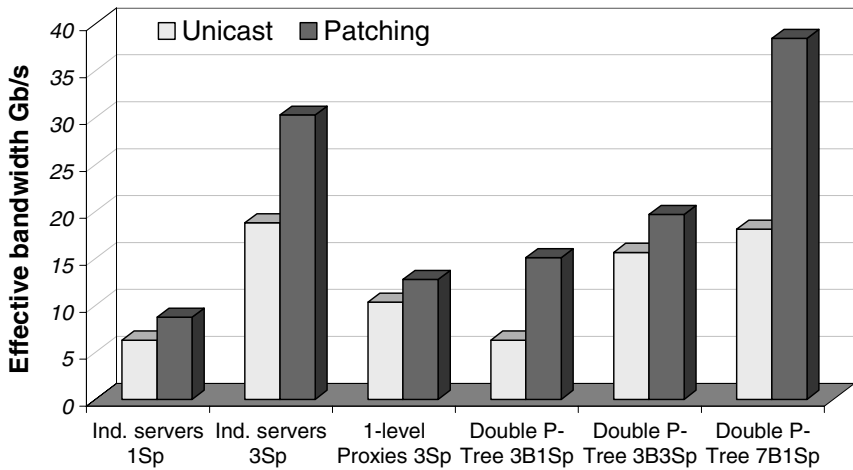


Fig. 2. Performance in Distributed VoD Architectures

hausted. When there is no bandwidth available, the system achieves its maximum streaming capacity and we evaluate the system performance (effective bandwidth using unicast and patching [12]).

Using unicast, we can see that the Independent servers with 3Sp is the architecture that obtains the best results, achieving the theoretical maximum stream capacity of 19Gbs (63 networks * 100Mbs * 3Sp). Meanwhile Double P-Tree with 7 brothers (7B) and 3Sp obtains an effective bandwidth of 18.2Gbs (4% less), due to the additional bandwidth required to attend distributed requests and lower storage requirements (20 as against 100 videos in every proxy). However, this underperformance is less than expected according to the criteria of mean service distance between Independent servers (1) and Double P-Tree architectures (1,747 according to Table 2). This result demonstrates the strength of our architecture in distributing and balancing traffic among topology ports in order to reduce network requirements. Also, as we can see, our architecture is better than 1-level proxies architecture (improved by 75%).

Double P-Tree architecture exploits its characteristics to realize its potential advantage when client streams are shared using a multicast technique (Patching, in our case). In this more realistic scenario, Double P-Tree is the best solution, improving Independent servers by 27% (38.4Gbs as against 30Gbs) and one-level proxies by 200% (38.4Gbs as against 12.7Gbs).

The principal argument for this improvement is that the Double P-Tree has a better connectivity than the Independent server and one-level architectures. This better connectivity means that, in Double P-Tree, mirror video streams can potentially be shared among requests coming from all adjacent networks, multiplying the sharing probability by topology connectivity. Meanwhile, in low interconnected architectures, potential stream sharing is limited only to local requests.

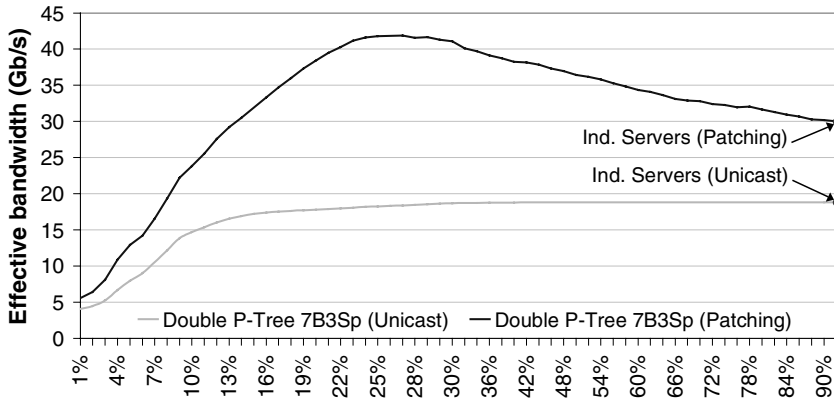


Fig. 3. Double P-Tree Effective Bandwidth & Proxy Storage Capacity

4.3 Effect of Proxy Storage Capacity

In this section, we evaluate the effect of proxy storage capacity on Double P-Tree performance. In Fig. 3, we can first see that a small proxy storage leads to low performance due to large service distance being required to attend remote requests, and to its over-bandwidth requirements.

With only storage for 15% of system videos, Double P-Tree performance (with patching) is equivalent to Independent servers performance, but uses 6 times less storage. Also, we notice that the highest performance is obtained with proxy storage of around 25%. In this case, Double P-Tree performs Independent servers in more than 38% (41.5Gbs against 30Gbs). From this point, we can observe that in the measure that storage capacity grows, performance decreases until reaching Independent server performance, in which case the proxy has enough capacity to store a full video catalog copy.

Why do more resources give less performance? The reasons for this interesting effect can be explained by the fact that, when Double P-Tree proxies have a lot of storage (more than 30%), their architectural behavior is very similar to that of Independent server systems. In this case, proxy mirrors have enough storage to reach all videos at distance-2 mirrors, therefore all remaining storage is assigned for caching.

Increasing proxy-cache size increases the number of requests attended locally, creating two consequences. First, server ports have more load, leading to network traffic unbalancing and a faster network saturation. Second, there are videos with a medium access pattern that were previously managed under mirroring scheme. Proxy-mirrors, where these were mapped, centralized all requests coming from adjacent nodes, improving stream sharing. If we now place these videos in the cache (replicating them in all proxies), we are decreasing access frequency for every video copy, reducing both the stream-sharing probability and system performance. This result clearly demonstrates the goodness of distributed mirroring as against several full mirror replications.

5 Conclusions

This paper deals with two decisive aspects for DVoD architectures performance: video placement policies in distributed mirrors and network-traffic balancing policies. The proposed policies attain a reduction in mean service distance and minimize network requirements for the Double P-Tree architecture.

Simulation results show that proposed policies substantially increase the number of concurrent clients who can be served by the system. The video mapping heuristic achieves an improvement of 7%, while the dynamic traffic-balancing policy yields an additional increase of 26%. These results clearly demonstrate the importance of network-traffic balancing policies as a fundamental instrument in diminishing the network bandwidth requirements in DVoD systems.

On the other hand, we have also shown the importance of topology connectivity in DVoD systems (in particular, in the Double P-Tree) in order to improve multicasting performance. The Double P-Tree using multicasting and similar resources clearly outperforms classical DVoD architectures, namely, Independent servers (by 38%) and one-level proxies (by 200%).

Finally, we have demonstrated that full mirror replication in every local network (as in Independent servers) not only requires more storage but also achieves a poorer performance in comparison to distributed mirroring (Double P-Tree).

References

1. S.A. Barnett and G. J. Anido, "A cost comparison of distributed and centralized approaches to video-on-demand," IEEE Journal on Selected Areas in Communications, vol. 14, pp. 1173–1183, August 1996.
2. S.-H. G. Chan and F. Tobagi, "Caching schemes for distributed video services", in Proc. of IEEE Int'l Conference on Communications (ICC'99), Canada, June 1999, pp. 994–1000.
3. S.-H. G. Chan and F. Tobagi, "Distributed Servers Architecture for Networked Video Services", in IEEE/ACM Transactions on Networking, Vol. 9, No. 2 April 2001.
4. A. Chankhunthod, P.B. Danzang, C. Neerdaels, M. F. Schwartz, and K.J. Worrell. "A Hierarchical Internetwork Object Cache". In Proceeding of the 1996 USENIX Technical Conference, San Diego, CA, January 1996.
5. J-M Choi, S-W Lee, K-d Chung, "A Multicast Scheme for VCR Operations in a Large VOD system", ICPADS 2001:555–561.
6. F. Cores, A. Ripoll, E. Luque, "Double P-Tree: A Distributed Architecture for Large-Scale Video-on-Demand", Euro-Par 2002, LNCS 2400, pp. 816–825, Aug. 2002.
7. A. Dan, D. Sitaram, and P. Shahabuddin, "Dynamic batching policies for an on-demand video server," Multimedia Systems 4, pp. 112–121, June 1996.
8. H. Fabmi, M. Latif, S. Sedigh-Ali, A. Ghafoor, P. Liu, L.H. Hsu, "Proxy servers for scalable interactive video support", IEEE Computer, Vol. 34 Iss. 9, pp. 54–60 Sept. 2001.
9. C. Griwodz. "Wide-Area True Video-on-Demand by a Decentralized Cache-based Distribution Infrastructure." PhD dissertation, Darmstadt Univ. of Technology, Germany, Apr. 2000.
10. K. A. Hua, Ying Cai and S. Sheu, Patching: A multicast technique for true video-on-demand services, ACM Multimedia'98, pages 191–200.

11. J. Y. B. Lee, "On a Unified Architecture for Video-on-Demand Services", IEEE Transactions on Multimedia, Vol. 4, No. 1, March 2002.
12. Cyrus Shahabi, Farnoush Banaei-Kashani, "Decentralized Resource Management for a Distributed Continuous Media Server", IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 7, July 2002
13. "Squid Internetwork Object Cache Users Guide". Available on line at <http://squid.nlanr.net>, 1997.
14. Xiaobo Zhou, R. Luling, Li Xie, "Solving a Media Mapping Problem in a Hierarchical Server Network with Parallel Simulated Annealing", Procs. 2000 International Conference on Parallel Processing, pp. 115–124, 2000.

On Transmission Scheduling in a Server-Less Video-on-Demand System¹

C.Y. Chan and Jack Y.B. Lee

Department of Information Engineering
The Chinese University of Hong Kong
{cychan2, yblee}@ie.cuhk.edu.hk

Abstract. Recently, a server-less video-on-demand architecture has been proposed which can completely eliminate costly dedicated video servers and yet is highly scalable and reliable. Due to the potentially large number of user hosts streaming video data to a receiver for playback, the aggregate network traffic can become very bursty, leading to significant packet loss at the access routers (e.g. 95.7%). This study tackles this problem by investigating two new transmission schedulers to reduce the traffic burstiness. Detailed simulations based on Internet-like network topologies show that the proposed Staggered Scheduling algorithm can reduce packet loss to negligible levels if nodes can be clock synchronized. Otherwise, a Randomized Scheduling algorithm is proposed to achieve consistent performance that is independent of network delay variations, and does not require any form of node synchronization.

1 Introduction

Peer-to-peer and grid computing have shown great promises in building high-performance and yet low cost distributed computational systems. By distributing the workload to a large number of low-cost, off-the-shelf computing hosts such as PCs and workstations, one can eliminate the need for a costly centralized server and at the same time, improve the system's scalability. Most of the current works on grid computing are focused on computational problems [1], and on the design of the middleware [2]. In this work, we focus on another application of the grid architecture – video-on-demand (VoD) systems, and in particular, investigate the problem of transmission scheduling in such a distributed VoD system.

Existing VoD systems are commonly built around the client-server architecture, where one or more dedicated video servers are used for storage and streaming of video data to video clients for playback. Recently, Lee and Leung [3] proposed a new server-less VoD architecture that does not require dedicated video server at all. In this

¹ This work was supported in part by the Hong Kong Special Administrative Region Research Grant Council under a Direct Grant, Grant CUHK4211/03E, and the Area-of-Excellence in Information Technology.

server-less architecture, video data are distributed to user hosts and these user hosts cooperatively serve one another's streaming requests. Their early results have shown that such a decentralized architecture can be scaled up to hundreds of users. Moreover, by introducing data and capacity redundancies into the system, one can achieve system level reliability comparable to or even exceeding those of high-end dedicated video servers [4].

Nevertheless, there are still significant challenges in deploying such server-less VoD systems across the current Internet. In particular, Lee and Leung's study [3] did not consider the issue of network traffic engineering. With potentially hundreds or even thousands of nodes streaming data to one another, the aggregate network traffic can become very bursty and this could lead to substantial congestion at the access network and the user nodes receiving the video data. Our study reveals that packet loss due to congestion can exceed 95% if we employ the common first-come-first-serve algorithm to schedule data transmissions.

In this study, we tackle this transmission scheduling problem by investigating two transmission scheduling algorithms, namely Staggered Scheduling and Randomized Scheduling. Our simulation results using Internet-like network topologies show that the Staggered Scheduling algorithm can significantly reduce packet loss due to congestion (e.g. from 95.7% down to 0.181%), provided that user nodes in the system are clock-synchronized using existing time-synchronization protocols such as the Network Time Protocol [5]. By contrast, the Randomized Scheduling algorithm does not need any form of synchronization between the user nodes albeit does not perform as well. Nevertheless, the performance of the Staggered Scheduling algorithm will approach that of the Randomized Scheduling algorithm when the network delay variation or clock jitter among nodes are increased. In this paper, we present these two scheduling algorithms, evaluate their performance using simulation, and investigate their sensitivity to various system and network parameters.

2 Background

In this section, we first give a brief overview of the server-less VoD architecture [3] and then define and formulate the transmission scheduling problem. Readers interested in the server-less architecture are referred to the previous studies [3-4] for more details.

2.1 Server-Less VoD Architecture

A server-less VoD system comprises a pool of fully connected user hosts, or called nodes in this paper. Inside each node is a system software that can stream a portion of each video title to as well as playback video received from other nodes in the system. Unlike conventional video server, this system software serves a much lower aggregate bandwidth and thus can readily be implemented in today's set-top boxes (STBs) and

PCs. For large systems, the nodes can be further divided into clusters where each cluster forms an autonomous system that is independent from other clusters.

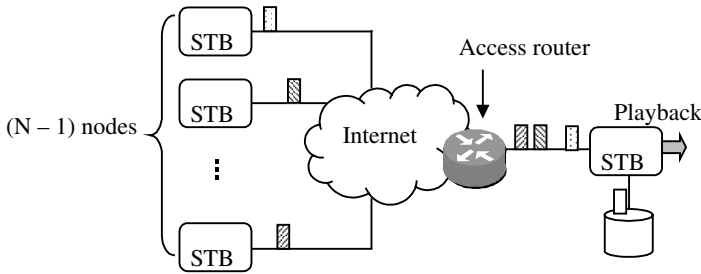


Fig. 1. A N -node server-less video-on-demand system

For data placement, a video title is first divided into fixed-size blocks and then equally distributed to all nodes in the cluster. This node-level striping scheme avoids data replication while at the same time share the storage and streaming requirement equally among all nodes in the cluster.

To initiate a video streaming session, a receiver node will first locate the set of sender nodes carrying blocks of the desired video title, the placement of the data blocks and other parameters (format, bitrate, etc.) through the directory service. These sender nodes will then be notified to start streaming the video blocks to the receiver node for playback.

Let N be the number of nodes in the cluster and assume all video titles are constant-bit-rate (CBR) encoded at the same bitrate R_v . A sender node in a cluster may have to retrieve video data for up to N video streams, of which $N - 1$ of them are transmitted while the remaining one played back locally. Note that as a video stream is served by N nodes concurrently, each node only needs to serve a bitrate of R_v/N for each video stream. With a round-based transmission scheduler, a sender node simply transmits one block of video data to each receiver node in each round. The ordering of transmissions for blocks destined to different nodes becomes the transmission scheduling problem.

2.2 Network Congestion

In an ideal system model, video data are transmitted in a continuous stream at a constant bit-rate to a receiver node. However, in practice data are always transmitted in discrete packets and thus the data stream is inherently bursty. In traditional client-server VoD system this problem is usually insignificant because only a single video server will be transmitting video data to a client machine and thus the data packets will be transmitted at constant time intervals. By contrast, video data are distributed across all nodes in a server-less VoD system and as a result, all nodes in the system participate in transmitting video data packets to a node for playback. If these data transmissions are not properly coordinated, a large number of packets could arrive at the receiver node's access network at the same time, leading to network congestion.

For example, Fig. 2 depicts a straightforward transmission scheduler - On Request Scheduling (ORS), which determines the transmission schedule based on the initial request arrival time. Specifically, a node transmits video data in fixed-duration rounds,

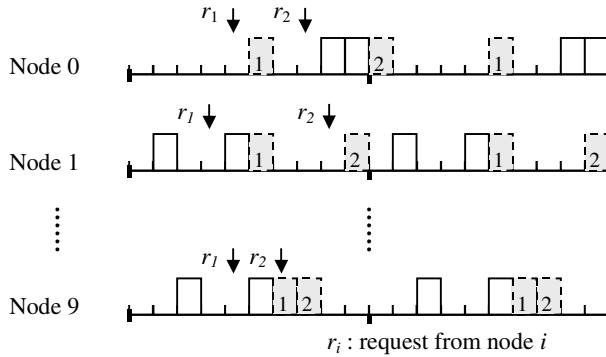


Fig. 2. Transmission schedules generated by the On Request Scheduling algorithm

with each round further sub-divided into N timeslots. The node can transmit one Q -byte data packet in each time slot. Let T_r be the length of round and $T_s = Q/R_v$ be the length of a timeslot, then with a video bit-rate R_v we can compute T_r and T_s from $T_r = NT_s = NQ/R_v$.

When a node initiates a new video streaming session, it will send a request to all nodes in the system. A node upon receiving this request will reserve an available timeslot in a first-come-first-serve manner to begin transmitting video data for this video session. For example, consider the scenario in Fig. 2 where there are 10 timeslots per round. Request r_1 reaches node 0, and is assigned to slot 5. On the other hand, when request r_2 reaches node 0 the upcoming slot has already been assigned to another stream and in this case the request will be assigned to the first available slot (i.e. slot 0). Note that for simplicity we do not consider disk scheduling in this study and simply assumed that data are already available for transmission.

It is easy to see that this ORS algorithm can minimize the startup delay experienced by end users as well as spread out data transmissions destined for different receivers to reduce burstiness of the aggregate network traffic leaving a node. While this algorithm may work well in traditional client-server VoD systems, its performance is unacceptably poor in a server-less VoD system. In our simulation of a 500-node system with $Q=8\text{KB}$ and $R_v=4\text{Mbps}$, this ORS algorithm can result in over 95% packet losses due to congestion in the access network.

The fundamental problem here is due to the very large number of nodes in the system and the fact that data transmissions are packetized. With the ORS algorithm, a new video session will likely be assigned to timeslots that are temporally close together. Thus once transmission begins, all nodes in the system will transmit video data packets to the receiver node in a short time interval, and then all cease transmission for T_r seconds before transmitting the next round of packets. While the average aggregate transmission rate is still equal to the video bit-rate, the aggregate

traffic is clearly very bursty and thus leads to buffer overflows and packet drops at the access network router connecting the receiver node.



Fig. 3. Transmission schedules generated by the Staggered Scheduling algorithm

3 Transmission Scheduling

To tackle the network congestion problem discussed earlier, we investigate in this section two transmission scheduling algorithms, namely the Staggered Scheduling (SS) and the Randomized Scheduling (RS) algorithms.

3.1 Staggered Scheduling (SS)

As mentioned in Section 2.2, the ORS algorithm can reduce the burstiness of the network traffic leaving a sender node, but the combined traffic from multiple sender nodes can still be very bursty. The fundamental problem is that the ORS algorithm attempt to schedule a video session to nearby timeslots in all nodes and thus rendering the combined traffic very bursty.

This observation motivates us to investigate a new scheduler – Staggered Scheduling, which schedules a video session to nodes in non-overlapping timeslots as shown in Fig. 3. Specifically, the timeslots are all pre-assigned to different receiver nodes. For node i serving node j data will always be transmitted in timeslot $(j-i-1) \bmod N$. For example, in Fig. 3 node 9 is served in timeslot 8 in node 0 while it is served in timeslot 7 in node 1. Thus the timeslot assignment of a video session forms a staggered schedule and hence the name for the algorithm.

Assuming the nodes are clock-synchronized, then transmissions from different nodes to the same receiver node will be separated by at least T_s seconds, thus eliminating the traffic burstiness problem in ORS. Nevertheless, the need for clock-synchronization has two implications. First, as clocks in different nodes cannot be precisely synchronized in practice, the performance of the algorithm will depend on the clock synchronization accuracy. Second, depending on the application, the

assumption that all nodes in the system are clock-synchronized may not even be feasible.

Table 1. Initial assignment of the parameters of the data loss problem model

Parameters	Values
Cluster size	500
Video block size	8KB
Video bitrate	4Mbps
Access network bandwidth	$1.1R_c$
Router buffer size (per node)	32KB
Mean propagation delay	0.005s
Variance of propagation delay	10^{-6}
Mean router queueing delay	0.005s
Variance of clock jitter	10^{-6}

3.2 Randomized Scheduling (RS)

Staggered Scheduling attempts to perform precise control of the data transmissions to smooth out aggregate network traffic. Consequently, close synchronization of the nodes in the system is essential to the performance of algorithm. In this section, we investigate an alternative solution to the problem that does not require node synchronization.

Specifically, we note that the fundamental reason why aggregate traffic in ORS is bursty is because data transmission times of all the sender nodes are highly correlated. Thus if we can decorrelate the data transmission times then the burstiness of the traffic will also be reduced. This motivates us to investigate a new Randomized Scheduling algorithm that schedules data transmissions for a video session in random timeslots. Moreover, the randomly assigned timeslot is not fixed but randomized in each subsequent round to eliminate any incidental correlations.

It is easy to see that under Randomized Scheduling, one does not need to perform clock synchronization among nodes in the system. Each node simply generates its own random schedule on a round-by-round basis. We compare the performance of Staggered Scheduling and Random Scheduling in the next section.

4 Performance Evaluation

We evaluate and compare the three scheduling algorithms studied in this paper using simulation. The simulator simulates a network with 500 nodes. To generate a realistic network topology, we implement the extended BA (EBA) model proposed by Barabási et al. [6] as the topology generator, using parameters measured by Govindan et al. [7].

To model access routers in the network, we assume an access router to have separate buffers for each connected node. These buffers are used to queue up incoming data packets for transmission to the connected node in case of bursty traffic. When the buffer is full, then subsequent arriving packets for the node will be discarded and thus resulting in packet loss.

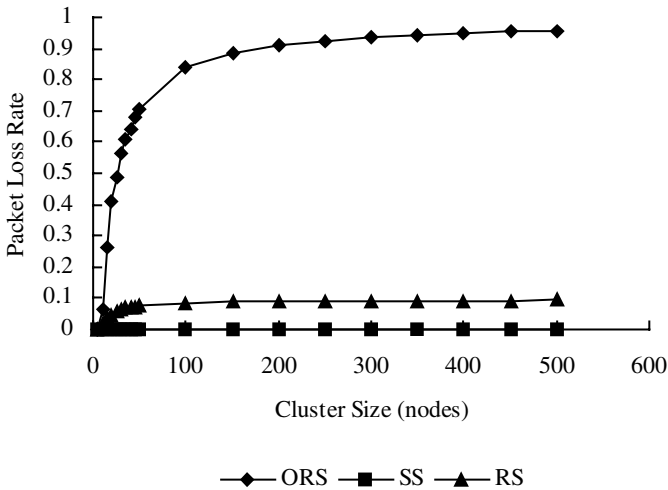


Fig. 4. Comparison of packet loss rate versus cluster size for ORS, SS, and RS

To model the network links, we separate the end-to-end delay into two parts, namely, propagation delay in the link and queueing delay at the router. While the propagation delay is primarily determined by physical distance, queueing delay at a router depends on the utilization of the outgoing links. We model the propagation delay as a normally distributed random variable and the queueing delay as an exponentially-distributed random variable [8].

To model clock synchronization protocol, we assume that the clock jitter of a node, defined as the deviation from the mean time of all hosts, is normally-distributed with zero mean. We can then control the amount of clock jitter by choosing different variances for the distribution.

Table 1 summarizes the default values of various system parameters. We investigate in the following sections the effect of four system parameters, namely cluster size, router buffer size, clock jitter, and queueing delay on the performance of the three scheduling algorithms in terms of packet loss rate. Each set of results is obtained from the average results of 10 randomly generated network topologies.

4.1 Packet Loss Rate versus Cluster Size

Fig. 4 plots the packet loss rate versus cluster size ranging from 5 to 500 nodes. There are two observations. First, the loss rate decreases rapidly at smaller cluster size and

becomes negligible for very small clusters. For example, for a 10-node cluster the loss rate is only 6.6%. This confirms that the traffic burstiness problem is unique to a server-less VoD system where the number of nodes is typically large. Second, comparing the three algorithms, On Request Scheduling (ORS), Staggered Scheduling (SS), and Randomized Scheduling (RS), ORS performs extremely poorly with loss rates as high as 95%, which is clearly not acceptable in practice. RS performs significantly better, with a loss rate approaching 9.3% when the cluster size is increased to 500. Finally, the SS algorithm performs best with 0.18% packet loss regardless of the cluster size, demonstrating its effectiveness in eliminating bursts in the aggregate traffic.

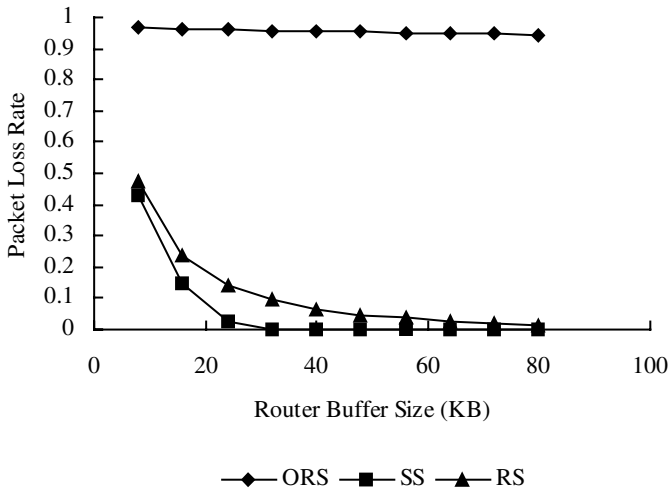


Fig. 5. Comparison of packet loss rate versus router buffer size for ORS, SS, and RS

4.2 Packet Loss Rate versus Router Buffer Size

Clearly, the packet loss rate depends on the buffer size at the access router. Fig. 5 plots the packet loss rate against router buffer sizes ranging from 8KB to 80KB. The packet size is $Q=8$ KB so this corresponds to the capacity to store one to ten packets. As expected, the loss rates for all three algorithms decrease with increases in the router buffer size. In particular, the performance of RS can approach that of SS when the router buffer size is increased to 80KB. Nevertheless, ORS still performs very poorly even with 80KB buffer at the routers and thus one cannot completely solve the problem by simply increasing router buffer size.

4.3 Packet Loss Rate versus Queueing Delay

On the other hand, delay variations in the network can also affect performance of the schedulers. To study this effect, we vary the routers' mean queueing delay from 0.0005 to 5 seconds and plot the corresponding packet loss rate in Fig. 6.

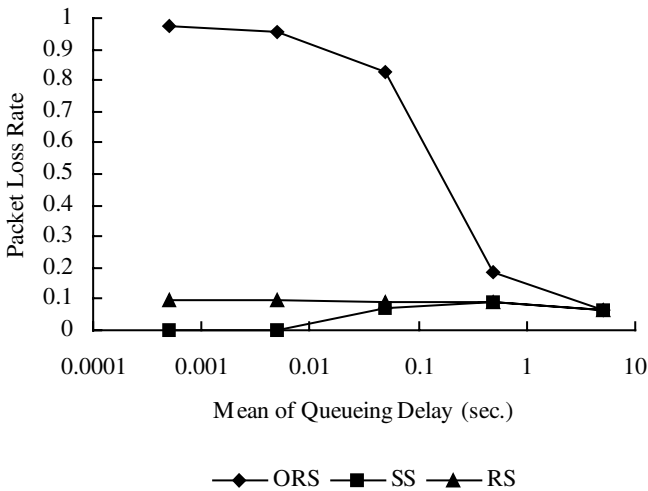


Fig. 6. Comparison of packet loss rate versus router queueing delay for ORS, SS, and RS

There are two interesting observations from this result. First, performance of the RS algorithm is not affected by changes in the mean queueing delay. This is because packet transmission times under RS are already randomized, and thus adding further random delay to the packet transmission times has no effect on the traffic burstiness.

Second, surprisingly performances of all three algorithms converge when the mean queueing delay is increased to 5 seconds. This is because when the mean queueing delay approaches the length of a service round (i.e. $T_s=8.192$ seconds), the random queueing delay then effectively randomize the arrival times of the packets at the access router and hence performances of both the ORS and SS algorithms converge to the performance of the RS algorithm.

This significance of this results is that transmission scheduling is effective only when random delay variations in the network are small compared to the service round length. Moreover, if the amount of delay variation is unknown, then the RS algorithm will achieve the most consistent performance, even without any synchronization among nodes in the system.

5 Conclusions

We investigated the transmission scheduling problem in a server-less VoD system in this study. In particular, for networks with comparatively small delay variations and with clock-synchronization, the presented Staggered Scheduling algorithm can effectively eliminate the problem of traffic burstiness and achieve near-zero packet loss rate. By contrast, the Randomized Scheduling algorithm can achieve consistent performance despite variations in network delay. More importantly, Randomized Scheduling does not require any form of node synchronization and thus is most suitable for server-less VoD systems that do not have any centralized control and management. Since the problem is defined in the packet level, we would expect these results can be easily applied to both stream-based and object-based media, given that the requested media is packed and sent in the scheduled slots.

References

1. A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly Press, USA, 2001.
2. M. Baker, R. Buyya, and D. Laforenza, "The Grid: International Efforts in Global Computing," *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, Rome, Italy, 31 July, 2000.
3. Jack Y. B. Lee and W. T. Leung, "Study of a Server-less Architecture for Video-on-Demand Applications," *Proc. IEEE International Conference on Multimedia and Expo.*, Lausanne, Switzerland, 26–29 Aug 2002.
4. Jack Y. B. Lee and W. T. Leung, "Design and Analysis of a Fault-Tolerant Mechanism for a Server-Less Video-On-Demand System," *Proc. 2002 International Conference on Parallel and Distributed Systems*, Taiwan, 17–20 Dec, 2002.
5. D. L. Mills, "Improved algorithms for synchronizing computing network clocks," *IEEE/ACM Transaction on Networks* 3, June 1995, pp. 245–254.
6. R. Albert, and A.-L. Barabási, "Topology of Evolving Networks: Local Events and Universality," *Physical review letters*, vol.85, 2000, p. 5234.
7. R. Govindan, and H. Tangmunarunkit, "Heuristics for Internet Map Discovery," *IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000, pp.1371–1380.
8. D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*, 3rd ed. New York: Wiley.

A Proxy-Based Dynamic Multicasting Policy Using Stream's Access Pattern

Yong Woon Park¹ and Si Woong Jang²

¹Department of Computer Science, Dong-eui Institute of Technology, san72, Yangjung Dong,
Pusanjin Ku, Pusan, Rep. of Korea
ywpark@dit.ac.kr

²Department of Computer Science, Dong-eui University, san24, Kaya Dong, Pusanjin Ku,
Pusan, Rep. of Korea
swjang@hyomin.dongeu.ac.kr

Abstract. This paper proposes a multicasting channel allocation policy in the proxy server that a multicasting channel is not allocated for all objects but for some objects whose access pattern satisfies some conditions. For an object, one unicasting channel is allocated from the origin server to the proxy server and subsequently the proxy server either unicasts or multicasts that object from the proxy server to the client(s) based on the number of active streams. The proxy server also caches each object partially or fully based on each object's access frequency : the cached portion of each object increases as its access frequency increases whereas the cached portion of each object decreases as its access frequency decreases.

1 Introduction

In the network based VoD systems, in order to service customers in a strictly on demand basis, VoD systems usually assigns a transmission stream to each client. So, the network resource requirement to transfer video streams over the network is overwhelming that for several hundred concurrent video streams, several giga bits per second of I/O and network bandwidth are required. Batching[1] is a stream service policy that a group of requests for the same object in a given time interval or batching window are collected and then serviced with one multicasting channel; the batching window size is defined either in terms of time or number of users[3]. So, the multicasting channel allocation scheme can reduce the transfer bandwidth. Moreover, caching media objects on the edge of the Internet become increasingly important in reducing the high start-up overhead and isochronous requirement as well[5]. It is especially effective in a batching environment with deterministic batching window. According to the simulation by [4], there is a considerable improvement in the cache hit ratio and mean disk I/O stream bandwidth reduction by adopting a caching policy. Proxy caching[2] has been widely known to reduce the Internet communication overhead but has mainly concerned with Web files. Some proxy caching schemes for continuous media objects are introduced but they are focused on reducing the initial latency or smoothing the burstiness of the VBR stream[8]. Partial segment caching of

media objects was proposed to be combined with a dynamic skyscraper broadcasting to reduce the media delivery cost from a remote server to regional servers[4]. The emphasis in [7] was on caching the initial segments of many media objects and relying on remote multicast delivery of the later segments, rather than fully caching fewer highly popular objects.

2 Multicasting Channel Allocation and Proxy Caching

Nodes connected to a LAN usually communicate over a broadcast network, while nodes connected to a WAN communicate via a switched network. In a broadcast LAN, a transmission from any one node is received by all the nodes on the network; thus, multicasting is easily implemented on a broadcast LAN. On the other hand, implementing multicasting on a switched network is quite challenging [6]. Hence, we assumed that in a LAN, both unicasting and multicasting facilities including proxy caching are available whereas in the WAN only unicasting is possible to transfer streams to the proxy server.

2.1 Multicasting Policy Base on Streams' Arrival Interval

1. A client sends a stream request to the proxy server.
2. The proxy server then immediately allocates one unicasting channel for that stream and transmits the requested object to the client. If the requested object is cached partially or not at all in the proxy server, then the proxy server sends a request on behalf of the client to the origin server and relays it to that client. At the same time, it caches the requested object partially or fully based on the caching status and access pattern of the requested object.
3. If there are more than two active unicasting streams for that object, the proxy server calculates the multicasting window size based on the streams' average arrival interval so as to minimize the number of the expected maximum concurrent streams.
4. If the calculated multicasting window size is smaller than the average arrival interval, those streams are grouped together into that multicasting window and a multicasting channel is allocated for that multicasting window. Otherwise, those streams will keep their already allocated unicasting channels.

2.2 Phased Proxy Caching

In our proposed caching policy, caching is done in three steps. Assume that the k^{th} request for object O_i and the j^{th} new multicasting window for O_i is generated.

1. First phase: In case O_i is not cached at all, as large initial portion of O_i as to hide the initial latency is prefetched
2. Second phase: In case O_i 's caching status is its first phase, part of O_i amounting to the j^{th} multicasting window size is cached.

3. Third phase: In this phase, caching is done only when $AVG_{arr}^{int}(o_i)_1^k > AVG_{arr}^{int}(o_i)_1^{k-1}$ ($AVG_{arr}^{int}(o_i)_1^k$ is the accumulated arrival rate from the first to the k^{th} streams for O_i); the remaining portion of O_i is cached step by step that as a result, some objects can be cached entirely to avoid server access.

3 Experimental Results

In this section, we measure the performance of the proposed dynamic multicasting policy and we use five different notations to express each algorithm. The *direct* means the case when only unicasting channels are used to service requests. The *cast* shows the result of our tests when only our proposed multicasting policy is applied. The *s+cast(cache size)* means the case when our proposed multicasting policy works with the caching policy where the caching step is limited to the second phase and the *t+cast(cache size)* means the case when our proposed caching policy works with the third phase caching policy. As the graphs in Figure 1 shows, our proposed multicasting channel policy shows about 50% of increased performance as compared with the case when all streams are serviced with unicasting channels without the proposed multicasting policy. Moreover, in case the multicasting policy works with our phased caching policy, it shows more enhanced performance than without caching policy; about 35% of decrease with respect to both the accumulated byte hits and peak time channel numbers. The performance gap between s+cast and t+cast is so small that it can be conjectured that in case of continuous media objects with the

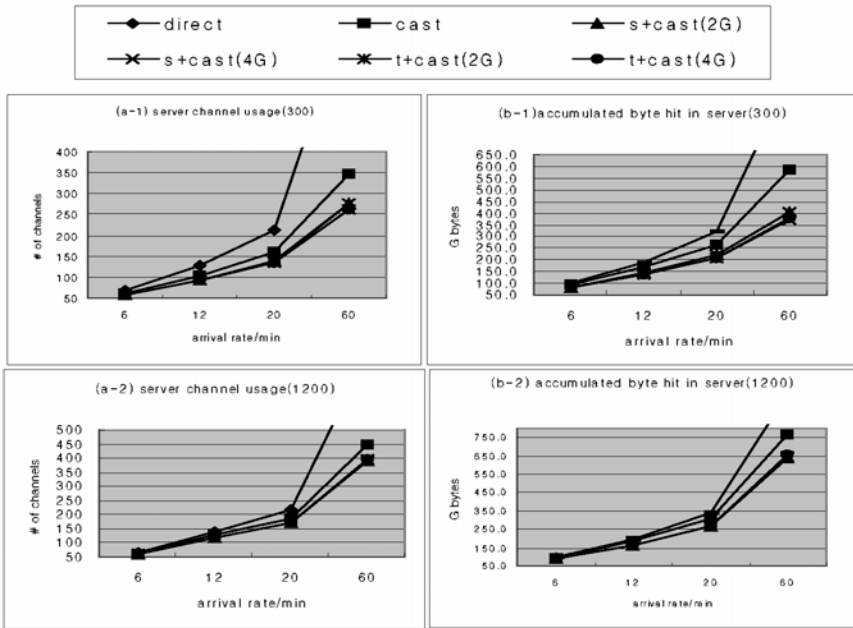


Fig. 1. Performance evaluation with different number of objects

multicasting policy, for an object, caching as much initial portion of that object as the length of the multicasting window is effective rather than caching the entire blocks of that object.

4 Conclusions and Future Research

In this paper, we proposed the access frequency-based dynamic multicasting policy based on the phased proxy caching for continuous media stream service. In the proposed caching policy, a multicasting channel for an object is started only after there are more than two active streams for that object. For proxy caching, each object is cached step by step to hide the initial latency as well as to manage cache space efficiently. With our proposed policy, the channel resource usage is reduced by 50%. It is known that when multicasting policy runs, instead of caching the entire or most of blocks of objects, caching as much portion of each object as the maximum multicasting window size is more effective. In addition, it is also known that increasing proxy capacity cannot reduce the initial latency when the arrival rate is low.

Acknowledgement. This work was supported by ECC(Electronic Ceramics Center) at Dong-eui University as RRC•TIC program which is financially supported by KOSEF(Korea Science and Engineering Foundation) under MOST(Ministry of Science and Technology), and ITEP(Korea Institute of Industrial Technology Evaluation and Planning) under MOCIE(Ministry of Commerce, Industry and Energy), and Busan Metropolitan City.

References

1. Aggarwal C. C., Wolf J.L., Yu P.S., "On Optimal Batching Policies for Video-on-Demand Storage Servers", Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, 1996, Page(s): 253–258
2. Aggarwal C., Wolf J.L., Yu P.S., "Caching on the World Wide Web", Knowledge and Data Engineering, IEEE Transactions on Volume: 11 1, Jan.-Feb. 1999 , Page(s): 94–1
3. Debasish Ghose, Hyoung Joong Kim, "Scheduling Video Streams in Video-on-Demand systems: A Survey", Multimedia Tools and Applications, 11, 167–195, 2000 Kluwer Academic Publishers
4. D. Eager, M. Ferris, and M. Vernon, "Optimized Regional Caching for On-Demand Data Delivery", in Proc. Multimedia Computing and Networking, Jan 1999
5. K.L. Wu, P.S. Yu, and J.L. Wolf, "Segment-based Proxy Caching of Multimedia Streams", Proceedings of 10th World Wide Web Conference, Elsevier Science, Amsterdam, 2001, pp. 36–44
6. Laxman H Sahasrabudhe, Biswanath Mukherjee, "Multicast Routing Algorithms and Protocols: A tutorial", IEEE Network Jan/Feb 2000 pp. 90–102
7. Rexford S., J. Towsley, "Proxy Prefix Caching for Multimedia Streams", INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE Volume: 3 , 1999 , Page(s): 1310–1319 vol.3
8. Zhi-Li Zhang, Du, D.H.C., Dongli S, Yuewei Wang, "A Network-Conscious Approach to End-to-End Video Delivery over Wide Area Networks Using Proxy Servers", INFOCOM '98. the 17th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE Volume: 2, 1998 , Page(s): 660–667 vol.2.

Topic 13

Theory and Algorithms for Parallel Computation

Christos Kaklamanis, Danny Krizanc, Pierre Fraigniaud, and
Michael Kaufmann
Topic Chairs

As theory of modelling and algorithms form the basis for parallel computing, the theory workshop is still a key part of Europar despite of the diversity of the topics of the conference. This year, the submissions mainly concentrate on algorithms to improve communication issues. Only a few contributions to other topics made it.

In more detail, Schmollinger presents a new algorithm for parallel radix sort emphasizing on the communication especially for unbalanced data and demonstrates considerable improvement compared to the traditional approach.

In a new paper on parallel list ranking, Sibeyn considers diverse algorithms under the aspect of global communication, and presents a new strategy and analysis when local communication is preferred.

The third paper by Laforest considers an interesting network design problem where a certain kind of graph spanner has to be constructed, and gives non-approximability results as well as efficient algorithms for trees.

Cordasi, Negro, Rosenberg and Scarano addresses the mapping problem of data structures to parallel memory modules in the paper “*c*-Perfect Hashing Schemes for Binary Trees, with Applications to Parallel Memories”.

A non-optimal but simple and elegant algorithm for parallel multiplication of large integers has been developed by Bunimov and Schimmler. Their parameter and analysis techniques are adapted from VLSI theory.

The paper “A Model of Pipelined Mutual Exclusion on Cache-Coherent Multiprocessors” by Takesue has a more practice-oriented topic. A new model of pipelined mutual exclusion is proposed and evaluated by diverse simulations.

Summarizing, we found the collection of papers satisfying and diverse enough to enlight several theoretical aspects of parallel computation. Finally, we wish to thank all the authors and the referees for their efforts.

Improving Communication Sensitive Parallel Radix Sort for Unbalanced Data

Martin Schmollinger

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
schmolli@informatik.uni-tuebingen.de

Abstract. Radix sort is an efficient method to sort integer keys on parallel computers. It is easy to parallelize and simple to implement. The main drawbacks of existing algorithms are load balancing problems and communication overhead. These problems are caused in data characteristics like data-skew and duplicates. There are several approaches how to parallelize the radix sort algorithm, which yield to reduce communication operations or to improve the load balance. If an algorithm has its focus on the optimization of the load balance then, its communication is inefficient. Otherwise, if the focus is on the communication minimization, then the algorithms are only efficient for well-distributed data. For the latter case, we will present an efficient improvement which helps to overcome the problems with unbalanced data characteristics. The suggested improvements are tested practically on a Linux-based SMP cluster.

1 Introduction and Related Work

Radix Sort is a method to sort a set of integer keys using their binary representation. The main idea is to sort the keys in several iterations. In each iteration the keys are sorted according to a certain number of bits (*radix* of size r), starting in the first iteration with the least significant bit. Assuming the keys consist of l bits, the algorithm needs l/r iterations to sort the keys. Detailed descriptions of sequential radix sort algorithms can be found in standard textbooks, see e.g. [13,10].

The main idea of the parallelization is that the processors are viewed as buckets. Roughly, each parallel iteration of radix sort consists of the following 3 steps. Each processor scans its local keys with a certain radix and stores them in the corresponding buckets. Then the sizes of the locally created buckets are exchanged by the processors in order to calculate a communication pattern for the buckets. The keys are exchanged according to the computed communication pattern. These steps are repeated until all bits of the keys were scanned. More detailed descriptions for shared-memory and distributed-memory machines can be found in [1,8].

The main problem with this approach is its irregularity in communication and computation which arise because of data characteristics like e.g. data skew or duplicates. Therefore, in [15] a load balanced parallel radix sort is presented. This algorithm splits the locally generated buckets with respect to balance the number of keys that have to be sent to each processor. Therefore, the resulting communication step is a real balanced all-to-all operation by the processors.

Until now, all mentioned algorithms need l/r key- and counter-communication steps. Especially for distributed-memory machines, where the communication is done over an interconnection network, this might be very time consuming. The communication and cache conscious radix sort algorithm (C3-Radix) [7] tries to improve this situation by starting the radix sort at the most significant bit. The intention of C3-Radix is to partition the data into several buckets which can be distributed equally among the processors. Therefore, the choice of the radix is very important. If the distribution of the buckets is not possible in a balanced way, the radix is enlarged and more fine-grained buckets are created. This is repeated until a good load-balance is achieved. Then the keys are sent with an all-to-all operation among the processors. The remaining task is to sort the buckets locally. For this step, the authors use a very efficient cache conscious radix sort algorithm which uses the fact that the data is already sorted for a certain number of bits. The algorithm has to communicate the keys only once, but because of the characteristics of the data there might be several iterations where the bucket counters have to be exchanged. Depending on the number of iterations and the size of the radix this can increase the running time of the algorithm extremely.

Another approach is based on *Sample Sort* which is often used in parallel sorting, see e.g. [6,3,2,14]. Each processor samples q keys from its N/P keys and exchanges them with the other processors. Sorting the set of samples in each processor makes it possible to create a set of $s - 1 < q$ equidistant keys called splitters. These splitters can be used to create s buckets of approximately size N/s . Load balance can be achieved, if s is a multiple of P . The parallel counting split radix sort algorithm (PCS-Radix) [9] uses Sample Sort for the partitioning of the data instead of using the radix of the keys directly. Radix sort is only used to sort the buckets locally. The prize for the independence of the data characteristics is the detection of global samples in the distributed system, but in a badly distributed environment this investment is worth doing. Despite of that, in environments where the data is mostly well distributed, the C3-Radix algorithm should be the algorithm of choice, because each processor is able to start the local creation of the buckets directly, and in general no further iterations are necessary to build the local buckets. On the other hand, also in a well-distributed environment, sometimes there may arise badly distributed data sets. The C3-Radix algorithm does not have the capability of being efficient in these cases, but it should be guaranteed that it is not far away from being efficient.

Hence, in the following we survey the possibilities for C3-Radix to be more stable and more predictable when working on unbalanced data distributions. In Section 2, we sketch the C3-Radix without going into detail for the steps we will not improve and without describing their optimizations made in [7]. In Section 3, we will explain the problems of C3-Radix in cases of unbalanced data more deeply, we will give an example, and we will describe improvements. In Section 4 we will give and interpret results of experimental test for the improved algorithm, and in Section 5 we conclude.

2 Outline of the C3-Radix Algorithm

The algorithm sorts N keys and each key consists of l bits. The used radix has length r . Initially, each processor stores N/P keys. The keys are globally sorted if the keys

within each processor are sorted, and there is a known order of the processors for which all keys of one processor are greater or equal to all keys in the preceding processor. Each processor builds buckets of keys by starting to observe the first r bits of each key. The initial length of the radix should be chosen in the way that $2^r > P$. Keys with the same radix belong to the same bucket.

(1) *Reverse Sorting*. Each processor scans its N/P integer keys using the first r bits beginning with the most significant bit, and building the corresponding 2^r buckets. During the creation of the buckets a counter array is built, too. Each entry in the counter array contains the number of keys in the corresponding bucket. (2) *Communication of Counters*. The local 2^r counters are exchanged between the processors. After this step, each processor knows the total amount of elements per bucket. (3) *Computation of bucket distribution*. Each processor computes locally a distribution of the buckets to the processors. If it is not possible to achieve a good load balance, then each processor starts again with step 1 and sets the new radix to ir , where i is the number of iterations. By extending the radix, the algorithm tries to produce more and smaller buckets that may lead to a better load balance. Otherwise the algorithm continues with step 4. (4) *All-to-All key communication*. The buckets are sent in an all-to-all fashion. After this step no more communication is necessary. (5) *Local sorting*.

3 Improvements

As we can see in Section 2, the main problem with the C3-Radix algorithm is that the first 3 steps may have to be repeated several times. The number of iterations depends on the data distribution, the initial radix chosen and the size of the integer keys. The more iterations are necessary, the bigger the radix gets. Since the number of buckets as well as the number of counters is 2^r , the allocated memory and the amount of data, that has to be communicated may increase the running time of the algorithm tremendously. What we want to improve is the way, the algorithm tries to distribute the data equally among the processors if more than 1 iteration is necessary. All other optimizations of the C3-Radix algorithms will not be changed. In order to explain the problem more detailed, we give an example. Let i be the number of iterations, then the accumulated number of counters in each processor is $\sum_{j=1}^i 2^{(j+1)r}$. In the experiments of [7] the radix is set to 5. If we assume there is a data distribution that leads to 6 iterations, then the number of counters broadcasted over the network is 1,108,378,656. If we further assume that a counter is a 32 bit integer (which is necessary for large N), then we see that each processor has to communicate about 4.129 GB of data. Each processor sends its counter array to all the other $P - 1$ processors. Assuming 16 processors, the total data transferred by the network is 990.96 GB. Besides this communication problem, there might arise memory problems. At least each processor has to store the counter array and the buckets constructed locally. Although there are more data structures like for example the initial data (N/P elements) or different counter arrays for the communication operation, the memory demand is dominated by the counter array. Therefore, we take its estimated memory demand as a key figure. Since the counter array has size 2^{30} in the sixth iteration and each entry stores one 32 bit integer, each processor needs at least 4 GB of main memory. Depending on the implementation of the counter communication (allreduce, broadcast, ...) the node has

to buffer up to P counter arrays which leads to 64 GB in our example. For a huge part of supercomputers, namely the PC-based SMP- or workstation-clusters, this size is not manageable. The program will abort due to out of memory errors. In order to avoid these large data arrays, our idea is not to rebuild all the buckets and counters with a larger radix within each iteration, but only to rebuild those that lie directly on at least one border of two processor data domains. (The data domain of a processor is its part of the sorted sequence of N integers. The N/P smallest integers are the data domain of processor 0, the next smallest N/P integers are the data domain of processor 1, and so on). Hence, at most $P - 1$ buckets need to be rebuilt.

In the first iteration the algorithm works as described in Section 2. The only difference is that after checking if another iteration is necessary, only selected buckets are rebuilt. For further iterations the first 3 steps are replaced by the following steps.

(1) *Reverse Sorting of special buckets.* Rebuild the buckets and counters that lie directly on the border of two processor data domains. (2) *Communication of Counters.* All-to-all communication of the new counter array. (3) *Computation of bucket distribution.* Check, if another iteration is necessary. If it is necessary, detect the buckets that have to be rebuilt with a higher radix. Otherwise proceed to step 4 of the algorithm described in Section 2.

Now, we want to analyze the behavior of this new algorithm in order to judge the received improvement. In each iteration, a certain number of buckets have to be rebuilt. We denote this number with a_j , where $1 \leq a_j < P$ and $j > 0$ is the number of the iteration ($a_1 = 0$). Let i be the number of iterations performed. The accumulated amount of counters in each processor is $\sum_{j=1}^i 2^r + a_j 2^r - a_j = i 2^r + (2^r - 1) \sum_{j=1}^i a_j$.

Looking at the scenario above ($r = 5, i = 6, P = 16$) and assuming the worst-case, where $a_j = P - 1$ for $j > 1$ the number of counters broadcasted over the network is 2517. Again we assume that a counter is a 32 bit integer, then we see that each processor has to communicate about 9.83 KB. In each iteration each processor sends its counter array to the other $P - 1$ processors. Hence, the overall amount of counters transferred by the network is 2359.2 KB.

The situation is also much better for the memory requirements. Storing the counter array in step 6 requires 19.66 KB per processor which normally is negligible compared to custom computer memory sizes. It should fit in the cache! Again, the maximum temporary buffer size needed for the communication and reduction step of the counters is 314.56 KB which is not critical, too. The analysis and the example show the potential of the formulated improvement.

Until now, we have just looked how the number of counters increases in both methods depending on the number of iterations. But we also have to care about the way the new buckets are created locally in the step 1 (*Reverse Sorting*). C3-Radix scans its N keys and rebuilds all buckets using the new radix. While building the buckets the counter array is updated, too. The scan can be done in $O(N)$, and an update of the buckets and counters can be done in $O(1)$. A practical drawback of this method is the following: As we know, the array of buckets and counters may get very big. Scanning the data means to access these arrays very irregularly. Hence, the memory hierarchy of the system is not

used efficiently, and the time needed for step 1 will grow quickly beginning at a critical size of the radix.

On the other side, the improved version does not always have to scan all local integer keys. It just scans the keys located in the buckets, which were decided to be rebuild. But in the worst case all local keys are contained in such buckets. Further, these N keys are not stored in one array consecutively, therefore, there is an additional overhead for switching between the buckets. After scanning the buckets and building new buckets out of them, these buckets have to replace the old buckets. Assuming that all buckets are stored in an array, this operation can be done in $O(2^r)$, whereby r is the size of the initial radix in the first iteration. For small initial radix sizes N should be $\geq 2^r$, hence, the additional time needed is not a problem. For larger sizes, step 1 is not the bottleneck of the algorithm as we saw in the example. Despite of that, we see that there might be situations for step 1, where C3-Radix is better than the improved version and vice versa, depending on the data distribution and the size of the radix.

In the following sections, we present experimental results that will show the behavior of the two algorithms for several data distributions. The improved algorithm is called balanced communication sensitive parallel radix sort (BCSP-Radix).

4 Experimental Tests

Our experimental tests were made on a Linux-SMP cluster with two Pentium III processors (650 MHz) and 1 GB main memory per node. The nodes are connected by a Myrinet 1.28 GBit/s switched LAN. The whole cluster consists of 98 nodes (Kepler-Cluster, [16]). As we showed in Section 3, it is sufficient to compare the first 3 steps of C3- and BCSP-Radix, because the steps 4 and 5 are the same as in the C3-Radix algorithm. We implemented them with C++ and TPO++ [4], which is an object-oriented message-passing system build on MPI [11,12]. Concerning the data distributions, we use two kinds of data sets. The first type are distributions already used in [15,7,9,5] and, therefore, are called standard data distributions. The second type are data distributions, which leads to worst-case behavior. We will explain them later on page 891. Duplicates are allowed in all data sets. For all experiments we try to achieve the best load balance possible. Since duplicates are allowed, a perfect load balance is not always possible. In general, we accept deviations of $\leq 1\%$ of N/P .

Our four **standard data distributions** are defined as follows, in which MAX is $(2^{31} - 1)$ for the integer keys, see also [15,7,9,5]. **Random [R]**, the data set is produced by calling the C library random number generator `random()` consecutively. The function returns integer values between 0 and $2^{31} - 1$. **Gaussian [G]**, an integer key is generated by calling the `random()` function 4 times, adding the return values and dividing the result by 4. **Bucket Sorted [B]**, the generated integer keys are sorted into P buckets, obtained by setting the first N/P^2 keys at each processor to be random numbers in the range of 0 to $(\text{MAX}/P - 1)$, the second N/P^2 keys in the range of MAX/P to $(2\text{MAX}/P - 1)$, and so forth. **Staggered [S]**, if the processor index i is $< P/2$, then all N/P integer keys at the processor are random numbers between $(2i + 1)\text{MAX}/P$. Otherwise, all N/P keys are random numbers between $(i - P/2)\text{MAX}/P$ and $((i - P/2 + 1)\text{MAX}/P - 1)$.

The main problem of using radix sort is that we do not know how the data is distributed and, therefore, with which size of the radix we should start. Our aim is to minimize the total running time. But without knowing details about the data it is not possible to choose the size of the radix optimally. The algorithm should grant that the running time should be close or equal to its optimum independent of the radix size and data distribution . Hence, in our test, we varied the initial size of the radix from 5 to 12. We made experiments using 16M integer keys and 8, 16 and 32 processors. All tests produced similar results, therefore, we give the explanation of the results by means of the 16 processors test. The running times of this test are presented in Table 1.

Table 1. Test with 16 processors (8 nodes with 2 processors) and 16M integer keys using 4 different data distributions, and varying the initial radix in a range from 5 to 12.. The first number in each cell is the mean running time of all used processors for the first 3 steps. The second number is the number of iterations needed for the execution

BCSP-Radix									
radix	5	6	7	8	9	10	11	12	total mean
[R]	0.95 1	1.06 1	1.12 1	1.24 1	1.24 1	1.23 1	1.20 1	1.15 1	1.15
[G]	2.63 3	2.42 2	1.85 2	1.68 2	1.71 2	1.77 2	1.07 1	1.11 1	1.78
[B]	2.87 3	2.87 3	2.51 2	2.44 2	2.10 2	1.94 2	0.93 1	0.96 1	2.08
[S]	2.76 3	2.53 3	2.94 2	2.34 2	1.98 2	1.79 2	1.01 1	1.09 1	2.05
C3-Radix									
radix	5	6	7	8	9	10	11	12	total mean
[R]	0.96 1	1.05 1	1.16 1	1.23 1	1.21 1	1.19 1	1.16 1	1.14 1	1.14
[G]	2.87 3	1.63 2	1.92 2	2.83 2	4.65 2	10.74 2	1.06 1	1.12 1	3.35
[B]	2.47 3	4.45 3	1.67 2	2.09 2	3.74 2	10.02 2	0.93 1	0.96 1	3.29
[S]	2.49 3	4.23 3	1.75 2	2.05 2	3.62 2	9.62 2	0.99 1	1.08 1	3.23

Concerning the [R] distribution, both algorithms achieve similar running times for all sizes of the radix because, if only 1 iteration is necessary then they are the same. In this case, the best running time can be achieved by using the smallest radix, which is obvious, because then the data structures are small, too. Computation and communication benefit by that. In general, the overall best running time for all distributions can be achieved, if we use the smallest size of the radix, with which only 1 iteration is necessary. In our sample this is $r = 11$ for [G], [B] and [S]. Again, both algorithms are the same. If the radix is chosen < 11 , then the situation is as follows. If the number of iterations needed to perform the algorithm is equal for several sizes of the initial radix, then C3-Radix is better if the size of the radix is minimal and BCSP-Radix is better for larger sizes of the radix. The reason why C3-Radix gets slower for larger sizes of the radix is founded in the increasing sizes of the bucket and counter arrays. Communication and computation gets much higher if the radix size increases dramatically due to further iterations. BCSP-Radix is much more stable, because the more iterations are necessary, the smaller the buckets are that have to be rebuild. Furthermore, the size of the counter

array for the communication is bounded by $O(2^r)$ as we saw in Section 3. Therefore, the communication cost is stable, too. If BCSP-Radix is worse, this is because of the running time of step 1. In these cases, due to the small radix, the data is distributed in a small number of buckets ($\leq P - 1$). Hence, each bucket has to be rebuilt and each processor has to scan all integer keys stored locally. The additional time needed for replacing all buckets by 2^r new buckets leads to the worse behavior. An essential observation can be made looking at the mean of the processor mean times (see column *total mean* in Table 1). BCSP-Radix is better for [S], [B], [G] and of course equal for [R]. While C3-Radix has some runaways in all distributions, BCSP-Radix is much more stable. This is important, because normally we do not know the behavior of the data with respect to the chosen radix. By using BCSP-Radix it is guaranteed that the total running time of the whole algorithm is not destroyed by the first 3 steps.

For the BCSP-Radix algorithm the **worst-case data-distribution** can be described as follows. After the first iteration, the data is partitioned into $P - 1$ buckets of equal size and the other buckets are empty. The algorithm decides to rebuild all $P - 1$ buckets with a larger radix. But the data is chosen in the way that the next bits are the same for all keys until the last r bits begin. That means, that in each iteration the $P - 1$ buckets remain unchanged until the last iteration. This is the worst case, because BCSP-Radix has to scan all keys in each iteration, and $P - 1$ is the maximum number of buckets which achieve this situation. Using this data set with C3-Radix does also lead to the maximum number of iterations possible. This data set is not constructed very artificially. The keys are uniformly distributed within a small range of bits and duplicates are allowed. Hence, this might also occur in a well-distributed environment. As we know from the example in Section 3, we cannot perform all iterations for 32 bit keys with the C3-Radix, because the main memory of our SMP-nodes is limited to 1 GB (using only 1 processor per node!). Therefore, the data set is constructed in the way that the nodes of the cluster will not collapse (≤ 20 bits). But we will show the behavior of BCSP-Radix performing all possible iterations.

Fig. 1 (left) presents an example of the comparison between the algorithms, where the data is distributed in the way that both algorithms have to perform 4 iterations, which means that the data can be partitioned looking at the first 20 bits. BCSP-Radix is much better than C3-Radix. The difference between both would even grow if further iterations were necessary. Unfortunately, C3-Radix cannot iterate further without aborting due to memory limitations.

In Fig. 1 (right), the running times for BCSP-Radix making 6 iterations (30 bits) are illustrated. We have to adhere that in this case BCSP-Radix is even faster than C3-Radix performing only 4 iterations. Although this is a worst-case situation for BCSP-Radix, the running time is better than the worst time achieved with C3-Radix for [S], [B] and [G] and radix 10 where 2 iterations were necessary (see Table 1).

5 Conclusions

In the previous sections, we suggested improvements with respect to the data decomposition for the currently fastest parallel radix sort algorithm. Instead of rebuilding all buckets for each iteration with a larger radix size, BCSP-Radix only rebuilds those buck-

Algorithm	Step 1	Step 2	Step 3	Σ
C3	4.79	5.88	0.15	10.82
BCSP	5.35	0.02	< 0.01	5.37

Algorithm	Step 1	Step 2	Step 3	Σ
BCSP	7.72	0.04	< 0.01	7.76

Fig. 1. Test with 16 processors, 16M integer keys, and an initial radix of 5. The numbers are the mean times (in sec.) of the processors for a worst case situation. (left) 4 iterations are necessary. (right) 6 iterations are necessary and only BCSP-Radix is able to terminate its execution

ets which help to find a better load balance for the proceeding steps of the algorithm. Due to this optimization, the computation, communication and memory requirements are much better. In experimental tests, we showed that the average general behavior of BCSP-Radix for standard data distributions is superior. While C3-Radix has several situations, where the execution time runs away, BCSP-Radix behaves much more stable and predictable. For worst-case data distributions the advantage of using BCSP-Radix is even bigger. While the non-optimized algorithm may not terminate due to memory constraints, BCSP-Radix is even in the position to achieve considerable running times. Even the memory requirements per processor are significantly within the capacity of custom workstations.

References

1. D. A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

2. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine. In *Proceedings of Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

3. A. V. Gerbessiotis and C. J. Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP Model. In *Proceedings oth the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages pp.223–232, 1996.

4. T. Grundmann, M. Ritt, and W. Rosenstiel. Object-Oriented Message-Passing with TPO++. In *EURO-PAR 2000, Parallel Processing*, volume 1900 of *LNCS*, pages 1081–1084. Springer Verlag, 2000.

5. D. R. Helman, D. A. Bader, and J. JáJá. Parallel Algorithms for Personalized Communication and Sorting With Experimental Study. In *Proceedings of the IEEE Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220, 1996.

6. D. R. Helman and J. JáJá. Sorting on Clusters of SMPs. *Informatica: An International Journal of Computing and Informatics*, 23, 1999.

7. D. Jiminez-Gonzales, J. Larriba-Pey, and J. Navarro. Communication Conscious Radix Sort. In *Proceedings of the International Conference on Supercomputing*, pages 76–82. ACM, 1999.

8. D. Jiminez-Gonzales, J. Larriba-Pey, and J. Navarro. *GI-Seminar: Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Case Study: Memory Conscious Parallel Sorting, pages 358–378. Springer Verlag, 2003. Advanced Lectures.

9. D. Jiminez-Gonzales, J. Navarro, , and J. Larriba-Pey. Fast Parallel In-Memory 64 Bit Sorting. In *Proceedings of the International Conference on Supercomputing*, pages 114–122. ACM, 2001.

10. D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.
11. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.
12. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997.
13. R. Sedgewick. *Algorithms*. Addison-Wesley, 1992.
14. H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
15. A. Sohn and Y. Kodama. Load Balanced Parallel Radix Sort. In *Proceedings of the International Conference on Supercomputing*, pages 305–312. ACM, 1998.
16. University of Tübingen (SFB-382). <http://kepler.sfb382-zdv.uni-tuebingen.de/>.

Minimizing Global Communication in Parallel List Ranking

Jop F. Sibeyn

Institut für Informatik
Universität Halle-Wittenberg
06120 Halle, Germany.

<http://www.mpi-sb.mpg.de/~jopsi/>

Abstract. Consider a parallel computer on which it is considerably cheaper to perform communications within close neighborhoods than globally. In such a context, it is not necessarily a good idea to simulate optimal PRAM algorithms. In this paper consider the list-ranking problem and present a new divide-and-conquer algorithm. For lists of length N , it performs $O(N \cdot \log N)$ work, but its communication is mainly local: there are only $6 \cdot 2^i$ communication operations within subnetworks of size $P/2^i$.

1 Introduction

A *linked list*, hereafter just *list*, is a basic data structure: it consists of nodes which are linked together, so that every node has precisely one predecessor and one successor, except for the *initial node*, which has no predecessor, and the *final node*, which has no successor. The *rank* of a node is defined as the number of links that must be traversed to reach the final node of the list. An important problem connected to the use of lists is the *list ranking* problem: the determination of the ranks of all nodes. Typically, the set of links constitutes a set of lists, and in that case for each node one also wants to compute the index of the last node of its list. Parallel list-ranking is a key subroutine in more complex problems such as edge coloring bipartite graphs and in the lowest-common-ancestor problem.

Parallel list ranking has been considered intensively [16,2,1,5,10,11,4,9,14], but there are still aspects which deserve further consideration. Imagine that one would like to solve a list-ranking problem on all the computers of a big institute or using even more distributed resources (using more computers also gives more main memory, for large problems this may avoid the considerable slow-down due to paging). Communication within small subnetworks may be performed without noticeable delay, while for more global communication the finite bandwidth of the network at its various levels may strongly reduce the processor-to-processor rate when all processors are communicating at the same time. In such a context, the focus of an algorithm should lie on minimizing the communication at the more global levels of the hierarchy, even if this implies some extra communication and work at the more local levels. In this paper we present an alternative algorithm with the following unique feature: there are only $6 \cdot 2^i$ operations within subnetworks of size $P/2^i$. In other words: most communication is performed locally.

2 Goal, Preliminaries, and Cost Model

The main purpose of this paper is to present and analyze an alternative list-ranking algorithm which implies an enrichment of the pool of available algorithms. The algorithm is formulated in a general recursive way with a cost expression given by a recurrence relation which is valid for any parallel computer model. Substituting the cost function for communication on the respective computer model allows to solve the recursion. In the full version of the paper we consider the performance on linear processor arrays and show that on this network the new algorithm is better than earlier ones and consequently also on all networks in which global communication is penalized even stronger.

The input consists of an array $\text{succ}[]$ of length N . The values in the array are all different. If $\text{succ}[u] = v$, with $0 \leq v < N$, then this is interpreted as v being the successor of u . In this case we will say that node u *points to* node v . If $v < 0$ or $v \geq N$, this is interpreted as u being the last node of its list. In this case we will say that u *points to null*, in an implementation it is convenient if then $v = u + N$. There are P PUs and the information related to node u is stored in $\text{PU } u \cdot P/N$.

Throughout this paper, for all algorithms we are considering, we are assuming that either all nodes were randomly renumbered at the beginning of the algorithm, or that we had a list given by a random permutation to start with. The communicational complexity of such a randomization is low: essentially it costs the same as routing all data through the network once. Alternatively, the data might be allocated to the PUs with help of a suitable randomly selected hash function mapping $\{0, 1, \dots, N-1\}$ to $\{0, 1, \dots, P-1\}$ in a sufficiently regular way. This assumption allows us to work with the expected sizes of the packets, in the analysis of the communication cost. For sufficiently large packets (as one should have anyway in order to amortize the cost to start-up a packet) the deviations from the expected values are small.

Our algorithm is based on the availability of two communication primitives. One is the *balanced k - k routing*, in which each processing unit, PU , sends k packets in total, k/P to each PU . The other is a *balanced k - k swapping*, in which, with respect to a given division of the network in two halves, each PU in one half sends $2 \cdot k/P$ packets to each PU in the other half. The time for these operations is denoted by $T_{\text{route}}(P, k)$ and $T_{\text{swap}}(P, k)$, respectively. On linear arrays $T_{\text{swap}}(P, k) = 2 \cdot T_{\text{route}}(P, k)$. On hierarchical networks the factor may be larger.

3 One-by-One Cleaning

All well-known parallel list-ranking algorithms perform are based on global all-to-all routings. Only the ‘one-by-one cleaning’ algorithm [13] solves the list ranking problem by a sequence of permutation routings. Its main ideas are repeated here, because the new algorithm is an alternative workout of the same underlying idea.

We define the concepts ‘clean’ and ‘cleaning’. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be subsets of the PUs. We say that $\mathcal{S}_1$ is *clean* for $\mathcal{S}_2$ if none of the nodes stored in a PU from $\mathcal{S}_1$ has a successor in $\mathcal{S}_2$. *Cleaning* $\mathcal{S}_1$ for $\mathcal{S}_2$ means to somehow achieve that $\mathcal{S}_1$ is clean for $\mathcal{S}_2$. In this context we distinguish two operations:

Autoclean(i): to clean PU i for itself;

Altroclean(i, j): to clean PU i for PU j .

For any PU i , $\text{autoclean}(i)$ can be performed without any communication by following the links until a final node or a link pointing to a node in another PU is found. For any PU i and j , $\text{altroclean}(i, j)$ can be performed by performing $\text{autoclean}(j)$ followed by a single pointer-jumping step: for each node u in PU i with $\text{succ}[u] = v$ in PU j , we perform $\text{succ}[u] = \text{succ}[\text{succ}[u]] = \text{succ}[v]$. Because we had first performed $\text{autoclean}(j)$, we can be sure that v does not point to another node in PU j .

In terms of autoclean and altroclean it is easy to formulate a list-ranking algorithm for a two-processor system, requiring only two routing operations independently of N :

1. For each $i \in \{0, 1\}$, $\text{autoclean}(i)$;
2. For each $i \in \{0, 1\}$, $\text{altroclean}(i, (i + 1) \bmod 2)$;
3. For each $i \in \{0, 1\}$, $\text{autoclean}(i)$.

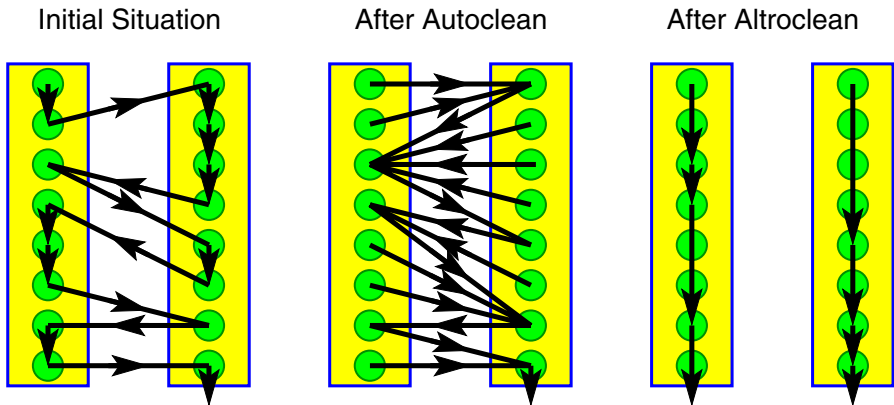


Fig. 1. List ranking on a two-processor system. After the first autoclean , all links point to nodes in the other PU, except for the final node which has a successor value not corresponding to any node index. After the altroclean , all non-final nodes point to a node in the own processor. One more autoclean solves the problem.

The problem with extending this idea to arbitrary P is that when PU i is clean for PU j and we perform $\text{altroclean}(i, k)$, for some other k , that then, if we do not take special care, we get again links to nodes stored in PU j , and PU i is no longer clean for PU j . In one-by-one cleaning, in each PU there are two sets of links, which are cleaned in different ways so that this problem is avoided. The algorithm consists of $P - 1$ rounds. An invariant is that after round t , $0 \leq t < P$, the ‘left-set’ of links in PU i is clean for $\{i, i - 1, \dots, i - t\}$ and the ‘right-set’ of links is clean for $\{i, i + 1, \dots, i + t\}$ (all indices are computed modulo P). Then, in round $t + 1$, the left-set of PU i is cleaned for PU $i - t - 1$ with help of the right-set of PU $i - t - 1$, and the right-set of PU i is cleaned for PU $i + t + 1$ with help of the left-set of PU $i + t + 1$. One more autoclean achieves the invariant for $t + 1$. The unique feature of this algorithm is that in any communication operation each PU is sending to only one other PU, the whole pattern constituting a permutation.

4 Divide and Conquer

The guiding idea in one-by-one cleaning is that if PU i is clean for PU j , one should be careful to keep it clean for PU j during the rest of the execution. This is a good strategy but also appears to require the given structure, not leaving room for further improvement. In this section we consider a divide-and-conquer algorithm using the same idea of alternating autoclean and altroclean. The algorithm is a variant of the ‘repeated-halving’ algorithm [12] modified so as to minimize the number of global communication operations.

With respect to a set $\mathcal{S}$ of the PUs, we will say that node u is *final* in $\mathcal{S}$ if the node $\text{succ}[u] = v$ is not stored in a PU of $\mathcal{S}$. In the following, if it is clear from the context which set $\mathcal{S}$ is meant, we will just say that node u is final. We say that the list-ranking problem has been *solved* for a subset $\mathcal{S}$ of the PUs, if each node stored in a PU from $\mathcal{S}$ knows a final node stored in a PU of $\mathcal{S}$. More precisely, we specify that after solving the problem for $\mathcal{S}$, $\text{succ}[u]$ should be unchanged for each final node u in $\mathcal{S}$, but for all non-final nodes in $\mathcal{S}$, afterwards $\text{succ}[u] = w$, for some final node w in $\mathcal{S}$.

In the following we present a recursive algorithm for solving the problem for $\mathcal{S}_1 \cup \mathcal{S}_2$, for two subsets of the PUs $\mathcal{S}_1$ and $\mathcal{S}_2$. The algorithm is actually quite simple, but it consists of many steps. These steps are divided in 6 phases. Two of these phases solve subproblems, the other four phases consist of a round of questions, answers and updates. Before the pseudo-code of the non-recursive phases a comment explains what is happening.

1. Solve the problem for $\mathcal{S}_1$ and for $\mathcal{S}_2$;

/* As a result of the solving in each subset all non-final nodes now know a final node. Each final node in $\mathcal{S}_1$ and $\mathcal{S}_2$ which is not a final node in $\mathcal{S}_1 \cup \mathcal{S}_2$ figures out the subsequent final node, and updates its *succ*-value accordingly. */

2. In each set, for each final node u pointing to a node v in the other subset, send a question to node $v = \text{succ}[u]$;
3. In each set, for each non-final node v which received a question sent in step 2, send a question to node $w = \text{succ}[v]$;
4. In each set, for each node w which received a question sent in step 3 from a node v , send an answer back to node v . If w is a final node in $\mathcal{S}_1 \cup \mathcal{S}_2$, then the answer consists of a placeholder, otherwise w returns $\text{succ}[w]$ and distance information;
5. In each set, for each node v which received a question sent in step 2 from a node u , send an answer back to node u . If v is a final node in $\mathcal{S}_1 \cup \mathcal{S}_2$, then the answer consists of a placeholder. Otherwise, if it is a final node in its own set or if it received a placeholder in step 4, v returns $\text{succ}[v]$ and distance information, else v returns $\text{succ}[w] = \text{succ}[\text{succ}[v]]$ and distance information;
6. In each set, for each final node u , which received a non-void answer sent in step 5 in reply to its question sent in step 2, update $\text{succ}[u]$ and distance information;

/* As a result of the update, a final node in $\mathcal{S}_1$ may now point to a non-final in $\mathcal{S}_1$, and similarly in $\mathcal{S}_2$. Such nodes now ask these non-final nodes for the final node they are pointing to, and update their *succ*-values accordingly. */

7. In each set, for each formerly final node u , which has become non-final as a result of the update in step 6, send a question to node $v = succ[u]$;

8. In each set, for each node v which received a question sent in step 7 from a node u , send an answer back to node u . If v was a final node before the update in step 6, then the answer consists of a placeholder, otherwise v returns $succ[v]$ and distance information.

9. In each set, for each formerly final node u , which received a non-void answer sent in step 8 in reply to its question sent in step 7, update $succ[u]$ and the distance information;

/ As a result of the updates, all final nodes point to final nodes again. A final node u in S_1 points to a node v in S_2 only when v is a final node in $S_1 \cup S_2$, and similarly for the final nodes in S_2 . The subproblems given by the linking of the final nodes are now solved recursively. */*

10. Solve the problem for all formerly final nodes in S_1 and S_2 which became non-final as a result of the update in step 6.

/ Now each final node in S_1 is either a final node in $S_1 \cup S_2$ or points to such a node, or points to a node which points to such a node, and similarly for the final nodes in S_2 . A final node in this last category asks the node it is pointing to for its successor and updates its $succ$ -value accordingly. */*

11. In each set, for each formerly final node u , which has become non-final as a result of the update in step 6, send a question to node $v = succ[u]$;

12. In each set, for each node v which received a question sent in step 11 from a node u , send an answer back to node u . If v is a final node in $S_1 \cup S_2$, then the answer consists of a placeholder, otherwise v returns $succ[v]$ and distance information.

13. In each set, for each formerly final node u , which received a non-void answer sent in step 12 in reply to its question sent in step 11, update $succ[u]$ and the distance information;

/ Now each final node in S_1 is either a final node in $S_1 \cup S_2$ or points to such a node, and similarly for the final nodes in S_2 . Each non-final node asks the final node it is pointing to for its successor and updates its $succ$ -value accordingly. As a result all nodes are either final nodes in $S_1 \cup S_2$ or are pointing to such a node. */*

14. In each set, for each non-final node u , send a question to node $v = succ[u]$;

15. In each set, for each node v which received a question sent in step 14 from a node u , send an answer back to node u . If v is a final node in $S_1 \cup S_2$, then the answer consists of a placeholder otherwise v returns $succ[v]$ and distance information.

16. In each set, for each non-final node u , which received a non-void answer sent in step 15 in reply to its question sent in step 14, update $succ[u]$ and the distance information.

The steps of the algorithm are illustrated in Figure 2. The algorithm will be used by calling it with S_1 and S_2 each consisting of $P/2$ PUs, and then recursively dividing

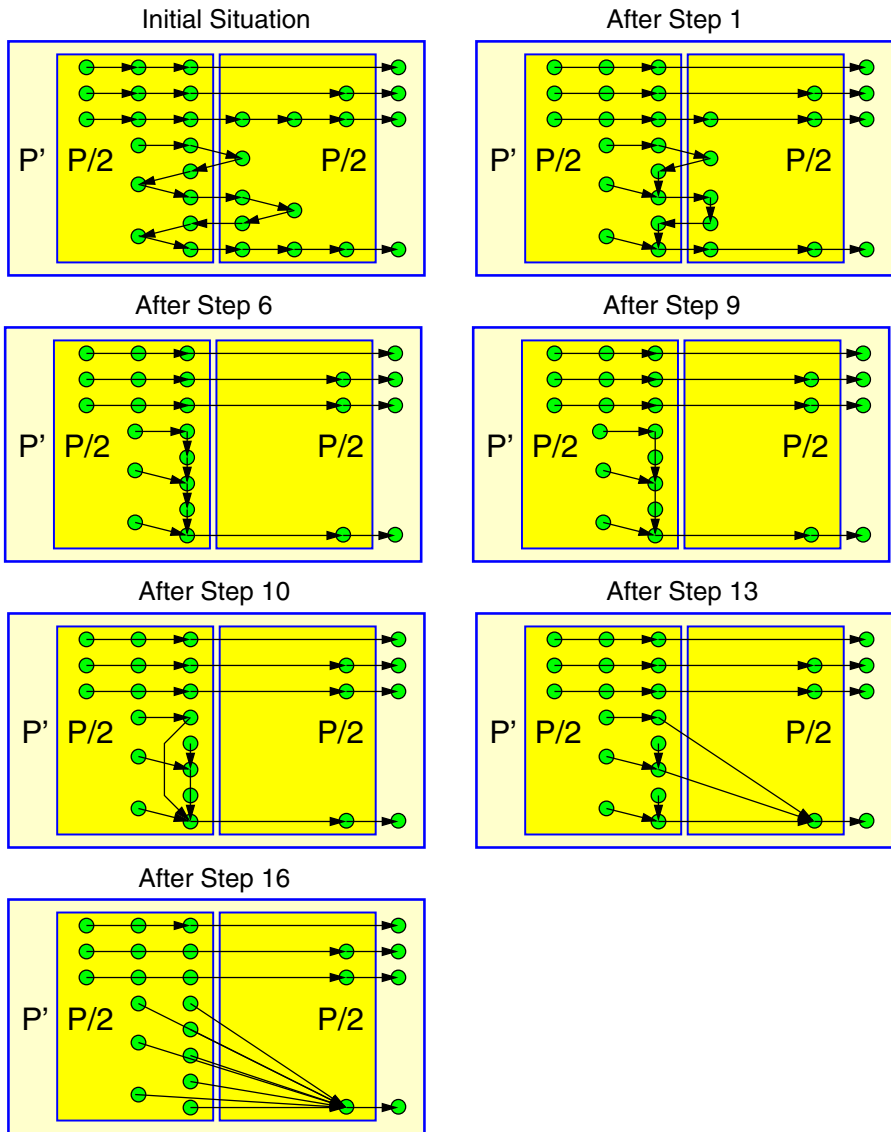


Fig. 2. The stages of the recursive list-ranking algorithm. On the right side of each picture only the nodes are shown which are still relevant for the nodes on the left side.

these sets. On a single PU, the problem is solved internally without further recursion. When answering a question (Step 4, 5, 8, 12, 15) from a node u we distinguish the case that the question was actually unnecessary because u was already pointing to a node with the desired property. In this case $\text{succ}[u]$ should not be updated, which is signaled to u by sending a placeholder.

The given algorithm constitutes an alternative generalization of the idea for the two-processor system depicted in Figure 1. Here we are solving (= searching for the last node in the own subset of PUs) rather than cleaning (= searching for the first node in the other subset of PUs) for efficiency reasons. This detail has a considerable impact on the recurrence relation we will get. The operation of the algorithm is summarized in the following lemma:

Lemma 1 *Let u be any node stored in a PU in $\mathcal{S}_1 \cup \mathcal{S}_2$, let $v = \text{succ}[u]$ before applying the algorithm and let $v' = \text{succ}[u]$ after applying the algorithm. If v is stored in a PU in $\mathcal{S}_1 \cup \mathcal{S}_2$, then*

- v' is stored in a PU in $\mathcal{S}_1 \cup \mathcal{S}_2$.
- $\text{succ}[v']$ is not stored in a PU in $\mathcal{S}_1 \cup \mathcal{S}_2$,

If v is not stored in a PU in $\mathcal{S}_1 \cup \mathcal{S}_2$, then $v' = v$.

Lemma 2 *The algorithm performs 2 routing operations within the whole network and $6 \cdot 2^i$ routing operations within subnetworks of size $P/2^i$, for all $1 \leq i < \log P$.*

Proof: In Step 2 and 5 (2 steps) the communication is global, in Step 3, 4, 7, 8, 11, 12, 14 and 15 (8 steps), the communication is limited to half the network. There are two recursive calls performing the same operations in the halves. Let $T(i)$ denote the number of operations within subnetworks of size $P/2^i$. The number of recursive calls in these subnetworks is 2^i . $T(i) = 2$. For all $i > 0$, we have

$$T(i) = 8 \cdot 2^{i-1} + 2 \cdot 2^i = 6 \cdot 2^i.$$

Here the first contribution is due to the routing operations from level $i - 1$, the second contribution to the global operations performed at level i of the recursion. $\square$

Let $T_{\text{solve}}(P, P', \alpha)$ denote the time for solving a problem on P PUs, with α list nodes per PU. Nodes, for which we already established that the successors lie in a set of P' PUs, for some $P' \geq P$. In the following analysis we assume that in the algorithm $\mathcal{S}_1$ and $\mathcal{S}_2$ each consist of half of the PUs.

Lemma 3 *The time consumption $T_{\text{solve}}(P, P', \alpha)$ of the algorithm is the sum of the following contributions:*

- Step 1:** $T_{\text{solve}}(P/2, P', \alpha)$,
- Step 2+5:** $T_{\text{swap}}(P, \alpha \cdot \beta) \cdot (2 + 2 \cdot \beta)$,
- Step 3+4:** $T_{\text{route}}(P/2, \alpha \cdot \beta \cdot \beta) \cdot (2 + \gamma)$,
- Step 7+8:** $T_{\text{route}}(P/2, \alpha \cdot \beta \cdot \gamma) \cdot (2 + \beta)$,
- Step 10:** $T_{\text{solve}}(P/2, P' - P/2, \alpha \cdot \beta \cdot \gamma)$,
- Step 11+12:** $T_{\text{route}}(P/2, \alpha \cdot \beta \cdot \gamma) \cdot (2 + \gamma)$,
- Step 14+15:** $T_{\text{route}}(P/2, \alpha \cdot \beta) \cdot (2 + \gamma)$,

Here $\beta = (P/2)/P'$, $\gamma = (P/2)/(P' - P/2)$ and

Proof: The result for step 1 is the simplest: we are recursing for $P/2$ PUs, each PU still holding the original number of participating nodes which are pointing to nodes stored in P' different PUs.

In step 2, all final nodes pointing to a node in the other subset are asking a question and get an answer in step 5. The questions consist of one integer each. The fraction of nodes pointing to a node in the other subset is $(P/2)/P' = \beta$. Here, we are using that, according to the first claim in Lemma 1, the number of nodes pointing into another set of PUs does not increase. Of the nodes which receive a question, exactly a fraction $P/P' = 2 \cdot \beta$ points to a node in $S_1 \cup S_2$ and answer in step 5 with two integers, the other nodes answer with a single integer. These routing operations are the only two performed in the whole set of P PUs, all other operations are limited to the subsets of size $P/2$.

In step 3, a fraction β of the questions is forwarded. The nodes which receive a question are final in their own subset of PUs. Thus, a fraction $(P/2)/(P' - P/2) = \gamma$ of them points to a node in the other subset of PUs and answer with two integers, the other nodes answer with a single integer.

In step 7, all nodes which became non-final are asking a question. A node becomes non-final, if and only if it got in reply to its question from step 2 the index of a node stored in its own subset of PUs. Because we know that these indices are arbitrary except for the fact that they do not point to nodes in the other subset, the fraction with this property is $(P/2)/(P' - P/2) = \gamma$. Of the nodes which receive a question a fraction $(P/2)/P' = \beta$ was non-final in their own subset of PUs and answer with two integers, the other nodes answer with a single integer.

In step 10, the recursion is performed for precisely these $\alpha \cdot \beta \cdot \gamma$ nodes in each PU which became non-final in step 6. The successors of these nodes are distributed over $P' - P/2$ different PUs.

In step 11, the same nodes are asking a question as in step 7. The only thing we know of the nodes which receive a question is that they are final in their own subset of PUs. Thus, a fraction $(P/2)/(P' - P/2) = \gamma$ of them points to a node in the other subset of PUs and answer with two integers, the other nodes answer with a single integer.

In step 14, all nodes which initially had their successors within the own subset of PUs ask a question. The fraction of these nodes is β . Of the nodes which receive a question we only know that originally they were final in their own subset. If they were pointing to a node in the other subset, which happens for a fraction γ of them, they answer with two integers, else with one. $\square$

For any given network with P PUs, substituting the costs of the routing operations, $T_{\text{solve}}(P, P, N/P)$ as expressed in Lemma 3 gives an exact estimate for the total communication time when solving a list-ranking problem of size N with the recursive algorithm.

5 Conclusion

We presented a new recursive parallel list-ranking algorithm. Due to its mainly local communication, it performs good on networks on which global communication is strongly penalized. The simplest example of such a network is the linea array, for which it can be shown that the new idea indeed leads to better performance than earlier algorithms.

References

1. Anderson, R.J., G.L. Miller, 'Deterministic Parallel List Ranking,' *Algorithmica*, 6, pp. 859–868, 1991.
2. Cole, R., U. Vishkin, 'Approximate Parallel Scheduling, Part I: the Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time,' *SIAM Journal on Computing*, 17(1), pp. 128–142, 1988.
3. Dehne, F., A. Fabri, A. Rau-Chaplin, Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers,' *Proc. 9th Computational Geometry*, pp. 298–307, ACM, 1993.
4. Dehne, F., S.W. Song, 'Randomized Parallel List Ranking for Distributed Memory Multiprocessors,' *Proc. Asian Computer Science Conference*, LNCS 1179, pp. 1–10, 1996.
5. Hayashi, T., K. Nakano, S. Olariu, 'Efficient List Ranking on the Reconfigurable Mesh with Applications,' *Proc. 7th International Symposium on Algorithms and Computation*, LNCS 1178, pp. 326–335, Springer-Verlag, 1996.
6. JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Inc., 1992.
7. Juurlink, B.H.H., H.A.G. Wijshoff, 'The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model,' *Proc. 2nd International Euro-Par Conference*, LNCS 1124, pp. 339–347, Springer-Verlag, 1996.
8. Ranade, A., 'A Simple Optimal List Ranking Algorithm,' *Proc. of 5th High Performance Computing*, Tata McGraw-Hill Publishing Company, 1998.
9. Reid-Miller, M., 'List Ranking and List Scan on the Cray C-90,' *Journal of Computer and System Sciences*, 53(3), pp. 344–356, 1996.
10. Ryu, K.W., J. JáJá, 'Efficient Algorithms for List Ranking and for Solving Graph Problems on the Hypercube,' *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 83–90, 1990.
11. Sibeyn, J.F., 'List Ranking on Meshes,' *Acta Informatica*, 35, pp. 543–566, 1998.
12. Sibeyn, J.F., 'Better Trade-offs for Parallel List Ranking,' *Proc. 9th Symposium on Parallel Algorithms and Architectures*, pp. 221–230, ACM, 1997.
13. Sibeyn, J.F., 'One-by-One Cleaning for Practical Parallel List Ranking,' *Algorithmica*, 32, pp. 345–363, 2002.
14. Sibeyn, J.F., F. Guillaume, T. Seidel, 'Practical Parallel List Ranking,' *Journal of Parallel and Distributed Computing*, 56, pp. 156–180, 1999.
15. Valiant, L.G., 'A Bridging Model for Parallel Computation,' *Communications of the ACM*, 33(8), pp. 103–111, 1990.
16. Wyllie, J.C., *The Complexity of Parallel Computations*, PhD Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

Construction of Efficient Communication Sub-structures: Non-approximability Results and Polynomial Sub-cases

Christian Laforest

LaMI, CNRS, Université d'Evry, 523, Place des Terrasses 91000 Evry, France
laforest@lami.univ-evry.fr

Abstract. In this paper, we study the following problem. Given a graph $G = (V, E)$ and $M \subseteq V$, construct a subgraph $G_M^* = (V^*, E^*)$ of G spanning M , with the minimum number of edges and such that for all $u, v \in M$, the distance between u and v in G and in G_M^* is the same. This is what we call an *optimal partial spanner* of M in G . Such a structure is "between" a Steiner tree and a spanner and could be a particularly performant and low cost structure connecting members in a network. We prove that the problem cannot be approximated within a constant factor. We then focus on special cases: We require that the partial spanner is a tree satisfying additional conditions. For this sub problem, we describe a polynomial algorithm to construct such a tree partial spanner.

1 Introduction

We are interested in this paper by a graph problem that is between two well known graph problems: The *Steiner tree* and the *spanner* problems.

Both are NP-complete for the majority of their formulations. The first one consists in constructing a tree of minimum weight connecting a subset of selected vertices and have been studied for a long time. Many *approximation algorithms* have been published (see for example [1,4] for general references on approximation algorithms and [5,16,17] for the Steiner problem).

The objective of the second one is, given a graph G and a *stretch factor* α , to construct a subgraph of minimum weight such that for any pair of vertices their distances in the graph and in the spanner is the same, up to a given factor α . If α is a *multiplicative* (resp. *additive*) factor, the corresponding spanners are called multiplicative (resp. additive) spanners (see for example [3,8,11,12,13,15]).

We can note that in the traditional Steiner problem, preserving distances between the selected vertices is not taken into account. Several recent works deal with maximal and/or average distances in approximated Steiner trees [6,9,10,14] (see [7] for the same kind of problem for spanning trees). The weight of obtained substructure is good (compared to optimal) but it fails keeping *exact* (optimal) distances between the selected vertices.

On the other hand, spanners deal with distances. But spanners are only considered as spanning subgraph that must include *all* the vertices of the graph.

In this paper, we try to generalise this concept and to consider spanners of minimum weight with multiplicative stretch 1 for subsets of selected vertices. This is what we call *partial spanners*. The study of this problem is motivated by *network* problems. The goal is to construct a subnetwork to connect members of a video-conference for example. The distance (latency) between participants and the total weight must be low to ensure good *Quality of Service* with a low cost.

The paper is organized as follows. Section 2 gives definitions, describe the problem of partial spanner and gives a general bound. In section 3 we prove that the problem cannot be approximated within a constant.

As the general problem is hard, we restrict our attention to particular cases of partial spanners. We require that they must be trees (for simplicity of routing mechanism) and additional properties that allow to implement these communication structures at the application layer of networks. Unfortunately such a partial spanner tree does not always exist. In section Section 4 we propose a polynomial algorithm that constructs this tree if and only if it exists.

2 Definitions, Problem, and General Bounds

In this paper, $G = (V, E)$ is a graph where V is the set of *vertices* (representing the set of nodes), E the set of *edges* (representing the set of links). Graphs are undirected and connected (see [2] for undefined terms of graph).

For each pair u, v of vertices of V , $d_G(u, v)$ is the *distance* between u and v ; this is the number of edges in a shortest path between u and v in G . The *weight* of G , $W(G)$ is $|E|$, its number of edges.

Definition 1 (Partial spanner). *Let $G = (V, E)$ be a graph and $M \subseteq V$. The sub-graph $S = (V_S, E_S)$ of G is a partial spanner of M in G if*

- $M \subseteq V_S \subseteq V$ and $E_S \subseteq E$.
- For all $u, v \in M$, $d_S(u, v) = d_G(u, v)$.

Note that when $M = V$ there is only one partial spanner, G itself. Here is now the main problem.

Problem 1 (Partial spanner problem).

INSTANCE: $G = (V, E)$ a graph and $M \subseteq V$.

SOLUTION: A partial spanner S of M in G .

MEASURE: $W(S)$, the weight of S .

Let $G = (V, E)$, $M \subseteq V$. A partial spanner of M in G of minimum weight is denoted by G_M^* and is called an *optimal partial spanner* of M .

The following result gives general tight bounds on the weight of an optimal partial spanner.

Theorem 1. *Let $G = (V, E)$ and $M \subseteq V$. The following bounds for $W(G_M^*)$ are tight:*

$$D_G(M) = \max\{d_G(u, v) : u, v \in M\} \leq W(G_M^*) \leq \frac{1}{2} \sum_{u, v \in M} d_G(u, v)$$

3 Complexity of Problem 1

In this section, we prove that the problem of constructing an optimal partial spanner cannot be approximated from a constant multiplicative factor. For that, we make a reduction with the following problem.

Problem 2 (Minimal set cover).

INSTANCE: X a set of elements and $\mathcal{F}$ a set of subsets of X satisfying: $X = \bigcup_{F \in \mathcal{F}} F$.

SOLUTION: A subset $C \subseteq \mathcal{F}$ covering X : $X = \bigcup_{F \in C} F$.

MEASURE: $|C|$, the size of C .

This problem is not APX (see [1], problem SP4).

Theorem 2. *Unless $P = NP$, for any constant α , there is no α approximation algorithm for Problem 1.*

Proof. We start the proof by describing a special instance of Problem 1, called *four levels instance*. The graph $G = (V, E)$ of this instance must satisfy the following requirements. Vertex set V contains two special vertices r and Y and two non empty sets V_1 and X plus additional vertices ensuring connectivity. Let $m = |X| + 1$. The connections in G are the following:

- Vertex r is connected to each vertex of V_1 by a path of length $m - 1$ (using additional vertices).
- Each vertex $u_i \in V_1$ must be connected by a path of length 3 to at least one vertex of X . We note C_i the set of vertices of X connected to u_i . To be a *four levels instance* we must have:

$$\bigcup_{u_i \in V_1} C_i = X$$

- Each vertex of X is connected (by an edge) to vertex Y .

There is no other connection in a *four levels instance*. The instance of Problem 1 is the constructed graph G and $M = X \cup \{r\}$. It is now easy to see that in polynomial time:

- From any instance $(X, \mathcal{F})$ of Problem 2, one can construct an associated *four levels instance* of Problem 1.
- From any *four levels instance* of Problem 1, one can construct an instance $(X, \mathcal{F})$ of Problem 2.

Let us now consider a *four levels instance* G, M . We can note that for all $u, v \in X$, $d_G(u, v) = 2$ and the only shortest path between u and v is via Y . Any partial spanner of M must contain *all* the $m - 1$ edges uY for all $u \in V_1$ to ensure the condition on distances.

The other edges to add in a partial spanner are just used to connect r to vertices of X . Note also that as for all $u \in X$, $d_G(r, u) = m + 2$, vertex r must be connected to u by a direct path via only one vertex of V_1 .

Hence, any partial spanner of M must contain at least a path of length 3 for each $u \in X$ ($3(m-1)$ edges) plus $m-1$ edges Yu ($u \in X$) plus $(m-1)s$ edges of paths between r and u_i ($u_i \in V_1$) where s is the number of vertices of V_1 used to connect r to vertices of X in the partial spanner. The weight of a partial spanner of M must be at least $3(m-1) + (m-1) + (m-1)s = (m-1)(s+4)$. Moreover, this number of edges is sufficient to construct a partial spanner of M .

Suppose first that there exists a partial spanner S of M , whose weight is $(m-1)(s+4)$ for a *four levels instance* of problem 1. Let us consider the s labels C_i of the vertices u_i such that there is a path between r and u_i in S . The union of these labels gives X (because of the previous remarks). Then this is a solution C of size s for the minimal set cover problem with X and $\mathcal{F} = \{C_i : i = 1, \dots, k\}$.

Let us see now the other direction of the reduction. Let $X, \mathcal{F}$ be any instance of Problem 2. Let C be a solution of size p . One can construct a partial spanner of M in the (associated four levels) graph by taking the following edges: The p paths of length $m-1$ between r and u_i for each i such that $C_i \in C$. For each $u \in X$ a path (of length 3) between u and u_i labelled C_i such that $u \in C_i$ (such a i exists because C covers X). Finally, each edge Yu for all $u \in M - \{r\}$. The constructed graph is a partial spanner of M , and its weight is exactly $(m-1)(p+4)$.

Hence, we have shown that for all s , there exists a solution for a *four levels instance* for problem 1, of weight $(m-1)(s+4)$ if and only if there exists a solution for an instance of problem 2, of size s .

Now, let α be a constant, $\alpha \geq 1$. Suppose that there exists an α approximation algorithm *Algo* for the partial spanner problem. Let $(X, \mathcal{F})$ be any instance of the minimal set cover problem. Let $G = (V, E)$ and $M \subseteq V$ be the four levels graph associated to the instance $(X, \mathcal{F})$. Let S be a partial spanner constructed by algorithm *Algo* applied on G and M . In addition to *Algo* we also suppress useless vertices of V_1 (i.e. not connected to r by a path of length $m-1$) and subtrees containing no vertex of M . As *Algo* is an α approximation algorithm, we have: $W(S) \leq \alpha W(G_M^*)$. Let A be the set of vertices of V_1 in S and $a = |A|$. As each vertex of A is in a shortest path from r to a vertex of X , by previous discussions we have: $(m-1)(a+4) \leq W(S)$. Now, let B be the set of vertices of V_1 in G_M^* and $b = |B|$. We have: $W(G_M^*) = (m-1)(b+4)$. From these notations, we have $(m-1)(a+4) \leq \alpha(m-1)(b+4)$ and $a+4 \leq \alpha(b+4)$. Moreover, with the previous discussions, $B = C^*$, that is B is a minimum covering for the instance $(X, \mathcal{F})$. Hence, by applying *Algo*, one can construct a covering A whose cardinality a satisfies: $a+4 \leq \alpha(|C^*|+4)$. That is: $a \leq \alpha|C^*| + 4(\alpha-1) \leq (5\alpha-4)|C^*|$. The covering problem can then be $(5\alpha-4)$ approximated. This contradicts the fact that it is not an APX problem (see [1]). $\square$

4 Partial Spanners That Are Trees

Section 3 shown that the general partial spanner problem is very hard (even to approximate and even for unweighted graphs). However, partial spanners remain very interesting substructures to connect computers. In this section, we

will require, in addition to be a partial spanner for M , to have the following properties:

1. The constructed partial spanner must be a *tree*.
2. The nodes that route information in the tree must be in M .

These two requirements are very useful to construct an efficient communication sub-structure because it can be realised at the *application layer* of network, i.e. without using intermediate routers to route messages or duplicate them, just by using tunnels.

Unfortunately, given a graph G and M there is not always such a tree partial spanner. In this section we propose a polynomial time algorithm that constructs such a tree if and only if there exists one in G .

To obtain this algorithm we need intermediate results and notions like L -trees, S -trees and Steiner trees. The relationship between S -tree and tree partial spanners is given in Corollary 1.

Definition 2 (Subtree of a graph). Let $G = (V, E)$ be a graph. The tree $T = (V_T, E_T)$ is a subtree of G if $V_T \subseteq V$ and $E_T \subseteq E$. Moreover T covers $M \subseteq V$ if $M \subseteq V_T$.

Definition 3 (Steiner tree of M). Let $G = (V, E)$ be a graph and $M \subseteq V$. A tree $T_0 = (V_0, E_0)$ is a Steiner tree of M in G if:

- T is a tree covering M in G : $M \subseteq V_0 \subseteq V$ and $E_0 \subseteq E$.
- T_0 is of minimum weight:

$$W(T_0) = \min\{W(T) : T \text{ covering } M \text{ in } G\}$$

Definition 4 (L -tree). Let $G = (V, E)$ be a graph and $M \subseteq V$. A tree $T = (V_T, E_T)$ (if it exists) is a L -tree of M if:

- $M \subseteq V_T \subseteq V$, $E_T \subseteq E$ and T has no subtree containing member of M .
- For any $u, v \in M$, $d_T(u, v) = d_G(u, v)$.

Lemma 1. Let $G = (V, E)$ be a graph and $M \subseteq V$. If G contains a L -tree T of M then T is a Steiner tree of M .

Proof. Let T_0 be a Steiner tree of M . Let $r_0 \in M$. Let us apply a depth first search of T_0 from r_0 . Let $r_0, \dots, r_k (= r_0)$ be the sequence of vertices of M visited during this tour ($k \geq |M|$). In order to simplify the presentation, we use the notation $i \oplus j$ for $i + j \pmod{k}$. As T is a L -tree, we have:

$$d_G(r_i, r_{i \oplus 1}) = d_T(r_i, r_{i \oplus 1}) \leq d_{T_0}(r_i, r_{i \oplus 1})$$

for all $i = 1, \dots, k$. Moreover, we have a well known property:

$$\sum_{i=0}^k d_{T_0}(r_i, r_{i \oplus 1}) = 2W(T_0)$$

The sequence of vertices contains several times each vertex of M . This leads to:

$$2W(T) \leq \sum_{i=0}^k d_T(r_i, r_{i \oplus 1}) \leq \sum_{i=0}^k d_{T_0}(r_i, r_{i \oplus 1}) = 2W(T_0)$$

Hence, $W(T) \leq W(T_0)$. As of course we have $W(T_0) \leq W(T)$ we get: $W(T_0) = W(T)$. T is a Steiner tree of M . $\square$

Definition 5 (S-tree). Let $G = (V, E)$ be a graph and $M \subseteq V$. A tree $T = (V_T, E_T)$ (if it exists) is a *S-tree* of M if:

- T is a *L-tree* of M .
- $S = \{u : d_T(u) \geq 3\} \cup \{(d_T(u) \geq 2 \text{ and } u \in M)\} \subseteq M$

The last condition says that what are usually called the "steiner points" (including duplication nodes and internal members of M) of T must be members of M only.

The following result show the relationship between optimal partial spanners and *L-trees*.

Corollary 1. Let $G = (V, E)$ be a graph and $M \subseteq V$. Any *L-tree* (hence any *S-tree*) T of M is an optimal partial spanner of M .

Proof. Definition 4 of a *L-tree* gives conditions on distances between members of M and Lemma 1 proves that such a covering tree is of minimum weight. $\square$

In the following, we will use the notion of *complete graph of distances* of M in graph G . It is a complete graph (each pair of vertices is connected by an edge) with vertex set equal to M . Each edge $e = uv$ ($u, v \in M$) is weighted by $w(e) = d_G(u, v)$. By extension of notation, for any subgraph or set of edges $G' = (V', E')$, we note:

$$W(G') = \sum_{e \in E'} w(e)$$

ALGORITHM *KS*

Input: $G = (V, E)$ and $M \subseteq V$.

Construct K_M the complete graph of distances of M .

Construct in K_M a minimum weight spanning tree T_P (with *Prim* algorithm for example).

For each edge $e = uv$ of T_P construct a shortest path between u and v in G .

Return the union of these paths.

Lemma 2. Let $G = (V, E)$ be a graph and $M \subseteq V$. If G contains a *S-tree* of M then *KS* applied to G and M returns a Steiner tree of M .

Proof. From Lemma 1, a *L-tree* (hence a *S-tree*) T_0 of M is a Steiner tree of M . Let T_1 be the graph returned by *KS*(G, M). The other notations are the same than in algorithm *KS*. We have: $W(T_0) \leq W(T_1) \leq W(T_P)$. As T_0 is a

S -tree, it is composed of $m - 1$ (with $m = |M|$) shortest paths of G between vertices of M . For each such shortest path, we associate a corresponding edge e of K_M . These edges composed a tree T_2 spanning K_M . Its weight is then greater than $W(T_P)$: $W(T_P) \leq W(T_2) = W(T_0)$. Combining the inequalities we have: $W(T_0) = W(T_1)$. T_1 is a Steiner tree of M . $\square$

Theorem 3. *Let $G = (V, E)$ be a graph and $M \subseteq V$.*

G contains a S -tree of $M \iff KS(G, M)$ returns a S -tree of M .

Proof. The $\Leftarrow$ part is trivial. Let us prove the other implication. Let T be a S -tree of M . As T is S -tree, it is composed of $m - 1$ (with $m = |M|$) shortest paths of G between vertices of M . At each of these shortest paths, we associate an edge e of K_M . The set of these edges is a tree T' spanning K_M . As T is a S -tree, we have: $W(T) = W(T')$. Like in the description of algorithm KS , let T_P be the minimum weight spanning tree (Prim) of K_M and T_1 be the graph returned by $KS(G, M)$.

As T' is a spanning tree of K_M and T_P is a Prim tree of K_M we have: $W(T_P) \leq W(T')$. From Lemma 1 we know that T is a Steiner tree of M . From Lemma 2, we know that T_1 is also a Steiner tree of M . Hence: $W(T) = W(T_1)$. From proof of Lemma 2 we have: $W(T_1) = W(T_P)$ Combining all these results gives:

$$W(T) = W(T') = W(T_P) = W(T_1) \quad (1)$$

We must prove now that $T' = T_P$. In this case T_1 is also a S -tree.

Suppose that $T' \neq T_P$.

For all $e = uv \in E(T_P)$, let $f(e)$ be the set of edges of $E(T')$ of the unique path between u and v in T' (if $e \in E(T')$, $f(e) = \{e\}$). As T' is a S -tree of M in K_M , we have:

$$\forall e \in E(T_P), \quad w(e) = W(f(e)) \quad (2)$$

Moreover, as $T_P \neq T'$, there exists $e \in E(T_P)$ such that $|f(e)| > 1$ and then:

$$\sum_{e \in E(T_P)} |f(e)| > |M| - 1 \quad (3)$$

We also have:

$$\forall e' \in E(T'), \exists e \in E(T_P), \quad e' \in f(e) \quad (4)$$

Indeed let $e' = uv \in E(T')$ and let V_u (resp. V_v) be the set of vertices of T' in the subtree of T' containing u (resp. v) obtained by deleting e' from T' . As T_P spans K_M , there exists an edge $e \in E(T_P)$ with an extremity in V_u and the other in V_v . It is then clear that $e' \in f(e)$.

From (3) we know that there exists $e_0 \in E(T')$ such that e_0 is in several $f(e)$ ($e \in E(T_P)$).

Combining this fact with (1), (2) and (4) we have:

$$W(T') = \sum_{e \in E(T_P)} w(e) = \sum_{e \in E(T_P)} W(f(e)) \geq w(e_0) + \sum_{e \in E(T')} w(e) > W(T')$$

that is a contradiction and $T' = T_P$. $\square$

Knowing either a given G contains a S -tree for M or not is then polynomial; just apply KS and control (in polynomial time) that the returned graph is or is not a S -tree of M .

Acknowledgements. Work supported by the French CNRS project *AcTAM*.

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti Spaccamela, and M. Protasi. *Complexity and approximation*. Springer, 1999.
2. J.A. Bondy and U.S.R. Murty. *Graph theory with applications*. North-Holland, 1976.
3. M. Elkin and D. Peleg. The harness of approximating spanner problems. In Springer Verlag, editor, *STACS*, volume 1770 of *Lecture notes in computer science*, pages 370–381, 2000.
4. D. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS publishing compagny, 1997.
5. S. Hougardy and H. Prömel. A 1.598 approximation algorithm for the steiner problem in graphs. In *Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM-SIAM*, pages 448–453, 1999.
6. A. Irlande, J.-C. König, and C. Laforest. Construction of low-cost and low-diameter steiner trees for multipoint groups. In G. Proietti M. Flammini, E. Nardelli and P. Spirakis, editors, *Proceedings of Sirocco 2000*, pages 197–210. Carleton Scientific, 2000.
7. S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, (14):305–321, 1995.
8. G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *J. algorithms*, (17):22–236, 1994.
9. G. Kortsarz and D. Peleg. Approximating shallow-light trees. In *Eighth ACM-SIAM Symp. on Discrete Algorithms*, pages 103–110, 1997.
10. C. Laforest. A good balance between weight and distance for multipoint trees. In *International Conference On Principles Of Distributed Systems (OPODIS'02)*, pages 195–204, 2002.
11. C. Laforest, A. Liestman, D. Peleg, T. Shermer, and D. Sotteau. Edge-disjoint spanners of complete graphs and complete digraph s. *Discrete mathematics*, 203:133–159, 1999.
12. C. Laforest, A. Liestman, T. Shermer, and D. Sotteau. Edge-disjoint spanners of complete bipartite graphs. *Discrete mathematics*, 234:65–76, 2001.
13. A.L. Liestman and T. Shermer. Grid spanners. *Networks*, (23):123–133, 1993.
14. M. Marathe, R. Ravi, R. Sundaram, S. Ravi, Rosenkrantz, and H. D. Hunt III. Bicriteria network design problems. *Journal of algorithms*, 28:142–171, 1998.
15. D. Peleg and Schöffer A. Graph spanners. *J. Graph Theory*, (13):99–116, 1989.
16. G. Robins and Zelikovsky. Improved steiner tree approximation in graphs. In *SODA 2000*.
17. H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. *Math. Jap.*, (24):573–577, 1980.

c -Perfect Hashing Schemes for Binary Trees, with Applications to Parallel Memories

(Extended Abstract)

Gennaro Cordasco¹, Alberto Negro¹, Vittorio Scarano¹, and
Arnold L. Rosenberg²

¹ Dipartimento di Informatica ed Applicazioni “R.M. Capocelli”,
Università di Salerno, 84081, Baronissi (SA) – Italy
{alberto,vitsca,cordasco}@dia.unisa.it

² Dept. of Computer Science,
University of Massachusetts Amherst
Amherst, MA 01003, USA
rsnbrg@cs.umass.edu

Abstract. We study the problem of mapping tree-structured data to an ensemble of parallel memory modules. We are given a “conflict tolerance” c , and we seek the smallest ensemble that will allow us to store *any* n -vertex rooted binary tree with no more than c tree-vertices stored on the same module. Our attack on this problem abstracts it to a search for the smallest *c -perfect universal graph for complete binary trees*. We construct such a graph which witnesses that only $O\left(c^{(1-1/c)} \cdot 2^{(n+1)/(c+1)}\right)$ memory modules are needed to obtain the required bound on conflicts, and we prove that $\Omega\left(2^{(n+1)/(c+1)}\right)$ memory modules are necessary. These bounds are tight to within constant factors when c is fixed—as it is with the motivating application.

1 Introduction

Motivation. This paper studies the efficient mapping of data structures onto a parallel memory system (*PMS*, for short) which is composed of several *modules* that can be accessed simultaneously (by the processors of, say, a multiprocessor system). Mapping a data structure onto a PMS poses a challenge to an algorithm designer, because such systems typically are single-ported: they queue up simultaneous accesses to the same memory module, thereby incurring delay. The effective use of a PMS therefore demands efficient mapping strategies for the data structures that one wishes to access in parallel—strategies that minimize, for each memory access, the delay incurred by this queuing. Obviously, different mapping strategies are needed for different data structures, as well as for different ways of accessing the same data structure. As a simple example of the second point, one would map the vertices of a complete binary tree quite differently when optimizing access to levels of the tree than when optimizing access to root-to-leaf paths.

The preceding considerations give rise to the problem studied here. Given a data structure represented by a graph, and given the kinds of subgraphs one wants easy access to (called *templates*), our goal is to design a memory-mapping strategy for the items of a data structure that minimizes the number of simultaneous requests to the same memory module, over all instances of the considered templates.

Our Results. This paper presents the first strategy for mapping a binary-tree data structure onto a parallel memory in such a way that *any* rooted subtree can be accessed with a bounded number of conflicts. Our results are achieved by proving a (more general) result, of independent interest, about the sizes of graphs that are “almost” *perfect universal* for binary trees, in the sense of [5].

Related Work. Research in this field originated with strategies for mapping two-dimensional arrays into parallel memories. Several schemes have been proposed [3,4,8,10] in order to offer conflict-free access to several templates, including rows, columns, diagonals and submatrices. While the strategies in these sources provide conflict-free mappings, such was not their primary goal. The strategies were actually designed to accommodate as many templates as possible.

Strategies for mapping tree structures considered conflict-free access for one *elementary* template—either complete subtrees or root-to-leaf paths or levels [6,7] or combinations thereof [2], but the only study that even approaches the universality of our result—i.e., access to *any* subtree—is the C-template (“C” for “composite”) of [1], whose instances are combinations of different numbers of distinct elementary templates. The mapping strategy presented for the C-templates’ instances of size K , with M memory modules, achieves $O(K/M + c)$ conflicts.

Background. For any binary tree T : $\text{Size}(T)$ is its number of vertices; $\text{Size}(T, i)$ is its number of level- i vertices; $\text{Hgt}(T)$ is its height (= number of levels).

For any positive integer c , a c -contraction of a binary tree T is a graph G that is obtained from T via the following steps.

1. Rename T as $G^{(0)}$. Set $k = 0$.
2. Pick a set S of $\leq c$ vertices of $G^{(k)}$ that were vertices of T . Remove these vertices from $G^{(k)}$, and replace them by a single vertex that represents the set S . Replace all of the edges of $G^{(k)}$ that were incident to the removed vertices by edges that are incident to S . The graph so obtained is $G^{(k+1)}$.
3. Iterate step 2 some number of times.

A graph $G_n = (V_n, E_n)$ is c -*perfect-universal* for the family $\mathbf{T}_n$ of n -vertex binary trees if every c -contraction of an n -vertex binary tree is a *labeled-subgraph* of G_n . By this we mean the following. Given any c -contraction $G^{(a)} = (V, E)$ of an n -vertex binary tree T , the fact that $G^{(a)}$ is a subgraph of G_n is observable via a mapping $f : V \rightarrow V_n$ for which each $v \in V$ is a subset of $f(V)$.

The simplest 1-perfect-universal graph for the family $\mathbf{T}_n$ is the height- n *complete binary tree*, $\mathcal{T}_n$, which is defined by the property of having all root-to-leaf paths of common length $\text{Hgt}(\mathcal{T}_n) = n$. The perfect-universality of $\mathcal{T}_n$ is witnessed by the identity map f of the vertices of any n -vertex binary tree to the

vertices of $\mathcal{T}_n$. Of course, $\mathcal{T}_n$ is a rather inefficient perfect-universal graph for $\mathbf{T}_n$, since it has $2^n - 1$ vertices, whereas each tree in $\mathbf{T}_n$ has only n vertices. It is natural to ponder how much smaller a 1-perfect-universal graph for $\mathbf{T}_n$ can be. The size—in number of vertices—of the smallest such graph is called the *perfection number* of $\mathbf{T}_n$ and is denoted $\text{Perf}(\mathbf{T}_n)$. In [5], $\text{Perf}(\mathbf{T}_n)$ is determined exactly, via coincident lower and upper bounds.

Theorem 1 ([5]) $\text{Perf}(\mathbf{T}_n) = (3 - (n \bmod 2)) 2^{\lfloor (n-1)/2 \rfloor} - 1$.

In this paper, we generalize the study of storage mappings for trees in [5] by allowing boundedly many *collisions* in storage mappings. We thus relax the “one to one” demands of perfect hashing to “boundedly many to one.”

2 Close Bounds on $\text{Perf}_c(\mathbf{T}_n)$

Our study generalizes Theorem 1 to c -perfect-universality, by defining $\text{Perf}_c(\mathbf{T}_n)$, for any positive integer c , to be the size of the smallest c -perfect-universal graph for $\mathbf{T}_n$; in particular, $\text{Perf}_1(\mathbf{T}_n) = \text{Perf}(\mathbf{T}_n)$.

We first derive our upper bound on $\text{Perf}_c(\mathbf{T}_n)$ by explicitly constructing a graph G_c that is c -perfect-universal for $\mathbf{T}_n$.

Theorem 2 *For all integers n and $c > 1$,*

$$\text{Perf}_c(\mathbf{T}_n) < \left(2 + 2c^{1-1/c}\right) 2^{(n+1)/(c+1)} + O(n). \quad (1)$$

We construct our c -perfect-universal graph G_c via an algorithm $\mathcal{A}_c$ that colors the vertices of $\mathcal{T}_n$ in such a way that the natural vertex-label-preserving embedding of any n -vertex tree T into $\mathcal{T}_n$ (which witnesses T ’s being a subgraph of $\mathcal{T}_n$) never uses more than c vertices of any given color as homes for T ’s vertices. When we identify each like-colored set of vertices of $\mathcal{T}_n$ —i.e., contract each set to a single vertex in the obvious way—we obtain the c -perfect-universal graph G_c , whose size is clearly an upper bound on $\text{Perf}_c(\mathbf{T}_n)$.

Algorithm $\mathcal{A}_c$ proceeds in a left-to-right pass along each level of $\mathcal{T}_n$ in turn, assigning a unique set of colors, C_i , to the vertices of each level i , in a round-robin fashion. $\mathcal{A}_c$ thereby distributes the 2^i level- i vertices of $\mathcal{T}_n$ equally among the $|C_i|$ level- i vertices of G_c . Clearly, thus, $\text{Size}(G_c) = \sum_i |C_i|$.

The remainder of the section is devoted to estimating how big the sets C_i must be in order for G_c to be c -perfect-universal for $\mathbf{T}_n$.

Auxiliary results. We begin by identifying some special subtrees of $\mathcal{T}_n$. Let m and i be integers such that¹ $\lceil \log m \rceil \leq i \leq n$, and let x be a binary string of length $i - \lceil \log m \rceil$. Consider the subtree $T_{(i,m)}(x)$ of $\mathcal{T}_n$ constructed as follows.

1. Generate a length- $(i - \lceil \log m \rceil)$ path from the root of $\mathcal{T}_n$ to vertex x .

¹ All logarithms are to the base 2.

2. Generate the smallest complete subtree rooted at x that has at least m leaves; easily, this subtree—call it $\mathcal{T}(x)$ —has height $\lceil \log m \rceil$.
3. Finally, prune the tree so constructed, removing all vertices and edges other than those needed to incorporate the leftmost m leaves.

Easily, every tree $T_{(i,m)}$ has exactly m leaves, all of length i .

Lemma 1. *The trees $T_{(i,m)}$ are the smallest rooted subtrees of $\mathcal{T}_n$ that have m leaves at level i .*

Proof. (Sketch) For any level j of any binary tree T , we have $\text{Size}(T, j-1) \geq \lceil \frac{1}{2} \text{Size}(T, j) \rceil$. One verifies easily from our construction that the trees $T_{(i,m)}$ achieve this bound with equality. $\square$

Lemma 2. *For all i and m ,*

$$2m + i - \lceil \log m \rceil - 1 \leq \text{Size}(T_{(i,m)}) \leq 2m + i - 1. \quad (2)$$

Proof. (Sketch) We merely sum the sizes of the complete subtree on $\mathcal{T}(x)$, plus the size of the various paths needed to “capture” them. Thus we have

$$\text{Size}(T_{(i,m)}) = (i - h_{p_m-1} + 1) + 2m - p_m \quad (3)$$

where p_m is the number of complete subtree on $\mathcal{T}(x)$ and h_{p_m-1} is the height of the smallest complete subtree on $\mathcal{T}(x)$. We obtain the bound of the lemma from the exact, but nonperspicuous, expression (3), from the facts that there is at least one 1 in the binary representation of m and that $p_m \geq 1$. $\square$

Let us now number the vertices at each level of $\mathcal{T}_n$ from left to right and say that the *distance* between any two such vertices is the magnitude of the difference of their numbers.

Lemma 3. *For any integers $0 \leq i \leq n$ and $0 \leq \delta_i < i$, the size of the smallest rooted subtree of $\mathcal{T}_n$ that has m leaves at level i of $\mathcal{T}_n$, each at distance $\geq 2^{\delta_i}$ from the others, is no smaller than $\text{Size}(T_{(i,m)}) + (m-1)\delta_i$.*

Proof. If two vertices on level i are distance $\geq 2^{\delta_i}$ apart, then their least common ancestor in $\mathcal{T}_n$ must be at some level $\leq i - (\delta_i + 1)$. It follows that the smallest rooted subtree T of $\mathcal{T}_n$ that satisfies the premises of the lemma must consist of a copy of some $T_{(i-\delta_i,m)}$, with m leaves at level $i - \delta_i$, plus m vertex-disjoint paths (like “tentacles”) from that level down through each of the δ_i levels $i - \delta_i + 1, i - \delta_i + 2, \dots, i$ of $\mathcal{T}_n$. Equation (3) therefore yields:

$$\text{Size}(T) \geq m\delta_i + \text{Size}(T_{(i-\delta_i,m)}) = (m-1)\delta_i + \text{Size}(T_{(i,m)}). \quad \square$$

The upper bound proof. We propose, in the next two lemmas, two coloring schemes for $\mathcal{A}_c$, each inefficient on its own (in the sense that neither yields an upper bound on $\text{Perf}_c(\mathbf{T}_n)$ that comes close to matching our lower bound), but which combine to yield an efficient coloring scheme. We leave to the reader the simple proof of the following Lemma.

Lemma 4. *Let $\mathcal{T}_n$ be colored by algorithm $\mathcal{A}_c$ using 2^{δ_i} colors at each level i . If each*

$$2^{\delta_i} \geq \kappa_i \stackrel{\text{def}}{=} \lceil 2^i / c \rceil,$$

then any rooted n -vertex subtree of $\mathcal{T}_n$ engenders at most c collisions.

Lemma 5. *Let $\mathcal{T}_n$ be colored by algorithm $\mathcal{A}_c$ using 2^{δ_i} colors at each level i . If each²*

$$2^{\delta_i} \geq \lambda_i \stackrel{\text{def}}{=} \frac{1}{4} \exp 2 \left(\left\lceil \frac{n + \log(c+1) - i}{c} \right\rceil \right), \quad (4)$$

then any rooted n -vertex subtree of $\mathcal{T}_n$ engenders at most c collisions.

Proof. We consider a shallowest tree T that engenders $c+1$ collisions at level i of $\mathcal{T}_n$. Easily, the offending tree T has $c+1$ leaves from level i of $\mathcal{T}_n$. By the design of Algorithm $\mathcal{A}_c$, these leaves must be at distance 2^{δ_i} from one another. We can, therefore, combine Lemma 3, the lower bound of (2) (both with $m = c+1$), and (4) to bound from below the size of the offending tree T .

$$\begin{aligned} \text{Size}(T) &\geq \text{Size}(T_{(i, c+1)}) + c\delta_i \\ &\geq 2(c+1) + i - \lfloor \log(c+1) \rfloor - 1 + c \left\lceil \frac{n - 2c + \log(c+1) - i}{c} \right\rceil \geq n + 1. \end{aligned}$$

Since we care only about $(\leq n)$ -vertex subtrees of $\mathcal{T}_n$, the lemma follows. $\square$

A bit of analysis verifies that the respective strengths and weaknesses of the coloring schemes of Lemmas 4 and 5 are mutually complementary. This complementarity suggests the ploy of using the κ_i -scheme to color the “top” of $\mathcal{T}_n$ and the λ_i -scheme to color the “bottom.” A natural place to divide the “top” of $\mathcal{T}_n$ from the “bottom” would be at a level i where $\kappa_i \approx \lambda_i$. Using this intuition, we choose level $i^* \stackrel{\text{def}}{=} \left\lceil \frac{n - c + 1}{c + 1} + \log c \right\rceil$ to be the first level of

the “bottom” of $\mathcal{T}_n$. (Since $c \leq n$, trivial calculations show that $n > i^*$.) Using our hybrid coloring scheme, then, we end up with a c -perfect-universal graph

G_c such that $\text{Size}(G_c) = \sum_{j=0}^{i^*-1} \kappa_j + \sum_{k=i^*}^{n-1} \lambda_k$. Evaluating the two summations

in turn, we find the following, under the assumption that $c > 1$ (since the case $c = 1$ is dealt with definitively in [5]; see Theorem 1).

$$\begin{aligned} \sum_{j=0}^{i^*-1} \kappa_j &= \sum_{j=0}^{i^*-1} \lceil 2^j / c \rceil \leq \sum_{j=0}^{i^*-1} (2^j / c + 1) = \frac{1}{c} 2^{i^*} + i^* - \frac{1}{c} \\ &\leq \frac{1}{c} \exp 2 \left(\left\lceil \frac{n - c + 1}{c + 1} + \log c \right\rceil \right) + O(n) < 2 \cdot 2^{(n+1)/(c+1)} + O(n) \quad (5) \end{aligned}$$

² To enhance the legibility of powers of 2 with complicated exponents, we often write $\exp 2(X)$ for 2^X .

$$\begin{aligned}
\sum_{k=i^*}^{n-1} \lambda_k &= \sum_{k=i^*}^{n-1} \exp 2 \left(\left\lceil \frac{n + \log(c+1) - k}{c} \right\rceil - 2 \right) < \frac{(c+1)^{1/c}}{2} \cdot \sum_{k=i^*}^{n-1} 2^{(n-k)/c} \\
&< 2 \sum_{k=i^*}^{i^*+c-1} 2^{(n-k)/c} \leq 2c \cdot \exp 2 \left(\frac{n}{c} - \left(\frac{n-c+1}{c(c+1)} + \frac{\log c}{c} \right) \right) \\
&< 2 c^{1-1/c} \cdot 2^{(n+1)/(c+1)}.
\end{aligned} \tag{6}$$

The bounds (5, 6) yield the claimed upper bound (1) on $\text{Perf}_c(\mathbf{T}_n)$. $\square$

Because of space limitations, we defer the proof of the following lower bound to the complete version of this paper.

Theorem 3 *For all $c > 1$ and all n ,*

$$\text{Perf}_c(\mathbf{T}_n) > \exp 2 \left(\left\lfloor \frac{n+1}{c+1} - \frac{11}{3} \right\rfloor \right).$$

Acknowledgment. The research of A.L. Rosenberg was supported in part by US NSF Grant CCR-00-73401.

References

1. V. Auletta, S. Das, A. De Vivo, M.C. Pinotti, V. Scarano, "Optimal tree access by elementary and composite templates in parallel memory systems". *IEEE Trans. Parallel and Distr. Sys.*, 13, 2002.
2. V. Auletta, A. De Vivo, V. Scarano, "Multiple Template Access of Trees in Parallel Memory Systems". *J. Parallel and Distributed Computing* 49, 1998, 22–39.
3. P. Budnik, D.J. Kuck. "The organization and use of parallel memories". *IEEE Trans Comput.*, C-20, 1971, 1566–1569.
4. C.J. Colbourn, K. Heinrich. "Conflict-free access to parallel memories". *J. Parallel and Distributed Computing*, 14, 1992, 193–200.
5. F.R.K. Chung, A.L. Rosenberg, L. Snyder. "Perfect storage representations for families of data structures." *SIAM J. Algebr. Discr. Meth.*, 4, 1983, 548–565.
6. R. Creutzburg, L. Andrews, "Recent results on the parallel access to tree-like data structures – the isotropic approach", *Proc. Intl. Conf. on Parallel Processing*, 1, 1991, pp. 369–372.
7. S.K. Das, F. Sarkar, "Conflict-free data access of arrays and trees in parallel memory systems", *Proc. 6th IEEE Symp. on Parallel and Distributed Processing*, 1994, pp. 377–383.
8. D.H. Lawrie. "Access and alignment of data in an array processor". *IEEE Trans. on Computers*, C-24, 1975, 1145–1155.
9. R.J. Lipton, A.L. Rosenberg, A.C. Yao, "External hashing schemes for collections of data structures." *J. ACM*, 27, 1980, 81–95.
10. K. Kim, V.K. Prasanna. "Latin Squares for parallel array access". *IEEE Trans. Parallel and Distributed Systems*, 4, 1993, 361–370.
11. A.L. Rosenberg and L.J. Stockmeyer, "Hashing schemes for extendible arrays." *J. ACM*, 24, 1977, 199–221.
12. A.L. Rosenberg, "On storing ragged arrays by hashing." *Math. Syst. Th.*, 10, 1976/77, 193–210.

A Model of Pipelined Mutual Exclusion on Cache-Coherent Multiprocessors

Masaru Takesue

Dept. Electronics and Information Engr., Hosei University, Tokyo 184-8584 Japan
takesue@ami.ei.hosei.ac.jp

Abstract. This paper proposes a model of *pipelined* mutual exclusion on cache-coherent multiprocessors that allows processors to concurrently access different memory locations of shared data in a pipelined manner. The model converts the *effective unit* of mutual exclusion from shared data to one memory location, by producing the total order of requests for each location while preserving the total order for shared data in the order for location. Two implementations of the model are evaluated.

Keywords: Models, mutual exclusion, pipelining.

1 Introduction

In cache-coherent multiprocessors, communication is performed through synchronization. Mutual exclusion is the synchronization operation for ensuring write atomicity on a set $\mathcal{D}$ of shared data. Since it serializes the accesses to $\mathcal{D}$, its latency with a naive scheme such as the test&set is surprisingly great when a large number of processors are competing for $\mathcal{D}$, leading to a crucial obstacle to fine-grain parallel computation.

The latency is greatly reduced, especially when the contention rate on $\mathcal{D}$ is large, by *pipelining* mutual exclusion so that processors can concurrently access different memory locations for $\mathcal{D}$ in a pipelined manner [1]; the pipelining is implemented with a software-hardware hybrid scheme. The contribution of this paper is to formalize a model of the pipelined mutual exclusion.

The key idea behind the model is to convert the *effective unit* of mutual exclusion from $\mathcal{D}$ to one memory location, ensuring mutual exclusion on $\mathcal{D}$. For this end, the model produces a total order of requests for *each memory location* of $\mathcal{D}$, preserving the total order of requests for $\mathcal{D}$ in the total order for each location. For simplicity of discussion, we restrict a memory *location* to a memory-block location in the paper, though it can be a memory-word location [1].

Related work: Of the previous work for mutual exclusion [2]-[8], software queue-based algorithms are most promising in heavily contentious cases [3]-[5], since the produced total order of requesters allows them to spin locally for the access right on $\mathcal{D}$. The QOLB [6] is one of the hardware queue-based schemes [8,6], and outperforms over the software counterparts [7]. However, none of the previous work addresses location-based pipelined mutual exclusion.

A recently proposed scheme [9] allows multiple threads to concurrently access *falsely shared locations* of $\mathcal{D}$ by speculatively removing unnecessary locking; so this scheme addresses a very restricted case in our model. As for the memory consistency models, the data-race-free-1 model [10] can release each memory block before the release lock on $\mathcal{D}$; this leads to the pipelining but needs a complex hardware scheme to match the arrival of the block with the access on it. The release consistency model [11] can overlap the writes in the critical section (CS) for a $\mathcal{D}$ and the reads in the CS for another $\mathcal{D}$ in *one processor*. Notice that this is not the pipelining among the CSs for a *same* $\mathcal{D}$ in *separate processors* as addressed in this paper.

The rest of the paper is organized as follows: Section 2 introduces the model. Sections 3 evaluates a software and a software-hardware hybrid implementations of the model, comparing with the most efficient non-pipelined software [3] and hardware [6] queue-based algorithms. Section 4 concludes the paper.

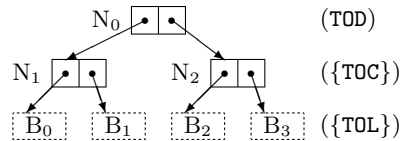
2 The Model

Let a set $\mathcal{D}$ of shared data be allocated to n memory blocks, B_0 to B_{n-1} . Each processor is assumed to *selectively* access some memory blocks of $\mathcal{D}$ that are required in the processor. Let TOD denote the *total order* of processors requesting for $\mathcal{D}$, and TOL_i be the *total order* of processors requesting for block B_i ; the set $\{TOL_i | 0 \leq i \leq n-1\}$ is denoted by $\{TOL\}$.

To preserve the TOD in each TOL, we introduce a tree data structure, called *order preserving tree* (OPT). The logical structure of an OPT node is shown in Fig. 1-(a). The tail pointer, **tail**, has the identifier of the tail processor in a total order (described shortly). An example of the OPT for a set $\mathcal{D}$ of 4 memory blocks is shown in Fig. 1-(b). The memory blocks for $\mathcal{D}$ are depicted by the dashed boxes since those are not included in the OPT but *associated* with the leaf nodes.

```
struct OPTnode {
    enum boolean nonleaf;
    struct {
        tail; /* of a TOC/TOL */
        struct OPTnode *ptr;
    } child[C];
};
```

(a) The logical structure of an OPT node



(b) An example of the OPT

Fig. 1. The logical structure of an OPT node, and an OPT for 4-block shared data

In reality, we map the **tail** pointers onto an array so that those in one node are packed in one memory block, and reorganize the **ptr**'s and arity of the tree as another data structure. Then the **tail** in a node indicates the tail processor in the total order, TOC, for its *child* node (memory block) if it is a **nonleaf**, or in the TOL for the *associated* memory block otherwise. For instance, in Fig. 1-(b), nonleaf node N_0 has the **tails** of the TOCs for the child nodes N_1 and N_2 , and leaf node N_1 has the **tails** of the TOLs for the associated blocks B_0 and B_1 .

With the OPT, the CS entry routine, **enterCS**, for $\mathcal{D}$ produces the $\{\text{TOL}\}$ from the TOD, preserving the TOD in each TOC and especially in each TOL. A processor accesses a memory block in the *blockwise critical section*, BwCS, for the block. Before and after the BwCS, we need an **enterBwCS** and an **exitBwCS** for the block to acquire and release the access right on the block.

The **enterCS** is shown in Fig. 2. The **root** points to the root node of the OPT. The **SCANQ** is a queue for scanning the tree in breadth-first order. A requesting processor I is put into the TOD by the **serialize**. The for-loop corresponds to the BwCS for memory block **myP** allocated to an OPT node. The **enterBwCS** and **exitBwCS** acquire and release the access right on block **myP**. In the BwCS, the **serializeChild** puts identifier I into the TOC or TOL for block **childP** (i.e., for the child node or the associated block of $\mathcal{D}$) if it is required in the processor.

```

enterCS(root,I)
{
  serialize(root,I); /* TOD */
  enqSCANQ(root);
  while (nonemptySCANQ()) {
    myP = deqSCANQ();
    enterBwCS(myP);
    for (i=0; i<C; ++i) /* BwCS(myP) */
      if (required(i)) {
        childP = myP->child[i].ptr;
        childT = myP->child[i].tail;
        serializeChild(childP,childT,I); /* TOC/TOL */
        updateTail(myP->child[i].tail,I);
        if (myP->nonleaf)
          enqSCANQ(childP);
      }
    exitBwCS(myP);
  }
}

```

Fig. 2. The **enterCS** routine

The CS routine, **CS**, for $\mathcal{D}$ consists of multiple tuples each of **enterBwCS**, BwCS, and **exitBwCS**. The number of tuples is equal to the number of memory blocks required in the processor, and the elements of one tuple can interleave with those of another tuple. No exit-CS routine is necessary because of the blockwise release of the access right in the **exitBwCS**. Now we define the model $\mathcal{M}$ of pipelined mutual exclusion, and prove that the pipelining is possible.

Definition 1. *Model $\mathcal{M}$ consists of the TOD, $\{\text{TOC}\}$, $\{\text{TOL}\}$, OPT, **enterCS**, and CS that comprises multiple tuples each of **enterBwCS**, BwCS, and **exitBwCS**.*

Theorem 1. *Model $\mathcal{M}$ ensures and pipelines mutual exclusion on a set $\mathcal{D}$ of shared data.*

Proof. Assume that processor I is a predecessor of processor J in the TOD and both access blocks B_i and B_j ($i \neq j$) of $\mathcal{D}$. If processor I is a predecessor of

processor J in the total order for an **OPT** node, the **enterCS** puts the former before the latter in the **TOC** for its child node or in the **TOL** for the associated memory block, since the former receives the **OPT** node earlier than the latter. So by induction, processor I is put before processor J , especially in the **TOLs** for blocks B_i and B_j . Thus processor I certainly accesses those blocks earlier than processor J . This ensures mutual exclusion on $\mathcal{D}$. As for the pipelining, since a processor releases the access right on a memory block basis, the successors in the **TOL** for the block can access it on receiving the access right.

Theorem 2. *Model $\mathcal{M}$ works with the relaxed memory consistency models.*

Proof. A program correctly synchronized under the sequential consistency model produces the same result as with any of the relaxed memory models (such as the release consistency model) [11,10]. In $\mathcal{M}$, the pair of an **enterBwCS**(B) and an **exitBwCS**(B) always synchronizes the memory operations on each block B .

3 Evaluation

This section describes our simulation environment, and shows the potential performance and overhead of the pipelined mutual exclusion.

3.1 Simulator and Evaluated Algorithms

We evaluate with an RTL (register transfer level) simulator of our system with up to 256 processing nodes (PNs). The system parameters are listed in Table 1. Four mutual exclusion algorithms are evaluated; the pipelined algorithms, **PpS** and **PpH**, based on model $\mathcal{M}$ with software and hardware queues, respectively, and the non-pipelined algorithms, **NpS** and **NpH**, with software and hardware queues. The software queue is implemented on an array by using the Fetch&Inc [3]. The hardware queue is organized in the requesting caches by specific instructions and an extended cache protocol; see [1] for the **PpH**, and [6] for the **NpH**.

Table 1. System parameters

Item	Description
processor (PE)	extended DLX [12]
cache	unified, blocking, 32-byte line, set-associative
network (NW)	hypercube, wormhole routing, 1 byte/hop-cycle
cycle time [clock]	PE/cache/NW-switch: 1, memory unit (in each PN): 10

For the **NpS** and **NpH**, we acquire the locks on all memory blocks of $\mathcal{D}$ in the **CS** (due to no **OPT**), while we access the blocks and release the locks on a block basis outside the **CS**, because this may lead to a sort of the pipelining, though the *degree of overlap* in the pipelining, i.e., the number of processors that are concurrently in the individual **CSs** for $\mathcal{D}$, may be small.

3.2 Results

The speedup of the PpS and NpS over the NpH is shown in the left part of Fig. 3; the degrees of overlap obtained with the NpH is always one (i.e., no overlapping). The PpS improves the performance of NpS by a factor of about 1.0 to 2.23, leading to a speedup of about 0.25 to 1.38 over the NpH. Surprisingly, the speedup reaches up to over 1.0 with no hardware support except of the Fetch&Inc. The PpH's speedup over the NpH is about 1.0 to 88.2 as shown in the right part of Fig. 3. As the sizes of data and system increase, a greater speedup is achieved owing to a larger degree of overlap up to about 91.2.

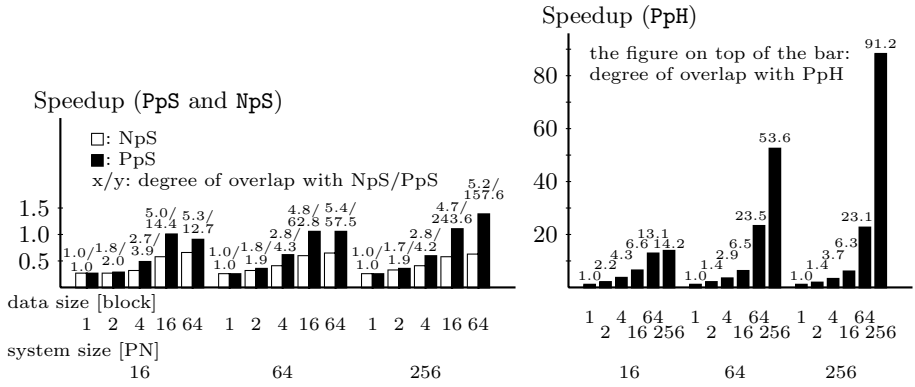


Fig. 3. Speedup of the PpS and NpS (left), and PpH (right) over the NpH

The overhead of PpS and PpH, i.e., their performance relative to the NpH's in light-load cases, is shown in Fig. 4, where n_p ($= 1, 2$, or 4) number of PNs are contesting for $\mathcal{D}$ on the 256-PN system. The relative execution time of PpS and PpH is about 1.8 to 5.7, and about 0.3 to 1.0.

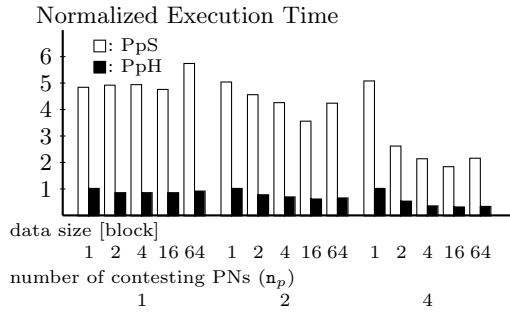


Fig. 4. Performance of the PpH and PpS relative to the NpH's in light-load cases

4 Conclusions

Our model of pipelined mutual exclusion converts effectively the unit of mutual exclusion from shared data to one memory location, ensuring mutual exclusion on shared data, so that processors can concurrently access different locations of shared data. The model is orthogonal to memory consistency models since it correctly synchronizes memory operations on a location basis.

A remarkable result of evaluation is that the software implementation (omitted for space, and will be presented in a future paper) of the model outperforms over a hybrid non-pipelined algorithm when the contention rate is high; then the hybrid implementation [1] is much more effective than the software one.

A scheme for pipelining conditional synchronization [2] will be reported in another paper [13]. For a smaller communication latency, we are designing a scheme for dynamically clustering the requests for the pipelined synchronization.

References

1. Takesue, M.: Pipelined Mutual Exclusion on Large-Scale Cache-Coherent Multiprocessors. Proc. the 14th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (2002) 745–754
2. Dinning, A.: A Survey of Synchronization Methods for Parallel Computers. IEEE Computer **22** (1989) 3–43
3. Anderson, T. E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. IEEE Trans. on Parallel and Distributed Systems, **1** (1990) 6–16
4. Graunke, G., Thakkar, S.: Synchronization Algorithms for Shared-Memory Multiprocessors. IEEE Computer **23** (1990) 60–69
5. Grummey, J. M. M., Scott, M. L.: Synchronization Without Contention. Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (1991) 269–278
6. Goodman, J. R., Vernon, M. K., Woest, P. J.: Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. Proc. 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (1989) 64–75
7. Kägi, A., Burger, D., Goodman, J. R.: Efficient Synchronization: Let Them Eat QOLB. Proc. 24th Int. Symp. on Computer Architecture (1997) 170–180
8. Lee, J., Ramachandran, U.: Synchronization with Multiprocessor Caches. Proc. 17th Int. Symp. on Computer Architecture (1990) 27–37
9. Rajwar, R., Goodman, J. R.: Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. Proc. 34th Int. Symp. on Microarchitecture (2001) 294–305
10. Adve, S. V., Hill, M. D.: A Unified Formalization of Four Shared-Memory Models. IEEE Trans. on Parallel and Distributed Systems **4** (1993) 613–624
11. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Proc. 17th Int. Symp. on Computer Architecture (1990) 15–26
12. Hennessy, J. L., Patterson, D. A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers (1990)
13. Takesue, M.: Pipelined Conditional Synchronization on Large-Scale Cache-Coherent Multiprocessors. Proc. the 16th ISCA Int. Conf. on Parallel and Distributed Computing Systems (2003) accepted for presentation

Efficient Parallel Multiplication Algorithm for Large Integers

Viktor Bunimov and Manfred Schimmler

Institute for Computer Engineering and Communication Networks,
Technical University of Braunschweig, Germany
v.bunimov@tu-bs.de, m.schimmler@tu-bs.de

Abstract. A new algorithm for calculating $A*B$ is presented. It is based on precomputation of a small number of values that allow for an efficient parallel implementation of multiplication. The algorithm requires $O(\log n)$ time and $O(n^2/\log n^2)$ area with small constant factors which makes it feasible for practical implementations.

Keywords: Computer arithmetic, large number arithmetic, redundant numbers, carry save addition, parallel multiplication, integer multiplication.

1 Introduction

Integer multiplication is one of the most important operations in computer arithmetic. There are many algorithms and architectures for integer multiplication. In classical algorithm for integer multiplication the bits of one operand are multiplied with shifted copies of the second operand and the results are added [5], [13]. If the operands are very long (the number of bits $n > 128$), the algorithm requires too many additions (n). One addition has time complexity of $\theta(\log n)$ [13] and area (hardware) complexity of $\theta(n)$, consequently the time complexity of this multiplication algorithm is $O(n * \log n)$ and the AT-complexity (area-time complexity) is $O(n^2 \log n)$. There are a many other algorithms for integer multiplication: Karatsuba multiplication [4], Toom multiplication [11], FFT multiplication, Schönhage-Strassen multiplication [8] and others. All these are sequential algorithms with good AT-complexity, but, compared to parallel methods which can be implemented in hardware, the time complexity is not good: The Karatsuba multiplication has time complexity of $O(n^{\log 3})$. Toom multiplication has time complexity of $O(n * \log^{1/2} n)$. Winograd [14], Cook [5], Zuras [15] and Knuth [5] developed versions of Toom multiplication with different improvements. Another interesting development is multiplication using Fast Fourier Transform (FFT) [2],[3]. One of the best algorithms is the Schönhage-Strassen multiplication with the time complexity of $O(n \log n \log \log n)$.

But the time complexity of every sequential algorithm is asymptotically greater than $O(\log n)$. The fastest parallel algorithms for integer multiplication has time complexity $O(\log n)$ [9], [12], [13] which has been proven to be optimal.

One of the possibilities to speed up the multiplication is the parallelisation of the suboperations of multiplication. There are a many parallel algorithms for integer multiplication: Wallace-tree multipliers [1], [10], [12] use a parallelisation of the

classical algorithm for multiplication with redundant numbers for intermediate representation. These provide an optimal time complexity of $O(\log n)$, but the area complexity is $O(n^2)$; consequently, the AT-complexity is $O(n^2 \log n)$. The parallel algorithm for the integer multiplication in [9] has an optimal time complexity of $O(\log n)$ too, and the AT-complexity is reduced to $O(n^3)$. The best parallel algorithm for parallel multiplication is the parallelisation of the Schönhage-Strassen multiplication [8],[13] with time complexity of $O(\log n)$ and area complexity of $O(n \log n \log \log n)$, thus the AT-complexity is $O(n \log^2 n \log \log n)$. But this algorithm is not used in practical applications, because the constants (which are hidden in the O -notation) are too large.

In this paper we present a new parallel algorithm for integer multiplication with optimal time complexity of $O(\log n)$ and AT-complexity of $O(n^2/\log n)$. In contrast to earlier time- and area-optimal algorithms, our method is suited for real world implementations because of its small constant.

This paper is organised as follows. In section 2, the idea of the new algorithm is presented. Section 3 gives a short introduction into fast adder technology using redundant number representations for intermediate results. The final version of our new algorithm exploiting these techniques to optimally speed up the algorithm of section 2 is given in Section 4. Finally, in Section 5 we show the exact complexity figures for an implementation example which concludes the paper.

2 Basic Version of the New Algorithm

Our goal is the computation of $P = X * Y$, where X and Y are large n -bit numbers (typical for our applications: $n \geq 1024$). We want to do this in optimal time, i.e. $T = O(\log n)$ and with limited area $A = O(n^2/\log^2 n)$. In addition we are interested in keeping the constants small, since the well known algorithms with optimal AT-complexity are not used in practical systems because their constant factors are too large. To shorten the notation throughout this paper we define two variables $k := \log^2 n$ and $t := \log n^{1/2}$.

Initially, we subdivide X into intervals of k bits. There are a n/k intervals of k bits. We subdivide each interval of k bits into subintervals of t bits:

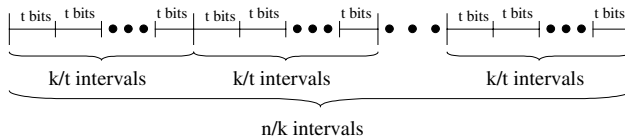


Fig. 1. Any interval of k bits is subdivided into intervals of t bits each.

Now our method proceeds in three steps:

Step 1 calculates the products of Y with every possible t -bit value and stores the results in registers for further usage.

Step 2 calculates the products of Y with all the k -bit-pieces of X .

Step 3 combines these partial products to get the product of X and Y .

We will show that each step can be performed with $O(n^{1/2} + n/\log^2 n)$ adders (and

registers) in at most $\log n$ steps, which meets the time and area requirements stated above.

Step 1: There are 2^t partial products to be computed. This operation is performed completely in parallel. 2^t adders are responsible for one value each. Each adder computes the product of one individual t -bit-number and Y . This multiplication is performed according to the classical school method, such that $\log t$ shifted copies of Y or 0, respectively, are added to each other.

This operation requires $\log n^{1/2}$ additions of length $n+t < 2n$. The results are stored in a set of registers in order to wait for subsequent computation in step 2.

In the step 1 we use 2^t adders and a lookup table with 2^t entries. Thus the area complexity of step 1 is

$$A_1 = O(n \cdot 2^t) = O(n^{3/2}) \quad (1)$$

The time complexity is given by the number of additions which is $\log n^{1/2}$ and therefore

$$T_1 = O(\log n) T_{\text{add}} \quad (2)$$

Step 2: In this step n/k partial products (the products of Y times the k -bit pieces of X) are computed in parallel. n/k adders are responsible for this task, each one for one partial product. Each computation consists of k/t additions, where the operands are taken from the registers whose values have been computed in Step 1. The process for each k -bit piece is purely sequential. The adders start with an initial value of 0 as intermediate result. In the first step each adder reads the value from one of the registers which corresponds to the least significant t bits of its k -bit interval of X . It is added to the intermediate result. After that the next t bits of the k -bit piece is taken. The corresponding value is taken from the appropriate register, it is shifted by t bits, and added to the intermediate result. In the same way the complete k -bit substring of X is worked through in k/t substeps corresponding to the k/t pieces of t bits. The final result is the product of Y and the k -bit substring of X .

Since we need n/k adders of size $2n$ each, the area respectively time complexity of step 2 is

$$A_2 = O(n^2/k) = O(n^2/\log^2 n), T_2 = O(k/t) T_{\text{add}} = O(\log^2 n / \log n^{1/2}) T_{\text{add}} = O(\log n) T_{\text{add}} \quad (3)$$

Step 3: The n/k partial products of step 2 are added to one final result. $n/2k$ adders are used to implement this addition according to a binary tree. Like in Step 2, each partial product is shifted with respect to the position of its corresponding k -bit substring of X . The area complexity of Step 3 results from the number of adders required. Again, each of them has a width of $2n$ bits.

$$A_3 = O(n^2/2k) = O(n^2/\log^2 n) \quad (4)$$

The time complexity is determined by the number of additions that one adder has to perform. It is equal to the depth of the binary tree.

$$T_3 = O(\log n/k) T_{\text{add}} \subseteq O(\log n) T_{\text{add}} \quad (5)$$

The result is $P=X*Y$.

The time complexity of the complete algorithm is obviously

$$T_1 + T_2 + T_3 = O(\log n) T_{\text{add}} \quad (6)$$

The area complexity is

$$A_1 + A_2 + A_3 = O(n^2 / \log^2 n + n^{3/2}) = O(n^2 / \log^2 n) \quad (7)$$

Consequently, the AT-complexity is

$$AT = O(\log n * n^2 / \log^2 n) = O(n^2 / \log n) T_{\text{add}} \quad (8)$$

3 Carry Save Adders and Redundant Number Representation

The central operation of most algorithms for integer multiplication including ours is addition. There are many different methods for an addition: ripple carry addition, carry select addition, carry look ahead addition and others [7]. The disadvantage of all of these methods is the carry propagation which makes the latency of the addition depend on the length of the operands. This is not a big problem for operands of size 32 or 64 but if the operand size is in the range of 1024. The resulting delay has a significant influence on the time complexity.

Carry save addition [6] is a method for an addition without carry propagation. It is simply a parallel collection of n full-adders without any interconnection. Its function is to add three n -bit integers X , Y , and Z to produce two integers C and S as results such that

$$C + S = X + Y + Z \quad (9)$$

The i^{th} bit of the sum s_i and the $(i+1)^{\text{st}}$ bit of carry c_{i+1} is calculated using the boolean equations

$$s_i = x_i \oplus y_i \oplus z_i, \quad c_{i+1} = x_i y_i \vee x_i z_i \vee y_i z_i, \quad (10)$$

in other words, a carry save adder cell is just a full-adder cell. When carry save adders are used in an algorithm one uses a notation of the form

$$(S, C) = X + Y + Z \quad (11)$$

to indicate that two results are produced by the addition.

The results are now represented in two binary n -bit words. Of course, this representation is redundant in the sense that we can represent one value in several different ways. This redundant representation has the advantage that the arithmetic operations are fast, because there is no carry propagation.

An n -bit carry save adder consists of n full adders (a circuit for a 1-bit wide addition). We define the area of one full adder as 1. Therefore, the area of an n -bit carry save adder is n .

We define the time complexity as the number of steps necessary to do the job. The time complexity of one full adder is T_{FA} . Thus, the time complexity of carry save adder is T_{FA} too, because all the full adders operate in parallel.

We define a new kind of redundant adder as an adder with 4 inputs and 2 outputs. This is a cascade of two carry save adders.

$$(S, C) = S_1 + C_1 + S_2 + C_2 = (S_1, C_1) + (S_2, C_2) \quad (12)$$

We will use a notation

$$S'' = S'_1 + S'_2 \quad (13)$$

whereby S' , S'_1 , and S'_2 are redundant numbers with

$$S' = (S, C), S'_1 = (S_1, C_1), \text{ and } S'_2 = (S_2, C_2) \quad (14)$$

The area complexity of a redundant n -bit adder of this kind is $2n$. The time complexity is $2T_{FA}$.

4 Optimised Version of the New Algorithm Using Redundant Representation of Numbers

The AT-complexity of our algorithm is

$$O(n^2/\log n) \quad (15)$$

The standard addition requires a time complexity of

$$O(\log n) \quad (16)$$

Nor there is straightforward way to speed up our algorithm by using the carry save addition with constant time complexity. Thus, we use redundant adders instead of standard adders. We calculate the product

$$P' = X * Y, \quad (17)$$

where P' is a redundant number with

$$P' = (S, C) \quad (18)$$

The time complexity and the area complexity is equal to the time and area complexity of the basic algorithm in section 2. The result consists now of two numbers S and C . Finally, we add S and C with a standard adder with area complexity of

$$O(n) \quad (19)$$

and time complexity of

$$O(\log n) \quad (20)$$

The time complexity of the new algorithm is

$$O(\log n + \log n) = O(\log n) \quad (21)$$

The area complexity of the new algorithm is

$$O(n^2/\log^2 n + n) = O(n^2/\log^2 n) \quad (22)$$

Consequently, the AT-complexity is

$$O(n^2/\log^2 n * \log n) = O(n^2/\log n) \quad (23)$$

The redundant representation doubles the amount of register space for all intermediate results, because every nonredundant n -bit number is now represented by two n -bit numbers. But this increase of a factor of two in area is not visible here due to the O -notation.

5 Conclusion

In this paper we have given a simple parallel algorithm for multiplication of large integers with optimal time ($O(\log n)$) and low area ($O(n^2/\log^2 n)$) complexity. It is well suited for an implementation in real world applications. We give an example for the precise complexity: For multiplication of two 1024-bit numbers it requires 104 nonredundant registers, 64 carry save adders and 1 standard adder. The time complexity of the new algorithm is $74T_{FA}$, where T_{FA} is the delay of a full adder.

References

- [1] Ciminiera, L., Montuschi, P.: *Carry-Save Multiplication Schemes Without Final Addition*, IEEE Transactions on Computers 45 (1996), 1050–1055.
- [2] Cooley, J. W., Tukey, J. W.: *An Algorithm for the machine calculation of complex Fouries series*, Mathematics of Computation 19 (1965), 297–301, MR 31 #2843.
- [3] Gentleman, W. M., Sande, G.: *Fast Fourier transforms—for fun and profit*, AFIPS 1966 Fall Joint Computer Conference, Spartan Books, Washington, 1966, 563–578
- [4] Karatsuba, A. A., Ofman, Y.: *Multiplication of multidigit numbers on automata*, Sovjet Physics Doklady 7 (1963), 595–596.
- [5] Knuth, D. E.: *The art of computer programming*, volume 2: seminumerical algorithm, 2nd edition, Addison-Wesley, Reading Massachusetts, 1981.
- [6] Koc, C. K.: *RSA Hardware Implementation*, RSA Laboratories, RSA Data Security, Inc. August 1995, <http://security.ece.orst.edu/koc/papers/reports.html>
- [7] Lee, S. C.: *Digital Circuits and Logic Design*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1976.
- [8] Schönhage, A., Strassen, V.: *Schnelle Multiplikation großer Zahlen*, Computing 7 (1971), 281–292. MR #1431.
- [9] Singer, B., Saon, G.: *An efficient algorithm for parallel integer multiplication*, Journal of Network and Computer Applications 19 (1996), 415–418.
- [10] Stelling, P. F., Martel, C. U., Oklobdzija, V. G., and Ravi, R.: *Optimal Circuits for Parallel Multipliers*, IEEE Transactions on Computers 47 (1998), 273–285.
- [11] Toom, A. L.: *The complexity of a scheme of functional elements realizing the multiplication of integers*, Sovjet Physics Doklady 3 (1963), 714–716.
- [12] Wallace, C. S.: *A Suggestion for a Fast Multipliers*, IEEE Transactions Electronic Computing, 13 (1964), 14–17.
- [13] Wegener, I.: *Efficiente Algorithmen für grundlegende Funktionen*, Chapter 3, 75–125, B. G. Teubner, 1989.
- [14] Winograd S.: *Arithmetic complexity of computations*, CBMS-NSF Regional conference Series in Applied Mathematics, Philadelphia, 1980. ISBN 0-89871-163-0. MR 81k:68039.
- [15] Zuras, D.: *More on squaring and multiplying large integers*, IEEE Transactions on Computers 43 (1994), 899–908.

Topic 14

Routing and Communication in Interconnection Networks

Jose Duato, Olav Lysne, Timothy Pinkston, and Hermann Hellwagner
Topic Chairs

Communication networks, protocols, and application programming interfaces (APIs) are crucial factors for the performance of parallel and distributed computations. This topic of Euro-Par 2003 is therefore devoted to all aspects of communication in on-chip interconnects, parallel computers, networks of workstations, and more widely distributed systems such as grids. Papers were solicited that examine the design and implementation of interconnection networks and communication protocols, advances in system area and storage area networks, routing and communication algorithms, and the communication costs of parallel and distributed algorithms. On-chip and power-efficient interconnects, I/O architectures and storage area networks, switch architectures as well as multimedia and QoS-aware communication were new topics introduced in this year's Call for Papers (CfP).

The CfP attracted 24 submissions to this topic, of which eight papers (33%) were selected for publication and presentation at the conference. Six of them were accepted as full papers, two as short ones. The selected papers cover a wide scope, ranging from convenient and efficient communication library designs to congestion control algorithms, QoS routing, packet forwarding in the InfiniBand Architecture, and deflection routing in unbuffered all-optical networks.

We would like to thank all the authors for their submissions to this Euro-Par topic. We owe special thanks to the 25 external referees who provided competent and timely review reports. Their effort has ensured the high quality of this part of Euro-Par 2003. We trust you will find this topic to be highly stimulating and quite informative.

Dynamic Streams for Efficient Communications between Migrating Processes in a Cluster

Pascal Gallard and Christine Morin

IRISA/INRIA – PARIS project-team

Pascal.Gallard@irisa.fr

<http://www.kerrighed.org>

Abstract. This paper presents a communication system designed to allow efficient process migration in a cluster. The proposed system is generic enough to allow the migration of any kind of stream: socket, pipe, char devices. Communicating processes using IP or Unix sockets are transparently migrated with our mechanisms and they can still efficiently communicate after migration. The designed communication system is implemented as part of Kerrighed, a single system image operating system for a cluster based on Linux. Preliminary performance results are presented.

1 Introduction

Clusters are now more and more widely used as an alternative to parallel computers as their low price and the performance of micro-processors make them really attractive for the execution of scientific applications or as data servers.

A parallel application is executed on a cluster as a set of processes which are spread among the cluster nodes. In such applications, processes may communicate and exchange data with each other. In a traditional Unix operating system, communication tools can be streams like **pipe** or **socket** for example. For load-balancing purpose, a process may be migrated from one node to another node. If this process communicates, special tools must be used in order to allow high performance communication after the migration.

This paper presents a new communication layer for efficient migration of communicating processes. The design of this communication layer assumes that processes migrate inside the cluster and do not communicate with processes running outside the cluster. In the Kerrighed operating system [5], depending on the load-balancing policy, processes may migrate at any time.

The remainder of this paper is organized as follows. Sect. 3 describes the dynamic stream service providing the dynamic stream abstraction and Kerrighed sockets. Sect. 4 shows how the dynamic stream service can be used to implement distributed Unix sockets. Sect. 5 presents performance results obtained with Kerrighed prototype. Conclusions and future works are presented in Sect. 6.

2 Background

The problem of migrating a communicating process is difficult and this explains why several systems, such as Condor [4], provide process migration only for non communicating processes.

The MOSIX[1] system uses *deputy* mechanisms in order to allow the migration of a communicating process. When a process migrates (from a *home*-node to a *host*-node), a link is created between this process and the *deputy*. Every communication from/to this process is transmitted to the *deputy* that acts as the process. In this way, migrated processes are not able to communicate directly with other processes and thus communication performance decreases after a migration. The Sprite Network operating system[3] uses similar mechanisms in order to forward kernel calls whose results are machine-dependent.

Several works like MPVM[2] or Cocheck[9] allow the migration of processes communicating by message passing. However, these middle-wares are not transparent for applications. Mobile-TCP[7] provides a migration mechanism in the TCP protocol layer using a *virtual port* linked to the real TCP socket. Mobility is one of the main features of IPv6[6] but communications can migrate only if the IP address migrates. In this case, one process must be attached to one IP address and each host must have several IP addresses (one for each running communicating process). Even in this case, only one kind of communication tools (inet sockets) can migrate. Another case of communication migration is detailed in M-TCP [11] where a running client, outside of the cluster, can communicate with a server through an access point. If processes on servers migrate, access points can hide the communication changes.

None of these proposals offer a generic and efficient mechanism for migrating streams in a cluster allowing a migrating process to use all standard communication tools of a Unix system.

We want to avoid message forwarding between cluster nodes when processes migrate. We propose a generic approach in order to provide standard local communication tools like **pipe**, **socket** and *char*-devices compliant with process migration (decided for load-balancing reasons or due to configuration changes in the cluster – addition/eviction of nodes). This approach has been implemented as part of Kerrighed project at the operating system level and in this way provides full migration transparency to communicating applications.

3 Dynamic Streams

Our work aims at providing standard communication interfaces such as Unix sockets or pipes to migrating processes in a cluster. Migrating a process should not alter the performance of its communications with other processes.

A communication comprises two distinct aspects: the binary stream between two nodes, and the set of meta-data describing the state of the stream and how to handle it. Our architecture is based on this idea.

We propose the concept of dynamic stream on which standard communication interfaces are built. We call the endpoints of these streams as "KerNet

Sockets” and these can be migrated inside the cluster. Dynamic streams and KerNet sockets are implemented on top of a portable high performance communication system providing a send/receive interface to transfer data between different nodes in a cluster.

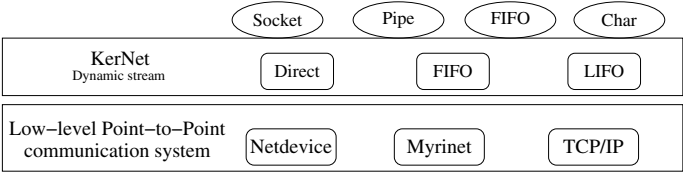


Fig. 1. Kerrighed network stack

The proposed architecture is depicted in Figure 1. Low-level Point-to-Point (PtP) communication service can be based on device drivers (such as myrinet), the generic network device in Linux kernel (netdevice) or a high-level communication protocol (such as TCP/IP). It is reliable and provides messages orders. On top of the low level point-to-point layer, we provide 3 kinds of dynamic streams: direct, FIFO and LIFO streams.

We use these dynamic streams, implemented by the *KerNet layer* to offer dynamic version of standard Unix stream interfaces (sockets, pipe...). It is a distributed service which provides global stream management cluster wide. In the remainder of this paper, we focus on the design and implementation of the KerNet layer and the Unix socket interface.

3.1 Dynamic Stream Service

We define a KerNet dynamic stream as an abstract stream with two or more defined KerNet sockets and with no node specified. When needed, a KerNet socket is temporarily attached to a node. For example, if two KerNet sockets are attached, send/receive operations can occur.

A KerNet dynamic stream is mainly defined by several parameters:

- **Type of stream:** it specifies how data is transfered using the dynamic stream. A stream can be:
 - DIRECT for one to one communication,
 - FIFO or LIFO for stream with several readers and writers.
- **Number of sockets:** number of existing sockets in the stream
- **Number of connected sockets:** it specifies the current number of attached sockets.
- **Data filter:** it allows modification of all data transmitted with the stream (in order to have cryptography, backup...).

Streams are managed by a set of stream managers, one executing on each cluster node. Kernel data structures related to dynamic streams are kept in a global directory which is distributed on cluster nodes.

3.2 KerNet Sockets

The KerNet service provides a simple interface to allow upper software layers implementing standard communication interface to manage KerNet sockets:

- **create/destroy** a stream,
- **attach**: to get an available KerNet socket (if possible),
- **suspend**: to unattach temporarily a socket (and to give an handle in order to be able to reclaim the KerNet socket later),
- **wakeup**: to attach a previously unattach KerNet socket,
- **unattach**: to release an attached KerNet socket,
- **wait**: to wait for the stream to be ready to be used (all required attachments completed).

KerNet provides two other functions (**send**, **recv**) for I/O operations.

The dynamic stream service is in charge of allocating KerNet sockets when it is needed, and of keeping track of these KerNet sockets. When the state of one KerNet socket changes, the stream's manager takes part in this change and updates the other KerNet sockets related to the stream. With this mechanism, each KerNet socket has got the address of each corresponding socket's node. In this way, two sockets can always communicate in the most efficient way.

At the end of a connection, a process is unattached from the stream. Depending on the stream type, the stream may be closed.

3.3 Example of Utilization of the KerNet API in the OS

Let us consider two kernel processes ($P1$ and $P2$) communicating with each other using a dynamic stream. They execute the following program:

<u>Process P1</u>	<u>Process P2</u>
(1) stream = create (DIRECT , 2);	
(2) socket1 = attach (stream);	socket2 = attach (stream);
(3) wait (stream);	wait (stream);
(4) ch = recv (socket1);	send (socket2 , ch);
(5)	sec2 = suspend (socket2);
	<u>Process P3</u>
(6)	socket3 = wakeup (stream , sec2);
(7) send (socket1 , ch);	ch = recv (socket3);

Initialization of a KerNet stream: **P1** creates the stream and requests a KerNet socket. Next **P1** wait for its stream to be ready, that is to say the two sockets to be attached. Assuming **P2** is running after the stream creation and has the stream identifier, it can get a KerNet socket, and then, wait for the correct state of the stream. The stream's manager sends the acknowledgement to all waiting KerNet sockets and provides the physical address of the other socket. With such information, KerNet sockets can communicate directly and *send/receive* communication can occur efficiently.

Migrating process using KerNet streams: If a process wants to migrate (or transfers its KerNet sockets to another process), it just uses the **suspend** function. When a process migration is needed, the KerNet socket is suspended on

the departure node and re-attached on the arrival node. In our example, **P2** suspends the socket, which is latter re-attached when **P3** executes. The dynamic stream service is in charge of ensuring that no message (or message piece) is corrupted or lost between the suspend and re-attached time. The stream manager updates the other KerNet socket so that it stops its communication until it receives new information from the stream manager. When the suspended socket is activated again, its new location is sent to the other KerNet socket and direct communication between the two KerNet sockets is restarted.

4 Implementation of Standard Communication Interface Using Dynamic Streams

Obviously, standard distributed applications do not use KerNet sockets. In order to create a standard environment based on dynamic streams and KerNet sockets, an interface layer is implemented at kernel level (see Figure 2). Each module of the interface layer implements a standard communication interface relying the interface of the KerNet service. The main goal of each interface module is to manage the standard communication interface protocol (if needed).

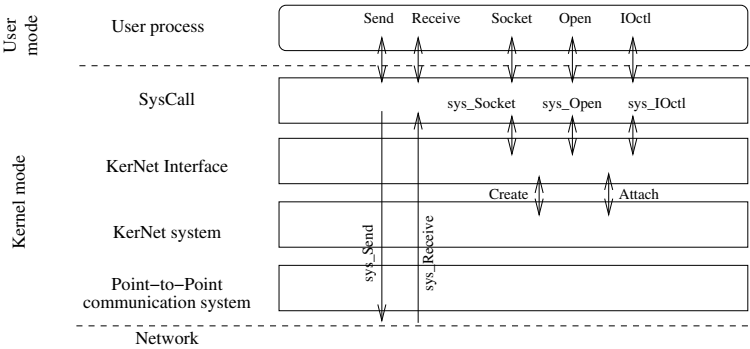


Fig. 2. Standard environment based on KerNet sockets

KerNet interfaces are the links between the standard Linux operating system and the Kerrighed dynamic communication service.

The Kerrighed operating system is based on top of a lightly-modified Linux kernel. All the different services, including the communication layer, are implemented as Linux kernel modules.

In Kerrighed operating system, the communication layer is made of two parts. A static high-performance communication system that provide a node to node service. On top of this system, the dynamic stream service manages the migration of streams's interfaces. Finally, the interface service replaces the standard functions for a given communication tool.

In the remainder of this section, we describe the Unix socket[10] interface on the KerNet sockets. We aim at providing a distributed Unix socket, transparent to the application.

In the standard Linux kernel, **Unix sockets** are as simple as unqueuing some packets from the sending socket and queuing them in the receiving socket. In this case the (physical) shared memory allows the operating system to access to the system structures of the two sockets. In the same way, the protocol management can be done easily. Obviously, in our architecture, the operating system of one node may not have access to the data structure of the other socket. Based on the KerNet services, the KerNet Unix sockets interface must manage the standard Unix sockets communication protocol.

When a new interface is defined, a corresponding class of stream is registered in the dynamic stream service. A class of stream is a set of streams that share the same properties. This registering step defines general properties of streams associated to this interface (stream type, number of sockets. . .) and returns it a stream class descriptor.

When a process makes an **accept** on a Unix socket, the Unix socket interface creates a new stream, attaches the first KerNet socket and waits for the stream allocate the other KerNet socket (as in **P1**). When another process (may be on another node) executes a **connect** on the Unix socket, an attach attempt is made (process **P2**). On success, the stream is completed and the two KerNet sockets (and by this way the two Unix sockets) can communicate directly.

The **accept/connect** example is a good representation of how we implement Unix sockets. With the same approach we have designed and implemented other standard socket functions like **poll** (in order to be able to use **select** syscall), **listen**, and so on. **Send** and **receive** functions are directly mapped on the **send** and **receive** KerNet socket functions.

Based on this interface, we may provide other standard interfaces such as **pipe**, **inet socket** and even some access to **char** device.

When a migration occurs (decided by the global scheduler or by the program itself), the migration service calls the **suspend** function and attaches the socket on the new node (such as **P3**).

5 Performance Evaluation

In the current implementation, KerNet provides standard Inet and Unix sockets interfaces. In order to have some performance evaluation of our communication system, we used the NetPipe[8] application. This benchmark is a ping-pong application with several packet's size. We use the *vanilla*-TCP version and a KerNet one in order to use **Unix socket**. Several physical networks are used inside the cluster: the loopback interface (Fig. 3(a)), FastEthernet (Fig. 3(b)), and Gigabits Ethernet (Fig. 3(c)).

In the FastEthernet and Gigabits Ethernet networks each node is a *Pentium III* (500MHz, 512KB cache) with 512MB of memory. The Kerrighed system used is an enhanced 2.2.13 Linux kernel.

In addition, we provide in all cases, the performance of our point to point communication layer (without any dynamic functionality). We must notice that, these measures represent PtP communications from KerNet to kernel with buffers physically allocated in memory. In this case, buffers are in contiguous memory area and their size are lower than 128KB. Thus, the PtP low-level performance is a maxima for our KerNet stream.

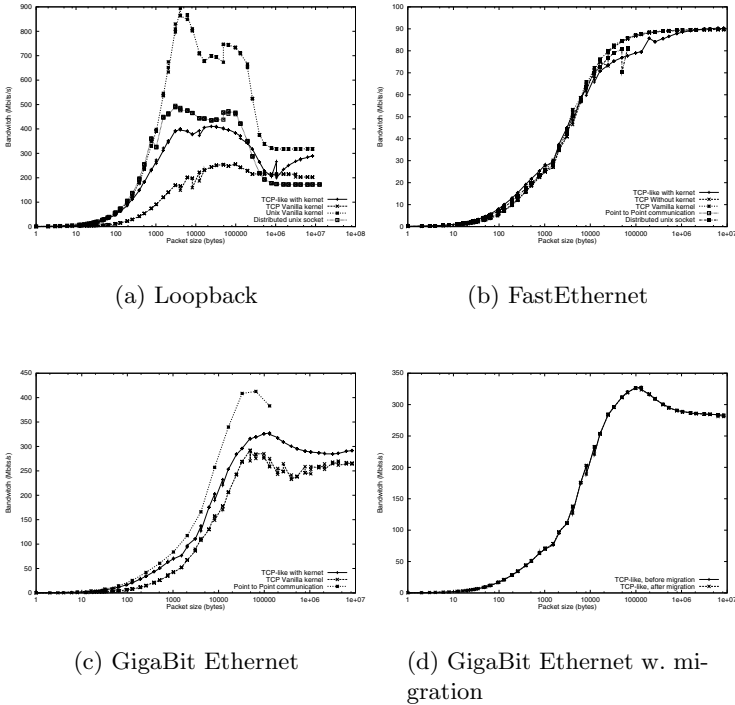


Fig. 3. Throughput of several communication systems

First, we notice that the interfaces have a low impact on the dynamic stream: Unix socket and TCP sockets have nearly the same results. With a *FastEthernet* network, the KerNet dynamic stream bandwidths are nearly the same than PtP low-level one. On *GigaBit ethernet* network, transfers between user-space and kernel-space are more perceptible.

When two communicating processes are on the same node, dynamic streams outperform the standard TCP sockets. This is mainly due to the small network stack in KerNet: IP stack provides some network services which are useless in a cluster or already performed by our low-level communication layer. However we do not reach (on a single node) the performance of a Unix socket. This is mainly due to the design of the low-level communication layer which as been designed for inter-node communications without any optimization for local communications.

When two communicating processes are not on the same node, KerNet outperformed TCP socket again. The reason are the same as above.

Other experiments have been performed to evaluate the impact of a migration on the dynamic stream performance. The NetPIPE application for TCP (NPtcp) has been modified to trigger a KerNet socket migration. Figure 3(d) shows that there is no overhead after a migration of a communicating process.

6 Conclusion

In this paper we have described the design and implementation of a distributed service allowing efficient execution of communicating processes after migration. We have introduced the concept of dynamic stream and mobile sockets. We have shown on the example of Unix sockets how standard communication interfaces can take advantage of these concepts.

The proposed scheme has been implemented in the Kerrighed operating system. We currently study communications in the context of the migration of standard MPI processes without any modification of the application and of the MPI library.

In future works on dynamic streams we plan to provide other stream standard communication interfaces like **pipes** and access to **char** devices. We also plan to study fault-tolerance issues in the framework of the design and implementation of checkpoint/restart mechanisms for parallel applications in Kerrighed cluster operating system.

References

1. A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.
2. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. Technical Report CSE-95-002, 1, 1995.
3. F. Douglass and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software-Practice & Experience*, 21(8), August 1991.
4. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.
5. C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an efficient single system image cluster operating system. In *ICA3PP*, 2002.
6. C. E. Perkins and D. B. Johnson. Mobility support in IPv6. In *Mobile Computing and Networking*, pages 27–37, 1996.
7. Xun Qu, J. Xu Yu, and R. P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Canberra 0200 ACT, Australia, 1997.
8. Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performance evaluator, 1996.
9. G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *International Parallel Processing Symposium*, 1996.
10. R. Stevens. *Unix Network Programming*, volume 1-2. Prentice Hall, 1990.
11. F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *Proceedings of The 22nd International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

FOBS: A Lightweight Communication Protocol for Grid Computing

Phillip M. Dickens

Department of Computer Science, Illinois Institute of Technology, 235-C Stuart Building,
Chicago, Illinois 60616
Dickens@iit.edu

Abstract. In this paper, we discuss our work on developing an efficient, lightweight application-level communication protocol for the high-bandwidth, high-delay network environments typical of computational grids. The goal of this research is to provide congestion-control algorithms that allow the protocol to obtain a large percentage of the underlying bandwidth when it is available, and to be responsive (and eventually proactive) to developing contention for system resources. Towards this end, we develop and evaluate two application-level congestion-control algorithms, one of which incorporates historical knowledge and one that only uses current information. We compare the performance of these two algorithms with respect to each other and with respect to TCP.

1 Introduction

The national computational landscape is undergoing radical changes as a result of the introduction of cutting-edge networking technology and the availability of powerful, low-cost computational engines. This combination of technologies has led to an explosion of advanced high performance distributed applications that, because of the limited bandwidth and best-effort nature of the Internet1 environment, were heretofore infeasible. Concurrently, research efforts have focused on the development of *Grid computing*, a fundamentally new set of technologies that create large-scale distributed computing systems by interconnecting geographically distributed computational resources via very high-performance networks. The advanced applications being developed to execute in Grid environments include distributed collaboration, remote visualization of terabyte (and larger) data sets, large-scale scientific simulations, Internet telephony, and advanced multimedia applications.

Arguably, Grid computing will reach its vast potential if, *and only if*, the underlying networking infrastructure (both hardware and software) is able to transfer vast quantities of data across (perhaps) quite long distances in a very efficient manner. Experience has shown, however, that advanced distributed applications executing in existing large-scale computational Grids are often able to obtain only a very small fraction of the available underlying bandwidth (see, for example, [21, 6, 11]). The reason for such poor performance is that the Transmission Control Protocol (TCP [3]), the communication mechanism of choice for most distributed applications, was not designed and is not well suited for a high-bandwidth, high-delay network

environment [21, 11, 17]. This issue has led to research aimed at improving the performance of the TCP protocol itself in this network environment [21, 1, 5, 10], as well as the development of application-level techniques that can circumvent the performance problems inherent within TCP [13, 12, 16, 19, 20].

Despite all of the research activity undertaken in this area, significant problems remain in terms of obtaining available bandwidth while being in some sense fair to competing flows. While TCP is able to detect and respond to network congestion, its very aggressive congestion-control mechanisms result in poor bandwidth utilization even when the network is lightly loaded. User-level protocols such as GridFTP[2], RUDP[16], and previous versions of FOBS [7-9], are able to obtain a very large percentage of the available bandwidth. However, these approaches rely on the characteristics of the network to provide congestion control (that is, they generally assume there is no contention in the network). Another approach (taken by SABUL [20]) is to provide an application-level congestion-control mechanism that is closely aligned with that of TCP (i.e. using the same control-feedback interval of a single round trip time).

We are attempting to approach the problem from a somewhat different perspective. Rather than attempting to control the behavior of the protocol on a time scale measured in milliseconds, we are interested in developing approaches that can operate on a much larger time scale while still providing very effective congestion control. We believe the best way to achieve this goal is to use the historical information that is available to the application to help drive the congestion-control algorithms. The work presented in this paper is aimed at showing that our approach is both feasible and useful.

The rest of the paper is organized as follows. In Section 2, we provide a brief overview of the components of the FOBS data transfer system. In Section 3, we discuss the design of two user-level congestion control algorithms. In Section 4, we discuss the design of the experiments developed to evaluate the performance of our control mechanisms and to compare their performance with that of TCP. In Section 5, we discuss the results of these experiments. We discuss related work in Section 6, and provide our conclusions and future research directions in Section 7.

2 FOBS

FOBS is a simple, user-level communication mechanism designed for large-scale data transfers in the high-bandwidth, high-delay network environment typical of computational Grids. It uses UDP as the data transport protocol, and provides reliability through an application-level acknowledgment and retransmission mechanism. Experimental results have shown that FOBS performs extremely well in a computational Grid environment, consistently obtaining on the order of 90% of the available bandwidth across both short- and long-haul network connections [7-9]. Thus FOBS addresses quite well the issue of obtaining a large percentage of the available bandwidth in a Grid environment. However, FOBS is in-and-of-itself a very aggressive transport mechanism that does not adapt to changes in the state of the end-to-end system. Thus to make FOBS useful in a general Grid environment we have developed congestion control mechanisms that are responsive to changes in system

conditions while maintaining the ability to fully leverage the underlying high-bandwidth network environment.

2.1 Rate-Controlled FOBS

Rate-Controlled FOBS enhances the basic FOBS mechanism by placing simple control agents at the communications endpoints. The agents collect and exchange information related to the state of the transfer and cooperatively implements the congestion control algorithms. The data transfer engine itself is (at least conceptually) reasonably simple. While it is beyond the scope of this paper to discuss in detail the implementation of FOBS and the technical issues addressed (the interested reader is directed to [8, 9, 18] for such a discussion), it is worthwhile to briefly discuss the implementation of the reliability mechanism.

FOBS employs a simple acknowledgment and retransmission mechanism. The file to be transferred is divided into data units we call *chunks*, and data is read from the disk, transferred to the receiver, and written to disk in units of a chunk. We have set the chunk size to 100 MBs, this number being chosen based on extensive experimentation. Each chunk is subdivided into *segments*, and the segments are further divided into *packets*. Packets are 1,470 bytes (within the MTU of most transmission mediums), and a segment consists of 10,000 packets. The receiver maintains a bitmap for each segment in the current chunk depicting the received/not-received status of each packet in the segment. A 10,000-packet segment requires a bitmap of 1,250 bytes, which will also fit in a data packet within the MTU of most transmission media. These bitmaps are sent from the data receiver to the data sender at intervals dictated by the protocol, and trigger (at a time determined by the congestion/control flow algorithm) a retransmission of the lost packets. The data to be transferred uses UDP sockets and the bitmaps are sent on TCP sockets.

There are two advantages of using this approach. First, such segmentation of the data de-couples the size of an acknowledgment packet from the size of the chunk. For example, a chunk size of 100 MBs would require on the order of 8,700 bytes for the acknowledgment packet (assuming one bit per data packet). While the size of the acknowledgment packet could be decreased with the use of negative acknowledgments, it is not difficult to imagine that it may still require bitmaps of a size (perhaps significantly) larger than the MTU of a given transmission medium. This makes it more difficult to deliver the feedback in a timely fashion (it would have to be fragmented along the way), which would have a very detrimental impact on performance.

Perhaps more importantly, these bitmaps provide a precise and detailed picture of the packet-loss patterns caused by the current state of the end-to-end system. We term these bitmaps *packet-loss signatures*, and preliminary research suggests that as the events that drive packet loss change the packet-loss signatures themselves also change. Currently, the packet-loss signatures are fed to a visualization system as the transfer is taking place, and we are attempting to develop an understanding of these signatures. The goal is to incorporate information gleaned from these bitmaps into the congestion-control mechanisms.

3 Congestion Control Algorithms

We are interested in evaluating approaches to congestion control that operate under a different set of assumptions than current techniques. One issue in which we are interested is whether the feedback-control interval must be pegged to roundtrip times in order to provide effective congestion control and ensure fairness. This approach is problematic for two reasons: First, a small spike in packet loss may well represent an event that has dissipated by the time the sender is even aware of its occurrence. Thus the congestion control mechanism may be reacting to past events that will not re-occur in the immediate future. Secondly, such an approach can become unstable. That is, the algorithm can get into cycles of increasing the sending-rate due to low packet loss, followed by a spike in packet-loss due to increasing the rate, followed by backing off in reaction to increased loss, and so forth. For these reasons, we believe it is important to explore alternative approaches.

We have developed and tested two alternative approaches to congestion control, one where the feedback-control interval is significantly lengthened and the increase/decrease parameters for the send-rate are linear. This approach incorporates historical knowledge into the design of the algorithm. The other approach is *state-based*, where the behavior of the algorithm depends only upon the current state and the current feedback.

It is important to note the way in which historical knowledge is incorporated into the first protocol. In particular, the algorithm was developed based on empirical observations made during the development and testing of the FOBS system. Thus the historical patterns were deduced by direct observation, and deducing such information from statistical measures would of course be significantly more complex. However, it does point to the fact that historical information can be important. One observation was that once the protocol found a sending rate that resulted in negligible packet-loss, it could remain at that rate for a reasonably long period of time (which in general was longer than it took to successfully transfer one complete chunk of data). Another observation was that once such a rate was established, the best approach was to stay at that rate. That is, there appeared to be an “optimal” sending rate such that being more aggressive tended to result in non-proportional increases in packet loss, and becoming less aggressive did not result in a meaningful decrease in packet loss. Finally, given a current “optimal” sending rate S , it was observed that the next such sending rate was quite often close to S . Such characteristics were not observed on all connections all the time. However, it was a pattern that emerged frequently enough to be noticed.

3.1 Historical Knowledge

These observations were incorporated into the protocol in the following ways. First, the feedback-control interval was extended to the time required to successfully transfer one complete chunk of data. The mean loss rate for an entire chunk was calculated once the data had been successfully transferred, and if this rate exceeded a threshold value the send-rate was decreased. The threshold value was $\frac{1}{2}$ of 1%, and the decrease parameter was a constant 5 Mbs on a 100 Mbs link and 50 Mbs on Gigabit connection. This slow decrease in the send rate is clearly not appropriate when the current conditions are causing massive packet loss *and* a reduction in sending rate (or several reductions) does not have a significant impact on the loss rate.

We are investigating techniques to detect such conditions and respond by either raising the decrease parameter or perhaps by switching to another protocol.

The increase parameter for this approach was also linear, but the algorithm does *not necessarily increase* the sending rate each time the loss-rate falls (or remains) below the current threshold value. Rather, the protocol keeps track of the current rate, the number of times the sending rate has been incremented from the current rate, and the number of times such an increase has resulted in a significant increase in packet loss. Based on this sample, the probability that increasing the sending rate will result in increased packet loss is computed, and the protocol then uses this probability to determine if the sending rate will be increased or remain unchanged.

It is important to note that this protocol incorporates two different types of historical knowledge. One type of knowledge was the empirical observations that suggested the basic framework of the algorithm. The other source of knowledge was obtained by tracking the historical relationship between an increase in the sending rate (from a given rate A to $A + \text{increase\_amount}$) and an increase in the loss-rate. Thus the protocol captured information obtained both within a given session and between multiple sessions.

3.2 State Based Approach

We also investigated what may be termed a *state-based* approach, where decisions regarding the send rate are based solely on the *current state* and *current feedback* information. We have developed such an algorithm and have implemented three states thus far: Green, Yellow, and Red. These states represent excellent, moderate, and poor system conditions respectively. Each state has its own minimum and maximum sending rate, its own increase and decrease parameters, and its own set of conditions under which it will change state. The feedback-control interval is the amount of time required to successfully transfer *one segment* of data to increase the responsiveness of the protocol to current conditions. The parameters are set such that the sending rate is increased at a high rate when there is little packet loss, and decreased very quickly in the event of massive packet loss.

4 Experimental Design

Our experiments were conducted on links between Argonne National Laboratory and the Center for Advanced Computing Resource (CACR), and links between Argonne National Laboratory and the National Center for Supercomputing Applications (NCSA). The round trip time between ANL and CACR (as measured by traceroute) was approximately 64 milliseconds, which we loosely categorize as a *long-haul* connection. The round trip time between ANL and NCSA was on the order of eight milliseconds, which we (again loosely) categorize as a *short-haul* connection. All links were connected through the Abilene backbone network. The slowest link between ANL and NCSA was an OC-12 connection from NCSA to Abilene; both ANL and CACR had to Gigabit Ethernet connections to Abilene.

Two hosts were tested within NCSA: a 64 processor SGI Origin 2000 running IRIX 6.5 and a 128 node Itanium cluster running Linux 2.4.16. Each node in the

Itanium cluster consisted of an Intel 800MHz Itanium CPU (dual processor), and each processor of the SGI was a 195MHz MIPs R10000. The host at CACR was a four-processor HP N4000 running HP-UX 11. The tests at Argonne National Laboratory were conducted on a dual processor Intel i686 running Linux RedHat 6.1. All experiments were conducted using a single processor.

We simulated the transfer of a 10-Gigabyte file from Argonne National Laboratory to the machines at both NCSA and CACR. That is, we performed the number of iterations necessary to complete a transfer of that size but did not actually perform the read from and write to disk (largely because of disk quotas on some hosts). We were unable to send in the opposite direction because of firewalls at ANL. The tests were conducted during normal business hours (for all hosts involved) to ensure there would be some contention for network and/or CPU resources. We were interested in the mean throughput and the overall loss percentage¹.

We were also interested in comparing the performance of Rate-Controlled FOBS with that of TCP, and conducted experiments between these same links to make such a comparison. The results obtained for TCP were extremely poor due to the small buffer size on the host at Argonne National Laboratory (64 KB), which could only be changed with root permissions (that we did not have). We present below these results, as well as results obtained in previous experiments where the Large Window extensions for TCP were enabled.

5 Experimental Results

The results of these experiments are shown in Table 1. The major difference between the two user-level algorithms is the higher loss rate experienced by the state-based approach on the link between ANL and CACR. This larger loss rate was a reflection the instability of the state-based approach on that particular link in the presence of contention. The instability arose from the fact that the algorithm immediately increases the sending rate whenever the loss rate fell (or remained) below the threshold value. Thus it spent a significant amount of time oscillating between sending at the “optimal” rate (as discussed in Section 3), and a rate that was not significantly higher but that resulted in significantly increased packet loss nonetheless. This phenomenon was due to the fact that negligible packet loss was experienced on the short-haul connections, and the algorithm thus rarely had to decrease the send rate.

Another interesting result was that FOBS was not able to obtain a very large percentage of the OC-12 or the Gigabit Ethernet connections we tested. There were two primary factors contributing to this relatively low utilization of network bandwidth. First, the experiments were run during the day when contention for CPU and network resources was quite high. The throughput was much higher when the experiments were run at night when there was little competition for CPU or network cycles (e.g., on the order of 94% of the maximum bandwidth on the OC-12 connection between ANL and NCSA).

¹ The loss percentage was calculated as follows. Let N be the total number of packets that had to be sent (assuming no packet loss), and let T be the total number of packets actually sent. Then the loss rate $R = (T - N) / N$.

The second factor has to do with inefficiencies of the FOBS implementation. Currently, the **select()** statement is used to determine those sockets ready for either sending or receiving. This is a heavyweight system call that is used repeatedly in the current implementation. A better approach would be to use threads to determine when a socket becomes available, and we are currently implementing a multi-threaded version of FOBS.

Table 1. This table shows the throughput and packet-loss percentage for FOBS using historical information (FOBS HI), FOBS using only current state information (FOBS CS) and TCP

Link	Protocol	Throughput	Loss
ANL to Itanium Linux Cluster at NCSA	FOBS HI	190 Mbs	< 0.1%
	FOBS CS	192 Mbs	< 0.1%
	TCP	7.2 Mbs	----
ANL to SGI Origin 2000 at NCSA	FOBS HI	161 Mbs	< 0.1%
	FOBS CS	156 Mbs	< 0.1%
	TCP	6.5 Mbs	----
ANL to HP N4000 at CACR	FOBS HI	86 Mbs	< 0.1%
	FOBS CS	84 Mbs	2.4 %
	TCP	1.6 Mbs	----

It is also quite clear that the application-level approach obtained performance significantly higher than that obtained by TCP across all experiments with very little packet loss. As noted above, the primary reason for TCP's very poor performance has to do with the very small TCP buffers available on terra (which has a significant negative impact on TCP's performance). In previous experiments that we conducted [7-9], using the same links but different hosts (that supported the Large Window extensions for TCP [17]), the results were significantly better. In particular, on the link between ANL and NCSA TCP obtained on the order of 90 Mbs, and obtained on the order of 50 Mbs on the link between ANL and CACR. However, even with the Large Window extensions, TCP's performance was significantly less than that obtained by FOBS.

6 Related Work

The work most closely related to our own are RUDP [16] and SABUL [20], both of which have been developed at the Electronics Visualization Laboratory at the

University of Illinois Chicago. RUDP is designed for dedicated networks (or networks with guaranteed bandwidth), making the issue of congestion control largely irrelevant. However, since FOBS is designed to operate in a general Grid environment the issues of congestion control must be addressed. SABUL does provide congestion control that uses a feedback-control interval pegged to round trip times. We believe this interval is too small for large-scale data transfers across high-performance networks, and the research presented in this paper is an attempt to move away from such a model of congestion control.

Research related to application-level scheduling and adaptive applications is also related (e.g. the Apples project [4] and the Grads project [14]). However, this body of research is focused on interactions between an application scheduler and a heavyweight Grid scheduler provided in systems such as Globus [13] and Legion [15]. FOBS, on the other hand, is a very lightweight communication protocol that operates outside of the domain of such large systems. However, an interesting extension of our work would be to modify FOBS such that it could interact with Grid-level schedulers to negotiate access to system resources.

7 Conclusions and Future Research

In this paper, we described a lightweight application-level communication protocol for computational Grids. The protocol is UDP-based, with a simple acknowledgment and re-transmission mechanism. We discussed two different application-level congestion-control mechanisms, and showed that Rate-Controlled FOBS was able to significantly outperform TCP across all connections tested with very minimal packet loss (especially when historical knowledge was incorporated into the algorithm).

There are many ways in which this research can be extended. The results provide herein suggest that integrating historical knowledge into the congestion control mechanism can significantly enhance its performance. This of course brings up the issue of how to automate the collection of historical data, which data has the best chance of providing insight into future behavior, and how such data can be incorporated into the control mechanisms. These are all issues we are currently pursuing.

It is our goal to make FOBS a widely used transport protocol for data transfers within computational Grids. Towards this end, we have developed an easy-to-use drag-and-drop interface, a 32-bit CRC for increased reliability, and a sophisticated system for visualization of the packet-loss signatures. Currently, we are developing a multi-threaded version of the protocol to increase efficiency and a version of FOBS capable of performing cluster-to-cluster data transfers.

References

- [1] *List of sack implementations. Web Page of the Pittsburgh Supercomputing Center.* http://www.psc.edu/networking/all_sack.html.
- [2] Allcock, W., Bester, J., Breshahan, J., Chervenak, A., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S. Secure, Efficient Data Transport and Replica Management for High-Performance Data_Intensive Computing. In *Proceedings of IEEE Mass Storage Conference*, 2001.

- [3] Allman, M., Paxson, V., and W.Stevens. *TCP Congestion Control*, RFC 2581. <http://www.faqs.org/rfcs/rfc2581.html>
- [4] *The AppLeS Homepage* <http://apples.ucs.d.edu/>
- [5] *Automatic TCP Window Tuning and Applications. National Laboratory for Advanced Networking Research Web Page* http://dast.nlanr.net/Projects/Autobuf_v1.0/autotcp.html
- [6] Boyd, E.L., Brett, G., Hobby, R., Jun, J., Shih, C., Vedantham, R., and Zekauska, M. *E2E piPEline: End-to-End Performance Initiative Performance Environment System Architecture*. July, 2002. <http://e2epi.internet2.edu/e2epipeline11.shtml>
- [7] Dickens, P. A High Performance File Transfer Mechanism for Grid Computing. In *Proceedings of The 2002 Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Las Vegas, Nevada, 2002.
- [8] Dickens, P., and Gropp, B. An Evaluation of Object-Based Data Transfers Across High Performance High Delay Networks. In *Proceedings of the 11th Conference on High Performance Distributed Computing*, Edinburgh, Scotland, 2002.
- [9] Dickens, P., Gropp, B., and Woodward, P. High Performance Wide Area Data Transfers Over High Performance Networks. In *Proceedings of The 2002 International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems.*, 2002.
- [10] *Enabling High Performance Data Transfers on Hosts: (Notes for Users and System Administrators)*. Pittsburgh Supercomputing Center http://www.psc.edu/networking/perf_tune.html#intro
- [11] Feng, W., and Tinnakornsrisuphap, P. The Failure of TCP in High-Performance Computational Grids. In *Proceedings of Proceedings of Super Computing 2000 (SC2000)*.
- [12] *The FOBS Data Transfer System* http://www.csam.iit.edu/~pmd/FOBS/fobs_index.html
- [13] *The Globus Project* <http://www.globus.org>
- [14] *The Grid Applications Development Software Project Homepage* <http://juggler.ucs.d.edu/~grads/>
- [15] Grimshaw, A., and Wulf, W. Legion-a view from 50,000 feet. In *Proceedings of Proceedings of High-performance Distributed Computing Conference (HPDC)*, 1996.
- [16] He, E., Leigh, J., Yu, O., and DeFanti, T. Reliable Blast UDP : Predictable High Performance Bulk Data Transfer. In *Proceedings of Proceedings of the IEEE Cluster Computing*, Chicago, Illinois, September 2002.
- [17] Jacobson, V., Braden, R., and Borman., D. TCP Extensions for high performance. RFC 1323, May 1992.
- [18] Kannan, V., Dickens, P., and Gropp, W. A Performance Study of Application Level Acknowledgment and Retransmission Mechanisms. In *Proceedings of The International Conference on High Performance Distributed Computing and Applications*, Reno, Nevada, 2003.
- [19] Sivakumar, H., Bailey, S., and Grossman, R. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Proceedings of Super Computing 2000 (SC2000)*.
- [20] Sivakumar, H., Mazzucco, M., Zhang, Q., and Grossman, R. Simple Available Bandwidth Utilization Library for High Speed Wide Area Networks. *Submitted to Journal of SuperComputing*.
- [21] *The Web100 Homepage* http://www.internet2.edu/e2epi/web02/p_web100.shtml

Low-Fragmentation Mapping Strategies for Linear Forwarding Tables in InfiniBand™^{*}

Pedro López, José Flich, and Antonio Robles

Dept. of Computing Engineering (DISCA)
Universidad Politécnica de Valencia
P.O.B. 22012, 46071 - Valencia, SPAIN
{plopez,jflich,arobles}@gap.upv.es

Abstract. The InfiniBand Architecture (IBA) supports distributed routing by using forwarding tables stored in each switch, which only consider the destination local identifier (LID) of the packet for routing. Each LID is mapped to a different table entry. Additionally, the IBA specifications allow each destination port to be assigned up to 2^n consecutive virtual addresses by masking the n least significant bits of its LID. Each range of virtual addresses must be mapped to consecutive table entries when IBA linear forwarding tables are used. However, the fact that each port may require a different number of virtual addresses and the fact that this number may not be a power of two could lead to waste some table entries, causing a fragmentation of the forwarding tables as a consequence of an inefficient mapping strategy of LIDs. Fragmentation of the forwarding tables could become critical as far as it reduces the number of available table entries to map LIDs, limiting, in turn, the number of ports that can be placed in the network. In this paper, we propose two effective mapping strategies to tackle the fragmentation effect on IBA forwarding tables. The first strategy is able to remove the fragmentation effect when the number of virtual addresses is a power of two for all destinations, introducing a fragmentation percentage lower than 20% in all cases. On the other hand, the second strategy is able to almost completely eliminate the possible fragmentation effect.

1 Introduction

InfiniBand [2] has been recently proposed as a standard for communication between processing nodes and I/O devices as well as for interprocessor communication. The InfiniBand Architecture (IBA) is designed around a switch-based interconnect technology with high-speed point-to-point links. An IBA network is composed of several subnets interconnected by routers, each subnet consisting of one or more switches and end node devices (processing nodes and I/O devices). Each end node contains one or more channel adapters (referred to as HCA in

^{*} This work was supported by the Spanish MCYT under Grants TIC2000-1151-C07 and 1FD97-2129, by the J.J.CC. de Castilla-La Mancha under Grant PBC-02-008, and the Generalitat Valenciana under grant CTIDIB/2002/288.

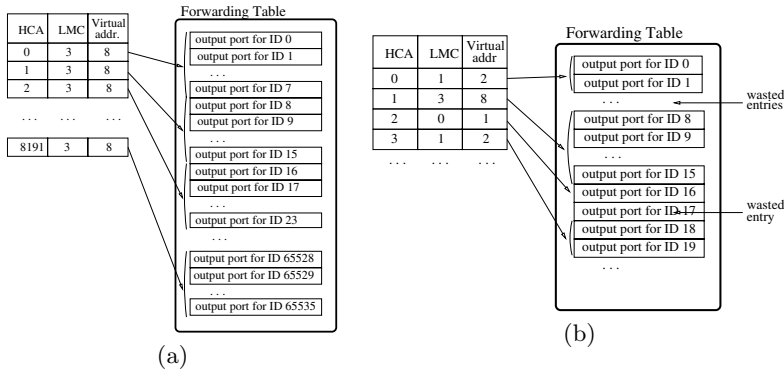


Fig. 1. Mapping of virtual addresses to a linear forwarding table. (a) LMC=3 for all HCAs and (b) different LMC values for each HCA.

the case of processor nodes and TCA in the case of I/O devices)¹. Each channel adapter contains one or more ports. Each port has a unique local identifier (LID) assigned by the subnet manager which is used by the subnet to forward packets towards their destination (the destination local identifier or DLID).

Routing in IBA subnets is distributed, based on forwarding tables stored in each switch, which only consider the packet DLID for routing [3]. IBA routing is deterministic since the routing tables only provide one output port per DLID. Forwarding tables in InfiniBand can be linear or random. In a linear table, the DLID of the incoming packet is used as an index into the table in order to obtain the output port. On the other hand, random tables are content-addressable memories. Thus, every entry in the table contains two fields: the DLID and the output port to be used. Clearly, the hardware cost of a linear table is lower than for a random table, as the access to the table is straightforward. In the random tables, additional logic must be used to compare the DLID of the packet within all the DLIDs stored in the table. The logic complexity increases with table size, leading to high access times and could prevent the use of large forwarding tables. Hence, from our point of view, linear tables will be the preferred choice for manufacturers. In this paper, we will focus on linear forwarding tables.

Additionally, IBA provides multiple virtual ports within a physical HCA port by defining a *LID Mask Control* or LMC [3]. The LMC specifies the number of least significant bits of the DLID that a physical port masks (ignores) when it validates that a packet DLID matches its assigned LID. Up to seven bits can be masked (3-bit field). As these bits are not ignored by the switches, from the subnet point of view, each HCA port has been assigned not a single address but a valid range of addresses (up to 2^{LMC} consecutive addresses). Moreover, each HCA port will accept all packets destined for any valid address within its range.

This IBA feature is originally intended to provide alternative routing paths within a subnet. The sender HCA port selects one of them by choosing the

¹ In what follows, we will use the term HCA port to refer to HCA ports and TCA ports indistinctly.

appropriate DLID when injecting a packet into the network. Additionally, this feature can also be used for other purposes. For instance, in [6] we proposed the Destination Renaming technique, which uses the virtual addressing feature to allow the implementation on IBA of any routing that take into account the input port for routing. The Destination Renaming technique uses a new virtual address for a particular destination HCA port whenever it encounters a routing conflict (two different routing options at the same switch leading to the same destination HCA port through different output ports). Moreover, in [4,5] virtual addressing is used to allow distributed adaptive routing in InfiniBand switches. By assigning several addresses to the same HCA port, several output ports are supplied by the forwarding tables.

The virtual addresses generated by the subnet manager for each HCA port must be mapped to the forwarding table of every switch. Figure 1.a shows the mapping of virtual addresses to a linear forwarding table when the same value of LMC is used by all HCA ports (LMC=3). In this case, the mapping is quite simple. As the LMC value is equal for all the HCA ports, all the ranges of virtual addresses for each HCA port are placed sequentially, starting at an entry whose index has its least LMC bits set to zero (entries 0, 8, 16, and so on).

However, all the HCA ports may not require the same number of virtual addresses. For instance, if HCA 0 in Figure 1.a only needed 2 virtual addresses, then DLIDs 2 through 7 would neither be used by HCA 0 nor by any other HCA port. To solve this problem, InfiniBand specifications state that each HCA port can be programmed with its own LMC value. Therefore, each HCA port adjusts its LMC value to the number of required virtual ports. Figure 1.b shows an example where each HCA port uses a different LMC value.

2 Motivation

In order to justify the motivation of this paper, in this Section we analyze what problems may arise when different LMC values are used by the HCA ports.

As the least LMC bits of the DLID are ignored by the HCA port, the range of virtual addresses for a particular HCA port must start at an address which has all its least LMC significant bits set to zero. Care must be taken in order to assign addresses to each port. Otherwise, gaps between two range of addresses for consecutive HCA ports may be created, leading to waste some entries of the forwarding table. As an example, Figure 1.b shows a straightforward mapping of the virtual addresses of different HCA ports with different LMC values to the forwarding table. The mapping is performed sequentially and searching in the forwarding table for the first suitable range of entries for each HCA port. As we can see, the first HCA port has a LMC value of 1, thus requiring only 2 virtual addresses. Therefore, the first two entries of the table are used to map the two virtual addresses of HCA 0 (DLID 0 and 1). However, as HCA 1 has a LMC value of 3, it needs a range of addresses that starts with the three least significant bits set to zero. The first available range of entries with such property is the range from DLIDs 8 through 15. Therefore, a gap has been created between the entry

for DLID 1 and the entry for DLID 8. Moreover, the entry for DLID 17 is also wasted, because HCA 2 only needs one table entry ($LMC=0$), whereas HCA 3 needs two entries ($LMC=1$). As a result, in our example, in order to map 13 virtual addresses for 4 HCA ports, 20 entries in the forwarding table have been used, 7 of them being wasted. In particular (50% more entries than the strictly needed are used). We say that the forwarding table is fragmented.

The fragmentation of the forwarding table may limit the number of HCAs we can connect in the subnet, and thus, it may limit the system scalability. If linear tables are implemented with full capacity for all the possible DLIDs, 49152 DLIDs can be used². In this situation, when 8 HCA ports per host are used and an average of 4 virtual addresses ($LMC=2$) per HCA port are needed, the system theoretically can support up to 1536 hosts. However, if a 35% of fragmentation was produced, the number of hosts that the system can support would drop to 998 hosts. In a more realistic scenario, IBA switches are shipped with smaller linear tables. In these situations, the fragmentation effect on forwarding tables could seriously affect system scalability.

The fragmentation will depend on the way mapping is performed. For instance, in Figure 1.b, if the mapping order of HCA ports is changed to HCA 1, HCA 0, HCA 3, and HCA 2, then no fragmentation is produced at all. In all cases, the least LMC significant bits of the starting DLID are set to zero. Therefore, by using a simple ordering of HCA ports, we can eliminate the fragmentation. However, another situation can occur that will cause fragmentation even when using the previous ordered mapping. It occurs when one or some HCA hosts require a number of virtual addresses that is not a power of 2. Therefore, the LMC value used for that particular HCA port will cover all the needed virtual addresses plus a few more virtual addresses that are not really needed by the HCA port.

Also, the fragmentation will depend on the required number of virtual addresses per port. This number will also depend on the applied routing algorithm, the network size and the network topology. Therefore, it is worth analyzing the impact on forwarding table fragmentation when different network configurations are used and different requirements of virtual addresses are imposed.

In this paper, we try to maximize the utilization of the forwarding tables by proposing two mapping strategies that will reduce the fragmentation effect in linear forwarding tables. The first one will be referred to as DLOM (Decreasing LMC-Ordered Mapping) and will eliminate the fragmentation effect in the case all the virtual addresses covered by the LMC values are used (i.e. if the number of virtual addresses required by each HCA port is a power of two). The second one will be referred to as FF (First Fit) and can eliminate all the possible fragmentation effect even if some virtual addresses covered by the LMC value are not really used by a particular HCA port (i.e. if the number of virtual addresses required by some HCA ports is not a power of two).

² Although 65536 DLIDs can be used, only 49152 DLIDs are available for unicast routing, bounding the number of HCA ports to 49152.

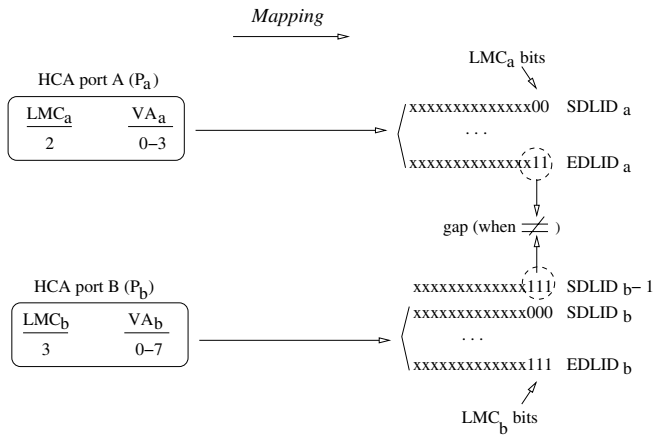


Fig. 2. DLOM strategy.

The rest of the paper is organized as follows. In Section 3, the proposed mapping strategies are presented. In Section 4 an analysis of the fragmentation caused by each mapping strategy is performed. Finally, in Section 5, some conclusions are drawn.

3 Mapping Strategies

The first mapping strategy will be referred to as DLOM strategy. It stands for Decreasing LMC-Ordered Mapping. As its name states, it is based on a previous ordering of HCA ports according to their LMC values. This strategy is able to fully eliminate fragmentation when all the virtual addresses covered by the LMC value are used by the HCA port.

The key idea of the strategy is shown in Figure 2. A gap in the forwarding table may be created between two consecutively mapped sets of virtual addresses (VA_a and VA_b) which correspond to two HCA ports (P_a and P_b , respectively) when their LMC values (LMC_a and LMC_b , respectively) are different. The mapping of VA_a should be performed with a starting DLID ($SDLID_a$) whose least LMC_a significant bits are set to zero. Also, VA_b should be mapped with a starting DLID ($SDLID_b$) whose least LMC_b significant bits are set to zero. Thus, the address immediately before $SDLID_b$ ($SDLID_{b-1}$) will have its least LMC_b significant bits set to ones. Also, the ending DLID of V_a ($EDLID_a$) will have its least LMC_a significant bits set to ones because all the virtual addresses covered by the LMC value are used.

No gap will be produced between both set of virtual addresses if the $EDLID_a$ has its least LMC_b bits set to ones. A sufficient condition to ensure this is that $LMC_a \geq LMC_b$. Effectively, if $LMC_a < LMC_b$ (as in Figure 2), then the $(LMC_b - LMC_a)$ bits after the LMC_a least significant bits may be zeros depending on the location where the VA_a addresses have been mapped.

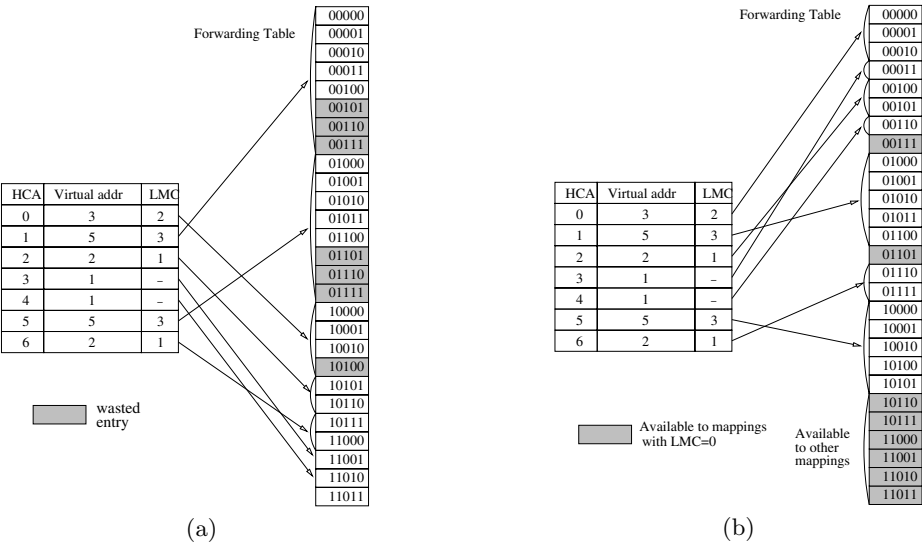


Fig. 3. (a) DLOM fragmentation. (b) FF technique.

Therefore, the DLOM strategy ensures that, for all the pairs of HCA ports mapped consecutively, P_a and P_b , it will be accomplished that $LMC_a \geq LMC_b$. This can be achieved by sorting the HCA ports in decreasing orders of their LMCs before mapping.

As previously commented, there can be cases where an HCA port does not need to use all the addresses that can be covered by its LMC value. Remember that a given LMC comprises 2^{LMC} addresses. In these cases, the DLOM mapping strategy does not guarantee a full defragmented table, as shown in Figure 3.a. In order to avoid wasting addresses, a second mapping strategy is proposed, which is referred to as First Fit (FF) strategy. Actually, this strategy will try to reuse the addresses that are not needed by a particular HCA port which have been reserved, though, due to its LMC value. Unlike the DLOM strategy, the FF strategy does not need to previously sort the HCA ports by their LMC values.

The FF strategy will reserve a valid range of DLID addresses to a particular HCA port from the first range of DLIDs not previously assigned. In other words, for each HCA port, it will search for the first suitable fit. Figure 3.b shows an example of the FF strategy. The first HCA port (HCA 0) needs only three virtual addresses. The FF strategy reserves DLIDs 0 through 2 for HCA 0. The next HCA port (HCA 1) needs 5 virtual addresses (LMC=3). The next range of addresses that starts with its three least significant bits set to zero is DLID 8 through 12. Therefore, a free range has been created from DLID 3 through DLID 7. The next HCA port (HCA 2) needs only 2 virtual addresses. The FF strategy chooses a valid range in the non-reserved zone. In particular, DLID 4 through DLID 5. The next two HCA ports (HCA 3 and HCA 4) need only one virtual address each one. The FF strategy uses two non-assigned DLIDs in the non-reserved zone. In particular, DLIDs 3 and 6 are chosen for HCA 3

Table 1. Renumbering HCA_{ports} after mapping.

Input data		DLOM			Renumbering
HCA port	LMC	Mapping order	Starting DLID	Ending DLID	New HCA port
0	2	2	8	11	8
1	3	1	0	7	0
2	1	4	16	17	16
3	2	3	12	15	12

and HCA 4, respectively. Notice that after mapping the seven HCA ports, only two virtual addresses will be left unassigned. As more HCA ports were mapped, those requiring only one virtual address would be mapped to these locations, thus leading to a fragmented-free mapping strategy.

A possible problem of the FF strategy is that it creates some DLIDs that are recognized as valid by several HCA ports. For instance, in Figure 3.b, DLID address 14 is a valid one for both HCA 1 and HCA 6 ports. Remember that a given HCA accepts packets destined to any address within its 2^{LMC} assigned ones. Hence, HCA 1 will accept any packet destined to DLIDs 8 to 15. This should not be a problem because DLID address 14 is really assigned to HCA 6, and, thus, forwarding tables have been programmed to forward packets whose DLID is 14 to HCA 6. But, if due to some network routing errors, a packet with DLID 14 arrives to HCA 1, it will be accepted, because it is in the range defined by its LMC. Unfortunately, the FF strategy could not be applied according to the current IBA specifications (1.0.a) because the subnet manager does not allow overlapping ranges of DLIDs.

Once the mapping is performed, each HCA port has an assigned range of DLIDs for its use. However, in order to each HCA port to use them correctly, the HCA port identifier must be set to the starting DLID. This is because the LMC bits of the packet DLID will be masked at destination to match the destination HCA port. Therefore, once the mapping is computed, the HCA port identifiers must be renumbered. Notice that the renumbering of HCA ports must be performed regardless of the applied mapping function. In Table 1 we can see an example of network with four HCA ports that have been mapped to the forwarding tables using the DLOM strategy. Firstly, the HCA ports are sorted by their LMC values. After sorting the HCA ports, the mapping is performed by assigning starting and ending DLIDs. As can be observed, for the original HCA port 1, its first address will be zero. Therefore, the HCA port must be renumbered to zero. The rest of HCA ports are renumbered in the same way according to the starting assigned address.

4 Evaluation of the Proposed Mapping Strategies

In this section, we evaluate the DLOM and FF strategies presented in this paper. For comparison purposes, we will evaluate the straightforward mapping strat-

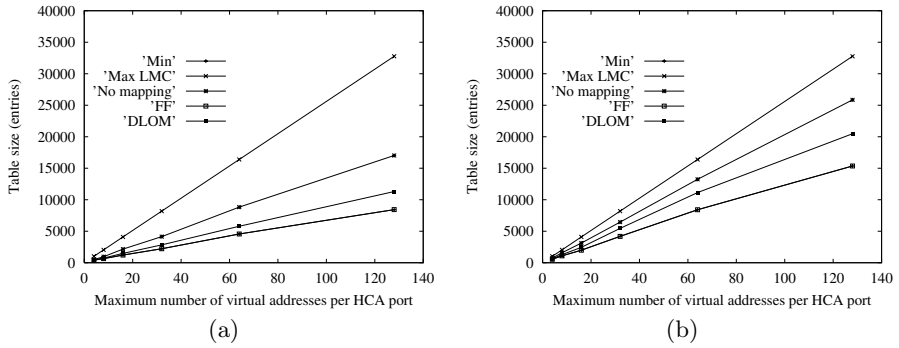


Fig. 4. Number of forwarding table entries required for different number of virtual addresses required. 256 hosts. In (a) 50% of hosts require more than one virtual address and in (b) all the hosts require more than one virtual address.

egy consisting of setting the LMC values of all the HCA ports to the highest LMC value (LMC_{max}) used for a particular HCA port (the easiest mapping strategy). With this mapping strategy, a high fragmentation is introduced as a large number of addresses is wasted. For a given HCA port with a required LMC, up to $2^{LMC_{max}-LMC}$ addresses are wasted. This strategy will be referred to as *MaxLMC*. We will also compare the strategies with the basic sequential (no ordering) mapping consisting of selecting the next valid range of free addresses. This strategy will be referred to as *No Mapping*. In addition, results for an optimal strategy that assigns the minimum number of addresses to HCA ports without any fragmentation is evaluated. The number of required addresses is simply the sum of the virtual addresses required by all the HCA ports. It will be referred to as *Min*.

In a first study, we will analyze the impact of the applied mapping strategy used on the fragmentation of the forwarding tables under different scenarios. These scenarios will be defined by the number of HCA ports in the system, the maximum possible number of virtual addresses required by any HCA port and the percentage of HCAs requiring more than one virtual address. In particular, we will analyze networks with 256 HCA ports. Regarding the percentage of HCA ports requiring more than one virtual address, we will show results for 50% and 100%. Finally, for each HCA port that requires more than one virtual address, we assign a number randomly selected between 2 and a maximum number. We will vary the maximum number of virtual addresses required by a particular HCA port from 2 up to 128 (the upper bound allowed by IBA). If virtual addresses are used to supply alternative paths to destinations, with the last two parameters we model the number of such paths. Notice that a large number of virtual addresses per HCA port and a high percentage of the number of HCA ports requiring more than one virtual address models a system with a high connectivity. In the analysis, we will measure fragmentation as the percentage of wasted entries relative to the minimum number of entries required in the table. In other words,

fragmentation keeps track of the additional useless entries in the forwarding tables.

Figure 4 shows the number of entries in the forwarding tables when different mapping strategies are used. In particular, Figure 4.a shows the number of entries when the percentage of HCA ports requiring more than one virtual address is 50%, whereas Figure 4.b show the results when all hosts use more than one virtual address. As can be seen, when no mapping strategy is used (*No mapping*) more than 60% of fragmentation is caused (assuming high connectivity). The easiest mapping strategy (*Max LMC*) exacerbates the fragmentation as it uses the maximum LMC used in the network for all HCA ports. On average, this strategy increases fragmentation up to 300% (≈ 33000 vs. ≈ 8000 in Figure 4.a).

As we can see, the fragmentation is reduced when the DLOM strategy is used. In particular, the fragmentation percentage introduced by DLOM decreases down to 30% in the worst case (Figure 4.a). Remember that the only fragmentation incurred by the DLOM strategy is due to the virtual addresses not used by HCAs but covered by the LMC values (i.e., when the required number of virtual addresses is not a power of two). This can be verified by the less fragmentation achieved by the FF strategy. As it can be observed, the FF strategy introduces no fragmentation in any case as it uses the minimum number of required entries.

In a second study, we will obtain results for topologies with different number of HCA ports. In particular, we will analyze irregular networks of 8, 12, 16, 24, and 32 switches randomly generated, taking into account some restrictions. First, we will assume that every switch in the network has the same degree. Four ports at each switch will be used to connect to HCA ports. The remaining ports will be used to connect to other switches. Switches with 7, 8, and 10 ports will be used. Second, each switch will be connected to other switch by exactly one link. Five different topologies will be generated for each network size and average results will be shown. The applied routing algorithm is the smart-routing [1] and only one path for each pair of source-destination ports will be used. The destination renaming technique proposed in [6] will be used to correctly compute the forwarding tables, which leads to the need of some virtual addresses per each HCA port. Notice that if several paths per each source-destination pair of hosts were allowed, more virtual addresses would be required.

Table 2 shows the average number of required entries and the average percentage of incurred fragmentation when using smart-routing. Also, results for different network connectivities are shown (different number of links connecting switches). As it can be observed, the FF mapping strategy is the only one that introduces no fragmentation in all the evaluated topologies. Also, it can be observed that the DLOM strategy does not achieve a very high reduction in the fragmentation when compared with *No mapping*. This is because most of the fragmentation is caused by virtual addresses not used by HCA ports but covered by their LMC value. Anyway, DLOM incurs in lower fragmentation than *No Mapping* in all cases.

Table 2. Average fragmentation of forwarding tables when using smart-routing under different topologies and network connectivity.

Sw	Links	Min	Max	LMC	FF		No mapping		DL0M	
		entries	entries	%	entries	%	entries	%	entries	%
8	3	36.00	44.80	24.44	36.00	0.00	36.00	0.00	36.00	0.00
8	4	68.20	128.00	87.68	68.20	0.00	78.00	14.37	75.20	10.26
8	6	98.40	128.00	30.08	98.40	0.00	121.60	23.58	121.60	23.58
12	3	74.40	134.40	80.65	74.40	0.00	78.00	4.84	77.40	4.03
12	4	85.00	153.60	80.71	85.00	0.00	86.40	1.65	86.20	1.41
12	6	108.00	192.00	77.78	108.00	0.00	124.40	15.19	121.00	12.04
16	3	92.20	204.80	122.13	92.20	0.00	96.00	4.12	95.40	3.47
16	4	104.60	204.80	95.79	104.60	0.00	110.60	5.74	109.80	4.97
16	6	139.00	256.00	84.17	139.00	0.00	157.60	13.38	153.40	10.36
24	3	194.60	384.00	97.33	194.60	0.00	214.00	9.97	209.40	7.61
24	4	244.20	537.60	120.15	244.20	0.00	293.20	20.07	286.20	17.20
24	6	255.80	537.60	110.16	255.80	0.00	316.40	23.69	303.00	18.45
32	3	282.00	512.00	80.92	283.00	0.00	329.60	16.47	319.60	12.93
32	4	323.40	512.00	58.32	323.40	0.00	389.60	20.47	380.20	17.56
32	6	456.47	682.67	48.90	458.47	0.00	563.87	22.99	550.07	19.98

5 Conclusions

IBA specifications allow each HCA port to be assigned a number of virtual addresses or LIDs, which must be mapped to a range of consecutive entries in the forwarding tables. However, the use of an inappropriate mapping strategy could lead to waste some table entries, causing fragmentation in the forwarding tables. This fact becomes critical as far as it reduces the number of HCA ports that can be placed in the network and/or avoids the implementation of forwarding tables on switches with a given table size.

In this paper, we have proposed two effective mapping strategies to eliminate the fragmentation effect in linear forwarding tables. The DL0M strategy is based on a previous ordering of the HCA ports according to their number of required virtual addresses. However, this strategy only eliminates the fragmentation effect when the number of virtual addresses required by each HCA port is a power of two. To overcome this problem, we have proposed the FF strategy, which is able to eliminate the fragmentation effect in all cases. This strategy will reserve a valid range of LID addresses to a particular HCA port from the first range of LIDs not previously assigned, trying to reuse the addresses not needed by a particular HCA port which, otherwise, would be reserved according to the LMC.

The evaluation of the proposed mapping strategies shows that their impact on the reduction in the fragmentation of the forwarding tables is more significant as far as the network size increases and a high number of virtual addresses are required by each HCA port. In particular, the fragmentation percentage introduced by DL0M decreases down to 20% whereas no fragmentation is introduced by applying the FF strategy. Moreover, when using smart routing under different topologies and network connectivities, the average fragmentation of the

forwarding tables introduced by using DLOM is always lower than that introduced when no mapping strategy is applied. Again, in all cases, no fragmentation is introduced by applying the FF strategy.

References

1. L. Cherkasova, V. Kotov, and T. Rokicki, "Fibre channel fabrics: Evaluation and design," in *Proc. of 29th International Conference on System Sciences*, Feb. 1995.
2. InfiniBandTM Trade Association. <http://www.infinibandta.com>.
3. InfiniBandTM Trade Association, *InfiniBandTM Architecture. Specification Volume 1. Release 1.0.a*. Available at <http://www.infinibandta.com>.
4. J.C. Martinez, J. Flich, A. Robles, P. López, and J. Duato, "Supporting Adaptive Routing in InfiniBand Networks," in *11th Euromicro Workshop in Parallel Distributed and Network-Based Processing*, February, 2003.
5. J.C. Martinez, J. Flich, A. Robles, P. López, and J. Duato, "Supporting Fully Adaptive Routing in InfiniBand Networks," in *Proc. of the International Parallel and Distributed Processing Symposium*, April 2003.
6. P. López, J. Flich, and J. Duato, Deadlock-free Routing in InfiniBandTM through Destination Renaming, in *Proc. of 2001 International Conference on Parallel Processing (ICPP'01)*, Sept. 2001.

A Robust Mechanism for Congestion Control: INC^{*}

Elvira Baydal and Pedro López

Dept. of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera s/n, 46071 – Valencia, SPAIN
elvira, plopez@disca.upv.es

Abstract. Several techniques to prevent congestion in multiprocessor interconnection networks have been recently proposed. Unfortunately, they either suffer from a lack of robustness or detect congestion relying on global information that wastes a lot of transmission resources.

This paper presents a new mechanism that uses only local information to avoid network saturation in wormhole networks. It is robust and works properly in different conditions. It first applies preventive measures of different intensity depending on the estimated traffic level; and if necessary, it uses message throttling during predefined time intervals that are extended if congestion is repeatedly detected. Evaluation results for different network loads and topologies show that the proposed mechanism avoids network performance degradation. Most important, without introducing any penalty for low and medium network loads.

1 Introduction

Massively parallel computers provide the performance that scientific and commercial applications require. Their interconnection networks offer the low latencies and high bandwidth that is needed for different kinds of traffic. Usually, wormhole switching with virtual channels and adaptive routing is used. However, multiprocessor interconnection networks may suffer from severe saturation problems with high traffic loads, which may prevent reaching the wished performance (see Figure 1). With low and medium network loads, the accepted traffic rate is the same as the injection rate. But if traffic increases and reaches (or surpasses) certain level (the saturation point), accepted traffic falls and message latency increases considerably. Performance degradation appears because with high network traffic several packets compete for the same resources (physical or virtual channels), but as only one packet can use them, the rest of the packets stay stopped in the network, thus blocking other packets and so on. When this situation is generalized, the network is congested and performance degradation appears.

One of the most popular approaches to solve this problem is to avoid that network traffic reaches the saturation point. Message throttling [4] (stop or slow down the injection rate of messages) has been the most frequently used method. Several mechanisms for multiprocessors have already been proposed but they have important drawbacks [2].

^{*} This work was supported by by the Spanish MCYT under Grants TIC2000-1151-C07 and 1FD97-2129

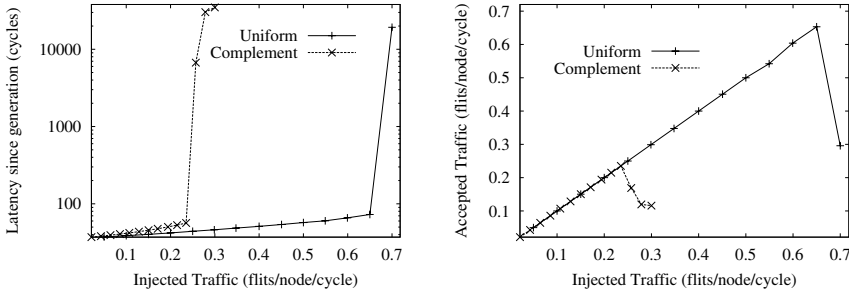


Fig. 1. Performance degradation in a 3-cube 8-ary (512 nodes) with a deadlock recovery-based fully adaptive routing algorithm with 3 virtual channels per physical channel. Uniform and complement traffic patterns. 16-flit messages.

In this paper, we propose and evaluate a new mechanism to avoid network saturation that tries to overcome these drawbacks. It is a robust mechanism that applies preventive measures of different intensity depending on the network traffic. Moreover, it progressively reduces the bandwidth available to inject messages, either by reducing the number of injection channels or, if it is necessary, by applying increasing intervals of forbidden message injection. The mechanism has been exhaustively tested, achieving good results in all the evaluated conditions and improving recent proposals [2], [15].

The rest of the paper is organized as follows. In order to design a good congestion control mechanism, Section 2 presents the main desirable features of such a mechanism. Section 3 describes the new proposal. Section 4 presents some simulation results. Finally, some conclusions are obtained.

2 Motivation

First, the mechanism should work properly for different conditions (robustness): different message destination distributions, message sizes, network topologies and even routing algorithms. As we can see in Figure 1, the network saturation point depends on the traffic pattern. Hence, a suitable mechanism for one distribution, may not work properly with another one. The same problem may appear when we change network topology. However, many of the previously proposed mechanisms have been analyzed with only one network size [1], [15] and for the uniform distribution of message destinations [4], [8], [13]. Other mechanisms do not achieve good results all traffic patterns considered [14]. On the other hand, some mechanisms are based on the idea that all the virtual channels of a physical channel are used in the same way [1], [2], thus being well suited for deadlock recovery schemes but may not for deadlock avoidance ones.

Second, the mechanism should not penalize the network when it is not saturated. Notice that this is the most frequent situation [12]. Thus, when with low and medium network traffic, the mechanism should not restrict or delay message injection into the network. However, some of the previous proposals increase message latency before the saturation point [4], [14], also reducing the network throughput.

Finally, the new mechanism should not generate new problems in the network. Those proposals that need to send extra information across the network increase network traffic and may make worse the congestion situation [8]. Moreover, when congestion detection relies on global information, mechanisms do not scale well since the amount of network status information increases with the network size [15].

3 The New Mechanism: Injection and Network Congestion

As we stated in Section 1, when network traffic is near to the saturation point, many packets block. Indeed, as wormhole switching is used, these packets stay stopped in the network, thus blocking other packets. In addition, as physical channels are multiplexed into more virtual channels, message advance speed decreases.

The new mechanism uses message advance speed in order to detect network congestion. In particular, it measures the number of transmitted flits during a fixed time interval t_m . Each virtual channel has an associated counter that increases each time one flit is transmitted. At the end of each interval, the channel is considered congested if it is busy and the number of transmitted flits is lower than a threshold fc (*flit counter*). When the header of a message reaches a congested area, either it will block or advance at a low speed. As a consequence, the remaining flits of the message will also block or decrease their advance speed. By tracking this speed, we try to detect congestion in network areas remote to the current node, but using only local information.

Once congestion is detected, the applied measures are different depending on whether the node is currently injecting messages towards the congested area. In particular, each node manage two different flags: the *injection* and the *network* flags. The *injection* flag is set when some of the virtual channels that the node is using to inject messages into the network are congested. On the contrary, the *network* flag is set when congestion is detected in any virtual channel that is being used by messages injected by other nodes. The mechanism will be referred to as INC, *Injection and Network Congestion* detection.

When either the injection or network flags are set, message injection restrictions are applied. These measures are more restrictive when the injection flag is set than when only network flag does. The injection flag setting means that the node is actively contributing to network congestion. On the contrary, when only the network flag is set, although the current node is not directly generating the problem, as congestion propagates quickly through the network, some preventive measures should be applied. In both cases, if congestion is repeatedly detected in later intervals, the measures get more restrictive. Thus, the mechanism works in a progressive way.

In particular, when injection congestion is detected, the number of enabled injection channels is divided (integer division) by a factor r_1 . If this quotient reaches zero, injection is completely stopped but only during some interval f_{ii} (*forbidden injection interval*). This interval starts when the last message that the node is currently injecting is completely injected into the network. After f_{ii} cycles, one injection channel is enabled regardless of the detected network traffic, and the first pending message, if any, is injected. As network traffic is estimated periodically, injection restrictions get harder or, on the contrary, are reduced. The mechanism uses limited forbidden injection intervals for two reasons. First, it avoids starvation and, second, it allows that other nodes also detect congestion and

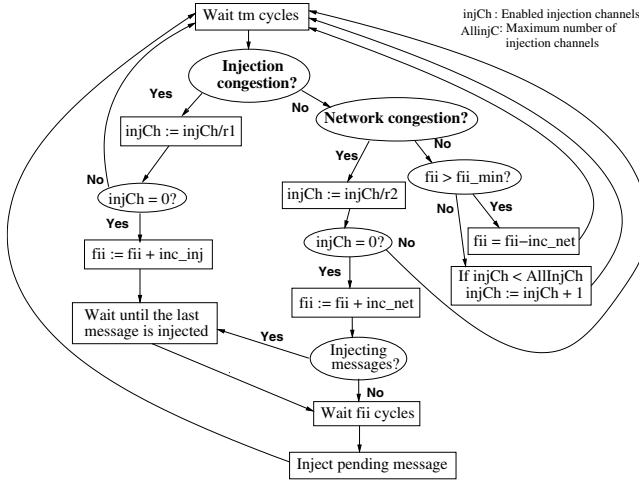


Fig. 2. Operation of the congestion control mechanism INC.

apply message injection limitation. If only network congestion is detected, the enabled injection channels are reduced by r_2 , with $r_2 < r_1$. Forbidden injection intervals are also used if network congestion is repeatedly detected, but they increase more slowly.

The values of r_1 and r_2 depend on the network radix k . Notice that when k increases also does the average distance traversed by the messages, which makes worse network congestion. Therefore injection restrictions have to be harder. After some tests, we have found that $r_1 = k/4$ and $r_2 = k/8$ work well, k being the network radix.

The forbidden injection interval is initialized to some value $f_{ii_{min}}$. Every time congestion is detected, it is incremented. This increment again depends on the network radix k . We have chosen it higher for injection than for network congestion. After several tests, we have found that values $inc_{inj} = (f_{ii_{min}} \times k)/16$ and $inc_{net} = (f_{ii_{min}} \times k)/32$ work well, for injection and network congestion, respectively.

Finally, injection bandwidth has to be recovered when congestion is no longer detected. However, injection restrictions are also reduced gradually. After an interval without detecting any congestion, injection limitation is smoothed. First, injection bandwidth is increased by reducing the forbidden interval (by inc_{net}), and later (when it reaches the minimum value $f_{ii_{min}}$), by increasing (by one channel at a time) the number of injection channels. Notice that the reduction of injection bandwidth is faster than the later recovering. Figure 2 summarizes the behavior of the mechanism.

4 Evaluation

In this section, we will evaluate by simulation the behavior of the INC mechanism. The evaluation methodology is based on the one proposed in [2]. The most important performance measures are latency (time required to deliver a message, including the time spent at the source queue, measured in clock cycles) and throughput (maximum traffic

accepted by the network, measured in flits per node per cycle). On the other hand, as message latency at saturation grows with time, another valuable measure of performance is the required time to deliver a given number of messages.

4.1 Network Model

Each node has a router, a crossbar switch and several physical channels. Routing time and both transmission time across the crossbar and across a channel are all assumed to be equal to one clock cycle. Each node has four injection/ejection channels. We explained the advantages of having several injection/ejection channels in [2].

Concerning deadlock handling, we have evaluated the new mechanism both with deadlock avoidance and recovery techniques. For deadlock recovery, we use software-based deadlock recovery [11] and a True Fully Adaptive routing algorithm (TFAR) [12,11]. This routing algorithm allows the use of any virtual channel of those physical channels that forwards a message closer to its destination. Deadlocks are detected with the mechanism proposed in [10] with a threshold equal to 32 cycles. With deadlock avoidance, we use the fully adaptive routing algorithm proposed in [6]. In both cases, we have tested the mechanism with 3 and 4 virtual channels per physical channel.

We have evaluated the performance of the proposed congestion control mechanism on different bidirectional k-ary n-cubes. In particular, we have used the following network sizes: 256 nodes ($n=2, k=16$), 1024 nodes ($n=2, k=32$), 512 nodes ($n=3, k=8$), and 4096 nodes ($n=3, k=16$).

4.2 Network Load

Each node generates messages independently, according to an exponential distribution. Destinations are chosen according to the *Uniform*, *Uniform with locality*, *Butterfly*, *Complement*, *Bit-reversal*, and *Perfect-shuffle* traffic patterns. Uniform distribution is the most frequently used one in the analysis of interconnection networks. The other patterns take into account the permutations that are usually performed in parallel numerical algorithms [9]. For message length, 16-flit, 64-flit and 128-flit messages are considered.

We have performed two kinds of experiments, using constant and dynamic loads. Experiments with constant loads are the usual evaluation tool for interconnection networks. In this case, the message generation rate is constant and the same for all network nodes. We analyze all the traffic range, from low load until saturation. Simulations finish after receiving 500,000 messages with the smallest networks (256 and 512 nodes) and 1,200,000 with the largest ones (1024 and 4096 nodes), but only the last 300,000 (1,000,000) messages are considered to calculate average latencies.

We will also evaluate the mechanism behavior when the traffic conditions change dynamically. In particular, we use a bursty load that alternates periods of a high message injection rate L_{sat} (that corresponds to a saturation interval), with periods of low traffic L_{low} . These loads are repeated twice ($L_{sat}L_{low}L_{sat}L_{low}$). Let L_s be the message generation rate just before the network reaches saturation. We have chosen a generation rate $L_{low} = 0.5 \times L_s$. On the other hand, in order to analyze the mechanism behavior at different saturation levels, we have tested two different values of the saturation load: $L_{sat} = 1.2 \times L_s$ and $L_{sat} = 1.8 \times L_s$. Concerning the number of sent messages, for

a given generation rate, each node generates the same number of messages. When a node has generated all the messages for one of the injection rates, then, it starts generating traffic with the next load level in sequence. Finally, when it has generated all the messages for the complete sequence, it does not generate new messages at all. The simulation finishes when all the injected messages arrive to their destinations.

We show results with 1,000 messages per node generated with the L_{sat} injection rate, and two different values for the load under the saturation (L_{low}): $M_{low} = 555$ and $M_{low} = 1111$ messages. The number of messages generated with the L_{low} rate establishes the elapsed time between two traffic bursts, and it allows us to analyze how the mechanism recovers the network after a congestion period.

4.3 Performance Comparison

In this section, we will analyze the behavior of the **INC** mechanism proposed in this paper. For comparison purposes, we will also evaluate the **U-channels** ([2]) and **Self-Tuned** ([15]) mechanisms. **U-channels** also uses only local information to avoid network saturation. Each node locally estimates traffic by using the percentage of free virtual output channels that can be used to forward a message towards its destination. When this number is below a threshold value, congestion network is assumed to exist and message throttling is applied. In the **Self-Tuned** mechanism, nodes detect network congestion by using global information about the number of full buffers in the network. If this number surpasses a threshold, all nodes apply message throttling. The use of global information requires to broadcast data among all the network nodes. A way of transmitting this control information is to use a sideband network [15]. As the mechanism proposed in this paper does not need to exchange control messages, to make a fair comparison, the bandwidth provided by the sideband network should be considered as additional available bandwidth in the main interconnection network. However, in the results that we present we do not consider this fact. If this additional bandwidth were considered, the differences, not only in throughput but also in latency, between **Self-Tuned** and the rest of the mechanisms would be greater than the ones shown.

As it was shown in [2], **U-channels** outperforms **Self-Tuned** with deadlock recovery-based TFAR routing, but it is not well suited for deadlock avoidance since, in this case, all the virtual channels are not used in the same way. For this reason, we compare **INC** with **U-channels** when TFAR routing is used, while we use **Self-Tuned** as the comparison with deadlock avoidance. Finally, in all cases, results without any congestion control mechanism (**No-Lim**) are also shown.

First of all, the **INC** mechanism has to be tuned. We have evaluated its behavior with different message destination distributions, message sizes and network topologies. In all cases, the number of transmitted flits per virtual channel is sampled every $t_m = 20$ cycles. As an example, Figure 3 shows the average message latency versus traffic for different threshold values for the uniform and perfect-shuffle traffic patterns with the TFAR routing algorithm based on deadlock recovery. Notice that both latency and accepted traffic are dependent variables on offered traffic. As you can see, threshold adjustment is not critical. In fact, several of them work fine since the mechanism tries to find the optimal injection bandwidth by applying progressive measures. Table 1 shows the values chosen for the evaluated networks, different number of virtual channels per physical channel,

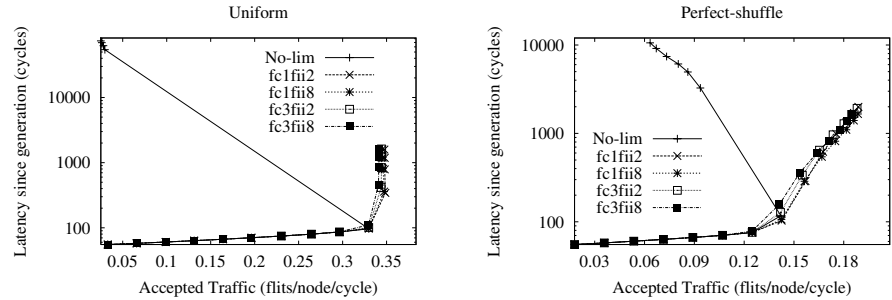


Fig. 3. Average message latency vs. accepted traffic for different threshold values for the *INC* mechanism. 16-ary 3-cube (4096 nodes). 3 virtual channels per physical channel. TFAR algorithm based on deadlock recovery.

Table 1. Threshold values for different topologies and number of virtual channels per physical channel.

Nodes	Virt. ch.	Deadlock recovery				Deadlock avoidance			
		fc	$f_{ii_{min}}$	inc_{inj}	inc_{net}	fc	$f_{ii_{min}}$	inc_{inj}	inc_{net}
256	3	1	4	4	2	1	4	4	2
	4	1	2	2	1	1	2	2	1
1024	3	4	2	4	1	6	8	16	8
	4	2	1	2	1	5	2	4	2
512	3	1	4	2	1	1	1	0	0
	4	1	1	0	0	1	1	0	0
4096	3	1	2	2	1	2	2	2	1
	4	1	2	2	1	1	2	2	1

and deadlock recovery and avoidance strategies. As you can see, the optimal values depend mainly on the topology radix k and the number of virtual channels per physical channel. This is not a problem, as these are design parameters that are fixed once the machine is built. Notice that for 512 nodes and 4 virtual channels, the f_{ii} is held constant ($inc_{inj} = inc_{net} = 0$). In this case, only reducing the number of injection channels is enough to avoid congestion.

Once tuned the new mechanism, we can compare it with the other ones. For the sake of shortness, we will only show some results for the 4096-node network ($k=16$, $n=3$) and 3 virtual channels per physical channel. The complete evaluation for all the tested networks, traffic patterns and different number of virtual channels per physical channel can be found in [3]. Figure 4 shows these results.

As we can see, the new mechanism **INC** avoids the performance degradation in all the evaluated cases, without increasing latencies for low and medium loads. In general, the results obtained with deadlock recovery are quite similar to the **U-channels** ones. On the other hand, with deadlock avoidance the **INC** mechanism improves in all the cases the **Self-tuned**. Although the **Self-Tuned** mechanism helps in alleviating network

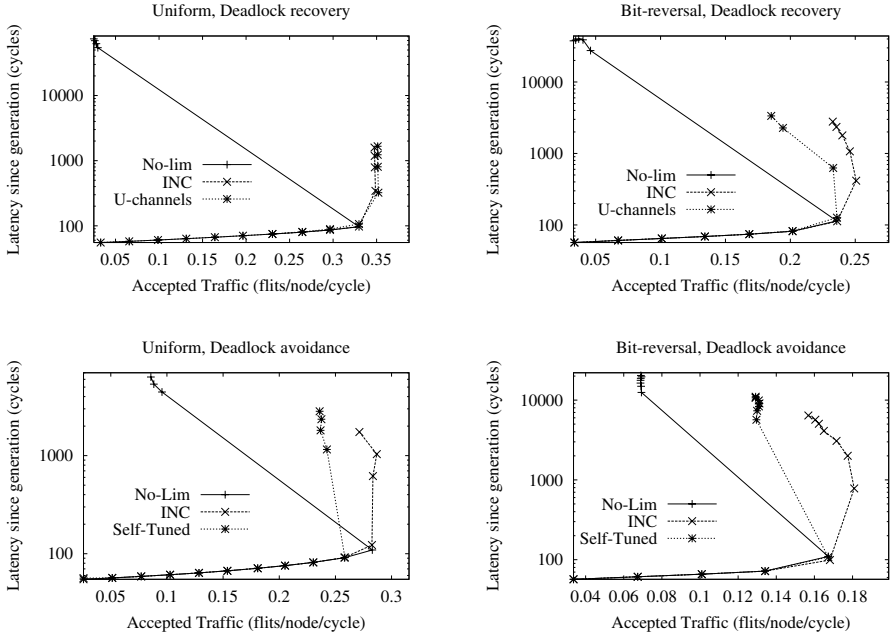


Fig. 4. Average message latency vs. accepted traffic. 16-flit messages. 8-ary 3-cube (4096 nodes).

congestion, it reduces network throughput and increases network latency with medium loads in some cases.

Figure 5 shows some results for dynamic loads with a uniform distribution of message destinations, $L_{sat} = 1.8 \times L_s$ and $M_{low} = 1111$ messages per node with L_{low} . As we can see, with the **INC** mechanism, the network accepts the injected bursty traffic without problems. On the contrary, when no congestion control mechanism is applied, as soon as the first burst is applied into the network, congestion appears. As a consequence, latency strongly increases and accepted traffic falls down. Later, after some time injecting with low rate, network traffic starts recovering but the arrival of a new traffic burst prevents it. Concerning **Self-Tuned** mechanism, we can see that it excessively limits injection rate, significantly reducing the highest value of accepted traffic, therefore increasing the time required to deliver all the injected messages. As we stated before, this time is another performance measure that is strongly affected by network saturation. Figure 5 shows that **INC** delivers the required number of messages in nearly half the time than **No-Lim** (74.36% of improvement), while **Self-Tuned** achieves a slightly smaller improvement (70.51%).

Table 2 shows the percentage of improvement in the time required to deliver all the injected messages, for different network sizes, when we apply the **INC** and **U-channels** congestion control mechanisms versus when no congestion control mechanism is used. Deadlock recovery is used. We have got improvements in the simulation time up to 156%

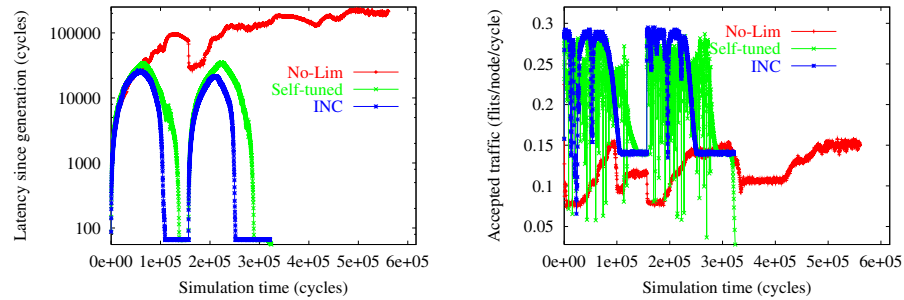


Fig. 5. Performance with dynamic loads. Uniform traffic pattern. 16-flit messages. 16-ary 3-cube (4096 nodes), 3 virtual channels. Fully adaptive algorithm with deadlock avoidance.

Table 2. Improvement (%) in the time required to deliver all the injected messages for the *INC* and *U-channels* mechanisms with different topologies, *TFAR* algorithm with deadlock recovery and 3 virtual channels. Uniform traffic pattern.

Nodes	Mechanism	$L_{sat} = 1.2 \times T_s$		$L_{sat} = 1.8 \times T_s$	
		$M_{low} = 1111$	$M_{low} = 555$	$M_{low} = 1111$	$M_{low} = 555$
256 (16 × 16)	INC	48.35	71.82	64.07	101.09
	U-channels	47.71	71.70	63.24	100.83
1024 (32 × 32)	INC	85.00	112.00	101.98	141.79
	U-channels	84.19	109.85	100.96	142.64
512 (8 × 8 × 8)	INC	22.85	44.23	34.69	69.33
	U-channels	22.29	42.47	33.94	66.97
4096 (16 × 16 × 16)	INC	96.29	125.59	111.28	156.19
	U-channels	94.86	122.92	109.59	152.64

with the uniform distribution of message destination and up to 86% with the complement traffic pattern¹. In general, the results are slightly better for the **INC** mechanism than for the **U-channels** mechanism. However, as the differences between both mechanisms do not exceed 5%, we can conclude that the progressive reduction/recovery provided by **INC** it is not an improvement but adds complexity.

Table 3 shows the same results when we use the **INC** and **Self-tuned** congestion control mechanisms versus **No-Lim**. Deadlock avoidance is used. In this case, the **INC** mechanism obtains improvements in the simulation time up to 150% versus **No-Lim** for a uniform distribution of message destinations, and up to a 22.5% for the complement traffic pattern¹. These improvements are far from the ones achieved by the **Self-Tuned** mechanism. Moreover, **INC** reaches this better performance without transmitting information through the network. Hence, for deadlock avoidance, **INC** is clearly a better option than **Self-tuned**.

¹ This result can be found in [3]

Table 3. Improvement (%) in the time required to deliver all the injected messages for the *INC* and *Self-Tuned* mechanisms with different topologies, fully adaptive routing algorithm with deadlock avoidance and 3 virtual channels. Uniform traffic pattern.

Nodes	Mechanism	$L_{sat} = 1.2 \times T_s$		$L_{sat} = 1.8 \times T_s$	
		$M_{low} = 1111$	$M_{low} = 555$	$M_{low} = 1111$	$M_{low} = 555$
256 (16 × 16)	INC	20.27	64.94	47.93	106.69
	Self-tuned	18.04	58.16	45.16	82.40
1024 (32 × 32)	INC	113.43	116.00	109.26	150.15
	Self-tuned	109.44	93.26	94.59	84.77
512 (8 × 8 × 8)	INC	0.00	2.33	0.00	24.49
	Self-tuned	-2.32	-0.22	-1.85	21.45
4096 (16 × 16 × 16)	INC	41.83	83.66	74.36	114.32
	Self-tuned	38.09	78.15	70.51	81.77

5 Conclusions

In this paper, we present a mechanism (**INC**) to avoid network saturation based on message throttling. It can be used with routing algorithms based either on deadlock recovery or avoidance schemes, and works well for different network topologies and network loads.

The mechanism tries to estimate network traffic using only local information available to the node, without sending control information through the network. However, it is able to estimate the status of remote areas by measuring the number of flits transmitted by each virtual channel during a fixed interval of time. If this number is lower than a threshold, the mechanism assumes that there is congestion in that virtual channel. In this situation, a flag is set and message injection restrictions are applied. Two kind of flags are used in order to adjust the injection bandwidth, depending on whether the node is already contributing to the congestion or not yet. In the latter case, applied measures are softer (they are only preventive). Moreover, if congestion is repeatedly detected, injection may be completely forbidden during increasing time intervals. Although both the threshold and the forbidden interval time have to be empirically adjusted, we have found that, as the mechanism works in a progressive way, their adjust is not critical.

The evaluation results show that the mechanism is able to avoid performance degradation for all the analyzed conditions, without introducing any penalty for low and medium network loads, when none congestion control mechanism is required. Moreover, it outperforms recent previous proposals.

References

1. E. Baydal, P. López and J. Duato, "A Simple and Efficient Mechanism to Prevent Saturation in Wormhole Networks", *Int. Parallel & Distributed Processing Symp.*, May 2000.
2. E. Baydal, P. López and J. Duato, "Avoiding Network Congestion with Local Information", *Int. Symp. High Performance Computing*, May 2002.

3. E. Baydal, *Una contribución al control de la congestión en redes de interconexión con conmutación wormhole*, (in Spanish), Ph. D. Thesis, Universidad Politécnica de Valencia, Nov. 2002. <http://www.gap.upv.es>.
4. W. J. Dally and H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, April 1993.
5. J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993.
6. J. Duato and P. López, "Performance evaluation of adaptive routing algorithms for k-ary n-cubes", *Parallel Computer Routing and Communication*, K. Bolding and L. Snyder (ed.), Springer-Verlag, pp. 45–49, 1994.
7. J. Duato, S. Yalamanchili and L.M. Ni, *Interconnection Networks: An Engineering Approach*, 2nd. ed. Morgan Kauffman, 2002.
8. J. H. Kim, Z. Liu and A. A. Chien, "Compressionless routing: A framework for Adaptive and Fault-Tolerant Routing", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 8, No. 3, 1997.
9. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, San Mateo, CA, USA, Morgan Kaufmann Publishers, 1992.
10. P. López, J.M. Martínez and J. Duato, "A Very Efficient Distributed Deadlock Detection Mechanism for Wormhole Networks", *Int. Symp. on High Performance Computer Architecture*, Feb. 1998.
11. J.M. Martínez, P. López and J. Duato, "A Cost-Effective Approach to Deadlock Handling in Wormhole Networks", *IEEE Trans. on Paral. and Distrib. Processing*, pp. 719–729, July 2001.
12. T.M. Pinkston and S. Warnakulasuriya, "On Deadlocks in Interconnection Networks", *Int. Symp. on Computer Architecture*, June 1997.
13. A. Smai and L. Thorelli, "Global Reactive Congestion Control in Multicomputer Networks", *Int. Conf. on High Performance Computing*, 1998.
14. M. Thottetodi, A.R. Lebeck, S.S. Mukherjee, "Self-Tuned Congestion Control for Multiprocessor Networks", *Technical Report CS-2000-15*, Duke University, Nov. 2000.
15. M. Thottetodi, A.R. Lebeck, S.S. Mukherjee, "Self-Tuned Congestion Control for Multiprocessor Networks", *Int. Symp. on High Performance Computer Architecture*, Feb. 2001.

RoCL: A Resource Oriented Communication Library

Albano Alves¹, António Pina², José Exposto, and José Rufino

¹ Instituto Politécnico de Bragança,
Campus Sta. Apolónia, 5301-857 Bragança-Portugal
albano@ipb.pt

² Universidade do Minho
pina@di.uminho.pt

Abstract. RoCL is a communication library that aims to exploit the low-level communication facilities of today's cluster networking hardware and to merge, via the resource oriented paradigm, those facilities and the high-level degree of parallelism achieved on SMP systems through multi-threading.

The communication model defines three major entities – contexts, resources and buffers – which permit the design of high-level solutions. A low-level distributed directory is used to support resource registering and discovering.

The usefulness and applicability of RoCL is briefly addressed through a basic modelling example – the implementation of TPVM over RoCL. Performance results for Myrinet and Gigabit Ethernet, currently supported in RoCL through GM and MVIA, respectively, are also presented.

Keywords: cluster computing, message-passing, directory, multi-threading.

1 Introduction

The appearing of commodity SMP workstations and high-performance SANs aroused the interest of researchers to the topic of cluster computing. Important tools have been developed and have provided an inexpensive vehicle for some classical problems. However, to address an important class of non-scientific large-scale multi-threaded applications and achieve the desired efficiency, in the presence of multiple communication technologies, current approaches and paradigms are not adequate.

1.1 Inter-node Communication

Cluster nodes are typically interconnected by means of high-performance networks like Myrinet or Gigabit Ethernet, but it is also very common to have an alternate low cost communication facility, like Fast Ethernet, to handle node setup and management. To interface high-performance NICs, user level communication libraries, like GM [9] and MVIA [11] (a VIA [4] implementation), became the right choice because it is possible to avoid context switching and memory copies.

Several runtime systems and programming environments have been ported to exploit those technologies (MPI over GM, MPI over Infiniband, PVM over VIA, etc). To provide a uniform interface to multiple communication technologies, intermediate-level communication libraries, such as Madeleine [3], had also been developed. Nevertheless,

it is not usual to combine multiple technologies in order to speed up application execution. Moreover, exploiting a secondary network like Fast Ethernet to perform specific communication tasks to alleviate the overhead of the main networking hardware seems to be an interesting topic that no one has properly addressed yet.

The point is, low-level communication libraries allow the exploitation of high-performance networking hardware but their interfaces are inadequate for application programming and their communication models, mainly concerned with node-to-node message exchanging, make difficult the integration with higher-level abstractions. For instance, GM supports up to eight communication end-points per node which are not enough to deal with the requirements of highly multi-threaded applications.

1.2 Execution Environment

Traditionally the use of parallelism and high-performance computation has been directed to the development of scientific and engineering applications. Multiple platforms are available to help programmers design and develop their applications but it is important to note that the majority of the runtime environments are practically static, since application modules cannot be started disregarding the running ones; in a PVM application, for example, an initial task launches some other tasks (processes), which in turn may launch other ones, and the identifiers of relevant tasks have to be announced by their creators to other participants; in a PM2 [10] application, for example, the programmer is responsible for defining the routines that will be used to deliver active messages.

To deal with the growing complexity of today's large-scale parallel applications, as is the case of a system that integrates crawling, indexing and querying facilities, it is necessary to have a flexible execution environment where: multiple applications from multiple users collaborate to reach a common goal; I/O operations (access to databases, for example) represent a significant part of the application execution time; system modules may execute intermittently; application requirements may vary unpredictably.

Execution environments that fulfil these requirements may be developed using the high-level abstractions provided by MPI, PANDA or others. However, the use of an existing platform imposes some constraints that are incompatible with the innovation needed to deal with the requirements of novel and more complex applications.

2 Resource Oriented Communication

RoCL, the Resource oriented Communication Library we present in this paper, uses existing low-level communication libraries to interface networking hardware. As a new intermediate-level communication library it offers a novel approach to system programmers to facilitate the development of a higher-level programming environment that supports the resource oriented computation paradigm of CoR[12].

2.1 General Concepts

RoCL communication model defines three major entities – contexts, resources and buffers – and uses the services provided by a specific low-level distributed directory.

A context is defined whenever the library is started up by calling the appropriate *init* primitive. Every context owns one or more low-level communication ports to send/receive messages, acting as a message store.

Resources are a common metaphor used to model both communication and computation entities whose existence is announced by registering them in a global distributed directory service. Every resource is associated to an existing context and possesses a unique identifier. RoCL does not define the properties of resources neither limits their definition. Resources are instances of application level concepts whose idiosyncrasies result from a set of attributes specified at creation. An attribute is a pair $\langle name, value \rangle$ where name is a string and value is a byte sequence. A resource R with n attributes is defined by the expression $R = \{\langle name_1, value_1 \rangle, \dots, \langle name_n, value_n \rangle\}$. To register a specific resource, the programmer must enumerate its attributes (see 3.1).

To minimize memory allocation and registering operations, RoCL uses a buffer management system. Messages are held on specific registered memory areas to allow zero-copy communication. Prior to sending messages the programmer must acquire adequate size buffers. At reception the library is responsible for providing the communication subsystems with the necessary pre-registered memory blocks.

Resource global identifiers are used to determine the origin and destination of messages. The identity of a resource may be previously known by the application or it may be obtained from the directory by querying it.

Figure 1 presents the steps required for the operation of a basic client/server interaction, according to the RoCL communication model.

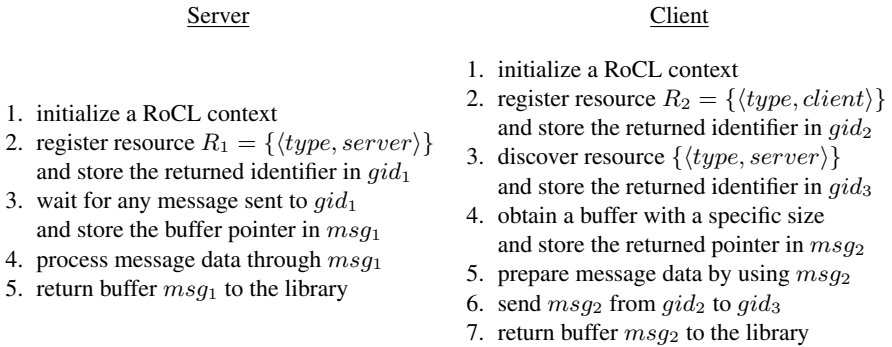


Fig. 1. A basic modelling example.

2.2 Basic Interface

The basic set of primitives that programmers may use to exploit RoCL is presented in table 1, organized according to the involved RoCL entities. Resource handling primitives are discussed through section 3. Buffer management and communication primitives have been presented in detail in a previous paper (see [1]).

In this context it is important to highlight that buffer and communication handling primitives were designed to keep up zero-copy messaging. The use of low-level communication libraries like GM and VIA does not automatically guarantee zero-copy communication; the higher-level abstraction layer must define an appropriate interface to

preserve the low-level features. SOVIA [8] and PVM over VIA [5], for example, use VIA, which permits zero-copy communication, but because users may continue to use the traditional sockets and PVM interfaces, those systems must copy or register user data (memory regions) before sending and can not avoid one copy at reception.

Table 1. RoCL basic primitives.

Contexts
<code>int rocl_init()</code> <code>rocl_exit()</code>
Resources
<code>int rocl_register(rocl_attr_t *attrs)</code> <code>rocl_delete(int gid)</code> <code>int rocl_query(int *gid, rocl_attr_t *attrs)</code>
Buffers
<code>void * rocl_bfget(int len)</code> <code>rocl_bfret(void *ptr)</code> <code>rocl_bftoret(void *ptr)</code> <code>int rocl_bfstat(void *ptr)</code>
Communication
<code>rocl_send(int ogid, int dgid, int tag, void *ptr, int len)</code> <code>int rocl_recv(int dgid, int ogid, int tag, void **ptr,</code> <code> int *aogid, int *atag, int *alen, int timeout)</code>

3 Directory Service

RoCL creates a fully dynamic system where communication entities may appear and disappear, at any moment, during application execution. To support these features we use the resource abstraction along with the facilities of a global distributed directory service. The importance of such a service is emphasized in [2] and [7].

RoCL directory service is a global distributed system that provides efficient and scalable access to the information about registered resources. This service enables the development of more flexible distributed computing services and applications.

3.1 Attribute Lists

A resource is defined/registered by specifying an attribute list. The primitives used to manipulate resource attribute lists are presented in table 2.

Attribute lists are used both for resource registering and querying. To successfully register a resource, all its attributes have to be completely specified, i.e. each attribute has to have a name and a value. For querying purposes, some attributes may be partially defined, i.e. attribute values may be omitted (*NULL* values) in order to inform the library about the attributes we want to know for a specific resource.

Table 2. RoCL primitives to handle attribute lists.

```
rocl_attrl_t * rocl_new_attrl(int maxlen)
int rocl_add_attr(rocl_attrl_t *attrs, char *name, void *val,
                 int len)
void * rocl_get_attr(rocl_attrl_t *attrs, char *name, int *len)
rocl_kill_attrl(rocl_attrl_t *attrs)
```

An attribute list is stored in a contiguous memory region to avoid memory copies when sending it to a server (see 3.2). In fact, an attribute list itself is used as a request/reply packet, requiring some space to be reserved at the attribute list head to allow the attachment of control information.

3.2 Local Operation

RoCL resources are registered at each cluster node using a local server that obtains a subset of global identifiers at start-up, thus minimizing inter-node communication (figure 2). The local resource database (DB in figure 2) is maintained in main memory and hashing techniques are used to accelerate query operations.

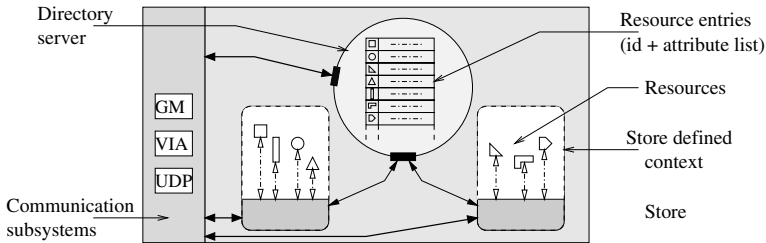


Fig. 2. Resource local registering.

A query received by a local server corresponds to a request packet that may contain: the resource identifier, some completely specified attributes (with valid names and values) and some partially specified attributes (with *NULL* values). If the resource identifier is present, the search mechanism is trivial; otherwise the completely specified attributes are used to produce hash indexes. After finding the right resource, all partially specified attributes are examined and each one will be completed if an attribute with a matching name was previously registered for that resource. Because the library reserves space in the attribute list to store expected values for incomplete attributes, request packets may be used as reply packets avoiding memory allocation and copying.

3.3 Global Operation

If a particular query can not be satisfied at the local server, a global search is initiated. In a global search, all servers running across the cluster receive the request but only the one where the query succeeds is committed to reply.

As a first approach to support global searches, we used UDP broadcast to spread requests through the Fast Ethernet network. This approach benefits from the native broadcasting support at protocol and hardware level.

Requests may also be delivered to servers by combining UDP broadcast and GM or VIA spanning trees. The general operation will be as follows: 1) local servers periodically announce their presence using UDP broadcast; 2) each server maintains a list with all active servers; 3) spanning trees are used to reach all active servers. The use of spanning trees results from the fact that Myrinet hardware does not support broadcasting and the connection oriented model, adopted by VIA, is not compatible with multicasting.

3.4 Multiple-Answer Queries

Queries that don't specify a resource identifier may result in multiple answers returned by one or more servers. This happens because different resources may share some (or even all) attribute names and values.

RoCL provides dedicated primitives, (see table 3) to manage multiple answers to a single query. This interface should not be used whenever we just want a single answer or if it is known in advance that there will be a sole answer to a particular query.

Table 3. RoCL primitives to handle multiple-answer queries.

```
rocl_handler_t * rocl_query_start(rocl_attrl_t *attrs)
int rocl_query_next(rocl_handler_t *handler, rocl_attrl_t *attrs)
int rocl_query_stop(rocl_handler_t *handler)
```

When using this interface, results are fetched from the local server, one at a time, as if multiple independent single answer queries were in progress. Each request/reply packet transports the data – an attribute list – corresponding to a single resource.

To support multiple answers, the local server maintains some control information for each query that is in progress, in order to be able to decide to: look for an answer in the local database, broadcast the query, store the answers returned by remote servers, search the local database for the next result, return an answer obtained from a remote server or request the next result from a remote server.

Each remote server may return only one result as a reply to a specific broadcast. The local server stores the multiple answers received and sends a unicast request to a specific remote server whenever a result previously returned by that server is used up.

4 Inter-resource Message Passing

RoCL applications address messages to resources previously located using the directory service. Inter-resource message passing raises two main problems: message routing, because there is not a direct mapping between resources and communication subsystem addresses, and message dispatching, because resources are animated by threads and multiple low-level communication ports must be multiplexed.

4.1 Message Routing

Contexts are the only valid end-points known by the communication subsystems. Therefore, resources must be mapped into contexts before message sending.

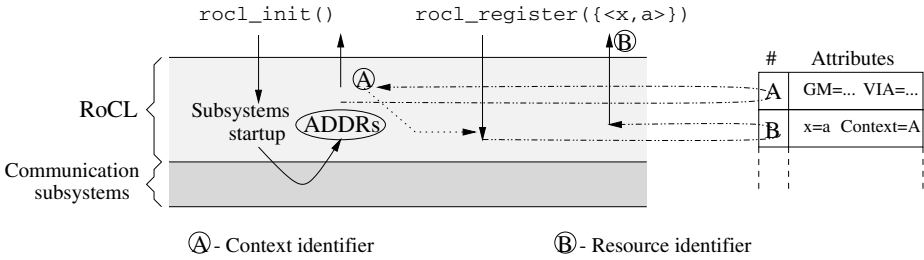


Fig. 3. Resource-context mapping.

The mapping between resources and contexts is handled as shown in figure 3. Contexts are registered at library initialization; they are managed as system resources. The addresses (or ports) of the communication subsystems are used as context attributes. The RoCL library uses the context global identifier, returned by the directory service, as an automatic attribute for resources, i.e. all resources will be tagged with the identifier assigned to the context where they belong.

This approach is quite inefficient because three steps will be required to send a message to a particular resource: 1) the resource context must be obtained by querying the directory system, 2) the context addresses must also be obtained and finally 3) the message will be sent. The two first steps will require some messages (requests and replies) to be exchanged and therefore communication latency will be unacceptable. To overcome this problem, the library uses two dedicated caches to store the most recently required mappings between resource identifiers and context identifiers and between context identifiers and their communication subsystem addresses.

4.2 Message Dispatching

RoCL resources are animated by threads, meaning that RoCL must support concurrent/parallel access to communication facilities. Besides that, message reception is totally asynchronous, meaning that message delivering must take into account that receivers may not be waiting the messages sent to them.

RoCL is a fully connectionless communication system that uses a dispatching mechanism based on system threads and message queues. System threads, one per communication subsystem, wait for messages using polling and interrupt handling techniques and store the received messages in a receiving queue. Resources access this queue to retrieve messages according to some selection criteria (message tag, origin identifier, etc). RoCL provides blocking and timed receivings, through the `timeout` parameter (see table 1): a negative timeout indicates a blocking behavior.

Sending primitives may directly access the communication subsystems but, whenever concurrent calls occur, messages are stored in a sending queue and the primitives return immediately.

Because receiving and sending queues only handle message descriptors, containing a pointer to the message data, no extra copies are introduced. A detailed explanation of message dispatching and the way it is related to quite distinct user-level communication protocols – GM and VIA – may be found in [1].

5 Applicability and Performance

Although RoCL was designed as an intermediate-level message-passing library to the development of a new programming platform that is still under construction, it constitutes per se a basic programming tool. So, it is already possible to present particular applicability examples along with raw performance results.

5.1 TPVM over RoCL

Assuming a simplified view of TPVM [6] we will examine the design/implementation of some of its basic functionality using RoCL primitives.

PVM tasks (processes under UNIX) will create a RoCL context at start-up when they start running. They also register themselves as task resources, using an attribute $\langle type, task \rangle$, and their global resource identifiers are used as PVM task identifiers (TIDs). A simplified PVM task spawning mechanism may be achieved as follows:

- the parent task uses *rsh* to execute a special process launcher passing to it the conventional arguments required to start the spawned program along with its TID;
- the launcher registers a temporary resource using as attributes its process identifier (PID) and the received parent task identifier ($\{\langle type, tmp \rangle, \langle pid, ... \rangle, \langle ptid, ... \rangle\}$);
- the launcher starts the target program using the *exec* primitive;
- the spawned task, at start-up, queries the directory service, using its process identifier, to find the temporary resource and to obtain the parent task identifier;
- the spawned task sends its global resource identifier to the parent/spawner process.

To remotely activate TPVM threads we will need a launcher thread, automatically created when the PVM process/task starts, that will block waiting for messages that request the activation of a specific TPVM thread. The launcher thread – the *pod* controller – registers itself as a system thread resource using an attribute $\langle type, systhread \rangle$.

The *tpvm_export* primitive will correspond to a simple RoCL register operation. A TPVM thread will be defined by using a name and the global resource identifier of its *pod* controller – $\{\langle type, threaddef \rangle, \langle name, ... \rangle, \langle systhread, ... \rangle\}$.

The *tpvm_spawn* primitive will use the thread name to query the directory service and find the global identifier of the *pod* controller. This global identifier is used to send a request message to the launcher thread, which will create the desired thread. Instantiated threads also register themselves using an attribute $\langle type, thread \rangle$ and send their global identifiers to the spawners. A spawned thread obtains its parent identifier directly; the request sent to the launcher thread carries the spawner global identifier.

A host database may also be provided as a collection of RoCL resources. However, since no PVM daemons are used, this database will only be useful to find out available machines and select a particular target node for spawning operations.

5.2 Inter-resource Message-Passing Performance

RoCL message-passing performance is influenced by the communication subsystems we support, the way we interface them and the capabilities of LINUX threads ¹.

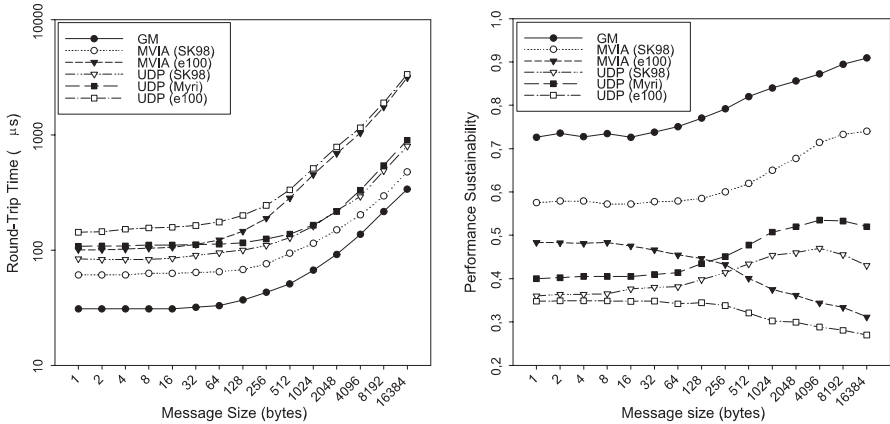


Fig. 4. RoCL performance.

Figure 4:left presents round-trip times relating to three networking technologies – Myrinet (LANai 9), Gigabit Ethernet (SysConnect 9821) and Fast Ethernet (Intel EtherExpress PRO/100) – exploited through GM, MVIA and UDP ². Performance tests were performed on dual Pentium III 733MHz workstations running RedHat 9.0. Myrinet and Gigabit adapters were attached to 64bits/66MHz PCI slots.

RoCL over GM allows to achieve a 30μs round-trip time for 1 byte messages, which corresponds to an overhead of 10μs when compared to GM node-to-node performance. MVIA for SysConnect, as expected, outperforms all communication subsystem alternatives but GM. Surprisingly, for small messages, SysConnect hardware produces better results than Myrinet, when using UDP.

In order to evaluate message-passing alternatives, it is mandatory to also analyze the impact of communication on computation and vice-versa. To evaluate this inter-dependence we had calculated the execution rate of a particular computation cycle, using one thread per processor, without performing any communication task, and then we run both the original round-trip and the computation benchmarks concurrently.

The impact on computation may be expressed by the ratio Ex_{CS}/Ex_0 , where Ex_{CS} stands for the execution rate obtained when we run, concurrently, the round-trip benchmark using one of the communication subsystem alternatives and Ex_0 stands

¹ Currently RoCL is only supported under LINUX.

² GM runs on Myrinet, MVIA runs on SysConnect and Intel and UDP runs on each of them.

for the execution rate obtained without background communication. Similarly, the ratio Rt_{CS_0}/Rt_{CS} , where Rt_{CS} and Rt_{CS_0} stand for the round-trip times obtained for a given communication subsystem, respectively, with or without computations taking place concurrently, express the impact on communication. To easily compare round-trip ratios from different communication subsystems, we use the ratio Rt_{GM_0}/Rt_{CS} , where Rt_{GM_0} stands for the round-trip times obtained using GM (the best round-trip times we may achieve with RoCL).

These two ratios express the performance sustainability of computation and communication, when they use the same CPU(s). As we consider that both communication and computation performances are equally fundamental to determine the success of high performance computing, we use an overall ratio, for each communication subsystem, calculated as a geometric average.

Figure 4: *right* presents the overall performance sustainability we can expect to achieve in RoCL applications. It is important to note that the RoCL impact on computation is in accordance to the selected hardware and communication subsystem: FastEthernet adapters perform badly and require higher CPU intervention as message size increases; UDP, a complex protocol, requires more CPU cycles than MVIA and GM and so the overall performance drops.

6 Conclusions

RoCL introduces a new communication paradigm to facilitate the design and implementation of high-level execution environments. In this paper, the key concepts related to inter-resource message-passing and the operation of a low-level distributed directory service – a very important component of RoCL – were presented.

The case study “TPVM over RoCL” shows that the abstractions provided by RoCL allow the rapid design and implementation of a great variety of high-level applications.

Performance values indicate that RoCL exploits efficiently the low-level communication subsystems and that multi-threaded dispatching mechanisms are now feasible.

Scalability evaluation of the directory service is still undergoing. Due to the limited number of cluster nodes available for testing, we intend to use simulation techniques.

References

1. A. Alves, A. Pina, J. Exposto, and J. Rufino. ToCL: a thread oriented communication library to interface VIA and GM low-level protocols. to appear ICCS '03, 2003.
2. M. Beck, J. Dongarra, G. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam. HARNESS: A next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5–6):571–582, 1999.
3. L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine: An Efficient and Portable Communication Interface for RPC-Based Multithreaded Environments. In *PACT '98*, 1998.
4. Compaq Computer Corp., Intel Corporation & Microsoft Corporation. Virtual Interface Architecture Specification. <http://www.vidf.org/info/04standards.html>, 1997.
5. R. Espenica and P. Medeiros. Porting PVM to the VIA architecture using a fast communication library. In *PVM/MPI '02*, 2002.

6. J. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *HPDC '95*, 1995.
7. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Perf. Distributed Computations. In *HPDC '97*, 1997.
8. J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *CLUSTER '01*, 2001.
9. Myricom. The GM Message Passing System. <http://www.myricom.com>, 2000.
10. R. Namyst and J. Méhaut. PM²: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo '95*, 1995.
11. National Energy Research Scientific Computing Center. M-VIA: A High Performance Modular VIA for Linux. <http://www.nersc.gov/research/FTG/via>, 2002.
12. A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR - a Prototype for Resource Oriented Computing. In *HPC '02*, 2002.

A QoS Multicast Routing Protocol for Dynamic Group Topology

Li Layuan and Li Chunlin

Department of Computer Science, Wuhan University of Technology, Wuhan 430063,
P. R.China
jwtu@public.wh.hb.cn

Abstract. This paper discusses the multicast routing problem with multiple QoS constraints, which may deal with the delay, delay jitter, bandwidth and packet loss metrics, and describes a network model for researching the routing problem. It presents a multicast routing protocol with multiple QoS constraints (MRPMQ). The MRPMQ attempts to significantly reduce the overhead of constructing a multicast tree with multiple QoS constraints. In MPRMQ, a multicast group member can join or leave a multicast session dynamically, which should not disrupt the multicast tree. It also attempts to minimize overall cost of the tree, and satisfy the multiple QoS constraints and least cost (or lower cost) requirements. In this paper, the proof of correctness and a complexity analysis of the MRPMQ are also given. Simulation results show that MRPMQ is a feasible approach to multicast routing with multiple QoS constraints.

Keywords: Multicast routing; protocol; multiple QoS constraints; QoS routing; NP-complete.

1 Introduction

The traditional multicast routing protocols, e.g., CBT and PIM [1-4], were designed for best-effort data traffic. They construct multicast trees primarily based on connectivity. Such trees may be unsatisfactory when QoS is considered due to the lack of resources. Several QoS multicast routing algorithms have been proposed recently. Some algorithms [5-7] provide heuristic solutions to the NP-complete constrained Steiner tree problem, which is to find the delay-constrained least-cost multicast trees. These algorithms however are most practical in the Internet environment because they have excessive computation overhead, require knowledge about the global network state, and do not handle dynamic group membership. Jia's distributed algorithm [2] does not compute any path or assume the unicast routing table can provide it. However, this algorithm requires excessive message processing overhead. The spanning join protocol by Carlberg and Crowcroft [3] handles dynamic membership and does not require any global network state. However, it has excessive communication and message processing overhead because it relies on flooding to find a feasible tree branch to connect a new member. QoSMIC [4], proposed by Faloutsos et al., alleviates but does not eliminate the flooding behavior. In addition, an extra control element, called Manager router, is introduced to handle the join requests of new members.

Multicast routing and its QoS-driven extension are indispensable components in a QoS-centric network architecture [5,7,9–10]. Its main objective is to construct a multicast tree that optimizes a certain objective function (e.g., making effective use of network resources) with respect to performance-related constraints (e.g., end-to-end delay bound, inter-receiver delay jitter bound, minimum bandwidth available, and maximum packet loss probability).

2 Network Model

As far as multicast routing is concerned, a network is usually represented as a weighted digraph $G = (V, E)$, where V denotes the set of nodes and E denotes the set of communication links connecting the nodes. $|V|$ and $|E|$ denote the number of nodes and links in the network, respectively. Without loss of generality, only digraphs are considered in which there exists at most one link between a pair of ordered nodes [8]. Associated with each link are parameters that describe the current status of the link.

Let $s \in V$ be a source node of a multicast tree, and $M \subseteq \{V - \{s\}\}$ be a set of end nodes of the multicast tree. Let R be the set of positive weights and R^+ be the set of nonnegative weights. For any link $e \in E$, we can define the following QoS metrics: delay function $delay(e): E \rightarrow R$, cost function $cost(e): E \rightarrow R$, bandwidth function $bandwidth(e): E \rightarrow R$, and delay jitter function $delay-jitter(e): E \rightarrow R^+$. Similarly, for any node $n \in V$, one can also define some metrics: delay function $delay(n): V \rightarrow R$, cost function $cost(n): V \rightarrow R$, delay jitter function $delay-jitter(n): V \rightarrow R^+$ and packet loss function $packet-loss(n): V \rightarrow R^+$. We also use $T(s, M)$ to denote a multicast tree, which has the following relations:

- 1) $delay(p(s, t)) = \sum_{e \in p(s, t)} delay(e) + \sum_{n \in p(s, t)} delay(n)$.
- 2) $cost(T(s, M)) = \sum_{e \in T(s, M)} cost(e) + \sum_{n \in T(s, M)} cost(n)$.
- 3) $bandwidth(p(s, t)) = \min_{e \in p(s, t)} \{bandwidth(e)\}$.
- 4) $delay-jitter(p(s, t)) = \sum_{e \in p(s, t)} delay - jitter(e) + \sum_{n \in p(s, t)} delay - jitter(n)$.
- 5) $packet-loss(p(s, t)) = 1 - \prod_{n \in p(s, t)} (1 - packet-loss(n))$

where $p(s, t)$ denotes the path from source s to end node t of $T(s, M)$.

Definition 1. The QoS-based multicast routing problem deals mainly with some elements: network $G=(V,E)$, multicast source $s \in V$, the set of end nodes $M \subseteq \{V - \{s\}\}$, $delay(\cdot) \in R$, $delay-jitter(\cdot) \in R^+$, $cost(\cdot) \in R$, $bandwidth(\cdot) \in R$, and $packet-loss(\cdot) \in R^+$. This routing problem is to find the $T(s, M)$ which satisfies some QoS constraints for all $t \in M$:

- 1) Delay constraint: $delay(p(s, t)) \leq D_t$
- 2) Bandwidth constraint: $bandwidth(p(s, t)) \geq B$

3) Delay jitter constraint: $delay-jitter(p(s,t)) \leq J$

4) Packet loss constraint: $packet-loss(p(s,t)) \leq L$

Simultaneously, the $cost(T(s, M))$ should be minimum. In the above QoS constraints, the bandwidth is a concave metric, the delay and delay jitter are additive metrics, and the packet loss is multiplicative metric. In these metrics, the multiplicative metric can be converted to an additive metric. For simplicity, we assume that all nodes have enough resources, i.e., they can satisfy the above QoS constraints. Therefore, we only consider the links' or edges' QoS constraints, because the links and the nodes are equivalent to the routing issue in question.

3 MRPMQ

The Join procedure of MRPMQ can be formally described as follows.

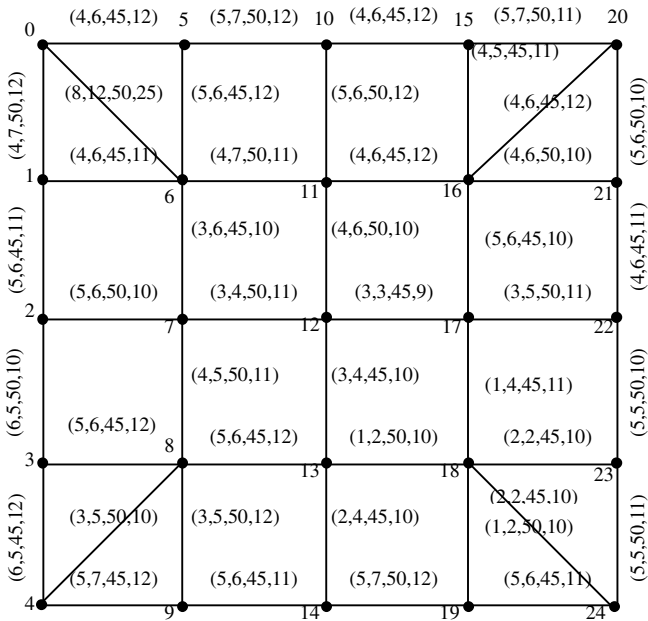
- 1) if a new member (t_i) wishes to join a $T(s, M) \rightarrow$
 the new member sends JOINreq to some neighbor t_j
- 2) if $(d(s, *) + d(j, i) \leq D) \wedge (dj(s, *) + dj(j, i) \leq J) \wedge (bw(t_u, t_v) \geq B) \rightarrow$
 {where $d(s, *)$ and $dj(s, *)$ are the delay sum and the delay jitter sum from the source s to all downstream nodes of a path, respectively, but except for the last pair of nodes. u and v are the sequence numbers between two adjacent nodes on the path from source to the new member}
 t_j transfers JOINack to t_i
 fi
 if $bw(t_u, t_v) < B \rightarrow$
 remove $e(u, v)$ from G
 fi
- 3) if $(d(s, *) + d(j, i) > D) \wedge (dj(s, *) + dj(j, i) > J) \rightarrow$
 if the next hop is the immediate upstream node t_j of $t_j \rightarrow$
 t_j transfers JOINreq to t_j .
 t_j adds JOINpend for t_i to the forwarding entry
 t_j transfers JOINack (or JOINnak) to t_j
 fi
 if the next hop is not the immediate upstream node $\rightarrow$
 if $(d(s, *) \geq D - d(j, i)) \wedge (dj(s, *) \geq J - dj(j, i))$
 t_j transfers JOINreq to t_{j^*}
 t_j adds the routing entry
 marks t_{j^*} as upstream node
 t_{j^*} transfers JOINack (or JOINnak) to t_j
 if t_j receives JOINack $\rightarrow$
 t_j forwards a pruning msg to t_j .
 fi
 fi
 fi
- 4) if $(d(s, *) + d(j, i) \leq D) \vee (dj(s, *) + dj(j, i) \leq J) \rightarrow$
 t_j computes a new path
 if $(p(s, i) = \min[d(s, *), D - d(j, i)]) \wedge$

```

    (dj(p(s,j))=min[dj(s,*), J-dj(j,i)]→
    tj receives JOINack
fi
    tj receives JOINnak
fi

```

We can use the following example to show how the MRPMQ works and how the multicast tree is constructed in a distributed fashion. Fig. 1 is a network graph. In this example, node t_0 is the multicast source. t_4 , t_9 , t_{14} , t_{19} and t_{24} are the joining nodes. Recall that the characteristics of a network's edge can be described by a fourtuple (D, J, B, C) . In this example shown in Fig. 1, suppose delay constraint is $D=20$, delay jitter constraint $J=30$ and bandwidth constraint $B=40$. t_4 wishes to join the group, it computes the paths according to the multiple QoS constraints. The path



$(t_0 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{13} \rightarrow t_{14})$ satisfy the delay, delay jitter and bandwidth constraints. The latter has the lower cost. Therefore, the join path should be the path $(t_0 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{13} \rightarrow t_{14})$. Meanwhile, t_6 should prune off from the original parent t_1 , the resulting tree is shown in Fig. 2(c) (see the bold lines of Fig. 2(c)). The tree after t_{19} joins the group is also shown in Fig. 2(d). When t_{24} joins the group, it computes the join paths. If t_{18} receives JOINreq from t_{24} , it finds out that the existing path $(t_0 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{13} \rightarrow t_{18})$ does not satisfy the delay constraint for the new member t_{24} , while the new path $(t_0 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{17} \rightarrow t_{18})$ does not satisfy the delay jitter constraint for t_{24} . t_{18} computes a new feasible path with delay constraint, which is given by

$$\begin{aligned} d(p(s,j)) &= \min[d(s,*), D-d(j,i)] \\ &= \min[(d(0,5)+d(5,6)+d(6,7)+d(7,12)+ \\ &\quad d(12,13)+d(13,18)), D-d(18,24)] \\ &= \min[19,18]=18 \end{aligned}$$

and delay jitter constraint, which can be given by

$$\begin{aligned} dj(p(s,j)) &= \min[dj(s,*), J-dj(j,i)] \\ &= \min[(dj(0,5)+dj(5,6)+dj(6,7)+dj(7,12)+ \\ &\quad dj(12,13)+dj(13,18)), J-dj(18,24)] \\ &= \min[28,28]=28 \end{aligned}$$

Thus, this new feasible path should be path $(t_0 \rightarrow t_6 \rightarrow t_7 \rightarrow t_{12} \rightarrow t_{13} \rightarrow t_{18})$. t_6 should prune off from the old parent t_5 , and the final tree can be shown in Fig. 2(e) (see the bold lines of Fig. 2(e)).

The loop-free routing for the above protocol can be achieved by maintaining a searching tree at any time.

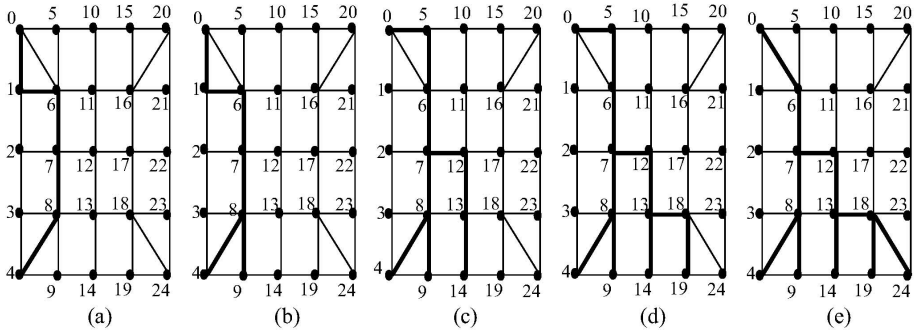


Fig. 2. Constructing multicast tree

4 Correctness and Complexity Analysis

Theorem 1. If a path from a new member to $T(s,M)$ has sufficient resources to satisfy the QoS constraints and has minimum cost, the algorithm searches only one path.

Proof. Note that a necessary condition for multiple paths to be searched is that a single path does not satisfy the QoS constraints, such as $(d(p(s,j)) \neq \min[d(s,*), D-d(j,i)]) \wedge (dj(p(s,j)) \neq \min[dj(s,*), J-dj(j,i)])$. However, if sufficient resources are

available on every link and node of the path, no node forwarding JOINreg will ever enter the multiple paths search state. Thus, the above theorem holds.

Lemma 1. Whenever during the routing process, all paths being searched form a $T(s, M)$ structure.

Proof. The paths being searched will be marked by the routing entries at the nodes. In MRPMQ, any routing entry has a single out interface and one or multiple in interfaces. Hence, the nodes will form a searching tree structure. This tree is just a $T(s, M)$.

Theorem 2. An available and feasible path found by MRPMQ is loop-free.

Proof. This Theorem follows directly from the above Lemma 1.

Theorem 3. MRPMQ can find an available and feasible path if one exists.

Proof. This theorem can be proved by contradiction. Suppose MRPMQ fails while an available and feasible path does exist. Let $e(i, j)$ be the first link in the path that the protocol did not explore. Since $e(i, j)$ is the first unexplored link of the path, t_i must have received a request message from the previous link or t_i is the new member issuing the request message. In either case, t_i is not in the initial state. Therefore, t_i is in the failure state, which requires t_i to explore all outgoing links including $e(i, j)$. It contradicts the assumption that $e(i, j)$ is not explored.

In MRPMQ, route computation can generally be made by the end node. If the join path is computed on-demand, the complexity depends on the unicast protocol. If QoS metrics are delay and bandwidth, there exist QoS routing heuristics which are $O(|V| \times |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in a network. For most networks, $|E| = O(|V|)$, hence the complexity is $O(|V|^2)$. For a multicast group with $|M|$ members, the computation overhead is $O(|V|^2 |M|)$. The study shows that computation complexities of CSPT and BSMA [5] are $O(|E| \log |V|)$ and $O(|V|^3 \log |V|)$, respectively. The study shows that the average message processing overheads to construct the multicast tree of MRPMQ, Jia's algorithm, and QoSMIC (centralized or distributed) are $K.2|M|$, $K.2|M|$, $|M| (w \cdot (w-1)^{(y-1)} + c - k) \cdot x$ (centralized QoSMIC) and $|M| (w \cdot (w-1)^{(y-1)} + |T|) \cdot x$ (distributed QoSMIC), respectively, where the x factor is added to reflect the fact that messages have to be processed at more than one node, w is the average degree of a node, y is the maximum TTL used for a search, $|T|$ is the tree size, c is the number of candidates for a BID-ORDER session and x depends on the topology and y , while $2 \leq x \leq 1 + K$.

5 Simulations

In the simulations, we compare the quality of routing trees by their network cost for constructing a multicast tree ($cost(T(s, M))$) [10]. The network cost is obtained as the mean value of the total number of simulation runs. At each simulation point, the simulation runs 80 times. Each time the nodes in the group G are randomly picked out from the network graph. The network cost is simulated against two parameters: delay bound D and group size. In order to simulate real situations, the group size is always made less than 20% of the total nodes, because multicast applications running in a wide area network usually involve only a small number of nodes in the network, such

as video conference systems, distance learning, co-operative editing systems, etc. [6–7].

Fig. 3 shows the network cost versus group size. In this round of simulations, the network size is set to 300 and D is $d_{\max}+3/8d_{\max}$. From Fig. 3, we can see when group size grows, the network cost produced by MRPMQ, BSMA and KMB increases at a rate much lower than CSPT. MRPMQ performs between BSMA and KMB. BSMA, KMB and the proposed MRPMQ can produce trees of comparable costs.

Fig. 4 is the network cost versus D . During this round of simulations, the network size is fixed at 300 nodes, group size is 20. From Fig. 4, it can be seen that the network cost of the CSPT algorithm is on the top and almost does not change as D increases. This is because the generation of the shortest path tree does not depend on D . Of the remaining three algorithms, the proposed MRPMQ has the lowest cost. From Fig. 4, we can also see that tree costs decrease for the MRPMQ, BSMA and KMB algorithms as the delay bound is relaxed. This shows all three schemes indeed can reduce the cost when the delay bound is relaxed. From Fig. 3 and Fig. 4, one can see that the MRPMQ, BSMA and KMB algorithms can produce trees of comparable costs. However, compared with the BSMA and KMB algorithms, the proposed MRPMQ has the advantage of being fully distributed and allowing incremental tree build-up to accommodate dynamic joining of new members. Furthermore, the MRPMQ is much less costly in terms of computation cost and in terms of cooperation needed from other network nodes compared with other schemes.

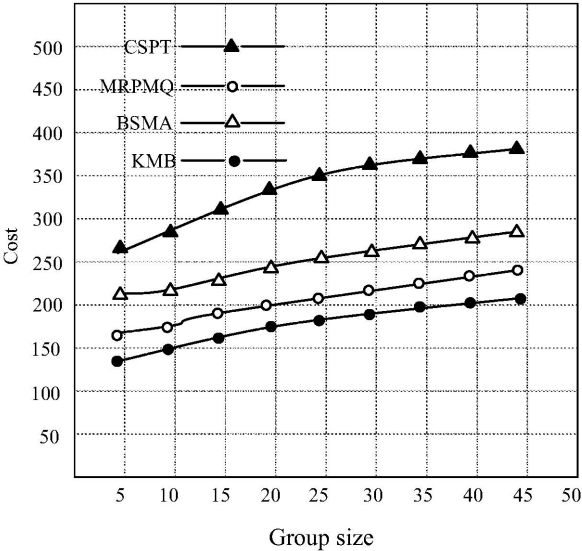


Fig. 3. Network cost vs. group size

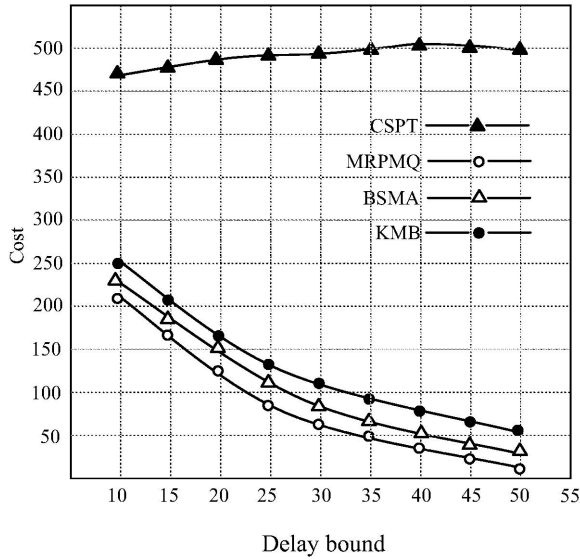


Fig. 4. Network cost vs. delay bound

6 Conclusion

In this paper, we discuss the multicast routing problem with multiple QoS constraints, which may deal with the delay, delay jitter, bandwidth and packet loss metrics, and describe a network model for researching the routing problem. We have presented a multicast routing protocol with multiple QoS constraints (MRPMQ). The MRPMQ can significantly reduce the overhead of establishing a multicast tree. In MRPMQ, a multicast group member can join or leave a multicast session dynamically, which should not disrupt the multicast tree. The MRPMQ also attempts to minimize the overall cost of the tree. This protocol may search multiple feasible tree branches in distributed fashion, and can select the best branch connecting the new member to the tree. The join of a new member can have minimum overhead to on-tree or non-tree nodes. The correctness proof and complexity analysis have been made. Some simulation results are also given. The study shows that MRPMQ is a feasible approach to multicast routing with multiple QoS constraints. Further work will investigate the protocol's suitability for inter-domain multicast and hierarchical network environments.

Acknowledgment. The work is supported by National Natural Science Foundation of China and NSF of Hubei Province.

References

- [1] Li Layuan and Li Chunlin, "The QoS routing algorithm for ATM networks", *Computer Communications*, Vol. 24, No. 3-4, 2001, pp. 416-421.
- [2] X. Jia. "A distributed algorithm of delay-bounded multicast routing for multimedia applications in wide area networks", *IEEE/ACM Trans. on Networking*, Vol. 6, No. 6, Dec. 1998, pp. 828-837.
- [3] K. Carlberg and J. Crowcroft, "Building shared trees using a one-to-many joining mechanism", *ACM Computer Communication Review*, Vol. 27, No. 1, Jan. 1997, pp. 5-11.
- [4] M. Faloutsos, A. Banerjee, and R. Pankaj, "QoSMIC: Quality of Service sensitive multicast Internet protocol", *SIGCOMM'98*, Vol. 28, September 1998.
- [5] Q. Zhu, M. Parsa, and J. J. Garcia-Luna-Aceves, "A source-based algorithm for delay-constrained minimum-cost multicasting," *Proc. IEEE INFOCOM 95*, Boston, MA, April 1995.
- [6] Y. Xiong and L.G. Mason, "Restoration strategies and spare capacity requirements in self-healing ATM networks", *IEEE/ACM Trans. on Networking*, Vol. 7, No. 1, Feb. 1999, pp. 98-110.
- [7] B. M. Waxman, "Routing of multipoint connections." *IEEE Journal on Selected Area in Communications*, Dec. 1998, pp. 1617-1622.
- [8] R. G. Busacker and T. L. Saaty, *Finite Graphs and Networks: An introduction with applications*, McGraw-Hill, 1965.
- [9] R. A. Guerin and A. Orda, "QoS routing in networks with inaccurate information: Theory and algorithms," *IEEE/ACM Trans. on Networking*, Vol. 7, No. 3, June 1999, pp. 350-363.
- [10] Li Layuan and Li Chunlin, *Computer Networking*, National Defense Industry Press, Beijing, 2001.

A Study of Network Capacity under Deflection Routing Schemes

Josep Fàbrega and Xavier Muñoz*

Departament de Matemàtica Aplicada IV, Universitat Politècnica de Catalunya
Campus Nord C3, Jordi Girona 1-3, 08034 Barcelona, Spain
`{matjfc,xml}@mat.upc.es`

Abstract. Routing in bufferless networks can be performed without packet loss by deflecting packets when links are not available. The efficiency of this kind of protocol (deflection routing) is highly determined by the decision rule used to choose which packets have to be deflected when a conflict arises (i.e. when multiple packets contend for a single outgoing link). As the load offered to the network increases the probability of collision becomes higher and it is to be expected that at a certain maximum offered load the network gets saturated. We present an analytical method to compute this maximum load that nodes can offer to the network under different deflection criteria.

1 Introduction

Deflection routing [1] is a routing scheme for bufferless networks based on the idea that if a packet cannot be sent through a certain link due to congestion, it is deflected through any other available one (instead of being buffered in a node queue) and rerouted to destination from the node at which the packet arrives. In this way, congestion causes packets admitted to the network to be misrouted temporarily, in contrast with traditional schemes where such packets might be buffered or dropped. This kind of protocol has been proposed, for instance, to route packets in all-optical networks because optical storage is not possible with nowadays technology [2,4,7] (Messages can only be shortly delayed by a fiber loop in order to wait for a quick processing of their headers, but cannot be buffered in queues without optical to electrical conversion.)

Many approximations have been proposed in the literature for implementing deflection routing [8]. The efficiency of the protocol is highly determined by the decision criteria used to deflect packets when collisions arise (i.e. when two packets should use the same outgoing link and one of them must be deflected). The strategies used to solve these conflicts can be divided into two categories. On one hand, those that give priority to the most disadvantaged packets in order to avoid deadlock or timeouts (and thus trying to guarantee that all packets arrive

* Research supported by the Ministry of Science and Technology, Spain, and the European Regional Development Fund (ERDF) under projects TIC-2001-2171 and TIC-2002-00155.

to destination): MAXDIST (packets that are further away to destination have a higher priority), MAXTIME (older packets are given a higher priority), and MAXDEFL (the larger the number of times a packet has been deflected, the higher the priority). On the other hand, strategies that give preference to those packets that might arrive to destination as soon as possible. The decision criteria within this group are analogous to the preceding case: MINDIST, MINTIME and MINDEFL.

As the load offered to the network increases, deflection routing becomes less efficient (the probability of collision increases) and at a certain maximum offered load R the network becomes saturated. Figure 1 shows the results of simulations under different decision criteria for a certain network topology. As the reader can check, the maximum allowed traffic highly depends on the decision rule used to deflect packets. This paper shows an analytical method to compute this maximum load R . Some specific theoretical results for the hypercube and shufflenet networks can be found in [11,10].

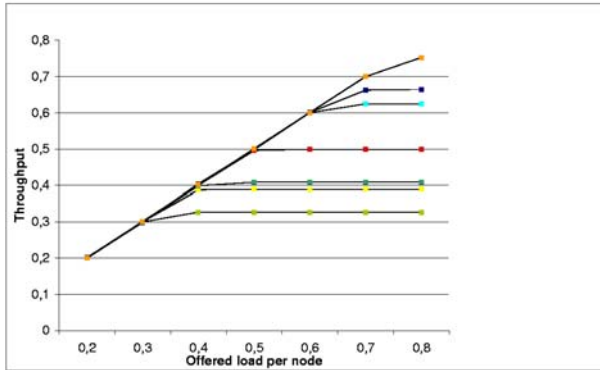


Fig. 1. Delivered vs. offered traffic

2 The Model

Network topology is another important aspect that determines the efficiency of deflection routing. We suppose here that the network is modeled by a directed graph (digraph) in which all nodes offer the same constant traffic load to the network by means of an input queue. We also assume that the communication model have the following properties: packets have fixed length and are transmitted synchronously, processing time at the intermediate nodes is zero, only one packet can travel through a given link at a time, and a packet in an input queue enters in the network as soon as a link is available and competes with other packets in the network for the same links (transmit no hold acceptance (txnh)). (Other queue management policies can be found in [5].) With respect to the topology and routing table the following restrictions are also supposed: all

nodes have the same number of outgoing and incoming links (i.e. the digraph is δ -regular), packets will always try to follow the shortest path to destination, and there is only one shortest path between any pair of nodes, which means that a packet remains at the same distance or further away to its destination after being deflected. Besides the previous assumptions we consider two more suppositions in order to simplify our problem. Firstly, a packet entering to a node exits through any outgoing arc with probability $1/\delta$ (uniform traffic distribution). This approximation is reasonable if the routing table is such that the edge-forwarding index [12] is close to the minimum. Secondly, since the purpose of our study is to deal under heavy traffic conditions, it will be assumed that each node has a packet ready to be transmitted as soon as a link is free. This supposition holds only for deterministic distribution of packet arrivals but it is not true for other traffic distributions such as Poisson or sporadic. Nevertheless the supposition is acceptable if the arrival ratio is high enough and if there are large input buffers which are overflowed.

With these assumptions, in steady-state the number of incoming packets to a node equals the number of incoming links to that node, and equals also the number of outgoing packets and outgoing links. In other words, the probability of a link being free during a time unit is zero. Hence the number of packets in the network equals the number of arcs in the digraph, and then, by Little's Law we have $nR\bar{t} = m$ where R stands for the maximum admissible throughput per node (i.e. the maximum load rate nodes can offer to the network) and $\bar{t}$ is the average time for a packet to arrive to destination. In the case of δ -regular directed graphs, we have $R = \delta/\bar{t}$. The following two sections will be devoted to give methods to compute R under different decision criteria.

3 Random Policy

To compute the average time $\bar{t}$ that (in steady-state) a packet is in the network hopping through its nodes, let us define an absorbing Markov chain with states corresponding to the possible distances that the packet could be to its destination: $0, 1, \dots, D$, where D is the diameter of the digraph and state zero stands for a packet that has arrived at destination (see Figure 2). The transition

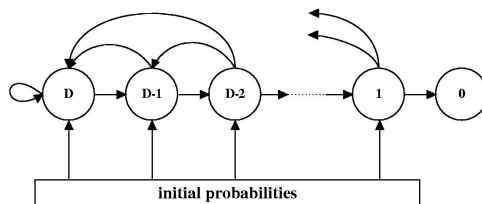


Fig. 2. Markov chain used to compute the average time to arrive to destination

probability matrix $M = (m_{ij})$ of this chain is given by $m_{ij} = P_a(i) = 1 - P_d(i)$ if $j = i - 1$ and $i > 0$, $m_{ij} = P_d(i)P_t(i, j)$ if $j \geq i$ and $i > 0$, and 0 otherwise, where $P_d(i)$ is the probability that a packet being at distance i is deflected and $P_t(i, j)$ is the conditional probability that a deflected packet which is at distance i to its destination z goes to a node which is at distance j to z .

The deflection probability $P_d(i)$ depends on the number of packets that should use the same link at a time. Given a packet that wants to use certain link, the probability that N other packets want to use the same outgoing link (N collisions) is given by $P_c(N) = \binom{\delta-1}{N} (1 - 1/\delta)^{\delta-1-N} (1/\delta)^N$. Therefore, the probability of being deflected conditioned to being at distance i to destination is $P_d(i) = \sum_{N=1}^{\delta-1} (1 - P_a(i | N)) P_c(N)$, where $P_a(i | N)$ stands for the probability $P_a(i)$ conditioned to N collisions. In the case of *random policy*, this probability does not depend on i , and according to assumption of uniform traffic, is given by $P_a(i | N) = 1/(N + 1)$. Consequently, in the case of random policy, the probability of deflection is, for any distance i , given by $P_d = \sum_{N=1}^{\delta-1} (1 - 1/(N + 1)) \binom{\delta-1}{N} (1 - 1/\delta)^{\delta-1-N} (1/\delta)^N = (1 - 1/\delta)^\delta$.

On the other hand, the probabilities $P_t(i, j)$ depend only on the network topology as well as the initial probability $P_{in}(i)$ of each state (i.e. the probability that a new packet entering the network is assigned a destination node at distance i from the source). A detailed analysis of these probabilities can be found in [9] for the case of Kautz networks [3].

Once defined the transition and initial probabilities, the Markov chain makes it possible to compute the average time that a packet is in the network by computing the mean time $\bar{t}$ to absorption to the zero state. This time will give us the maximum admissible throughput $R = \delta/\bar{t}$.

4 Distance Priority Criteria

In case of distance criteria, the probability that a packet advances towards destination depends on its distance to that destination, but also on the probability that competitors are closer to or further to its own destinations. In other words, to compute the probability of advancing $P_a(i) = 1 - P_d(i)$ we need to know the probability of being at each state in the Markov chain. Hence, a more complex analysis must be performed.

Since packets fighting for a certain outgoing link have not arrived yet to destination (i.e. they are not in the zero state) we can consider the probabilities of being at each state conditioned to not being in state 0. Let us call these new conditioned state probabilities $P'(i)$. In order to compute $P'(i)$ and the new transition probabilities it is convenient to state the problem in a different way: there is a fixed number of packets (equal to the number of links) hopping through the nodes in such a way that they will never exit the network, but once a packet reaches its destination, a new random destination is immediately assigned to it. The new ergodic Markov chain associated to this problem is shown in Figure 3. Because of the assumption that each node has a packet ready to be transmitted as soon as a link is free, the traffic in the network will be exactly the same as in

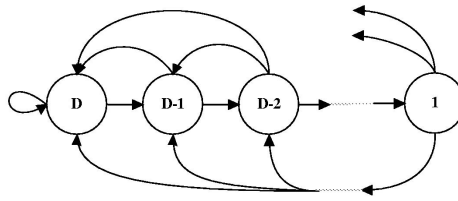


Fig. 3. Conditioned Markov chain

the original problem and, moreover, the transition probabilities will also be the same for $i \geq 2$. More precisely, the entries m'_{ij} of the new transition probability matrix $M' = (m'_{ij})$ are $1 - P_d(i)$ if $j = i - 1$ and $i \neq 1$, $P_d(i)P_t(i, j)$ if $j \geq i$ and $i \neq 1$, $P_d(i)P_t(i, j) + (1 - P_d(i))P_{in}(j)$ if $i = 1$, and 0 otherwise, where $P_{in}(j)$ stands for the probability that the new assigned destination is at distance j from source, as in the preceding section.

Each deflection probability $P_d(i)$ is now a function of the state probabilities $P'(1), \dots, P'(D)$. For instance, if MINDIST criteria is used, $P_d(i)$ can be computed as in the previous section, but now with the advancing probability $P_a(i|N) = \sum_{k=0}^N \binom{N}{k} (P'(d > i)^{N-k} P'(i)^k) / (k + 1)$ if $i < D$ and $P_a(i|N) = P'(D)^N / (N + 1)$ if $i = D$, where $P'(d > i)$ stands for the probability that a competitor packet is at distance greater than i to its destination.

Table 1. Results from theory and simulation

d	D	MINDIST		MAXDIST		Random policy	
		Theor.	Simul.	Theor.	Simul.	Theor.	Simul.
2	3	0,52	0,54	0,37	0,39	0,46	0,46
2	4	0,37	0,38	0,23	0,22	0,30	0,30
3	3	0,62	0,69	0,39	0,41	0,54	0,55
3	4	0,43	0,47	0,21	0,20	0,33	0,33
4	3	0,75	> 0,80	0,45	0,45	0,64	0,65
4	4	0,53	0,58	0,22	0,20	0,39	0,38

Even if the transition probabilities are unknown, M' is the probability matrix of an ergodic Markov chain (finite, irreducible and aperiodic). Let $\mathbf{V}' = (V'(1), \dots, V'(D))$ be the stationary distribution of M' (i.e. a probability left eigenvector associated to the eigenvalue 1), each $V'(i)$ being a function of $\mathbf{P}' = (P'(1), \dots, P'(D))$. The probabilities $P_d(i)$ can be computed then by solving the equation $\mathbf{V}' = \mathbf{P}'$. Finally, we consider again the absorbing Markov chain M and compute the expected time to absorption in state 0, as in Section 3.

5 Validation through Simulation

Table 1 gives, for the case of the d -regular Kautz network $K(d, D)$ with diameter D , a comparison of theoretical results obtained by applying our analytical model with results from simulation.

In the computation of the theoretical results, the values of $P_{in}(i)$ and $P_t(i, j)$ are those obtained in [9]. The comparison is a validation of the hypothesis and assumptions used in paper. Further details and description of the simulation can be found in [14].

References

1. Baran, P.: On distributed communications networks. *IEEE Trans. Comm. Sys.* **12** (1964) 1–9
2. Baransel, C., Dobosiewicz, W., Gburzynski, P.: Routing in multihop packet switching: Gb/s challenge. *IEEE Network* **9** (1995) 38–61
3. Bermond, J.C.; Peyrat, C.: De Bruijn and Kautz networks: A competitor for the hypercube? In: Andre, F., Verjus, J.P. (eds): *Hypercube and Distributed Computers*, North-Holland, Amsterdam (1989) 279–294.
4. Borgonovo, F., Fratta, L.: Deflection networks: Architectures for metropolitan and wide area networks. *Comput. Networks ISDN Syst.* **2** (1992) 171–183
5. Bononi, A., Pruncal, P.R.: Analytical evaluation of improved access techniques in deflection routing networks. *IEEE/ACM Trans. on Networking* **4**(5) (1996) 726–730
6. Chich, T.: Optimisation du routage par déflexion dans les réseaux de télécommunications métropolitains. Ph.D. Thesis. LRI Paris (1997)
7. Chich, T.; Cohen, J.; Fraigniaud, P.: Unslotted deflection routing: a practical and efficient protocol for multihop optical networks. *IEEE/ACM Trans. on Networking* **9** (2001) 47–59
8. Clérot., F.: Réseaux fonctionnant par déflexion: deux ans déjà... Internal Research Rapport. CNET (1996)
9. Fàbrega, J.; Muñoz, X.: How to find the maximum admissible throughput in deflection routing networks. Internal Research Rapport. MA4. UPC (2002)
10. Gary, S.-H., Kobayashi, H.: Performance analysis of Shufflenet with deflection routing. *Proceedings of IEEE Globecom 93* **2** (1993) 854–859
11. Greenberg, A.G.; Hajek, B.: Deflection routing in hypercube networks. *IEEE Trans. Comm.* **40** (1992) 1070–1081
12. Heydemann, M.-C., Meyer, J. C., Sotteau, D.: On forwarding indices of networks. *Disc. Appl. Math.* **23** (1989) 103–123
13. Homobono, N.: Résistance aux pannes des grands réseaux d'interconnexion. Ph.D. Thesis. LRI Paris (1987)
14. Ortega, D.: Simulación de encaminamientos por deflexión sobre algunas familias de digrafos. Master Thesis. UPC. Barcelona (2000)

Implementation and Performance Evaluation of M-VIA on AceNIC Gigabit Ethernet Card*

In-Su Yoon¹, Sang-Hwa Chung¹, Ben Lee², and Hyuk-Chul Kwon¹

¹ Pusan National University
School of Electrical and Computer Engineering
Pusan, 609-735, Korea

{isyoon, shchung, hckwon}@pusan.ac.kr

² Oregon State University
Electrical and Computer Engineering Department
Owen Hall 302, Corvallis, OR 97331
benl@ece.orst.edu

Abstract. This paper describes the implementation and performance of M-VIA on the AceNIC Gigabit Ethernet card. The AceNIC adapter has several notable hardware features for high-speed communication, such as jumbo frames and interrupt coalescing. The M-VIA performance characteristics were measured and evaluated based on these hardware features. Our results show that latency and bandwidth improvement can be obtained when the M-VIA data segmentation size is properly adjusted to utilize the AceNIC's jumbo frame feature. The M-VIA data segmentation size of 4,096 bytes with MTU size of 4,138 bytes showed the best performance. However, larger MTU sizes did not necessarily result in better performance due to extra segmentation and DMA setup overhead. In addition, the cost of M-VIA interrupt handling can be reduced with AceNIC's hardware interrupt coalescing. When the parameters for the hardware interrupt coalescing were properly adjusted, the latency of interrupt handling was reduced by up to 170 μ s.

1 Introduction

Gigabit Ethernet based clusters are considered as scalable, cost-effective platforms for high performance computing. However, the performance of Gigabit Ethernet has not been fully delivered to the application layer because of the TCP/IP protocol stack overhead. In order to circumvent these problems, a group of user-level communication protocols has been proposed. Examples of user-level communication protocol are U-Net [1], Fast Message [2], Active Message [3] and GAMMA [4]. The Virtual Interface Architecture (VIA) [5] has emerged to standardize these different user-level communication protocols. Since the introduction of VIA, there have been several software and hardware implementations of

* This work has been supported by Korean Science and Engineering Foundation (Contract Number: R05-2003-000-10726-0) and National Research Laboratory Program (Contract Number: M10203000028-02J0000-01510).

VIA. M-VIA (Modular VIA) [6] is a software implementation that employs Fast or Gigabit Ethernet as the underlying platform.

This paper discusses the implementation of M-VIA on the AceNIC Gigabit Ethernet card. The AceNIC Gigabit Ethernet card has several notable hardware features which are jumbo frames and interrupt coalescing. Therefore, this paper presents a study of what effects jumbo frames and interrupt coalescing features have on the performance of M-VIA.

2 M-VIA Overview

M-VIA is implemented as a user-level library and at least two loadable kernel modules for Linux. The core module (*via_ka* module) is device-independent and provides the majority of functionality needed by VIA. M-VIA device drivers implement device-specific functionality. In M-VIA device drivers, the *via_ering* module includes operations such as construction and interpretation of media-specific VIA headers and mechanisms for enabling VIA to co-exist with traditional networking protocols, i.e., TCP/IP. In this paper, we present our implementation of M-VIA on the AceNIC by developing a new AceNIC driver module (*via_acenic* module) for the M-VIA. Also, the *via_ering* module was modified to support different M-VIA segmentation sizes.

3 AceNIC Hardware Features

3.1 Jumbo Frames

Although jumbo frames are available to transfer large data, the original M-VIA segmentation size was designed to support the standard Ethernet MTU size of 1,514 bytes. When M-VIA transfers data, the *via_ering* module organizes data into pages and then each page is divided into the M-VIA segmentation size. Then, the *via_acenic* module writes the physical address, length, and other information of each data segment on the AceNIC's descriptor. Finally, each data segment is transferred to the AceNIC's buffer via DMA. Since segmentation and DMA setup require substantial amount of processing time, it is important to reduce the number of data segments.

In our implementation, the M-VIA segmentation size was adjusted to utilize AceNIC's jumbo frame feature. M-VIA segmentation size of 8,958 bytes is obtained by subtracting M-VIA data header of 42 bytes from an MTU of 9,000 bytes. With large MTU and segmentation size, the number of M-VIA packets is significantly reduced. When M-VIA segmentation size is made equal to the page size, the *via_ering* module needs to only generate one segment for each page. In this case, we can use an MTU of 4,138 bytes, which is obtained by adding a data header to M-VIA segment size of 4,096 bytes.

3.2 Interrupt Coalescing

Interrupt coalescing delays the generation of an interrupt until a number of packets arrive. The waiting time before generating interrupt is controlled by setting the AceNIC's internal timer. The internal timer starts counting clock ticks from the time when the first packet arrives. The number of coalescing clock ticks can be changed by modifying module parameters. With the hardware interrupt coalescing, the host can amortize the interrupt handling cost over a number of packets and thus save host processing cycles.

Because many Ethernet cards do not support hardware interrupt coalescing, M-VIA implements this feature in software. When M-VIA sends intermediate data segments, it does not mark the completion flags of the descriptors, which are ignored by the interrupt handler. When the completion flag of the final descriptor is marked, it indicates the completion of the interrupt coalescing. M-VIA's software interrupt coalescing conflicts with AceNIC's because an interrupt can be generated by the expired timer before M-VIA marks the completion flag on the final descriptor. Therefore, we maximized the number of coalescing clock ticks to prevent interrupts from being generated during send operations. However M-VIA does not implement the interrupt coalescing when it receives data, and instead depends entirely on the receive interrupt handler of NIC. The cost of M-VIA's receive interrupt handling is reduced using the AceNIC's interrupt coalescing.

4 Experimental Results

The performance of M-VIA was measured using 800 MHz Pentium III PCs with 256 MB SDRAM and 33 MHz/32-bit PCI bus. The AceNIC Gigabit Ethernet card used was a 3Com 3C985B-SX. The PCs were running with Linux kernel 2.4. M-VIA latency and bandwidth were measured using *vnettest* program.

4.1 M-VIA Data Segmentation Size and Hardware MTU

Figure 1 shows the performance of M-VIA with various segmentation sizes. One interesting observation is that M-VIA segmentation size of 4,096 bytes shows a sawtooth shape of the curve. This is because fewer packets are generated when the data size is multiples of the page size. For M-VIA segmentation size of 8,192 bytes, the performance is worse than that of 4,096-byte case until the data size reaches approximately 50 KB. Although AceNIC was configured to carry 8,192-byte frames, the *via_ering* module segments data by page size. Therefore, an 8,192-byte frame requires two segmentations and DMA initiation processes resulting in extra overhead. However, the receiving side can benefit from larger MTUs for bulk data due to reduced number of interrupts. The segmentation size of 8,958 bytes shows even worse performance because of extra segmentation and DMA initiation costs. For data size from 8 KB to approximately 36 KB, 8,958-byte segment size resulted in even worse performance compared to the

1,472-byte case. This is due to the fact that small frames allow packets to be sent faster to the receiving side, while large frames have to wait until they are filled up. However, for data size larger than 36 KB, 8,958-byte case performs better because small frames generate more frequent interrupts on the receiving side.

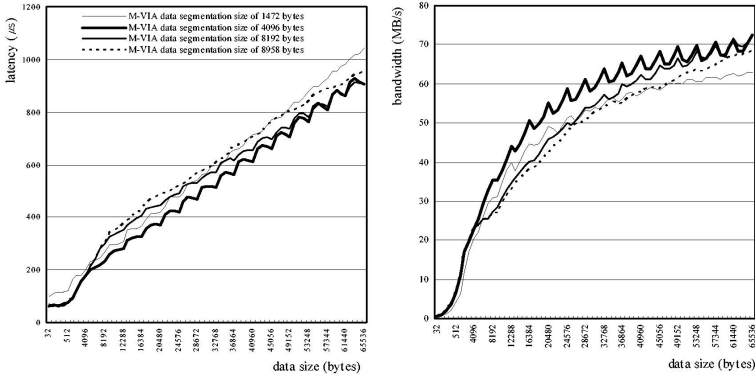


Fig. 1. Performance with various M-VIA segmentation sizes

4.2 Hardware Interrupt Coalescing Feature

The interrupt coalescing of AceNIC is controlled by a pair of parameters (*tx_coal*, *rx_coal*), which specify the number of coalescing clock ticks for transmission and reception, respectively. These parameters indicate the duration of packet send/receive before interrupting the host. Figure 2 shows the M-VIA latency for AceNIC using 1,514-byte MTU.

When the parameters are set to (1, 1), the interrupt coalescing is disabled for both transmission and reception. Thus, AceNIC invokes the interrupt handler routine as soon as a packet arrives. This leads to a minimum latency of 67 μ s for a 32-byte data, but results in significantly longer latencies for larger messages. The maximum latency difference between (1, 1) and (2000, 90) is approximately 400 μ s for 64 KB data. To evaluate the latency of the M-VIA’s software interrupt coalescing, the parameters were set to (2000, 1). The *tx_coal* value of 2,000 is sufficient for M-VIA to complete its transmit operations before the expired timer generates an interrupt. This results in significantly lower latency than disabling the interrupt coalescing. The approximate minimum and maximum latency differences are 2 μ s and 212 μ s, respectively. To confirm that the hardware interrupt coalescing for receiving data improves performance, experiments with parameters set to (2000, 90) were performed. The value 90 was determined experimentally to give the best performance. For data sizes larger than 17 KB, lower latencies were observed compared to when the parameters were set to

(2000,1). The maximum latency difference between (2000, 90) and (2000, 1) is approximately 170 μ s for sending a 64 KB data. For data sizes smaller than 17 KB, slightly higher latencies were observed because of the increased waiting time on the receiving side, but the difference was negligible.

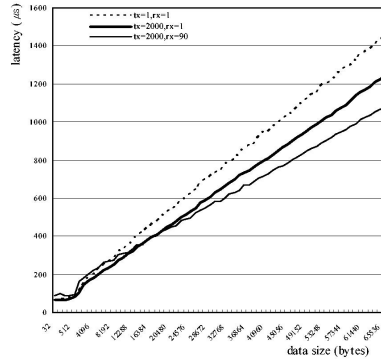


Fig. 2. M-VIA latency with hardware interrupt coalescing feature

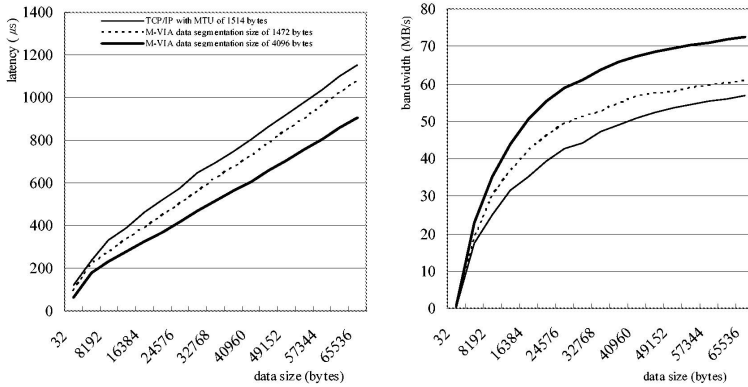


Fig. 3. M-VIA vs. TCP/IP

4.3 Comparison of M-VIA and TCP/IP

Figure 3 shows a comparison between M-VIA and TCP/IP in terms of latency and bandwidth. In this experiment, the 8,192-byte and 8,958-byte segment sizes were excluded because 4,096-byte segment size resulted in better performance. When both M-VIA and TCP/IP use the same MTU size of 1,514 bytes, M-VIA has lower latency than TCP/IP. M-VIA and TCP/IP have minimum latencies

of 89 μ s and 123 μ s respectively. The minimum latency difference is 15 μ s with 4 KB data. For data sizes larger than 16 KB, the latency difference is approximately 76 μ s. M-VIA and TCP/IP have maximum bandwidths of 60.9 MB/s and 56.9 MB/s, respectively. Comparing M-VIA using MTU size of 4,138 bytes with TCP/IP, the minimum latency difference is 57 μ s with 4 KB data and the maximum latency difference is 246 μ s with 64 KB data. M-VIA has a maximum bandwidth of 72.5 MB/s with segmentation size of 4096 bytes.

5 Conclusion

We presented our implementation and performance study of M-VIA on the AceNIC Gigabit Ethernet card by developing a new AceNIC driver for M-VIA. In particular, we focused on AceNIC's jumbo frame and interrupt coalescing features for M-VIA. We experimented with the various M-VIA data segmentation sizes and MTUs. The M-VIA data segmentation size of 4,096 bytes with MTU size of 4,138 bytes showed the best performance. Comparing M-VIA using MTU size of 4,138 bytes with TCP/IP, M-VIA latency improves by approximately 57~246 μ s and results in maximum bandwidth of 72.5 MB/s. Also the latency time of M-VIA's interrupt handling was reduced by up to 170 μ s with the AceNIC's hardware interrupt coalescing.

References

1. T. Von Eicken, A. Basu, V. Buch, and W. Vogels: "U-NET: A User Level Network Interface for Parallel and Distributed Computing", Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP), Colorado, December 1995
2. S. Pakin, M. Lauria, and A. Chien: "High Performance Messaging on Workstation: Illinois Fast Message (FM) for Myrinet", Proc. of Supercomputing '95, December
3. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer: "Active Messages: a Mechanism for Integrated Communication and Computation", 19th International Symposium on Computer Architecture, May 1992
4. G. Chiola and G. Ciaccio: "GAMMA: a Low-cost Network of Workstations Based on Active Messages", Proc. of 5th EUROMICRO workshop on Parallel and Distributed Processing, London, UK, January 1997
5. Intel, Compaq and Microsoft Corporations: Virtual Interface Architecture specification version 1.0, December 1997, <http://developer.intel.com/design/servers/vi/>
6. P. Bozeman and B. Saphir: "A Modular High Performance Implementation of the Virtual Interface Architecture", Proc. of the 2nd Extreme Linux Workshop, June 1999

Topic 15

Mobile and Ubiquitous Computing

Max Mühlhäuser, Karin A. Hummel, Azzedine Boukerche, and Alois Ferscha
Topic Chairs

The growing availability of mobile and wireless network infrastructures and, at the same time, the development of powerful and smart mobile computing devices opens up new possibilities and challenges, but also poses new issues for distributed systems and applications. One of primary issues of ubiquitous computing appears to be the integration of the potentials of mobile computing, context-awareness and seamless connectivity of services and devices — one of its primary aims is the provision of an omnipresent but calm computing landscape, empowering people with smart assistance and intelligent guidance.

We formed this workshop to reflect on the evident changes of research interests from parallel and distributed computing towards mobile computing and ubiquitous computing issues. We framed the call for papers text to solicit papers contributing to one or more of following topics: mobility, ubiquity, awareness, intelligence, and natural interaction. Mobility addresses solutions that help to make time, geographic, media and service boundaries less and less important. Ubiquity refers to a situation in which we are surrounded by a multitude of interconnected embedded systems, which are (mostly) invisible and moved into the background of our surrounding (workplace, building, home, outdoor). Awareness refers to the ability of the system to recognise and localise objects as well as people and their intentions. Intelligence refers to the ability of an environment to adapt to the people that live in it, learn from their behaviour, and possibly recognise as well as show emotion. Natural interaction finally refers to advanced modalities like natural speech- and gesture recognition, as well as speech-synthesis, which will allow a much more human-like communication with the digital environment than is possible today.

The papers selected — after a rigorous reviewing process — for publication in Topic 15 all address problems ultimately related to the design and development of ubiquitous computing systems: Related to networking, a few contributions study enhanced mobility management and QoS strategies, like improved routing protocols and mobile IP optimization and utilization. Contributions addressing embedded and context aware services introduce new models, frameworks and software architectures which incorporate sensor interfaces. Furthermore, models for context-aware behavior are presented as well as middleware adaptations due to mobile coordination patterns. Location and position tracking methods are discussed based on outdoor technologies, like mobile communication infrastructures and triangulation protocols, and indoor radio frequency technologies. Rewarding results are presented in terms of accuracy for different outdoor loca-

tion estimation methods. In order to argue the new possibilities of ubiquitous computing, various contributions present attracting real-world demonstrations and applications.

This is the first year that “mobile and ubiquitous computing” has been solicited from within Euro-Par, and the number of submissions indicate an overwhelming appreciation of the topic. In numbers, 39 papers have been submitted to the topic. Out of these contributions, 14 have been accepted as regular papers [36%] and 4 as short papers [10%] based on the results of three referees.

We owe special thanks to the Euro-Par Organizing Committee for their efforts and support, as well as to the referees providing detailed and helpful comments. We would also like to thank all the authors who have considered Euro-Par as the outlet for their valuable research work, and would at the same time like to encourage potential authors for this consideration for then next Euro-Par.

A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks^{*}

I. Chatzigiannakis^{1,2}, T. Dimitriou³, M. Mavronicolas⁴, S. Nikolettseas^{1,2}, and P. Spirakis^{1,2}

¹ Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
`{ichatz,nikole,spirakis}@cti.gr`

² Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece

³ Athens Information Technology (AIT)
`tdimitriou@ait.gr`

⁴ Department of Computer Science,
University of Cyprus, CY-1678 Nicosia, Greece
`mavronic@cs.ucy.ac.cy`

Abstract. Smart Dust is comprised of a vast number of ultra-small fully autonomous computing and communication devices, with very restricted energy and computing capabilities, that co-operate to accomplish a large sensing task. Smart Dust can be very useful in practice i.e. in the local detection of a remote crucial event and the propagation of data reporting its realization to a control center.

In this work, we have implemented and experimentally evaluated four protocols (PFR, LTP and two variations of LTP which we here introduce) for local detection and propagation in smart dust networks, under new, more general and realistic modelling assumptions. We comparatively study, by using extensive experiments, their behavior highlighting their relative advantages and disadvantages. All protocols are very successful. In the setting we considered here, PFR seems to be faster while the LTP based protocols are more energy efficient.

1 Introduction, Our Results, and Related Work

Recent dramatic developments in micro-electro-mechanical (MEMS) systems, wireless communications and digital electronics have already led to the development of small in size, low-power, low-cost sensor devices. Such extremely small devices integrate sensing, data processing and communication capabilities. Examining each such device individually might appear to have small utility, however the effective *distributed co-ordination* of large numbers of such devices may lead to the efficient accomplishment of large sensing tasks.

^{*} This work has been partially supported by the IST Programme of the European Union under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS).

Large numbers of sensor nodes can be deployed in areas of interest (such as inaccessible terrains or disaster places) and use *self-organization and collaborative methods* to form a sensor network. Their wide range of applications is based on the possible use of various sensor types (i.e. thermal, visual, seismic, acoustic, radar, magnetic, etc.) in order to monitor a wide variety of conditions (e.g. temperature, object presence and movement, humidity, pressure, noise levels etc.). Thus, sensor networks can be used for continuous sensing, event detection, location sensing as well as micro-sensing. Hence, sensor networks have important applications, including (a) military (like forces and equipment monitoring, battlefield surveillance, targeting, nuclear, biological and chemical attack detection), (b) environmental applications (such as fire detection, flood detection, precision agriculture), (c) health applications (like telemonitoring of human physiological data) and (d) home applications (e.g. smart environments and home automation). For an excellent survey of wireless sensor networks see [1] and also [6, 12].

Note however that the efficient and robust realization of such large, highly-dynamic, complex, non-conventional networking environments is *a challenging algorithmic and technological task*. Features including the huge number of sensor devices involved, the severe power, computational and memory limitations, their dense deployment and frequent failures, pose *new design and implementation aspects* which are essentially different not only with respect to distributed computing and systems approaches but also to ad-hoc networking techniques.

Contribution: We focus on an important problem under a particular model of sensor networks that we present. More specifically, continuing the research of our team (see [2], [4]), we study the problem of *local detection and propagation*, i.e. the local sensing of a crucial event and the energy and time efficient propagation of data reporting its realization to a (fixed or mobile) control center. This center may be some human authorities responsible of taking action upon the realization of the crucial event. We use the term “*sink*” for this control center. We note that the protocols we present here can also be used for the more general problem of data propagation in sensor networks (see [10]).

As opposed to [5] (where a 2-dimensional lattice deployment of particles is used) *we extend the network model* to the general case of particle deployment according to a random, uniform distribution. We study here *the more realistic case* when the control center is not a line in the plane (i.e. as in [2]) but a single point.

Under these more general and realistic (in terms of motivation by applications) modelling assumptions, we implemented and experimentally evaluated four information propagation protocols: (a) The Probabilistic Forwarding Protocol (PFR) that avoids flooding by favoring in a probabilistic way certain “close to optimal” transmissions, (b) the Local Target Protocol (LTP), where data is propagated by each time selecting the best (with respect to progress towards the sink) particle for passing information and (c) we propose two variations of LTP according to different next particle selection criteria. We note that we had to carefully design the protocols to work under the new network models.

The extensive simulations we have performed show that all protocols are very successful. In the setting we considered here, PFR seems to achieve high success rates in terms of time and hops efficiency, while the LTP based protocols manage to reduce the energy spent in the process by activating less particles.

Discussion of Selected Related Work:

A family of negotiation-based information dissemination protocols suitable for wireless sensor networks is presented in [9]. In contrast to classic flooding, in SPIN sensors negotiate with each other about the data they possess using meta-data names. These negotiations ensure that nodes only transmit data when necessary, reducing the energy consumption for useless transmissions.

A data dissemination paradigm called *directed diffusion* for sensor networks is presented in [10]. An observer requests data by sending *interests* for named data; data matching the interest is then “drawn” down towards that node by selecting a single path or through multiple paths by using a low-latency tree. [11] presents an alternative approach that constructs a greedy incremental tree that is more energy-efficient and improves path sharing.

In [8] a clustering-based protocol is given that utilizes randomized rotation of local cluster heads to evenly distribute the energy load among the sensors in the network. In [15] a new energy efficient routing protocol is introduced that does not provide periodic data monitoring (as in [8]), but instead nodes transmit data only when sudden and drastic changes are sensed by the nodes. As such, this protocol is well suited for time critical applications and compared to [8] achieves less energy consumption and response time.

We note that, as opposed to the work presented in this paper, the above research focuses on energy consumption without examining the time efficiency of their protocols. Furthermore, note that our protocols are quite general in the sense that (a) they do not assume global network topology information, (b) do not assume geolocation information (such as GPS information) and (c) use very limited control message exchanges, thus having low communication overhead.

2 The Model

Sensor networks are comprised of a vast number of ultra-small homogenous sensors, which we call “*grain*” *particles*. Each grain particle is a fully-autonomous computing and communication device, characterized mainly by its available power supply (battery) and the energy cost of computation and transmission of data. Such particles (in our model here) cannot move.

Each particle is equipped with a set of monitors (sensors) for light, pressure, humidity, temperature etc. Each particle has a *broadcast* (digital radio) *beacon mode* which can be also a directed transmission of angle α around a certain line (possibly using some special kind of antenna, see Fig. 2).

We adopt here (as a starting point) a two-dimensional (plane) framework: A *smart dust cloud* (a set of particles) is spread in an area (see Fig. 1). Note that a two-dimensional setting is also used in [8,9,10,11,15].

Definition 1. Let n be the number of smart dust particles and let d (usually measured in numbers of *particles*/ m^2) be the *density* of particles in the area. Let $\mathcal{R}$ be the maximum (beacon/laser) transmission range of each grain particle.

There is a single point in the network area, which we call the sink $\mathcal{S}$, and represents a control center where data should be propagated to. Note that, although in the basic case we assume the sink to be static, in a variation it may be allowed to move around its initial base position, to possibly get data that failed to reach it but made it close enough to it.

Furthermore, we assume that there is a set-up phase of the smart dust network, during which the smart cloud is dropped in the terrain of interest, when using special control messages (which are very short, cheap and transmitted only once) each smart dust particle is provided with the direction of $\mathcal{S}$. By assuming that each smart-dust particle has individually a *sense of direction* (e.g. through its magnetometer sensor), and using these control messages, each particle is aware of the general location of $\mathcal{S}$.

We feel that our model, although simple, depicts accurately enough the technological specifications of real smart dust systems. Similar models are being used by other researchers in order to study sensor networks (see [8,15]). In contrast to [10,13], our model is weaker in the sense that *no geolocation abilities* are assumed (e.g. a GPS device) for the smart dust particles leading to more generic and thus stronger results. In [7] a thorough comparative study and description of smart dust systems is given, from the technological point of view.

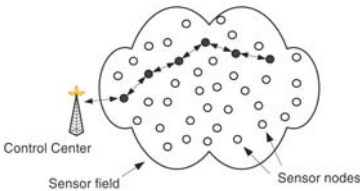


Fig. 1. A Smart Dust Cloud

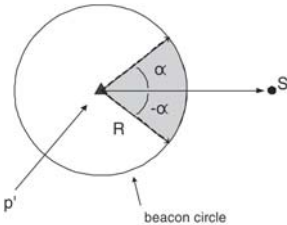


Fig. 2. Example of the Search Phase

3 The Problem

Assume that a single particle, p , senses the realization of a *local crucial event* $\mathcal{E}$. Then the *propagation problem* $\mathcal{P}$ is the following:

“How can particle p , via cooperation with the rest of the grain particles, efficiently propagate information $\text{info}(\mathcal{E})$ reporting realization of event $\mathcal{E}$ to the sink $\mathcal{S}$?”

Because of the dense deployment of particles close to each other, multi-hop communication consumes less power than a traditional single hop propagation. Also, multi-hop communication can effectively overcome some of the signal propagation effects in long-distance wireless transmissions. Furthermore, short-range hop-by-hop transmissions may help to smoothly adjust propagation around obstacles. Finally, the low energy transmission in hop-by-hop may enhance security, protecting from undesired discovery of the data propagation operation.

Because of the above reasons, data propagation should be done in a multi-hop way. To minimize the energy consumption in the sensor network we wish to *minimize the number of hops (directed transmissions)* performed in the data propagation process. Note that the number of hops also characterizes propagation time, assuming an appropriate MAC protocol [18].

Furthermore, an interesting aspect is how close to the sink data is propagated (in the case where data does not exactly reach the sink). Note that proximity to the sink might be useful in the case where the sink is mobile or it performs a “limited flooding” to get the information from the final position it arrived.

4 The Probabilistic Forwarding Protocol (PFR)

The basic idea of the protocol lies in probabilistically favoring transmissions towards the sink within a *thin zone* of particles around the line connecting the particle sensing the event $\mathcal{E}$ and the sink (see Fig. 3). Note that transmission along this line is optimal w.r.t. energy and time. However it is not always possible to achieve this optimality, basically because, even if initially this direct line was appropriately occupied by sensors, certain sensors on this line might become inactive, either permanently (because their energy has been exhausted) or temporarily (because they might enter a sleeping mode to save energy). Further reasons include (a) *physical damage* of sensors, (b) *deliberate removal* of some of them (possibly by an adversary), (c) *changes in the position* of the sensors due to a variety of reasons (weather conditions, human interaction etc). and (d) *physical obstacles* blocking communication.

The protocol evolves in two phases:

Phase 1: The “Front” Creation Phase. Because of the probabilistic decision on whether to forward messages or not, initially we build (by using flooding) a sufficiently large “front” of particles, in order to guarantee the survivability of the propagation process. During this phase,

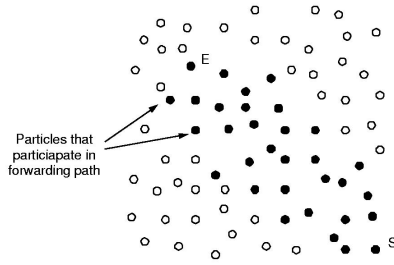


Fig. 3. Thin zone of particles around the line connecting the particle sensing the event $\mathcal{E}$ and the sink $\mathcal{S}$.

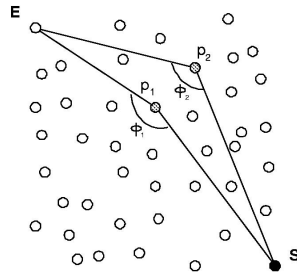


Fig. 4. Angle ϕ and closeness to optimal line

each particle having received the data, deterministically forwards them towards the sink. In particular, and for a sufficiently large number of steps, each particle broadcasts the information to all its neighbors, using a directed (towards the sink) angle transmission.

Phase 2: The Probabilistic Forwarding Phase. During this phase, each particle possessing the information under propagation, probabilistically favors its transmission within a thin zone of sensors lying close to the (optimal) line between the particle that sensed $\mathcal{E}$ and $\mathcal{S}$. In other words, data is propagated with a suitably chosen probability p , while it is not propagated with probability $1 - p$, based on a random choice. This probability is calculated as follows: Let ϕ the angle defined by the line connecting $\mathcal{E}$ and the sensor performing the random choice, and the line defined by the position of this particle and $\mathcal{S}$ (see Fig. 4). To limit the propagation zone we choose the forwarding probability $\mathbb{P}_{fwd}$ to be

$$\mathbb{P}_{fwd} = \frac{\phi}{\pi} \tag{1}$$

Remark that indeed a bigger angle ϕ suggests a sensor position closer to the direct line between $\mathcal{E}$ and $\mathcal{S}$. Clearly, when $\phi = \pi$, then the sensor lies on this line. Also note that calculations of ϕ needs only local information. Figure 4 displays this graphically.

Thus, we get that $\phi_1 > \phi_2$ implies that for the corresponding particles p_1, p_2, p_1 is closer to the $\mathcal{E}$ - $\mathcal{S}$ line than p_2 , thus

$$\mathbb{P}_{fwd}(p_1) = \frac{\phi_1}{\pi} > \frac{\phi_2}{\pi} = \mathbb{P}_{fwd}(p_2) \quad (2)$$

Remark: Certainly, there might exist other probability choices for favoring certain transmissions. We are currently working on such alternative choices.

5 The Local Target Protocol (LTP) and Its Variations

We now present a protocol for smart dust networks which we call the “Local Target” protocol and two variations that use different next particle selection criteria. In this protocol, each particle p' that has received $info(\mathcal{E})$ from p (via, possibly, other particles) does the following:

Phase 1: The Search Phase. It uses a periodic low energy broadcast of a beacon in order to discover a particle nearer to $\mathcal{S}$ than itself. Among the particles returned, p' selects a unique particle p'' that is “best” with respect to progress towards the sink. We here consider two criteria for measuring the quality of this progress and also a randomized version to get a good average mix:

- (a) *Euclidean Distance.* In this case, the particle p''_E that among all particles found achieves the bigger progress on the $p'S$ line, should be selected. We call this variation of the our protocol LTP_E . This is considered as the “basis” LTP.
- (b) *Angle Optimality.* In this case, the particle p''_A such that the angle $\widehat{p''_A p' S}$ is minimum, should be selected. We call this variation of the our protocol LTP_A .
- (c) *Randomization.* Towards a good average case performance of the protocol, we use randomization to avoid bad behavior due to the worst case input distributions for each selection (i.e. particles forming big angles with the optimal line in LTP_E and particles resulting to small Euclidean progress in LTP_A). Thus, we find p''_E, p''_A and randomly select one of them, with probability $\frac{1}{2}$. We call this protocol LTP_R .

Phase 2: The Direct Transmission Phase. Then, p' sends $info(\mathcal{E})$ to p'' and sends a *success* message to p (i.e. to the particle that it originally received the information from).

Phase 3: The Backtrack Phase. If the *search phase* fails to discover a particle nearer to $\mathcal{S}$, then p' sends a *fail* message to p .

In the above procedure, propagation of $info(\mathcal{E})$ is done in two steps; (i) particle p' locates the next particle (p'') and transmits the information and (ii) particle p' waits until the next particle (p'') succeeds in propagating the message further towards $\mathcal{S}$. This is done to speed up the backtrack phase in case p'' does not succeed in discovering a particle nearer to $\mathcal{S}$.

6 Efficiency Measures

Definition 2. Let h_A (for “active”) be the *number of “active” sensor particles* participating in the data propagation and let E_{TR} be the *total number of data transmissions* during propagation. Let T be the *total time* for the propagation process to reach its final position and H the *total number of “hops”* required.

Clearly, by minimizing h_A we succeed in avoiding flooding and thus we minimize energy consumption. Remark that in LTP we count as active those particles that transmit *info*($\mathcal{E}$) at least once.

Note that h_A , E_{TR} , T and H are random variables. Furthermore, we define the success probability of our algorithm where we call success the eventual data propagation to the sink.

Definition 3. Let P_s be the *success probability* of our protocol.

We also focus on the study of the following parameter. Suppose that the data propagation fails to reach the sink. In this case, it is very important to know “*how close*” to the sink it managed to get. Propagation reaching close to the sink might be very useful, since the sink (which can be assumed to be mobile) could itself move (possibly by performing a random walk) to the final point of propagation and get the desired data from there. Even assuming a fixed sink, closeness to it is important, since the sink might in this case begin some “limited” flooding to get to where data propagation stopped. Clearly, the closer to the sink we get, the cheaper this flooding becomes.

Definition 4. Let $\mathcal{F}$ be the final position of the data propagation process. Let $\mathcal{D}$ be $\mathcal{F}$ ’s (Euclidean) distance from the sink $\mathcal{S}$.

Clearly in the case of total success $\mathcal{F}$ coincides with the sink and $\mathcal{D} = 0$.

7 Experimental Evaluation

We evaluate the performance of the four protocols by a comparative experimental study. The protocols have been implemented as C++ classes using the data types for two-dimensional geometry of LEDA [16] based on the environment developed in [2], [4]. Each class is installed in an environment that generates sensor fields given some parameters (such as the area of the field, the distribution function used to drop the particles), and performs a network simulation for a given number of repetitions, a fixed number of particles and certain protocol parameters. After the execution, the environment stores the results in files so that the measurements can be represented in a graphical way.

In the full paper ([3]), we provide for all protocols a more detailed description of the implementation, including message structures, data structures at a particle, initialization issues and pseudo-code for the protocol.

In our experiments, we generate a variety of sensor fields in a 100m by 100m square. In these fields, we drop $n \in [100, 3000]$ particles randomly uniformly

distributed on the smart-dust plane, i.e. for densities $0.01 \leq d \leq 0.3$. Each smart dust particle has a fixed radio range of $\mathcal{R} = 5m$ and $\alpha = 90^\circ$. The particle p that initially senses the crucial event is always explicitly positioned at $(x, y) = (0, 0)$ and the sink is located at $(x, y) = (100, 100)$. Note that this experimental setup is based on and extends that used in [8,11,15]. We repeated each experiment for more than 5,000 times in order to achieve good average results.

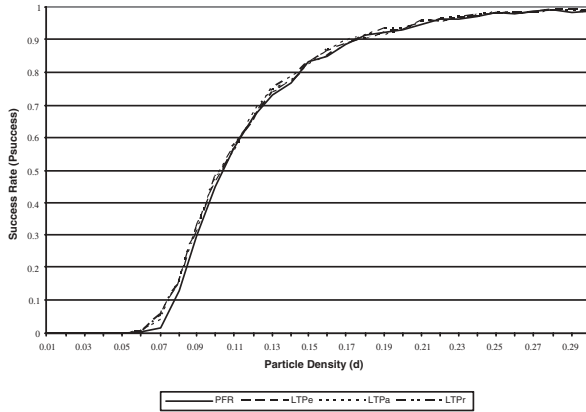


Fig. 5. Success Probability ($\mathbb{P}_s$) over particle density $d = [0.01, 0.3]$.

We start by examining the success rate of the four protocols (see Fig. 5), for different particle densities. Initially, when the density is low (i.e. $d \leq 0.06$), the protocols fail to propagate the data to the sink. However as the density increases, the success rate increases quite fast and for high densities, all four protocols almost always succeed in propagating the data to the sink. Thus, all protocols are very successful. We remark a similar shape of the success rate function in terms of density. This is due to the fact that all protocols use local information to decide how to proceed by basically selecting (all protocols) the next particle with respect to a similar criterion (best progress towards the sink).

In the case when the protocols fail to propagate the data to the sink, we examine “how close” to the sink they managed to get. Figure 6 depicts the distance of the final point of propagation to the position of the sink. Note that this figure should be considered in conjunction with Fig. 5 on the success rate. Indeed, failures to reach the sink are very rare and seem to appear in very extreme network topologies due to bad particle distribution on the network area.

Figure 7 depicts the ratio of active particles over the total number of particles ($r = \frac{h_A}{n}$) that make up the sensor network. In this figure we clearly see that PFR, for low densities (i.e. $d \leq 0.07$), it indeed activates a small number of particles (i.e. $r \leq 0.3$) while the ratio (r) increases as the density of the particles increases.

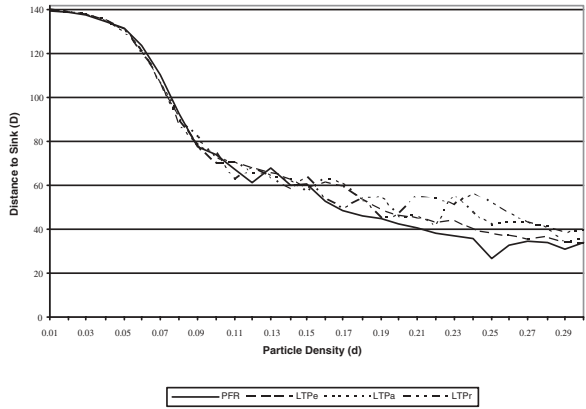


Fig. 6. Average Distance from the Sink ($\mathcal{D}$) over particle density $d = [0.01, 0.3]$.

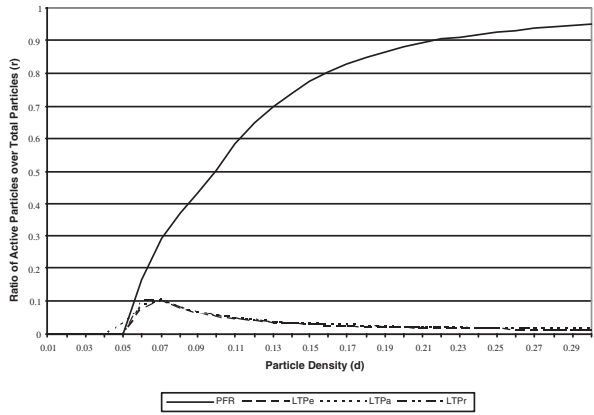


Fig. 7. Ratio of Active Particles over Total Particles (r) over particle density $d = [0.01, 0.3]$.

The LTP based protocols do not behave in the same way, since the ratio (r) of active particles seems to be independent to the total number of particles.

Remark: Because of the way PFR attempts to avoid flooding (by using angle ϕ to capture “distance” from optimality) its merits are not sufficiently shown in the setting considered here. We expect PFR to behave significantly better with respect to energy in much larger networks and in cases where the event is

sensed in an average place of the network. Also, stronger probabilistic choices (i.e. $\mathbb{P}_{fwd} = \left(\frac{\phi}{\pi}\right)^a$, where $a > 1$ a constant) may further limit propagation time.

Furthermore, examining the total number of transmissions performed by the particles (see Fig. 8), it is evident that because the LTP based protocols activate a small number of particles, the overall transmissions are kept low. This is a surprising result, since the PFR protocol was originally designed to work without the need of any control messages so that the energy consumption is low. However, the comparative study clearly shows that avoiding the use of control messages does not achieve the expected results. So, even though all four protocols succeed in propagating the data, it is evident that the LTP based protocols are more energy efficient in the sense that less particles are involved in the process.

We continue with the following two parameters: (a) the “hops” efficiency and (b) the time efficiency, measured in terms of rounds needed to reach the sink. As can be seen in Fig. 9, all protocols are very efficient, in the sense that the number of hops required to get to the sink tends below 40 even for densities $d \geq 0.17$. The value 40 in our setting is close to optimality since in an ideal placement, the diagonal line is of length $100\sqrt{2}$ and since the transmission range $\mathcal{R} = 5$ the optimal number of hops (in an ideal case) is roughly 29. In particular PFR achieves this for very low densities ($d \geq 0.07$). On the other hand, the LTP based protocols exhibit a certain pathological behavior for low densities (i.e. $d \leq 0.12$) due to a high number of executions of the backtrack mechanism in the attempt to find a particle closer to the sink (see also Fig. 10).

As far as time is concerned, in the full paper [3] we demonstrate (see Fig. 10 there) that time and number of hops are of the same order and exhibit a very similar behavior.

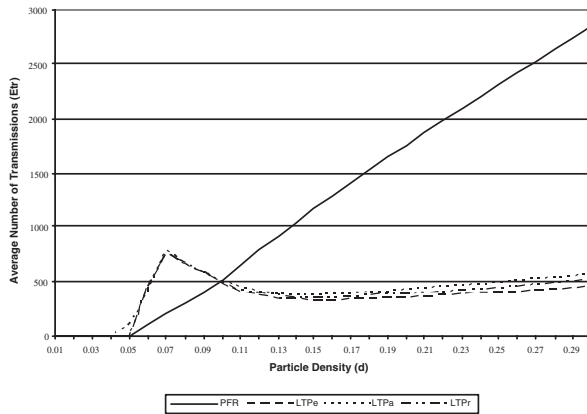


Fig. 8. Average Number of Transmissions (E_{TR}) over particle density $d = [0.01, 0.3]$.

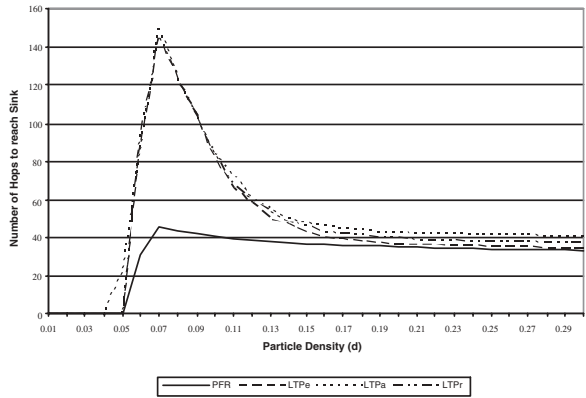


Fig. 9. Average Number of Hops to reach the sink (H) over particle density $d = [0.01, 0.3]$.

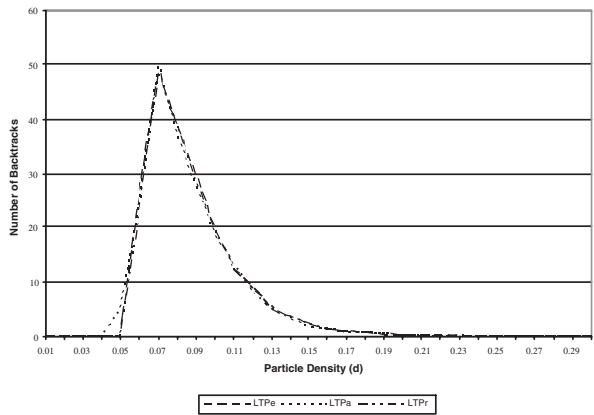


Fig. 10. Average Number of Backtracks over particle density $d = [0.01, 0.3]$.

Finally, in Fig. 10 we compare the three LTP based protocols and the number of backtracks invoked in the the data propagation. It is evident that for very low particle densities (i.e. $d \leq 0.12$), all three protocols perform a large number of backtracks in order to find a valid path towards the sink. As the particle density increases, the number of backtrack reduces fast enough and almost reaches zero.

8 Future Work

We plan to also investigate alternative probabilistic choices for favoring certain data transmissions for the PFR protocol and consider alternative backtrack mechanisms for the LTP protocol. Also, we wish to study different network shapes, various distributions used to drop the sensors in the area of interest and the fault-tolerance of the protocols. Finally, we plan to provide performance comparisons with other protocols mentioned in the related work section, as well as implement and evaluate hybrid approaches that combine the PFR and LTP protocols in a parameterized way.

References

1. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci: Wireless sensor networks: a survey. In the Journal of Computer Networks, Volume 38, pp. 393–422, 2002.
2. I. Chatzigiannakis, S. Nikolettseas and P. Spirakis: Smart Dust Protocols for Local Detection and Propagation. In *Proc. 2nd ACM Workshop on Principles of Mobile Computing* – POMC’2002.
3. I. Chatzigiannakis, T. Dimitriou, M. Mavronicolas S. Nikolettseas and P. Spirakis: A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks. FLAGS, TR, <http://ru1.cti.gr/FLAGS>.
4. I. Chatzigiannakis and S. Nikolettseas: A Sleep-Awake Protocol for Information Propagation in Smart Dust Networks. In *Proc. 3rd Workshop on Mobile and Ad-Hoc Networks (WMAN)–IPDPS Workshops*, IEEE Press, p. 225, 2003.
5. I. Chatzigiannakis, T. Dimitriou, S. Nikolettseas and P. Spirakis: A Probabilistic Forwarding Protocol for Efficient Data Propagation in Sensor Networks. FLAGS Technical Report, FLAGS-TR-14, 2003.
6. D. Estrin, R. Govindan, J. Heidemann and S. Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing* – MOBICOM’1999.
7. S.E.A. Hollar: COTS Dust. Msc. Thesis in Engineering-Mechanical Engineering, University of California, Berkeley, USA, 2000.
8. W. R. Heinzelman, A. Chandrakasan and H. Balakrishnan: Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proc. 33rd Hawaii International Conference on System Sciences* – HICSS’2000.
9. W. R. Heinzelman, J. Kulik and H. Balakrishnan: Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing* – MOBICOM’1999.
10. C. Intanagonwiwat, R. Govindan and D. Estrin: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proc. 6th ACM/IEEE International Conference on Mobile Computing* – MOBICOM’2000.
11. C. Intanagonwiwat, D. Estrin, R. Govindan and J. Heidemann: Impact of Network Density on Data Aggregation in Wireless Sensor Networks. Technical Report 01-750, University of Southern California Computer Science Department, November, 2001.
12. J.M. Kahn, R.H. Katz and K.S.J. Pister: Next Century Challenges: Mobile Networking for Smart Dust. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing*, pp. 271–278, September 1999.

13. B. Karp: Geographic Routing for Wireless Networks. Ph.D. Dissertation, Harvard University, Cambridge, USA, 2000.
14. μ -Adaptive Multi-domain Power aware Sensors: <http://www-mtl.mit.edu/research/icsystems/uamps>, April, 2001.
15. A. Manjeshwar and D.P. Agrawal: TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks. In *Proc. 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, satellite workshop of *16th Annual International Parallel & Distributed Processing Symposium* – IPDPS'02.
16. K. Mehlhorn and S. Näher: LEDA: A Platform for Combinatorial and Geometric Computing. *Cambridge University Press*, 1999.
17. TinyOS: A Component-based OS for the Network Sensor Regime. <http://webs.cs.berkeley.edu/tos/>, October, 2002.
18. W. Ye, J. Heidemann and D. Estrin: An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proc. 12th IEEE International Conference on Computer Networks* – INFOCOM'2002.
19. Wireless Integrated Sensor Networks: <http://www.janet.ucla.edu/WINS/>, April, 2001.

Network Based Mobile Station Positioning in Metropolitan Area^{*}

Karl R.P.H. Leung¹, Joseph Kee-Yin Ng², Tim K.T. Chan²,
Kenneth M.K. Chu², and Chun Hung Li²

¹ Department of Information & Communications Technology,
Hong Kong Institute of Vocational Education (Tsing Yi), Hong Kong
`kleung@computer.org`

² Department of Computer Science,
Hong Kong Baptist University, Hong Kong
`{jng, timchan, mkchu, chli}@comp.hkbu.edu.hk`

Abstract. Recently, mobile station positioning is drawing considerable attention in the field of wireless communications. Many location estimation techniques have been proposed. Although location estimation algorithms based on received signal strength technique may not be the most promising approach for providing location services, signal strength is the only common attribute available among various kind of mobile network. In this paper, we report our study of a Crude Estimation Method (CEM) which estimate the location of a mobile station based on the ratio of the signal strengths received from different base transceiver stations. We conducted series of field tests with the networks of two major mobile phone operators of Hong Kong. Among 6120 real world readings obtained from these field tests, the average errors of CEM is 49.03 meter with a variance of 538.14. When comparing the results of CEM with the Center of Gravity Method (CGM), another mobile location estimation also based on signal strength ratio, CEM has an improvement of 33.85% with about the same variance. These results are very encouraging in the study of mobile network based mobile station positioning in metropolitan areas.

1 Introduction

Positioning is the base technology for most location-based applications such as fleet management. Although Global Positioning System (GPS) is a common method used for estimating locations, using GPS in metropolitan areas, like Hong Kong, is not an effective method. It is because satellite signals are often reflected, deflected and blocked by skyscrapers and cause many blind spots when using GPS in metropolitan areas.

The U.S. Federal Communication Commission (FCC) requires all cellular operators must be able to estimate the mobile device locations with an accuracy of

^{*} The paper is partially supported by the Hong Kong SAR Government Innovation Technology Fund – ITSP under ITS/22/02.

100 meters in 67 percent of the time for the network-based solution when making an emergency call [1]. CDMA mobile phone network architecture is being used in the United States. Hence many methods which make use of the features of the existing CDMA network are studied to perform location estimation. These methods have been utilizing the Time-Of-Arrival (TOA), Angle of Arrival (AOA), Time- Different of Arrival (TDOA) and Signal Strength, for estimating the cellular phone location. However, in European countries and also regions like Hong Kong, most of mobile phone service providers are using Global System of Mobile Network Architecture (GSM), instead of CDMA. Most of the features being used for location estimation in CDMA are either not available or with much poor precision.

Using the GSM network for positioning seems to be impractical in many countries. Because of limited information about the TOA, AOA or TDOA, signal strengths received from the serving base station and its neighboring base stations seem to be the only information for location estimation. However, this method has to deal with the large and random deviations of the receiving signals [2]. Second, due to the high penetration power of GSM, in many sparsely populated European cities, only 2 to 3 towers of macro cell sites is sufficient for providing mobile phone services for a huge area. This highly centralized network topology causes more difficulties for location estimation because of too few distinct references. However, cities like Hong Kong and similar metropolitan areas are densely populated with skyscrapers. Hence more cell sites are needed to provide a good telecommunication coverage than in sparsely populated areas.

Willassen and Andresen proposed a method for estimating MS location based on GSM network [3]. By mapping signal strength received by the MS to the actual distance, triangulation method was applied to estimate the location of the MS. This method was studied intensively by simulation. However, the method had not been tested in a real GSM network.

The HATA formula [4] proposed by Okumura for mapping the signal strength to the actual distance is one the most widely used propagation-path models. The model is based on the empirical measurements of radii 1-10 km in Tokyo. However, variations of terrain factors, transmission frequencies, and cell types of the mobile stations are necessary factors to be considered in this formula. These factors make the calculation complicated, are location dependent, and impose extra maintenance cost to the network operators.

Although location estimation algorithms based on signal attenuation may not be the most promising approach for providing location services, signal strength is the only common attribute available among various kind of mobile network. Furthermore, the cell layout in metropolitan areas like Hong Kong is very different. It is our hypothesis that in city areas like Hong Kong, there are methods which can estimate MS locations by signal strengths as accurate as those methods used in CDMA network. A Center of Gravity Method (CGM) has proposed for estimating GSM mobile station locations [5] which come up with encouraging results.

In the paper, we report our Crude Estimation Method (CEM). This is a mobile network based mobile station positioning method which makes use of

the received signal strengths received by the mobile station. We also give a brief summary of the Weighted Center of Gravity Method in Section 2. This is the method which we are going to compare with extensively. Then the Crude Estimation Method is described in Section 3. The design of field tests is discussed in Section 4 which is followed by the experiment and discussion of experimental results in Section 5. Finally, we conclude our work in Section 6.

2 Weighted Center of Gravity Method

A Center of Gravity Method (CGM) is proposed for estimating locations of GSM handsets [5]. Theoretically, the relationship between the distance and the transmitted signal strength is based on the inverse square law. Thus, the weighting assignment function should be $1/x^2$. However, the terrain and the interference distort the law with high density of buildings in the area like Hong Kong. So the weighting function becomes $1/x^\alpha$. With the concept of Center of Gravity and the weighting function, the algorithm of Weighted CG is constructed for the location estimation. By using the reciprocal function of the signal strengths as a weighting, the location estimation formula can be defined as follows:

Let x, y be the xy -coordinate of the mobile phone, and $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ the actual position(s) of the BTSs¹. With n being the number of reference points, which should be less than the number of received cell sites. $dB_1, dB_2, \dots, dB_n$ represent the corresponding signal strength for each BTS. Thus, the location estimation formula for x , and y are:

$$x = \frac{x_1/dB_1^\alpha + x_2/dB_2^\alpha + x_3/dB_3^\alpha + \dots + x_n/dB_n^\alpha}{1/dB_1^\alpha + 1/dB_2^\alpha + 1/dB_3^\alpha + \dots + 1/dB_n^\alpha} \quad (1)$$

$$y = \frac{y_1/dB_1^\alpha + y_2/dB_2^\alpha + y_3/dB_3^\alpha + \dots + y_n/dB_n^\alpha}{1/dB_1^\alpha + 1/dB_2^\alpha + 1/dB_3^\alpha + \dots + 1/dB_n^\alpha} \quad (2)$$

where $3 \leq n \leq 9$

While 3 is the minimum number of channels for the algorithm to work, and 9 is the maximum number of channels a Nokia handset² can provide. The major disadvantage of CGM is, due to the property of center of gravity, the estimated MS location can only be inside the convex hull of the BTSs. This is a strong restriction in location estimation. The Crude Estimation Method (CEM) that we are going to propose, is an investigation on a location estimation method based on signal strength and is not subject to this restriction.

3 Crude Estimation Method

The *Crude Estimation Method* (CEM) proposed in this paper, similar to the CGM, makes use of the signal strengths received by all the Base Transceivers

¹ The BTS can be uniquely identified by the Broadcast Channel (BCCH) and the BSIC from the operator.

² Although all our proposed methods are handset independent, all our experiments and measurements are based on the Nokia handset (Model 6150).



Fig. 1. Locus Method

Stations (BTSS) when a call is made. This method consists of two phases, *Crude Estimation* phase and *Location Recommendation* phase. In the Crude Estimation phase, the *Crude Estimation Algorithm (CEA)* is applied to estimate a location from every combination of two signal strengths received by the MS. This will generate a bag of possible locations. Then in the Location Recommendation phase, by manipulating the bag of possible locations obtained in Crude Estimation, a point is recommended as the estimated location. These processes are visualized in Figure 1.

3.1 Crude Estimation Algorithm

Let d be the distance between a MS and a BTS. Let s be the signal loss in the communication between the BTS and MS. Our CEA assumes that the relation $s \propto 1/d^e$, for some e , holds.

The intuition of CEM is assuming that the location of a MS lies on the locus which satisfy the ratio of the signal strength in the communication with the two BTSS. With the geographical location of the two BTSS and the relation between signal strength and distance, a locus of possible locations of the MS can be constructed. We choose the turning points of the locus as the crude estimated locations. This crude estimation approach caters for the possibilities that the estimated locations are not lying on the straight line between the two BTSS. Hence the estimated locations can be outside the convex hull of the BTSSs. The derivation of the CEM is shown as follows.

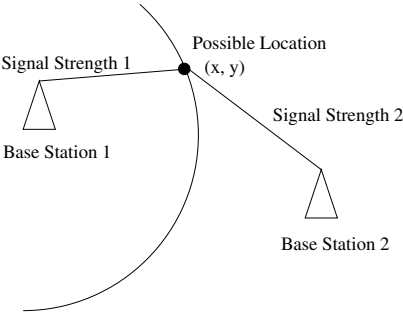


Fig. 2. Locus formed by the ratio of signal strengths from two Base Stations

Let the location of two BTSS be (α_1, β_1) and (α_2, β_2) . Let the signal strengths from the two BTSS to the MS be s_1 and s_2 . Then we have

$$s_1 \propto \frac{1}{d_1^e} \quad (3) \quad s_2 \propto \frac{1}{d_2^e} \quad (4)$$

Let (x, y) be the location of MS. The relations of the distances between MS's location and the two BTSs are,

$$(x - \alpha_1)^2 + (y - \beta_1)^2 = d_1^2 \quad (5) \quad (x - \alpha_2)^2 + (y - \beta_2)^2 = d_2^2 \quad (6)$$

By dividing Equation 5 by Equation 6, we have Equation 7, which is the ratio of the distances between the MS and the two BTSs.

By substituting Equation 3, and Equation 4 into Equation 7, for some k we have Equation 8.

$$\frac{(x - \alpha_1)^2 + (y - \beta_1)^2}{(x - \alpha_2)^2 + (y - \beta_2)^2} = \frac{d_1^2}{d_2^2} \quad (7) \quad \frac{(x - \alpha_1)^2 + (y - \beta_1)^2}{(x - \alpha_2)^2 + (y - \beta_2)^2} = k \left\{ \frac{s_2}{s_1} \right\}^{\frac{2}{e}} \quad (8)$$

Let $Q = k \left\{ \frac{s_2}{s_1} \right\}^{\frac{2}{e}}$ then Equation 8 becomes

$$\begin{aligned} (x - \alpha_1)^2 + (y - \beta_1)^2 &= Q((x - \alpha_2)^2 + (y - \beta_2)^2) \quad (9) \\ \Rightarrow x^2 - 2\alpha_1 x + \alpha_1^2 + y^2 - 2\beta_1 y + \beta_1^2 &= Q(x^2 - 2\alpha_2 x + \alpha_2^2 + y^2 - 2\beta_2 y + \beta_2^2) \quad (10) \\ \Rightarrow (1 - Q)x^2 - 2(\alpha_1 - Q\alpha_2)x + (1 - Q)y^2 - 2(\beta_1 - Q\beta_2)y + \alpha_1^2 + \beta_1^2 - \\ &\quad Q(\alpha_2^2 + \beta_2^2) = 0 \quad (11) \end{aligned}$$

Let $A = (1 - Q)$, $B = \alpha_1 - Q\alpha_2$, $C = \beta_1 - Q\beta_2$, $D = \alpha_1^2 + \beta_1^2 - Q(\alpha_2^2 + \beta_2^2)$. Then Equation 11 becomes Equation 12. By differentiating Equation 12 once, we have Equation 13

$$Ax^2 - 2Bx + Ay^2 - 2Cy + D = 0 \quad (12) \quad \frac{dy}{dx} = \frac{B - Ax}{Ay - C} \quad (13)$$

By putting Equation 13 equal to zero, we have

$$x = \frac{B}{A} \text{ and } y = \frac{C \pm \sqrt{C^2 - B^2 - AD}}{A}$$

Then when $C^2 - B^2 - AD \geq 0$, we can obtain one or two estimated locations. These locations are the crude estimated locations.

3.2 Location Recommendation

We collect all the estimated locations by CEA for every combination of two received signal strengths. This form a bag of estimated locations. In this location recommendation phase, one recommended location is generated from this bag of estimated locations.

We assumed that the set of result points is in a normal distribution. Thus, statistical approach is used in the recommendation process. We first remove the outlier estimated locations which are not within the standard deviations of the bag of estimated locations. Then we pick the center of gravity of the remained estimated locations as the recommended estimated location. Hence the recommended estimated location is calculated as follows.

Let N be the number of estimated locations obtained by CEA for every combination of two received signal strengths, and let the i -th location be (x_i, y_i) .

So, the mean and variance of the location (x_μ, y_μ) and (x_σ, y_σ) are,

$$x_\mu = \frac{\sum_{i=1}^N x_i}{N}, y_\mu = \frac{\sum_{i=1}^N y_i}{N}, x_\sigma = \sqrt{\frac{\sum_{i=1}^N (x_i - x_\mu)^2}{N - 1}}, y_\sigma = \sqrt{\frac{\sum_{i=1}^N (y_i - y_\mu)^2}{N - 1}}$$

Thus, the set of clustered result points S is

$$S ::= \{i : 1..N \mid (x_i \leq (x_\mu \pm x_\sigma) \wedge y_i \leq (y_\mu \pm y_\sigma)) \bullet (x_i, y_i)\}$$

The recommended point (x_r, y_r) is

$$(x_r = \frac{\sum_{i=1}^D x_i}{D}, y_r = \frac{\sum_{i=1}^D y_i}{D})$$

where, D is the cardinality of S and $(x_i, y_i) \in S$.

4 Field Test

We obtained supports from two major mobile operators of Hong Kong in conducting this research. We call them Operator A and Operator B³. We conducted three field tests with two different mobile phone networks in two different areas in Hong Kong. With Operator A, we conducted two field tests. One of these field tests was conducted in the Prince Edward district and the other was conducted in the Mong Kok district. Both districts are urban areas where Mong Kok is more crowded and consists of building taller than Price Edward on the average. We could only conduct field tests in Mong Kok area with Operator B.

In our field tests, we first planned some check points in the districts for our study. We call these points *Marker Location*. Then we get the geo-location of the active BTSs in the districts on the days of field tests. The networks of Operator A and Operator B are very different from each other. For Operator A, in the Prince Edward district, there were six (6) macro cells and forty-four (44) micro cells. The average distance among all these fifty BTSs was 804.87 meter. In the Mong Kok district, Operator A had fifty-four (54) micro cells and no macro cell. The average distance among all these fifty-four BTSs was 510.18 meter. On the other hand, Operator B had eighteen (18) macro cells and only four (4) micro cells in the Mong Kok district. The average distance among all these twenty-two BTSs was 889.59 meter.

In our study, we collect the signal strengths from the MS. Nokia Model 6150 handsets were used in the field tests. A maximum of nine signal strengths together with their corresponding BTSs can be collected from the Net Monitor of

³ The mobile phone operators find the network information are sensitive to their business. Hence we have to hide their names and network information according to our Non-disclosure agreements.

the handset. We collected 120 readings at each marker location. As the hardware restriction that the interval between each successive shots was 1 second, it took two minutes to collect all 120 readings. We had chosen fifteen markers in the Prince Edward district and eighteen markers in the Mong Kok district. The marker locations of these two districts are shown in Figure 3. Hence we collected 1800 readings at the Prince Edward district and 2160 readings from each of the operators in the Mong Kok district. In total, we have 6120 sets of real world data for our experiment.

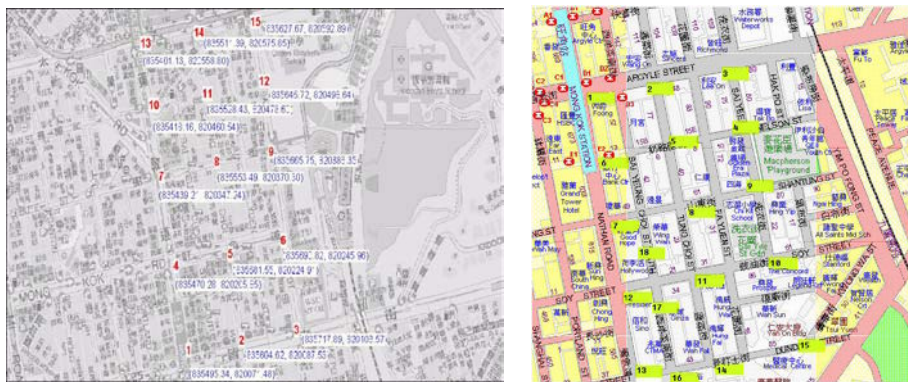


Fig. 3. Marker locations in Prince Edward (Left) & Mongkok (Right)

5 Experiment

5.1 Determining Value of e

In our experiment, we first validate our assumption that $s \propto 1/d^e$ for some e and also determine the value of e . We validate the relation and determine the value of e by plotting graphs of signal strength against distance with the data we collected. Since the networks of the two operators were different, we plot graphs for each of the operator. Furthermore, we eliminate some noises by averaging the signal strengths of each of the marker locations. The graphs of the signal strength from operator A and operator B are shown in Figure 4 and Figure 4, respectively.

The curve obtained from Figure 4 is $y = 125.02x^{-0.2122}$ and the curve obtained from Figure 4 is $y = 129.8x^{-0.2056}$. Hence, we can draw the following conclusions: our assumption that $s \propto 1/d^e$ is valid; and the value of e is around 0.2.

5.2 Experiment Results

Assuming that k in Equation 8 is related to the signal strengths in the communication, k is a function of the signal strengths received by the two BTSSs. CEM assumes that the MS lies on the locus which is the ratio of the two received signal strengths. We experimented with different relations between the two received

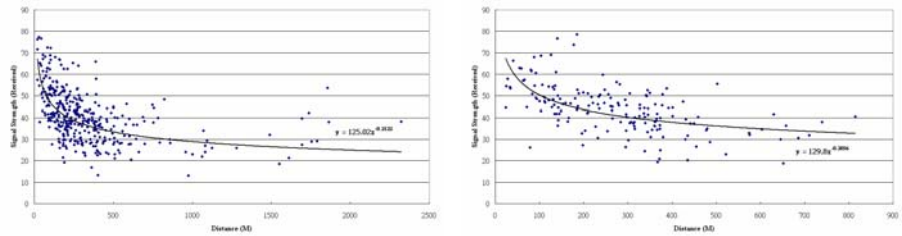


Fig. 4. Signal Strength vs Distance of Operator A (Left) & Operator B (Right)

signal strengths and find out that k yields good results when it is expressed as follows:

$$k = \frac{\sqrt{s_1}}{\sqrt{s_1} + \sqrt{s_2}} \quad \text{where } s_1 > s_2.$$

CEM is applied to the data collected from the field tests with the e and k . For each reading, the MS could receive signals from up to nine BTSs. For the signals from every combination of two BTSs received, we applied the CEM once. Hence we had at most $9C_2 \times 2$ estimated locations for each reading. Since we took 120 readings at each marker location, we then have at most $9C_2 \times 2 \times 120$ estimated locations for each marker location. We studied the average error of all these estimated locations, their variance and their best and worst cases as well. Afterwards, we looked at the overall average, variance, best and worst cases. We also compare CEM results of those obtained from CGM by using the same set of data. These results and comparisons are shown in Table 3 and Table 2.

5.3 Discussion of Experimental Results

A summary of the overall results of Table 3, Table 2, is shown in Table 1. The overall average errors of CEM in all tests are better than CGM by 33.85% which varies from 12.45% to 67.5%. The overall variance of CEM is slightly better than CGM by 5.05%. In analyzing the variance of CEM and CGM in the three tests, there are much less fluctuation of CEM than CGM in the Prince Edward district. However, the fluctuation of CEM is higher than CGM in the Mong Kok district. With these analysis, we can conclude that CEM can provide better location estimation than CGM on the average. However, the fluctuation of accuracy of CEM may not be better than CGM.

6 Conclusions and Future Work

We proposed a Crude Estimation Method for mobile station location estimation. This method is based on the signal strengths received by the mobile station and the geo-location of the base transceiver stations. We conducted field tests with two different real world GSM mobile phone network in Hong Kong. From the

Table 1. Comparison of CEM & CGM

	Average	Variance	Best	Worst
Prince Edward (average BTSs distance: 804.87m)				
CEM	51.51827944	791.0404695	7.532442214	126.1238434
CGM	86.2916469	991.52	5.73326	464.98298
diff	-67.5	-25.344	23.886	-268.672
Mong Kok - Operator A (average BTSs distance: 510.18m)				
CEM	42.26	365.21	0.66	282.52
CGM	50.64	297.88	0.57	193.85
diff	-19.83	18.44	13.57	31.38
Mong Kok - Operator B (average BTSs distance: 889.59m)				
CEM	53.32	458.18	1.76	208
CGM	59.96	406.61	1.3	282.65
diff	-12.45	11.26	26.26	-35.89
Overall results				
CEM	49.03	538.14	0.66	282.52
CGM	65.63	565.34	0.57	464.98
diff	-33.85%	-5.05%	13.57%	-64.59%

6120 sets of readings, the average error of CEM is 49.03 meters with a variance of 538.14. In comparing these results with other similar methods including the Weighted Center of Gravity Method (CGM), another method proposed by our research team, CEM is a promising method. Furthermore, CEM does not have the disadvantage that the MS must lie inside the convex hull of the base transceiver stations. We have experimented the CEM method with two real world networks in two metropolitan areas in Hong Kong. Our next step is to study the theoretical aspects of this method. These include finding the intrinsic contribute to the value of e and the function k , and investigating the theoretical aspects of this method.

References

1. Federal Communications Commission: Revision to the commission's rules to ensure compatibility with enhanced 911 emergency calling system. Technical Report CC Docket No. 94-102 (1996)
2. Geoffery G. Messier, Michel Fattouche and Brent R. Petersen: Locating an IS-95 mobile using its signal. In: The Tenth International Conference on Wireless Communications (Wireless 98). Volume II., Calgary, AB, Canada (1998) 562-574 sponsored by TRLabs, the Communications Research Centre and IEEE Canada.
3. Willassen, S.Y.: A method of implementing mobile station location in gsm (1998) URL:<http://www.willassen.no/msl/bakgrunn.html>.
4. Y. Okumura, et.al.: Field strength and its variability in vhf and uhf and mobile service. Technical report, Rev. Elec. Comm. Lab. (1968)
5. Joseph K. Ng, Stephen K. Chan, and Kenny K. Kan: Location Estimation Algorithms for Providing Location Services within a Metropolitan Area based on a Mobile Phone Network. In: Proceedings of the 5th International Workshop on Mobility Databases and Distributed Systems (MDDS 2002), Aix-en-Provence, France (2002) 710-715

A Test Results

Table 2. Results of CEM and CGM in Mong Kok

Marker	CEM (Op. A)		CGM (Op. A)		CEM (Op. B)		CGM (Op. B)	
	Average	Variance	Average	Variance	Average	Variance	Average	Variance
1	17.26	10.13	36.58	46.86	34.32	66.95	14.39	52.81
2	29.79	341.25	15.5	85.93	40.87	38.01	31.44	74.18
3	42.37	347.42	35.05	236.43	64.08	270.38	52.37	179.74
4	46.99	256.09	31.89	84.3	84.53	27.85	42.91	210.57
5	16.27	100.99	15.81	95.24	23.11	6.93	14.14	97.8
6	26.51	89.51	26.14	142.99	60.67	565.82	31.13	365.03
7	19.76	11.86	25.26	102.57	71.31	919.58	50.57	193.87
8	42.12	191.95	42.09	84.9	37.41	164.75	49.66	212.95
9	83.31	634.0	48.49	152.55	82.68	1046.13	44.15	473.96
10	42.36	661.31	155.39	124.59	77.83	3146.26	68.23	702.45
11	66.78	339.72	93.06	1068.56	58.52	361.19	59.24	152.24
12	23.23	174.03	84.84	946.42	31.86	11.27	50.63	549.46
13	46.76	110.01	53.37	232.45	23.96	268.76	63.76	1314.5
14	29.85	76.78	83.06	62.33	54.84	232.18	112.6	485.28
15	134.2	2651.7	50.61	1488.21	98.14	529.16	173.05	1633.21
16	44.14	311.29	29.83	258.22	69.01	199.59	101.44	260.23
17	34.38	126.31	64.09	127.73	25.65	229.37	75.12	284.25
18	14.05	139.36	18.51	21.56	21.07	163.1	44.62	76.45
OverAll	42.26	365.21	50.64	297.88	53.32	458.18	59.96	406.61

Table 3. Results of CEM and CGM in Prince Edward (Operator A)

Marker	CEM				CGM			
	Average	Variance	Best	Worst	Average	Variance	Best	Worst
1	43.96	94.54	8.31	55.39	78.94	285.31	20.7	130.15
2	57.75	351.63	15.3	76.70	48.03	295.42	10.12	72.53
3	33.79	7.30	28.31	41.09	66.91	2799.16	9.36	207.39
4	18.3	2.42	15.36	22.25	31.88	29.54	22.63	50.71
5	9.39	18.02	7.53	42.88	76.3	254.52	47.49	114.8
6	99.39	19.06	67.19	100.69	101.51	18.12	92.92	113.89
7	65.46	81.08	15.76	82.2	86.68	5001.58	21.54	464.98
8	45.46	101.61	27.91	79.69	117.41	221.35	66.14	152.56
9	92.54	43.32	82.15	110.06	108.44	2062.54	13.37	203.33
10	27.1	80.7	12.0	49.67	64.27	726.86	14.76	172.91
11	70.78	571.76	21.23	126.12	222.36	35.46	207.95	240.06
12	71.5	198.45	36.54	95.79	72.75	770.26	5.73	127.51
13	28.7	5.74	23.88	33.37	45.28	99.23	15.63	83.26
14	74.65	73.02	24.06	106.23	134.16	1920.08	35.84	275.31
15	33.99	0.38	33.54	37.48	39.45	353.31	12.3	112.18
Overall	51.52	791.04	7.53	126.12	86.29	991.52	5.73326	464.98

Programming Coordinated Motion Patterns with the TOTA Middleware

Marco Mamei^{1,2}, Franco Zambonelli², and Letizia Leonardi¹

¹Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
Via Vignolese 905 – Modena – ITALY

²Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
Via Allegri 13 – Reggio Emilia – ITALY

{mamei.marco, franco.zambonelli, letizia.leonardi}@unimo.it

Abstract. In this paper, we present TOTA (“Tuples On The Air”), a novel middleware to coordinate the movements of a large number of autonomous components (i.e. agents) in a ubiquitous computing scenario. The key idea in TOTA is to rely on spatially distributed tuples for both representing contextual information and supporting uncoupled and adaptive interactions between application components. The TOTA middleware takes care both of propagating tuples across a network on the basis of application-specific rules and of adaptively re-shaping the resulting distributed structures accordingly to changes in the network structures. Application agents – via a simple API – can locally sense such distributed structures to achieve context-awareness and to effectively coordinate their movements.

1 Introduction

Computing is becoming intrinsically ubiquitous and mobile [6]. Computer-based systems are going to be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of interacting with each other in the context of complex distributed applications, e.g., to support our cooperative activities [4], to monitor and control our environments [2], and to improve our interactions with the physical world [9]. Also, since most of the embeddings will be intrinsically mobile, distributed software processes and components (from now on, we adopt the term “agents” to generically indicate the active components of a distributed application) will have to effectively interact with each other and effectively orchestrate their motion coordination activities despite the network and environmental dynamics induced by mobility.

The above scenario introduces peculiar challenging requirements in the development of distributed software systems: (i) since new agents can leave and arrive at any time, and can roam across different environments, applications have to be *adaptive*, and capable of dealing with such changes in a flexible and unsupervised way; (ii) the activities of the software systems are often contextual, i.e., strictly related to the environment in which the systems execute (e.g., a room or a street), whose

characteristics are typically a priori unknown, thus requiring to dynamically enforce *context-awareness*; (iii) the adherence to the above requirements must not clashes with the need of promoting a *simple programming model* requiring light supporting infrastructures, possibly suited for resource-constrained and power-limited devices [15].

Unfortunately, current practice in distributed software development are unlikely to effectively address the above requirement: (i) application agents are typically strictly coupled in their interactions (e.g., as in message-passing models), thus making it difficult to promote and support spontaneous interoperations; (ii) agents are provided with either no contextual information at all or with only low-expressive information (e.g., raw local data or simple events), difficult to be exploited for complex coordination activities; (iii) due to the above, the results is usually in an increase of both application and supporting environment complexity.

Here we focus on the problem of motion coordination in ubiquitous computing environments, because we think that this problem resembles perfectly the peculiar challenges introduced before and highlights the ineffectiveness of current practices.

The approach we propose in this paper builds on the lessons of uncoupled coordination models like event-based [5] and tuple space programming [4] and aims at providing agents with effective contextual information that – while preserving the lightness of the supporting environment and promoting simplicity of programming – can facilitate both the contextual activities of application agents and the definition of complex distributed motion coordination patterns.

In TOTA (“Tuples On The Air”), all interactions between application agents take place in a fully uncoupled way via tuple exchanges. However, unlike in traditional tuple-based model, there is not any notion like a centralized shared tuple space. Rather, tuples are some how injected into the network and, in the network, can propagate and diffuse accordingly to a tuple-specific propagation pattern, automatically re-adapting such patterns accordingly to the dynamics changes that can occur in the network. All interactions are mediated by these distributed tuples, which can express both information explicitly provided by other agents or a local view of some global property of the network. As we will show later in this paper, this facilitate the ease definition of very complex coordination activities.

The contribution of this paper is to motivate and present the key concepts underlying TOTA, and to show how it can be effectively exploited to develop in a simple way adaptive and context-aware distributed applications, suitable for the dynamics of pervasive computing scenarios. To this end, the following of this paper is organized as follows. Section 2 overviews the TOTA approach. Section 3 details how to program distributed motion coordination in TOTA. Section 4 discusses related works. Section 5 concludes and outlines future works.

2 The Tuples on the Air Approach

TOTA proposes relying on distributed tuples for both representing contextual information and enabling uncoupled interactions among distributed application agents. Unlike traditional shared data space models [4], tuples are not associated to a specific

node (or to a specific data space) of the network. Instead, tuples are injected in the network and can autonomously propagate and diffuse in the network accordingly to a specified pattern. Thus, TOTA tuples form a sort of spatially distributed data structure able to express not only data to be transmitted between application agents but, more generally, some property of the distributed environment.

To support this idea, TOTA is composed by a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. Each TOTA node holds references to a limited set of neighboring nodes. The structure of the network, as determined by the neighborhood relations, is automatically maintained and updated by the nodes to support dynamic changes, whether due to nodes' mobility or to nodes' failures. The specific nature of the network scenario determines how each node can found its neighbors: e.g., in a MANET scenario, TOTA peers are found within the range of their wireless connection; in the Internet they can be found via an expanding ring search. More details on the architecture of the TOTA middleware can be found in [9].

Upon the distributed space identified by the dynamic network of TOTA peers, each agent is capable of locally storing tuples and letting them diffuse through the network. Tuples are injected in the system from a particular node, then they spread hop-by-hop accordingly to their propagation rule, generally being stored in the nodes they visit. Accordingly, a tuple in TOTA is defined in terms of a "content", and a "propagation rule".

$$T=(C,P)$$

The content C of a tuple is basically an ordered set of typed fields representing the information carried on by the tuple. The propagation rule P determines how a tuple should be distributed and propagated in the TOTA network. This includes determining the "scope" of the tuple (i.e. the distance at which such tuple should be propagated and possibly the spatial direction of propagation) and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rules can determine how tuple's content should change while it is propagated. Tuples do not necessarily have to be distributed replicas: by assuming different values in different nodes, tuples can be effectively used to build a distributed data structure expressing some kind of spatial/contextual information. We emphasize that the TOTA middleware supports tuples propagation actively and adaptively: by constantly monitoring the network local topology and the income of new tuples, the middleware automatically re-propagates tuples as soon as appropriate conditions occur. For instance, when new peers get in touch with a network, TOTA automatically checks the propagation rules of the already stored tuples and eventually propagates the tuples to the new peers. Similarly, when the topology changes due to peers' movements, the distributed tuple structure automatically changes to reflect the new topology.

From the application agents' point of view, executing and interacting basically reduces to inject tuples, perceive local tuples, and act accordingly to some application-specific policies. Software agents on a TOTA node can inject new tuples in the network, defining their content and their propagation rule. They have full access to the

local content of the middleware (i.e., of the local tuple space), and can query the local tuple space – via a pattern-matching mechanism – to check for the local presence of specific tuples. Moreover, the TOTA middleware offers a read-only, view of the tuple spaces of the one-hop TOTA neighbors. This feature is fundamental since the main TOTA algorithms require the knowledge of tuples present in at least a one-hop neighborhood (e.g. to evaluate the tuples' gradients as described in 2.1).

In addition, agents can be notified of locally occurring events (i.e., changes in tuple space content and in the structure of the network neighborhood).

2.1 Motion Coordination in TOTA

To fix ideas on a case study, we focus on applications in which a group of autonomous and mobile agents have to globally coordinate their respective movements. The goals of their coordination can be various: letting agents meet somewhere [4], move avoiding traffic jams [1], distribute themselves accordingly to a specific geometry [11, 13], etc.

Our specific case study, consists in how to program a group of agents, each running on the TOTA middleware, moving maintaining a suitable formation. For example, we can imagine security guards in a museum to move and monitor the museum in a coordinated way, i.e., according to a specific formation in which they have to preserve a specified distance from each other. Security guards can be provided with wireless enabled palm computer connected each other in an ad-hoc network and running the TOTA middleware. TOTA agents could direct guards movements to help them maintaining the formation.

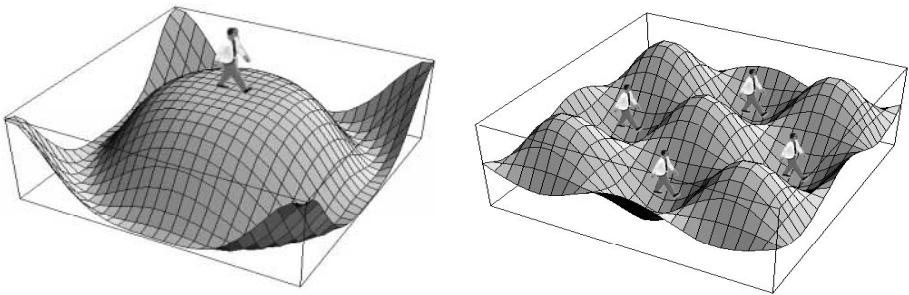


Fig 1. Distribution of a single flocking tuple (left); Regular formation of flocking peers (right)

To this end, we can take inspiration from the work done in the swarm intelligence research [1]: flocks of birds stay together, coordinate turns, and avoid each other, by following a very simple swarm algorithm. Their coordinated behavior can be explained by assuming that each bird tries to maintain a specified separation from the nearest birds and to match nearby birds' velocity. To implement such a coordinated behavior in TOTA and apply it in our case study, we can have that each security guard generates a tuple $T=(C,P)$ with following characteristics:

$C = (\text{peerName}, \text{val})$

$P = (\text{"val" is initialized at 2, propagate to all the peers decreasing by one in the first two hops, then increasing "val" by one for all the further hops})$

This tuple creates a distributed data structure in which the *val* field assumes the minimal value at specific distance from the source (e.g., 2 hops), distance expressing the intended spatial separation between security guards. For a tuple, the *val* field assumes a distribution approaching the one showed in Figure 1-left. The TOTA middleware ensures dynamic updating of this distribution to reflect peers' movements. To coordinate movements, peers have simply to locally perceive the generated tuples, and to follow downhill the gradient of the *val* fields. The result is a globally coordinated movement in which peers maintain an almost regular grid formation (see Figure 1-right).

2.2 Implementation

From an implementation point of view, we developed a first prototype of TOTA running on laptops and on Compaq IPAQs equipped with 802.11b and Personal Java. IPAQ connects locally in the MANET mode (i.e. without requiring access points) creating the skeleton of the TOTA network. Tuples are being propagated through multicast sockets to all the nodes in the one-hop neighbor. Actually we own only a dozen of IPAQs and laptops on which to run the system. Since the effective testing of TOTA would require a larger number of devices, we have implemented an emulator to analyze TOTA behavior in presence of hundreds of nodes. The emulator, developed in Java, enables examining TOTA behavior in a MANET scenario, in which nodes topology can be rearranged dynamically either by a drag and drop user interface or by autonomous nodes' movements. The strength of our emulator is that, by adopting well-defined interfaces between the emulator and the application layers, the same code "installed" on the emulated devices can be installed on Java real devices (e.g. Compaq IPAQs) enabled with wireless connectivity. This allow to test application first in the emulator, then to transfer them directly in a network of real devices. More details on the implementation can be found in [10].

3 TOTA Programming

When developing applications upon TOTA, one has basically to know:

- what are the primitive operations to interact with the middleware;
- how to specify tuples and their propagation rule;
- how to exploit the above to code agent coordination.

3.1 TOTA Primitives

TOTA is provided with a simple set of primitive operations to interact with the middleware.

```

public void inject (TotaTuple tuple);
public Vector read (Tuple template);
public Vector delete (Tuple template);
public void subscribe (Tuple template, ReactiveComponent comp, String
rct);
public void unsubscribe (Tuple template, ReactiveComponent comp);

```

Inject is used to inject the tuple passed as an argument in the TOTA network. Once injected the tuple starts propagating accordingly to its propagation rule (embedded in the tuple definition). The read primitive accesses the local TOTA tuple space and returns a collection of the tuples locally present in the tuple space and matching the template tuple passed as parameter. The delete primitive extracts from the local middleware all the tuples matching the template and returns them to the invoking agent. In addition, subscribe and unsubscribe primitives are defined to handle events. These primitives rely on the fact that any event occurring in TOTA (including: arrivals of new tuples, connections and disconnections of peers) can be represented as a tuple. Thus: the subscribe primitive associates the execution of a reaction method in the agent in response to the occurrence of events matching the template tuple passed as first parameter. The unsubscribe primitives removes matching subscriptions.

3.2 Specifying Tuples – The Flocking Tuple

TOTA relies on an object oriented tuple representation. The system is preset with an object oriented hierarchy of tuples. A programmer can inherit from this hierarchy to create his own application specific tuple. We are still developing a full hierarchy, but we already completed the definition of the branch leading to tuple whose content is a function of the hop distance from the source. The root of the full hierarchy is the **abstract class TotaTuple**, that provides a general framework for tuples programming. The class at the basis of the hop-based tuples is the **abstract class HopBasedTuple**, it inherits from *TotaTuple*, implements a general purpose breadth first, expanding ring propagation with a method called **propagate**. In this class three abstract methods, controlling the tuple propagation, its content update and its behavior have been defined:

```

public abstract boolean decidePropagation();
public abstract Tuple changeTupleContent();
public abstract boolean decideStore ();

```

These methods must be implemented when subclassing from the abstract class *HopBasedTuple* to create and instantiate actual tuples. This approach is very handy when programming new tuples, because there is no need to re-implement every time the propagation and maintenance mechanism from scratch, but it allows to customize the same breadth first, expanding ring propagation for different purposes.

In particular the core structure of the method **propagate** is the following:

```

if(this.decidePropagation()) {
    updatedTuple = this.changeTupleContent();
    if(this.decideStore()) {
        tota.store(updatedTuple);
    }
}

```

All these methods acts on the tuple itself (the “this” in OO terminology) and can access the local TOTA middleware in which they execute (i.e., in which a propagating tuple has arrived).

The first method (*decidePropagation*) is executed upon arrival of a tuple on a new node, and returns true if the tuple has to be propagated in that middleware, false otherwise. So for example an implementation of this method that returns always true, realize a tuple’s propagation that floods the network because the tuple is propagated in a breadth first, expanding ring way to all the peers in the network. Vice-versa an always false implementation creates a tuple that does not propagate, because the expanding ring propagation is blocked in all directions. In general more sophisticate implementations of this method are used to narrow down tuple spreading to avoid network flooding.

The second method (*changeTupleContent*) creates a new tuple that updates the previous one. This method is used because a tuple can change its content during the propagation process. Basically before propagating a tuple to a new node, the tuple is passed to this method that returns an eventually modified version of it, to be propagated. So for example, if the tuple content is an integer value and this method returns a tuple having that value increased by one, we are creating a tuple whose integer content increases by one at each propagation hop.

The third method (*decideStore*) returns true if the tuple has to be stored in the local middleware, false otherwise. This method can be used to create transient tuples, i.e., tuples that roam across the network like messages (or events) without being stored.

Eventually, once the above method determined that a tuple has to be locally stored, a method of the middleware with restricted access (the *store* method) is invoked on the local TOTA middleware (*tota*) to store a tuple in the local tuple space.

To clarify these points we consider the flocking motion coordination problem introduced in 2.1. Let’s create the tuple *FlockingTuple*: the content C of the tuple is encoded by the object state, the method propagate P is inherited from the abstract class *HopBasedTuple*, the three abstract methods, controlling the tuple propagation, its content update and its behavior must be implemented.

```

public class FlockingTuple extends HopBasedTuple {
    public String peerName;
    public int val = 2;
    /* always propagate */
    public boolean decidePropagation() {
        return true;
    }
    public HopBasedTuple changeTupleContent() {
        /* here it must be a function of the hop value. The correct hop
        distance is maintained in the super-class HopBasedTuple */
        // flocking shape
        if(hop <= 2) val--;
    }
}

```



```

    else if(hop >2) val++;
    return this;
}
/* always store */
public boolean decideStore() {
    return true;
}
}

```

The *decidePropagation* and *decideStore* methods return always true, because the flocking tuple should be spread to the whole network and being stored in every node. This is in order to create just one large flock, rather than some separate flocks. The *changeTupleContent* method changes the tuple being propagated by decreasing its *val* field for the first two hops then letting it increase. In this way the *val* field has a minimum located two-hops away from its source.

3.3 The Flocking Agent

The algorithm followed by flocking agents is very simple: they simply have to determine the closest peer, and then move by following downhill that peer's flocking tuple. To achieve this goal each agent will evaluate the gradient of the flocking tuple it is going to follow, by comparing tuple's values in its neighborhood.

```

public class FlockingAgent extends Thread implements AgentInterface {
    private TotalMiddleware tota;
    ...
    public void run() {
        // inject the meeting tuple to participate the meeting
        FlockingTuple ft = new FlockingTuple ();
        ft.setContent(peer.toString());
        tota.inject(ft);

        while(true) {
            // read other agents' flocking tuples
            FlockingTuple query = new FlockingTuple();
            Vector v = tota.read(query);
            /* evaluate the gradients and select the peer to which the
            gradient goes downhill */
            GenPoint destination = getDestination(v);
            // move downhill following the meeting tuple
            peer.move(destination);
        }
    }
    ...
}

```

4 Related Works

Several proposals in the last few years are challenging the traditional ideas and methodologies of distributed software engineering by proposing frameworks and models inspired by physical and biological systems to facilitate, among the others, distributed motion coordination.

In robotics, the idea of potential fields driving robots movement is not new [8]. For instance, one of the most recent re-issue of this idea, the Electric Field Approach (EFA) [7], has been exploited in the control of a team of Sony Aibo legged robots in the RoboCup domain. Following this approach, each Aibo robot builds a field-based representation of the environment from the images captured by its head mounted camera, and decides its movements by examining the fields' gradients of this representation. Although very close in spirit, EFA and TOTA are very different from the implementation point of view: in TOTA, fields are distributed data structure actually spread in the environment; in EFA, fields are just an agent internal representation of the environment and they do not actually exists. TOTA requires a supporting infrastructure to host fields' data structures, but it completely avoids the complex algorithms involved in field representation and construction.

Smart Messages [2], rooted in the area of active messages, is an architecture for computation and communication in large networks of embedded systems. Communication is realized by sending "smart messages" in the network, i.e., messages which include code to be executed at each hop in the network path. The execution of the message at each hop determines the next hop in the path, making messages responsible for their own routing. Smart Messages share with TOTA, the idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunk of code to determine their propagation. However, in Smart Messages code is used mainly for routing or packets mobility purposes. In TOTA instead, the tuples are not simply routed though the network, but can be persistent and create a distributed data structure that remains stored in the network, and TOTA propagation code can also be used to change the message/tuple content, in order to realize distributed data structure and not just replicas.

As a final note, we emphasize that (i) recent approaches in the area of modular robots [13] exploit the idea of propagating "coordination field" across the robot agents so as to achieve a globally coherent behavior in robot's re-shaping activities; (ii) in the popular simulation game "The Sims" [14], characters move and act accordingly to specific fields that are assumed to be spread in the simulated environment and sensed by characters depending on situations (e.g., they sense the food field when hungry); (iii) ant-based optimization systems [1, 12] exploit a virtual environment in which ants can spread pheromones, diffusing and evaporating in the environment according to specific rules. (iv) amorphous computers [3, 11] exploit propagation of fields to let particles self-organize their activities. Although serving different purposes, these approaches definitely share with TOTA the same physical inspiration.

5 Conclusions and Future Works

Tuples On The Air (TOTA) promotes programming motion coordination activities by relying on distributed data structures, spread over a network as sorts of electromagnetic fields, and to be used by application agents both to extract contextual information and to coordinate with each other in an effective way. As we have tried to show in this paper, TOTA suits the needs of pervasive computing environments (spontaneous interoperation, effective context-awareness, lightness of supporting

environment) and facilitates both the access to distributed information and the expression of distributed coordination patterns.

Several issues are still to be investigated to make TOTA a practically useful framework for the development of pervasive applications. First, proper access control models must be defined to rule accesses to distributed tuples and their updates (and this is indeed a challenging issue for the whole area of pervasive and mobile computing). Second, because the maintenance of all the distributed tuples may create scalability issues, we think it would be necessary to enrich TOTA with mechanisms to compose tuples with each other, so as to enable the expression of unified distributed data structures from the emanation of multiple sources. Finally, deployment of TOTA applications in real-world scenarios will definitely help identify current shortcomings and directions of improvement, possibly suggesting application even beyond the motion coordination area.

Acknowledgements. Work supported by the Italian MIUR and CNR in the “Progetto Strategico IS-MANET, Infrastructures for Mobile ad-hoc Networks”.

References

- [1] E. Bonabeau, M. Dorigo, G. Theraulaz, “Swarm Intelligence”, Oxford University Press, 1999.
- [2] C. Borcea, et al., “Cooperative Computing for Distributed Embedded Systems”, 22th International Conference on Distributed Computing Systems, Vienna (A), IEEE CS Press July 2002.
- [3] W. Butera, “Programming a Paintable Computer”, PhD Thesis, MIT Media Lab, Feb. 2002.
- [4] G. Cabri, L. Leonardi, F. Zambonelli, “Engineering Mobile Agent Applications via Context-Dependent Coordination”, IEEE Transactions on Software Engineering, 28(11), Nov. 2002.
- [5] G. Cugola, A. Fuggetta, E. De Nitto, “The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS”, IEEE Transactions on Software Engineering, 27(9): 827–850, Sept. 2001.
- [6] D. Estrin, D. Culler, K. Pister, G. Sukjatme, “Connecting the Physical World with Pervasive Networks”, IEEE Pervasive Computing, 1(1):59–69, Jan. 2002.
- [7] S. Johansson, A. Saffiotti, “Using the Electric Field Approach in the RoboCup Domain”, RoboCup 2001, LNAI, Springer Verlag, pp. 399–404, Seattle, WA, USA, 2001.
- [8] O. Khatib, “Real-time Obstacle Avoidance for Manipulators and Mobile Robots”, The International Journal of Robotics Research, 5(1):90–98, 1986.
- [9] M. Mamei, L. Leonardi, M. Mahan, F. Zambonelli, “Coordinating Mobility in a Ubiquitous Computing Scenario with Co-Fields”, Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices, AAMAS 2002, Bologna, Italy, July 2002.
- [10] M. Mamei, F. Zambonelli, L. Leonardi, “Tuples On The Air: a Middleware for Context-Aware Computing in Dynamic Networks”, 1st International ICDCS Workshop on Mobile Computing Middleware (MCM03) Providence, Rhode Island. May 2003.
- [11] R. Nagpal, “Programmable Self-Assembly Using Biologically-Inspired Multiagent Control”, 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna (I), ACM Press, July 2002.

- [12] V. Parunak, S. Bruekner, J. Sauter, "ERIM's Approach to Fine-Grained Agents", NASA/JPL Workshop on Radical Agent Concepts, Greenbelt (MD), Jan. 2002.
- [13] W. Shen, B. Salemi, P. Will, "Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots", IEEE Transactions on Robotics and Automation, Oct. 2002.
- [14] The Sims, www.thesims.com
- [15] F. Zambonelli, V. Parunak, "From Design to Intention: Signs of a Revolution", 1st Intl. ACM Conference on Autonomous Agents and Multiagent Systems", Bologna (I), July 2002.

iClouds – Peer-to-Peer Information Sharing in Mobile Environments

Andreas Heinemann*, Jussi Kangasharju, Fernando Lyardet, and Max Mühlhäuser

Telecooperation Group, Department of Computer Science
Darmstadt University of Technology
Alexanderstr. 6, 64283 Darmstadt, Germany
aheine@gkec.tu-darmstadt.de
{jussi, fernando, max}@tk.informatik.tu-darmstadt.de

Abstract. The future mobile and ubiquitous computing world will need new forms of information sharing and collaboration between people. In this paper we present *iClouds*, an architecture for spontaneous mobile user interaction, collaboration, and transparent data exchange. iClouds relies on wireless ad hoc peer-to-peer communications. We present the iClouds architecture and different communication models, which closely resemble familiar communication forms in the real world. We also design a hierarchical information structure for storing the information in iClouds. We present our prototype implementation of iClouds which runs on wireless-enabled PDAs.

1 Introduction

People living in the future ubiquitous computing world will need new ways to share information and interests as well as collaborate with each other. Given the success of wireless communications, such as mobile telephones, 802.11b, or Bluetooth, many of these activities will benefit from or rely on wireless ad hoc communications.

This paper presents the iClouds project in which we study the architectures and mechanisms required to support mobile user interaction, collaboration, and transparent data exchange. The iClouds project is part of the Mundo research activity, which we will present in the next section.

Our motivation behind iClouds can be expressed as follows: “Whenever there is a group of people, they may share a common goal or have a related motivation. Information of interest may be in possession of only a few of them.” The goal of iClouds is to make this information available to the whole group, based on individual user contribution, through peer-to-peer communications and data exchange.

* The author’s work was supported by the German National Science Foundation (DFG) as part of the PhD program “Enabling Technologies for Electronic Commerce” at Darmstadt University of Technology.

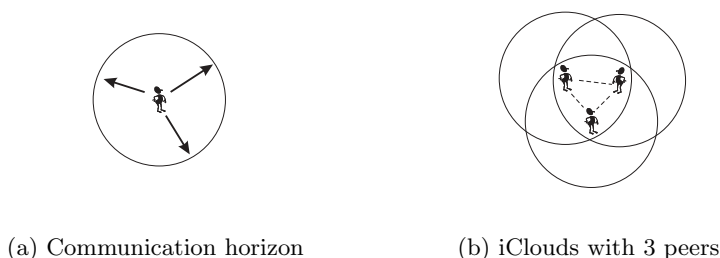


Fig. 1. Information clouds

Consider a person walking with a wireless-enabled PDA. The communication range of the wireless device defines a sphere around that node. We call this sphere or communication horizon an *information cloud* or *iCloud* (see Fig. 1(a)). In practice this will not be an ideal sphere due to radio signal interference with buildings and other structures. The limited communication range (a few hundred meters at most) is a desired property, because it allows for easy ad hoc meetings and collaboration. When several nodes come close together, as shown in Fig. 1(b), the devices can communicate with each other and exchange information depending on what information the users provide and need. This exchange happens automatically, without any need for direct user intervention.

We have identified several application scenarios in which iClouds is beneficial:

- **Local Information Acquisition.** Residents of a city publish information about their city which tourists are interested in. This could be information about sights, restaurants, or useful telephone numbers, such as taxi number, etc.
- **Common Goal Pursuit.** iClouds can bring people with common interests together to help them collaborate. For example, consider students in a classroom. Some students may have formed study groups and others might be interested in joining those groups. Students already in the groups could publish the groups and interested students could directly contact them and join the group.
- **Advertisement and mCommerce.** A store can publish ads which are picked up by interested customers. These customers further pass the ads along to other interested users when they are away from the store, thus increasing the reach of the ads. If any of the users who have received ads in this way actually make a purchase, the store could give a bonus to the person who passed the ad along. This bonus could be, for example, points or a discount on the next purchase.

This paper is organized as follows. Section 2 describes the underlying Mundo project. Section 3 presents the iClouds architecture and communication mechanisms. In Section 4 we describe our prototype implementation of iClouds. Sec-

tion 5 discusses related work. Finally, Section 6 concludes the paper and presents directions for future work.

2 Mundo Overview

In this section, we provide a brief overview of the Mundo project. A more complete description can be found in [1]; below we will briefly present the different entities in Mundo.

ME (Minimal Entity)

We consider ME devices as the representation of their users in the digital world. The personal ME is the only entity always involved in the user's activities. Our decision for one individually owned device rather than a publicly available infrastructure comes from *trust establishment*. As an increasing amount of computer based activities have (potential) legal impact and involve privacy issues, it is vital that users can trust the device carrying out such transactions.

US (Ubiquitous aSsociable object)

Minimalization pressure will not permit feature-rich MEs in the near future. Hence, they must be able to connect to local devices such as memory, processors, displays, and network devices in an ad hoc manner. We call this process association and we call such devices *ubiquitous associable objects* (US). Through association, the ME can personalize the US to suit the user's preferences and needs. For privacy reasons, any personalization of an US must become unavailable if it is out of range of the user's ME device.

IT (smart ITeM)

There are also numerous smart items that do not support association that would turn them into an US. Vending machines, goods equipped with radio frequency IDs, and landmarks with "what is" functionality are just a few examples. We define such *smart items* as ITs. An IT is any digital or real entity that has an identity and can communicate with the ME. Communication may be active or passive; memory and computation capabilities are optional.

WE (Wireless group Environment)

We expect ad hoc networking to be restricted to an area near to the user of a ME device, as connections with remote services will involve a non-ad hoc network infrastructure. The functionality of a WE is to bring together two or more personal environments consisting of a ME and arbitrary US entities each. A WE makes connections between the devices possible and also allows for sharing and transferring hardware (e.g., US devices) and software or data between WE users.

iClouds is our most advanced project (in terms of realization) within the WE sub-domain of Mundo. While iClouds focuses on pure information sharing, other WE-related activities consider hardware sharing and corresponding security aspects.

THEY (Telecooperative Hierarchical ovErlaY)

We regard overlay cells as the backbone infrastructure of Mundo. These *telecooperative hierarchical overlay cells* (THEY) connect users to the (non-local)

world, deliver services, and data to the user. THEY support transparent data access, for example, by caching frequently used data on US devices, and offer transparent cooperation between different physical networks. Because iClouds is based on ad hoc networking, it does not require the services of THEY.

3 iClouds System Description

iClouds devices are small mobile devices (like PDAs) with mobile communication support for a maximum of a few 100 meters; one example is a PDA with 802.11b support. There is no need for any central servers in the iClouds architecture; instead, each device is completely independent.

The diameter of the iClouds communication horizon (Fig. 1(a)) should not exceed a few hundred meters. We want to give iCloud users the option for spontaneous collaboration and when two iCloud users “see” each other, they should be within a short walking distance from each other (a couple of minutes at maximum). To allow for this easy collaboration, we specifically exclude multi-hop communication. Therefore, iClouds does not require any routing protocols; all communications happen directly between the concerned parties.

3.1 Data Structures, Communication Pattern, and Information Exchange

The two most important data structures found on the iClouds device are two information lists (*iLists* for short):

- *iHave*-list (information have list or information goods)
The iHave list holds all the information the user wants to contribute to the iCloud. The items could be, for example, simple strings, or more complex entities expressed in XML. We discuss these issues in Section 3.2.
- *iWish*-list (information wish list or information needs)
In the iWish list, the user specifies what kind of information he is interested in. The exact format of the items on this list depend on the format of the entries on the iHave list. Typically they would be search patterns which are matched against the entries on the iHave lists of other users. Note that the items on the iWish list are more private, since they reflect the needs of the user, which the user may not want to disclose to others.

Each iClouds device periodically scans its vicinity to see if known nodes are still active and in communication range and also to see if any new nodes have appeared. Information about active nodes is stored in a data structure called *neighborhood*.

In the second stage, the iClouds devices align their information goods and needs. This is achieved by exchanging iLists. Items on the *iWish*-lists are matched against items on the *iHave*-lists. On a match, information items move from one iHave-list to the other.

Table 1. Information Flow Semantics (from Alice’s point of view)

	pull (from Bob)	push (to Bob)
iHave-List	Standard search	Advertise
iWish-List	Active service inquiry	Active search

For example, consider two iClouds users, Alice and Bob, who meet on the street. When their iClouds devices discover each other, they will exchange their iHave lists and match them locally against their iWish lists. If an item on Bob’s iHave list matches an item on Alice’s iWish list, her iClouds device will transfer that item onto her iHave list.

We have two main communication methods for transferring the iLists. Peers can either *pull* the iLists from other peers or they can *push* their own iLists to peers they encounter. In addition, either of these two operations is applicable to both lists, which gives us four distinct possibilities of communication. We summarize these possibilities, along with their real-world equivalents, in Table 1.

In each of the four cases shown in Table 1, the matching operation is always performed on the peer who receives the list (Alice’s in pull and Bob’s in push). Each of the four possible combinations corresponds to some interaction in the real world:

- *Standard search.* Alice pulls iHave-List from Bob.
This is the most natural communication pattern. Alice asks for the information stored on Bob’s device and performs a match against her information needs (specified in her iWish-List) on her device. We can also see the user as just passively “browsing” what is available.
- *Advertise.* Alice pushes her iHave-List to Bob.
This is a more direct approach. Alice gives her information goods straight to Bob and it’s up to Bob to match this against the things he is interested in. As an example, consider iClouds devices mounted on shopping mall doorways pushing advertisements onto customer devices when they enter the building.
- *Active service inquiry.* Alice pulls iWish-List from Bob.
This is best suited for shopping clerks. They learn at a very early stage, what their customers are interested. An example of this query could be: “*Can I help you, please show me what are you looking for?*”.
In general, especially for privacy reasons and user acceptance, we believe it is a good design choice to leave the iWish-list on the iClouds device. Hence, this model of communication would likely be extremely rare in the real world.
- *Active search.* Alice pushes her iWish-List to Bob.
With active search, we model the natural “*I’m looking for X. Can you help me?*”. This is similar to the standard search mechanism, except that the user is actively searching for a particular item, whereas in the standard search the user is more passive.

3.2 Information Modeling and List Matching

A key issue in iClouds is a set of rules or constraints for the information items on the iHave-list, as well as for the search patterns stored on the iWish-list.

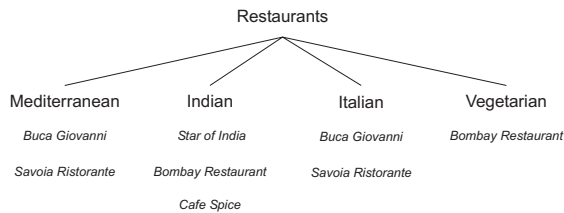


Fig. 2. Hierarchical data organization

For example, consider free text for both lists. A hypothetical item *Best Italian Restaurant on Market Street* will not match an information wish *Interested in Mediterranean Food*. Therefore the chances are close to zero, that any information is passed through iClouds, although one node holds information the other node is interested in.

To overcome this, we propose a hierarchical system, as shown in Fig. 2, similar to the organization of the Usenet, the product catalog of eBay, or the online catalog of Yahoo. Items are divided into categories (e.g., restaurants, hotels, etc.) which are further divided into subcategories (e.g., Italian, Indian, etc. restaurants). Note that entries may belong to several categories. For example, the Bombay Restaurant in Fig. 2 is listed under Indian and Vegetarian, because it offers both Indian and vegetarian food.

These hierarchies can be stored at a central server and can be downloaded to an iClouds device while it is in its cradle during power recharge or a normal sync operation. This system should be open for extension and categories could be moderated by different people to improve quality.

Hierarchies are well understood by users and, in addition, technologies, such as XML and XPATH, support the construction of hierarchically organized information and search very well. The names of the most common tags, such as address or telephone, will need to be standardized. We will also need a standardized way for user extensions to the hierarchy. These are topics of our on-going work.

3.3 User Notification and Privacy

Collaboration regarding a common goal or a related motivation requires that users are notified by the iClouds device on a successful match. This can be a simple beep or a vibrating alarm as found in mobile phones today. For each item (search pattern) on the iWish-list, the user can specify, whether a notification should be sent or not. This enables a user to differentiate between pure collect pattern, e.g. *Good Restaurants?*, and patterns that require some kind of action, e.g. talk to the other person.

iClouds devices are linked to their owners, broadcast information, and are traceable, hence they raise the question of user privacy. To protect user privacy, iWish-lists never leave the device, unless explicitly allowed by the user. Therefore, to construct a user profile is not possible. A user can mark each item on the

iHave-list as private. Private items will then be unavailable to other parties. In addition, the comparatively short communication range constitutes a natural protection for user privacy.

4 Prototype

To gain more practical experiences with iClouds, we have built a first prototype and set up a testbed. The prototype runs on Toshiba Pocket PC e740 (Windows CE) with integrated 802.11b cards. For the underlying link layer network connectivity, we run the PDAs in 802.11b ad hoc mode with statically assigned IP addresses. The prototype was developed using PersonalJava from the Java2 Micro Edition (J2ME).

Our information list data structures consist of strings. Currently, we have not yet implemented hierarchies. iList comparison is based on a simple substring matching function. A successful match will copy iHave-list items to new devices. The user is notified of this event by a beep from the device. This allows the user to check her updated information goods and plan further action.

We use a UDP based ping/pong mechanism for scanning the vicinity for new nodes. New node discovery is done by periodically pinging every IP address within a given set. A node sends a ping message to other nodes in and waits for a pong message. Upon receiving a pong, the new node is added to the active neighborhood. Otherwise, after a certain timeout, the node is removed from the neighborhood. When a node encounters a new node, it pulls the iHave list from the new node using an HTTP GET request.

Because a PDA is not a good device for data input, we believe that the iLists should be managed in a desktop application and the lists should be synced to the PDA during the normal sync operation.

5 Related Work

The Proem Platform [3] targets very similar goals. The main difference to iClouds is that they focus on Personal Area Networks (PAN) for collaboration. We believe that it is fruitful to focus on a wider area (mobile networks that cover several 100 meters in diameter) and that it is not necessary to encounter communication partners physically for information exchange.

Sharing information among mobile, wireless users is also subject of the 7DS Architecture [5,6]. In contrast to iClouds, in 7DS the users are intermittently connected to the Internet and cache information, i.e., HTML pages, which is accessed during that time frame. Later these caches are used to fulfill requests among the nodes.

The Usenet-on-the-fly system [2] makes use of channels to share information in a mobile environment. But the information spreading is limited by a hop count in the message. This has the disadvantage, that an unlucky user might be one hop too far away from the information source, although she might be interested in receiving the information.

Mascolo *et al.* describe XMIDDLE [4], a peer-to-peer middleware that allows the synchronization of XML documents across several mobile devices of one user. The data modeling and exchange in iClouds has to fulfill similar tasks, it has to work between different users though.

Basic information services require contributions from users. This is true for many current systems. The Usenet news is certainly one of the most prominent and successful systems. Tveit [7] proposes a peer-to-peer based network of agents, that support product and service recommendations for mobile users. Recommendations are provided by aggregating and filtering individual user input. Tveit focuses on infrastructured wireless networks, e.g. mobile phone networks.

6 Conclusion

In this paper we have presented iClouds, an architecture for supporting spontaneous user interaction, collaboration, and transparent data exchange in a mobile ubiquitous computing context. iClouds achieves these goals through ad hoc peer-to-peer communications between wireless-enabled devices. iClouds supports many natural forms of interaction, such as browsing for information, searching, and advertising. We have devised a hierarchical information structure for storing and matching information in iClouds. We have also implemented a prototype of iClouds which runs on PDAs with wireless LAN cards.

As part of our future work, we will develop a model for specifying the iClouds information hierarchies and ways to extend them. In addition, we plan on investigating ontologies as a possible way to improve the matching between items on the iLists.

References

1. Hartl, A. *et al.*, Engineering Multimedia-Aware Personalized Ubiquitous Services. In IEEE MSE 2002, Newport Beach, CA, December 2002.
2. Becker, C., Bauer, M., Hähner, J., Usenet-on-the-fly - Supporting Locality of Information in Spontaneous Networking Environments, Workshop on Ad Hoc Communications and Collaboration in Ubiquitous Computing Environments (at CSCW), New Orleans, LA, November 2002.
3. Kortuem, G. *et al.*, When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad-hoc Networks, International Conference on Peer-to-Peer Computing, Linköping, Sweden, August 2001.
4. Mascolo, C., Capra, L., Emmerich, W.: An XML-based Middleware for Peer-to-Peer Computing. International Conference on Peer-to-Peer Computing, Linköping, Sweden, August 2001.
5. Papadopouli, M., Schulzrinne, H., Seven degrees of separation in mobile ad hoc networks, IEEE GLOBECOM, 2000.
6. Papadopouli, M., Schulzrinne, H., Design and implementation of a peer-to-peer data dissemination and prefetching tool for mobile users, First NY Metro Area Networking Workshop, 2001.
7. Tveit, A., Peer-to-peer based Recommendations for Mobile Commerce, First International Workshop on Mobile Commerce, Rome, Italy, 2001.

Support for Personal and Service Mobility in Ubiquitous Computing Environments*

K. El-Khatib**, N. Hadibi, and Gregor v. Bochmann

School of Information Technology & Engineering, University of Ottawa
161 Louis Pasteur St., Ottawa, Ont., K1N 6N5, Canada
{elkhatib, nhadibi, bochmann}@site.uottawa.ca

Abstract. This paper describes an agent-based architecture that extends personal mobility to ubiquitous environment. A software agent, running on a portable device, leverages the existing service discovery protocols to learn about all services available in the vicinity of the user. Short-range wireless technology such as Bluetooth can be used to build a personal area network connecting only devices that are close enough to the user. Acting on behalf of the user, the software agent runs a QoS negotiation and selection algorithm to select the most appropriate available service(s) to be used for a given communication session, as well as the configuration parameters for each service, based on session requirements, the user preferences and the constraints of the devices that provide the service(s). The proposed architecture supports also service hand-off to take account of service volatility as a result of user mobility.

1 Introduction

Our work here is motivated by the growing need to provide personal mobility for persons roaming in ubiquitous computing environments. Personal mobility [1] is defined as the ability of a user to get access to telecommunication services from any terminal (e.g. workstations, notebooks, Personal Digital Assistants (PDA), cellular phones) at any time and from any place based on a unique identifier of the user, and the capability of the network to provide services in accordance with the user's service profile. Closely related to the subject of personal mobility is service or session mobility [6], which refers to the possibility of suspending a service on a device and picking it up on another device at the same point where it was suspended. An example of service mobility is a call transfer from the mobile phone of the user to his office phone.

Ubiquitous computing is a new trend in computation and communication; it is at the intersection of several technologies, including embedded systems, service discovery, wireless networking and personal computing technologies. It is best described by Mark Weiser, father of ubiquitous computing, as the world with "invisible" machines; a world such that "its highest ideal is to make a computer so imbedded, so

* This work was partially supported by the Communications and Information Technology Ontario (CITO)

** Currently working at the National Research Council of Canada.

fitting, so natural, that we use it without even thinking about it”[2]. In such a ubiquitous environment, devices are only visible through the services they provide; specific information about the devices such as location, address or configuration parameter is transparent to the user. A user roaming in a ubiquitous environment might be surrounded by several services, depending on his/her current location.

One of the major contributing factors to the big interest in ubiquitous computing is the advance in short-range radio frequency communication that created the notion of personal level communication infrastructure, referred to as Wireless Personal Area Networking (WPAN). Bluetooth [3] is an example of WPAN technology. Devices connected to the WPAN have the capability to locate, communicate, and provide services for each other. This capability additionally allows these devices to collaboratively provide an ad hoc distributed computing environment and to deliver services that cannot be possible delivered with only one device. For instance, a video display service can use an audio playing service in its vicinity to play an audio and video recording. Also, an audio play-out service can use the service of a PC connected to the Internet to download and play music from the Internet.

Given these trends in personal communication, personal mobility comes as a challenge to ubiquitous environments. The architecture proposed in this paper is an improvement on our personal mobility architecture [8] for classical non-ubiquitous environments. It leverages technologies in short-range wireless communication, such as Bluetooth, to construct the WPAN; it also leverages service discovery protocols, such as Jini, SDP, and Salutation, to discover services available just in the WPAN of the roaming user. Running a service discovery protocol over a WPAN restricts the services available for the use of the user to devices that are within the WPAN and hence close enough to the user. The architecture supports also optional service mobility, which allows, when possible, the service to follow the user as he moves. A major component of our architecture is a Personal Agent (PA) process running on a personal device carried by the user; the PA triggers the service discovery, service selection and service mobility. It also enforces the user’s policies and preferences stated in the user profile.

Throughout the paper, we will show how the presented architecture is used during a communication session between Alice, a team manager on a business trip, and her team-members. The elaboration of the scenario is presented in Section 3. Before the meeting, Alice would like to have small chat with Bob, who is a team leader in her group. Using the multimedia workstation in the business office of the hotel, Alice sends an invitation to Bob ten minutes before the meeting time. Bob, sitting in the lounge area, receives the call on his PDA for a multimedia conversation with Alice. Since Bob has indicated in his profile that he is always willing to accept calls from his manager Alice, Bob’s PDA tries to find a microphone, a speaker, a video display service and a camera to make for a full multimedia session. Assuming that such services exist in the lounge area, the PDA can discover and reserve them for the communication session with Alice. The PDA sends back the addresses of the play-out services, and the videoconference is started between Alice and Bob.

When the time for the meeting comes, Bob moves with his PDA into the conference room where all the team members are waiting. Bob’s PDA detects that the services that Bob was using are not available anymore, and since he has already set the FOLLOW-ME option of the session to ON, his PDA tries to discover similar services to continue the session in the conference room. The PDA then detects and

selects the big screen, the camera, the speaker as well as the microphone of the conference room; Alice's picture appears on the big screen and she is now ready to participate in the meeting.

The rest of the paper is organized as follows: Section 2 starts by presenting a literature review of a number of architectures for personal mobility with highlights to their limitations in ubiquitous environments. We then propose our architecture for supporting personal mobility in ubiquitous environment and its main components. After that, we present the algorithm used for service and Quality of Service (QoS) parameter values selection and explain how our architecture supports service mobility. Section 3 continues with more details about the usage scenario introduced in Section 1. We finally conclude in Section 4.

2 Architecture for Personal and Service Mobility

2.1 Related Architectures

A number of architectures [4, 5, 6, 7, 8] were proposed to solve the personal mobility problem in the context of the Internet. All these architectures share the same concept of a Directory Service (DS) that provides a user-customized mapping from a unique user identifier to a device that is best suitable for the user to use. The basic idea of these architectures is that the user keeps a profile in the DS including static information about all the devices he/she uses, and the logic or policy for when to use these devices (user preferences). During session initiation, the DS executes the logic in the user profile and handles the communication request according to the user's specified preferences.

While these architectures provide personal mobility for users with multiple telecommunication devices (telephone, PDA, laptop, cellular phone, pager), they all fail short to extend personal mobility to ubiquitous environments because:

- the information in the directory service is static,
- there is no support for service or device discovery,
- they lack support for service mobility, and finally
- they lack support for complex (combined) services.

A number of researchers have addressed the problem of service selection in ubiquitous computing environments. The work in [9] presented two approaches for selecting services based on the physical proximity and line of sight of the handheld device relative to the service. The authors in [10] used a centralized approach where a central gateway is responsible for delegating rendering tasks to devices in the environment of the user. A PDA carried by the user is responsible for detecting the devices, and sending the list of available devices to the central gateway. Both these architectures suffer from the drawback of using infrared communication for finding and selecting services. Because infrared communication requires the knowledge of the location of the existing devices due to the line-of-sight restriction, these architectures cannot be used in ubiquitous environments because they require user's awareness of the available services. Service mobility and QoS issues are also not discussed in these works. In another work [11], the authors investigated the use of a browser running on a PDA to enable ubiquitous access to local resources as well as resources on the

World Wide Web. The browser, called the Ubicompbrowser, detects devices and resources in the environment of the user, and delegates the rendering of the requested resources to these devices in order to overcome the limitation of the PDA. A major drawback of the Ubicompbrowser is that it requires the user to know its current location to restrict the set of available services. Additionally, the Ubicompbrowser does not deal with the issue of QoS negotiation, neither with the issue of service mobility.

2.2 Proposed Architecture

Our work presented here is inspired by the Ubicompbrowser project, and is intended to support personal mobility in ubiquitous environments. Our architecture builds on our previous architecture for personal mobility [8] and includes additional functionalities to overcome its shortcomings in ubiquitous environments. The modified architecture uses the short-range Bluetooth wireless communication to construct the user's WPAN, and to restrict the domain of services available to the user just to those running on devices that are within this WPAN. Our architecture differs also from the architecture in [9,10] in that service selection is done automatically on behalf of the user without requiring the user to point and select each service individually using infrared, since the user might not (and should not) be aware of the services and their locations. We also address the problem of service mobility by using periodical search for services similar¹ to the services currently used by the user, in order to provide smooth hand-off for these services.

Our previous architecture for personal mobility [8] is based on the concept of a Home Directory. The Home Directory (HD) has two functions: (a) storage of the user's profile and (b) forwarding of incoming communication requests. As a storage facility, the HD is a personalized database of users profiles, with each profile holding the user's contact information, current access information, preferences for call handling, presence service, authentication, authorization, and accounting information. With this logic stored with the data in the profile, end users can control the access permission to their devices according to their own preferences. To forward communication requests, a Home Directory Agent (HDA) executes the logic in the user profile every time the user receives an incoming call through the HD. The user can also invoke the HDA in order to find (and establish) the best way to place an outgoing communication request.

In a ubiquitous computing environment, the set of available devices for the user may change continuously as the user changes his/her location. Updating manually the information about currently available services is not a feasible. Additionally, discovering the services and sending update messages to the HDA is not a practical solution since the set of available services might change very often, which results in many update messages. Moreover, an update message might incur a certain delay so that by the time the message is received by the HDA, the information included in the message might be outdated.

To overcome these limitations, we propose to run a modified version of the HDA on a hand-held device, such as a PDA, that is carried by the user. We call this

¹ We say that two services are similar if they serve the same purpose, for instance a TV and a wall projector, or a PC speakers and a mini-stereo.

modified version of the HDA the Personal Agent (PA), and it is responsible for detecting devices in the vicinity of the user as well as managing the user's communication sessions. We require that the hand-held device, on which the Personal Agent runs, have access to the Internet (through a wireless modem or IEEE 802.11[12] connection) in order to retrieve the user profile and send/receive communication requests through the HDA. The PDA is also supposed to be able to construct a Wireless Personal Area Network (such as Bluetooth WPAN) in order to be able to detect and communicate with other wireless devices just around the user. For the rest of the paper, we will assume that the PA is running on a PDA that satisfies these communication requirements.

At any one time, either the HDA or the PA is providing personal mobility service to the user. When the PDA is switched ON, the PA contacts the HDA to retrieve the user profile. From that point on until the PDA is switched OFF, the PA is responsible for executing the logic in the user profile, and the HDA would act only as a proxy for incoming call requests. To ensure that the HDA is aware of the status of the PA, replies to communication requests are sent back through the HDA. The HDA can detect when the PA is not running or the PDA is currently out of reach if the HDA does not see a reply to a forwarded call after a certain time-out period. The HDA would then switch into active mode, and handle the communication request according to the rules specified in its local copy of the user profile.

The architecture of the Personal Agent has four major components: a **Service Discovery Agent**, a **User Activity Monitor Agent (UAMA)**, a **QoS Selection and Negotiation Agent (QSNA)**, and a **Service Registry**. Fig. 1 shows the architecture of the Personal Agent with its components. Following is a detailed description of each of these components.

- **Service Discovery Agent and Service Registry:** the function of the **Service Discovery Agent (SDA)** is to search for all services in the WPAN of the user. The SDA provides the **QoS Selection and Negotiation Agent (QSNA)** (discussed below) with the list of currently available services. Since different services might be using different service discovery protocols (JINI [13], SDP [14], SLP [15]), the SDA shall act as a service discovery client in multiple service discovery protocols. Once a session has been started, the SDA periodically searches for services that are similar to the services currently used in the session, and stores this information in the **Service Registry (SR)**. Information in the SR allows smooth service mobility, as discussed in Section 2.4.
- **User Activity Monitor Agent (UAMA):** The UAMA keeps track of the current activity of the user (idle, busy...) as well as the services he is currently using. The UAMA assists the QSNA during the service selection phase by providing up-to-the-minute information about the user's status. The UAMA can also incorporate information about the context of the user in the service selection process. Context information includes the location of the user, and whether he is by himself or surrounded by other people, as suggested in [11,16].
- **QoS Selection and Negotiation Agent (QSNA):** based on the session requirements², the information in the user profile, the list of available services

² Session requirements may be described using the Session Description Protocol (SDP) carried in the SIP INVITE message.

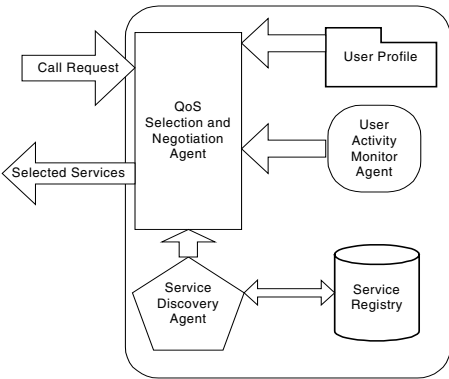


Fig. 1. Components of the Personal Agent

provided by the SDA, and the user’s current activity provided by the UAMA, the QSNA selects the best services that satisfy the session requirements, and complying to the preferences of the user (described in the user profile). The QSNA might also generate a mix and match several currently available services to satisfy the requirements of the session. The QSNA implements the service and QoS parameter values selection algorithm presented in the next section.

2.3 Service and QoS Parameter Values Selection Algorithm

To provide personal mobility in a ubiquitous environment, a decision has to be made concerning (a) the service(s) (camera, speakers, microphone...) that should be used among the discovered services that meet the requirements of the communication session, and (b) what Quality of Service (QoS) parameter values (frame rate, frame resolution, audio quality...) should be used for each service. These decisions are affected by many factors including the media type of the exchanged content, the hardware and software capabilities of the devices where the services are running (also called device limitations), the preferences of the user and finally his current context.

When the PA receives an incoming call through the HDA, and assuming that the user is willing to accept calls now, the first step in the selection process consists of selecting the services that are required for establishing the communication session. Based on the session requirements, the QSNA queries the SDA for the necessary services for the session. For instance, a communication session that includes the receipt of audio media requires the discovery and selection of an audio playing service.

The second step in the selection process consists of selecting the QoS parameter values for each service, such as the audio quality for an audio service and the video frame rate and resolution for a video service. This selection is based on the hardware/software limitations of the device, network condition, and the preferences of the user. For instance, the network access line (e.g. modem or wireless connection) of one of the devices may limit the maximum throughput of that device. Also, a video content that requires a high-resolution display eliminates the possibility of using the

display service of a monochrome device, especially if the user expressed his desire to receive only high quality video. This selection process also deals with the case when multiple similar services exist in the environment of the user. The QSNA always selects the services that give the user the highest possible satisfaction.

We base the selection of QoS parameters for each service on the concept of maximizing the user's satisfaction. In [8], we presented an extension to the work presented in [17], wherein, the QoS parameters for each service are selected based on a user satisfaction function. In his/her profile, the user indicates a satisfaction function that maps a QoS parameter value to a satisfaction value in the $[0..1]$ range. The user also assigns a weight value for each media type. The total satisfaction value of the user for a combination of services is based on weighted combination of his/her satisfaction with the media type and the individual QoS parameter value of the service. Using all possible combinations of QoS parameters of all available services, we select the combination that generates the maximum satisfaction within the restrictions of the device where the service is running and the preferences of the user. More details about the selection algorithm are given in [8].

2.4 Support for Service Mobility

During a communication session, a nomadic user in a ubiquitous environment might move away from one device and get closer to another device that provides a similar service. Service mobility is required since the life span of the communication session might be longer than the time that the currently used device is available in the user's WPAN. When a device that is currently used becomes unavailable, the Personal Agent should switch to another device providing a similar service if available or it should inform the user about the disappearance of the service. For instance, a user moving away from his computer and entering the conference room should have, if desired, the multimedia session transferred from his computer to the TV and stereo system in the conference room. If the conference room does not have a TV set, the user should be warned that the video display service would be discontinued if he/she stays in the conference room.

Since our architecture is designed to support nomadic users in ubiquitous environments, the architecture supports service mobility through service hand-off, transparently to the user. A smooth transparent service handoff requires continuous discovery of currently available services and updating of the service registry to provide smooth service transfer. The SDA periodically updates the information in the local service registry (SR). Updates are carried out only when a communication session is in progress. When a connection to a service is fading, the SDA informs the QSNA about a replacement service, which sends the coordinates of the replacement service to the other party.

The Personal Agent may also use several heuristics during service selection such as selecting the services that have the longest life span, even with lower quality, in order to avoid service disruption and minimize the number of service hand-off.

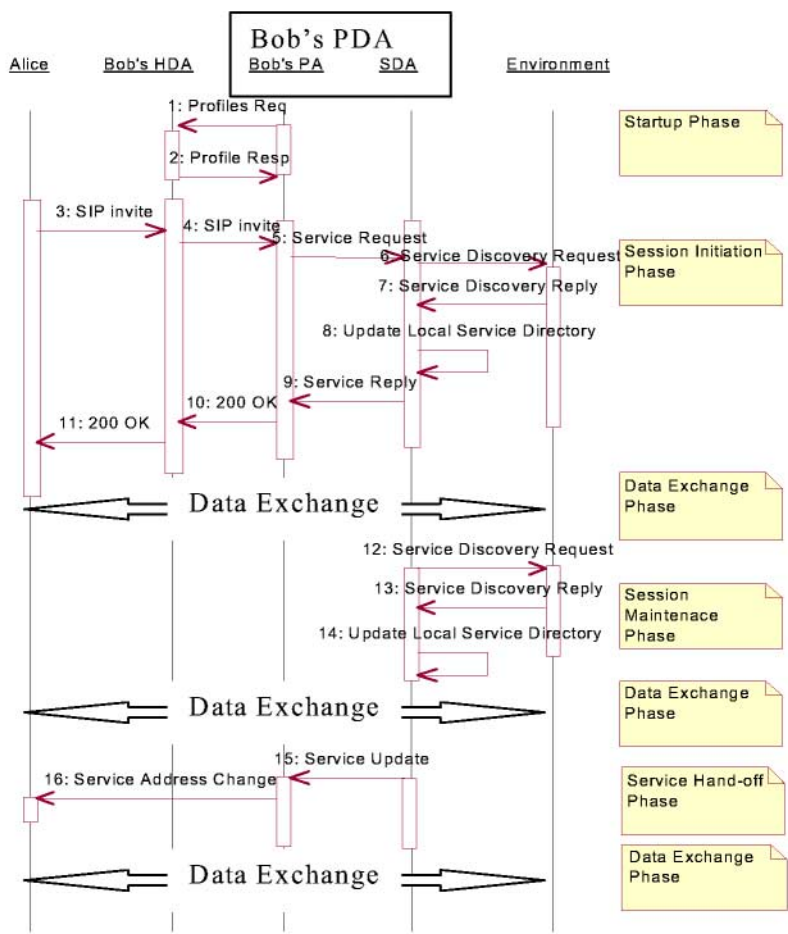


Fig. 2. Session establishment based on the Personal Agent

3 Usage Scenario (Continued)

In this section, we will elaborate more on the scenario presented in Section 1. We will assume that the SIP [18] signaling protocol is used to establish and maintain the communication session. The scenario of Alice trying to reach Bob, who is in the lounge area, is divided into five phases (Fig. 2), with the first phase executed only once when Bob switches ON his PDA. We will assume that Bob has enabled the service mobility option. Due to the space limitation, we will only give a short description of each phase:

- **Startup Phase:** The Personal Agent retrieves the user's profile from the Home Directory Agent (**Messages 1-2**).
- **Session Initiation Phase:** Bob's Home Directory Agent forwards the request to the Personal Agent (**Messages 3-5**). The SDA uses the service discovery protocol to discover the available services for the session and update the SR (**Messages 6-8**). The QSNA selects from the SR the services for the session based on the session requirements, the user profile, network condition, and device/service profile, as described in Section 2.4. The QSNA might also mix-and-match several devices to provide compound services. The Personal Agent sends back to Alice the address of the selected services. (**Messages 9-11**)
- **Data Exchange Phase:** The data is exchanged between Alice's device and the selected devices from Bob's environment.
- **Session Maintenance Phase:** As long as the session is still running, the SDA periodically queries the environment for services that are similar to the services used in the session. This information is used to update the SR in order to reduce the delay in service mobility. (See Section 2.4 for more details). The SDA will also keep querying for services that temporary were not available but were optional for the session. When Bob moves to the conference room, the SDA detects all the audio and video services of the conference room. (**Messages 12-14**)
- **Service Hand-off Phase:** In case a service that is currently used becomes unavailable because of the mobility of the user, the SDA informs the QSNA of the replacement service(s) (in this scenario, the replacement services are the services of the conference room). The QSNA in turns sends an update message to Alice with the new address of the service. (**Messages 15-16**)

4 Conclusion

In this paper, we have presented an architecture for supporting personal mobility in ubiquitous environments. The architecture allows nomadic users to benefit from the availability of large number of hidden services in a ubiquitous environment to establish communication sessions. To construct this architecture, we introduced a new component that we called the Personal Agent (PA) that acts on behalf of the user during the service discovery and selection process. The Personal Agent also provides support for service mobility through periodic updates of currently available services into a local service registry. We have also shown the functionality of the Personal Agent during a typical communication session using an example scenario. Presently, we have implemented the architecture and we are studying its feasibility and usability.

Acknowledgment. The authors would like to thank Eric Zhen Zhang for his helpful comments and for implementing a prototype of the architecture.

References

1. Schulzrinne, H.: Personal mobility for multimedia services in the Internet. In European Workshop on Interactive Distributed Multimedia Systems and Services (IDMS), (Berlin, Germany), Mar. 1996.
2. <http://www.ubiq.com/weiser>
3. Haartseen, J., Naghshineh, M., Inouye, J.: Bluetooth: Vision, Goals, and Architecture. *ACM Mobile Computing and Communications Review*, Vol. 2, No. 4, October 1998, pp. 38–45
4. Anerousis, N. et. al.: TOPS: An architecture for telephony over packet networks,” *IEEE Journal of Selected Areas in Communications*, Jan. 1999
5. Roussopoulos, M., Maniatis, P., Swierk, E., Lai, K., Appenzeller, G., Baker, M.: Person-level Routing in the Mobile People Architecture. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999
6. Kahane, O., Petrack, S.: Call Management Agent System: Requirements, Function, Architecture and Protocol. IMTC Voice over IP Forum Submission VOIP97-010, 1997
7. Wang, H. et. al.: ICEBERG: An Internet-core Network Architecture for Integrated Communications. *IEEE Personal Communications* (2000): Special Issue on IP-based Mobile Telecommunication Networks
8. He, X., El-Khatib, K., Bochmann, G.v.: A communication services infrastructure including home directory agents. U. of Ottawa, Canada. May 2000
9. Barbeau, M., Azondekon, V., Liscano, R.: Service Selection in Mobile Computing Based on Proximity Confirmation Using Infrared. *MICON 2002*
10. Schneider, G., Hoymann, C., Goose, S.: Ad-hoc Personal Ubiquitous Multimedia Services via UpnP. *Proceedings of the IEEE International Conference on Multimedia and Exposition*, Tokyo, Japan; Aug. 2001
11. Beigl, M., Schmidt, A., Lauff, M., Gellersen, H.W.: The UbicompBrowser. *Proceedings of the 4th ERCIM Workshop on User Interfaces for All*, October 1998, Stockholm, Sweden
12. IEEE802.11: <http://grouper.ieee.org/groups/802/11/main.html>
13. JINI (TM): [Http://java.sun.com/product/JINI/](http://java.sun.com/product/JINI/). 1998
14. Bluetooth Special Interest Group (SIG): Service Discovery Protocol,” *SIG Specification version 1.0, Volume 1 Core, part E*, pp 233–384
15. Guttman, E., Perkins, C., Veizades, J., Day, M.: Service Location Protocol. Version 2. <http://ietf.org/rfc/rfc2608.txt>.
16. Horvitz, E., Jacobs, A., Hovel, D.: Attention-Sensitive Alerting. In: *Proceedings of UAI '99*, Stockholm, Sweden, July 1999, pp. 305–313. San Francisco: Morgan Kaufmann
17. A. Richards, A., Rogers, g., Witana, V., Antoniadis, M.: Mapping User Level QoS from a Single Parameter. In *Second IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, Versailles, November 1998
18. Handley, M., Schulzrinne, H., Scholler, E., Rosenberg, J.: SIP: Session Initiation Protocol. RFC2543, IETF, March 1999

Dynamic Layouts for Wireless ATM^{*}

Michele Flammini¹, Giorgio Gambosi², Alessandro Gasparini¹, and
Alfredo Navarra^{1,3}

¹ Dip. di Informatica, University of L'Aquila, Italy.
{flammini,gasparin,navarra}@di.univaq.it

² Dip. di Matematica, University of Rome "Tor Vergata", Italy.
gambosi@mat.utovrm.it

³ MASCOTTE project, INRIA/Université de Nice-Sophia Antipolis, France.
anavarra@sophia.inria.fr

Abstract. In this paper we present a new model able to combine quality of service (QoS) and mobility aspects in wireless ATM networks. Namely, besides the standard parameters of the basic ATM layouts, we introduce a new one, that estimates the time needed to reconstruct the virtual channel of a wireless user when it moves through the network. QoS guarantee dictates that the rerouting phase must be imperceptible. Therefore, a natural combinatorial problem arises in which suitable trade-offs must be determined between the different performance measures. We first show that deciding the existence of a layout with maximum hop count h , load l and distance d is NP-complete, even in the very restricted case $h = 2$, $l = 1$ and $d = 1$. We then provide optimal layout constructions for basic interconnection networks, such as chains and rings.

1 Introduction

Wireless ATM networks are emerging as one of the most promising technologies able to support users mobility while maintaining the QoS offered by the classical ATM protocol for Broadband ISDN [2]. The mobility extension of ATM gives rise to two main application scenarios, called respectively *End-to-End* WATM and WATM *Interworking* [13]. While the former provides seamless extension of ATM capabilities to users by allowing ATM connections that extend until the mobile terminals, the latter represents an intermediate solution used primarily for high-speed transport over network backbones by exploiting the basic ATM protocol with additional mobility control capabilities. Wireless independent subnets are connected at the borders of the network backbone by means of specified ATM interface nodes, and users are allowed to move among the different wireless

^{*} Work supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), by the EU RTN project ARACNE, by the Italian project REAL-WINE, partially funded by the Italian Ministry of Education, University and Research, by the French MASCOTTE project I3S-CNRS/INRIA/Univ. Nice-Sophia Antipolis and by the Italian CNR project CNRG003EF8 – “Algoritmi per Wireless Networks” (AL-WINE).

subnets. In both scenarios, the mobility facility requires the efficient solution of several problems, such as handover (users movement), routing, location management, connection control and so forth. A detailed discussion of these and other related issues can be found in [13,6,5,19,17].

The classical ATM protocol for Broadband ISDN is based on two types of predetermined routes in the network: *virtual paths* or VPs, constituted by a sequence of successive edges or physical links, and *virtual channels* or VCs, each given by the concatenation of a proper sequence of VPs [15,14,18]. Routing in virtual paths can be performed very efficiently by dedicated hardware, while a message passing from one virtual path to another one requires more complex and slower elaboration.

A graph theoretical model related to this ATM design problem has been first proposed in [12,7]. In such a framework, the VP layouts determined by the VPs constructed on the network are evaluated mainly with respect to two different cost measures: the *hop count*, that is the maximum number of VPs belonging to a VC, which represents the number of VP changes of messages along their route to the destination, and the *load*, given by the maximum number of virtual paths sharing an edge, that determines the size of the VP routing tables (see, e.g., [8]). For further details and technical justifications of the model for ATM networks see for instance [1,12].

While the problem of determining VP layouts with bounded hop count and load is NP-hard under different assumptions [12,9], many optimal and near optimal constructions have been given for various interconnection networks such as chain, trees, grids and so forth [7,16,10,11,20,4] (see [21] for a survey).

In this paper we mainly focus on handover management issues in wireless ATM. In fact, they are of fundamental importance, as the virtual channels must be continually modified due to the terminals movements during the lifetime of a connection. In particular, we extend the model of [12,7] in order to combine QoS and mobility aspects in wireless ATM networks.

Typical handover managements issues are the path extension scheme, in which a VC is always extended by a virtual path during a handover [5], or the anchor-based rerouting and the nearest common node rerouting [13,3], that involve the deletion of all the VPs of the old VC and the addition of all the VPs of the new one after a common prefix of the two VCs. Other handover strategies can be found in [13,6,5].

Starting from the above observations, besides the standard hop count and load performance measures, we introduce the new notion of virtual channel distance, that estimates the time needed to reconstruct a virtual channel during a handover phase. In order to make the rerouting phase imperceptible to users and thus to obtain a sufficient QoS, the maximum distance between two virtual channels must be maintained as low as possible. Therefore, a natural combinatorial problem arises in which suitable trade-offs must be determined between the different performance measures.

The paper is organized as follows. In the next section we introduce the model, the notation and the necessary definitions. In Section 3 we provide hardness

results for the layout construction problem. In Section 4 and 5 we provide optimal layouts for chains and rings, respectively. Finally, in Section 6, we give some conclusive remarks and discuss some open questions.

2 The WATM Model

We model the network as an undirected graph $G = (V, E)$, where nodes in V represent switches and edges in E are point-to-point communication links. In G there exists a subset of nodes $U \subseteq V$ constituted by cells with corresponding radio stations, i.e., switches adapted to support mobility and having the additional capability of establishing connections with the mobile terminals. A distinguished source node $s \in V$ provides high speed services to the users moving along the network. We observe that, according to the wireless nature of the system, during the handover phase mobile terminals do not necessarily have to move along the network G , but they can switch directly from one cell to another, provided that they are adjacent in the physical space. It is thus possible to define a (connected) adjacency graph $A = (U, F)$, whose edges in F represent adjacencies between cells.

A layout Ψ for $G = (V, E)$ with source $s \in V$ is a collection of paths in G , termed *virtual paths* (VPs for short), and a mapping that defines, for each cell $u \in U$, a virtual channel $VC_\Psi(u)$ connecting s to u , i.e., a collection of VPs whose concatenation forms a shortest path in G from s to u .

Definition 1. [12] The hop count $h_\Psi(u)$ of a node $u \in U$ in a layout Ψ is the number of VPs contained in $VC_\Psi(u)$, that is $|VC_\Psi(u)|$. The maximal hop count of Ψ is $\mathcal{H}_{\max}(\Psi) \equiv \max_{u \in U} \{h_\Psi(u)\}$.

Definition 2. [12] The load $l_\Psi(e)$ of an edge $e \in E$ in a layout Ψ is the number of VPs $\psi \in \Psi$ that include e . The maximal load $\mathcal{L}_{\max}(\Psi)$ of Ψ is $\max_{e \in E} \{l_\Psi(e)\}$.

As already observed, when passing from a cell $u \in U$ to an adjacent one $v \in U$, the virtual channel $VC_\Psi(v)$ must be reconstructed from $VC_\Psi(u)$ changing only a limited number of VPs. Once fixed $VC_\Psi(u)$ and $VC_\Psi(v)$, denoted as $VC_\Psi(u, v)$ the set of VPs in the subchannel given by the longest common prefix of $VC_\Psi(u)$ and $VC_\Psi(v)$, this requires the deletion of all the VPs of $VC_\Psi(u)$ that occur after $VC_\Psi(u, v)$, plus the addition of all the VPs of $VC_\Psi(v)$ after $VC_\Psi(u, v)$. The number of removed and added VPs, denoted as $D(VC_\Psi(u), VC_\Psi(v))$, is called the distance of $VC_\Psi(u)$ and $VC_\Psi(v)$ and naturally defines a channel distance measure d_Ψ between pairs of adjacent nodes in A .

Definition 3. The channel distance of two nodes u and v such that $\{u, v\} \in F$ (i.e., adjacent in A) is $d_\Psi(u, v) = D(VC_\Psi(u), VC_\Psi(v)) = h_\Psi(u) + h_\Psi(v) - 2|VC_\Psi(u, v)|$. The maximal distance of Ψ is $\mathcal{D}_{\max}(\Psi) \equiv \max_{\{u, v\} \in F} \{d_\Psi(u, v)\}$.

It is now possible to give the following definition on WATM layouts.

Definition 4. A layout Ψ with $\mathcal{H}_{\max}(\Psi) \leq h$, $\mathcal{L}_{\max}(\Psi) \leq l$ and $\mathcal{D}_{\max}(\Psi) \leq d$ is a $\langle h, l, d \rangle$ -layout for G , s and A .

In the following, when the layout Ψ is clear from the context, for simplicity we will drop the index Ψ from the notation. Moreover, we will always assume that all the VPs of Ψ are contained in at least one VC. In fact, if such property does not hold, the unused VPs can be simply removed without increasing the performance measures h , l and d .

3 Hardness of Construction

In this section we show that constructing optimal dynamic layouts is in general an NP-hard problem, even for the very simple case $h = 2$ and $l = d = 1$.

Notice that when $d = 1$, for any two cells $u, v \in U$ adjacent in $A = (U, F)$, during an handover from u to v by definition only one VP can be modified. This means that in every $\langle h, l, 1 \rangle$ -layout Ψ , either $VC(v)$ is a prefix of $VC(u)$ and thus $VC(v)$ is obtained from $VC(u)$ by adding a new VP from u to v , or vice versa. In any case, a VP between u and v must be contained in Ψ . As a direct consequence, the virtual topology defined by the VPs of Ψ coincides with the adjacency graph A .

Theorem 1. *Given a network $G = (V, E)$, a source $s \in V$ and an adjacency graph $A = (U, F)$, deciding the existence of a $\langle 2, 1, 1 \rangle$ -layout for G , s and A is an NP-complete problem.*

For $h = 1$, any l and any d , the layout construction problem can be solved in polynomial time by exploiting suitable flow constrictions like the ones presented in [9].

4 Optimal Layouts for Chain Networks

In this section we provide optimal layouts for chain networks. More precisely, we consider the case in which the physical graph is a chain C_n of n nodes, that is $V = \{1, 2, \dots, n\}$ and $E = \{\{v, v+1\} | 1 \leq v \leq n-1\}$, and the adjacency graph A coincides with C_n . Moreover, without loss of generality, we take the leftmost node of the chain as the source, i.e. $s = 1$, as otherwise we can split the layout construction problem into two equivalent independent subproblems for the left and the right hand sides of the source, respectively. Finally, we always assume $d > 1$, as by the same considerations of the previous section the virtual topology induced by the VPs of any $\langle h, l, 1 \rangle$ -layout Ψ coincides with the adjacency graph A and thus with C_n . Therefore, the largest chain admitting a $\langle h, l, 1 \rangle$ -layout is such that $n = h + 1$.

In the following we denote by $\langle u, v \rangle$ the unique VP corresponding to the shortest path from u to v in C_n and by $\langle \langle s, v_1 \rangle \langle v_1, v_2 \rangle \dots \langle v_k, v \rangle \rangle$ or simply $\langle s, v_1, v_2, \dots, v_k, v \rangle$ the virtual channel $VC(v)$ of v given by the concatenation of the VPs $\langle s, v_1 \rangle$, $\langle v_1, v_2 \rangle$, ..., $\langle v_k, v \rangle$. Clearly, $s < v_1 < v_2 < \dots < v_k < v$.

Definition 5. *Two VPs $\langle u_1, v_1 \rangle$ and $\langle u_2, v_2 \rangle$ are crossing if $u_1 < u_2 < v_1 < v_2$. A layout Ψ is crossing-free if it does not contain any pair of crossing VPs.*

Definition 6. A layout Ψ is *canonic* if it is crossing-free and the virtual topology induced by its VPs is a tree.

According to the following definition, a $\langle h, l, d \rangle$ -layout for chains is optimal if it reaches the maximum number of nodes.

Definition 7. Given fixed h, l, d and a $\langle h, l, d \rangle$ -layout Ψ for a chain C_n , Ψ is *optimal* if no $\langle h, l, d \rangle$ -layout exists for any chain C_m with $m > n$.

We now prove that for every h, l, d , the determination of an optimal $\langle h, l, d \rangle$ -layout can be restricted to the class of the canonic layouts.

Theorem 2. For every h, l, d , any optimal $\langle h, l, d \rangle$ -layout for a chain is canonic.

Motivated by Theorem 2, in the remaining part of this section we focus on canonic $\langle h, l, d \rangle$ -layouts for chains, as they can be the only optimal ones.

Let us say that a tree is ordered if it is rooted and for every internal node a total order is defined on its children. As shown in [12], an ordered tree induces in a natural way a canonic layout and vice versa.

Therefore, there exists a bijection between canonic layouts and ordered trees.

We now introduce a new class of ordered trees $\mathcal{T}(h, l, d)$ that allows to completely define the structure of an optimal $\langle h, l, d \rangle$ -layout. Informally, denoted as $\mathcal{T}(h, l)$ the ordered tree corresponding to optimal layouts with maximum hop count h and load l without considering the distance measure [11], $\mathcal{T}(h, l, d)$ is a maximal subtree of $\mathcal{T}(h, l)$ with the additional property that the distance between two adjacent nodes in the preorder labelling of the ordered tree, and thus between two adjacent nodes in the induced layout, is always at most d . Moreover, the containment of $\mathcal{T}(h, l, d)$ in $\mathcal{T}(h, l)$ guarantees that the hop count h and the load l are not exceeded in the induced layout.

The definition of $\mathcal{T}(h, l, d)$ is recursive and the solution of the associated recurrence gives the exact number of the nodes reached by an optimal $\langle h, l, d \rangle$ -layout. Before introducing $\mathcal{T}(h, l, d)$, let us define another ordered tree that is exploited in its definition.

Definition 8. Given any h, l, d , $T(h, l, d)$ is an ordered tree defined recursively as follows. $T(h, l, d)$ is obtained by joining the roots of $\min\{h, d - 1\}$ subtrees $T(i, l - 1, d)$ with $h - \min\{h, d - 1\} + 1 < i \leq h$ in such a way that the root of $T(i - 1, l - 1, d)$ is the rightmost child of the root of $T(i, l - 1, d)$. A last node is finally added as the rightmost child of $T(h - \min\{h, d - 1\} + 1, l - 1, d)$. Trees $T(0, l, d)$ and $T(h, 0, d)$ consist of a unique node.

Definition 9. The ordered tree $\mathcal{T}(h, l, d)$ is defined recursively as the join of the roots of the tree $\mathcal{T}(h - 1, l, d)$ and the tree $T(h, l - 1, d)$ in such a way that the root of $\mathcal{T}(h - 1, l, d)$ is the rightmost child of the root of $T(h, l - 1, d)$. Trees $\mathcal{T}(0, l, d)$ and $\mathcal{T}(h, 0, d)$ consist of a unique node.

The following lemma establishes that $\mathcal{T}(h, l, d)$ is the ordered tree induced by an optimal $\langle h, l, d \rangle$ -layout.

Lemma 1. *The layout Ψ induced by $\mathcal{T}(h, l, d)$ is a $\langle h, l, d \rangle$ -layout. Moreover, every canonic $\langle h, l, d \rangle$ -layout Ψ induces an ordered subtree of $\mathcal{T}(h, l, d)$.*

Let $\mathcal{T}_n(h, l, d)$ and $T_n(h, l, d)$ denote the number of nodes in $\mathcal{T}(h, l, d)$ and in $T(h, l, d)$, respectively. Directly from Definition 8 and 9, it follows that $\mathcal{T}_n(h, l, d) = T_n(h, l - 1, d) + T_n(h - 1, l, d) = \sum_{k=0}^h T_n(k, l - 1, d)$, where the value of every $T_n(k, l - 1, d)$ for $0 \leq k \leq h$ is obtained by the following recursive equation:

$$T_n(h, l, d) = \begin{cases} 1 & \text{if } l = 0 \text{ or } h = 0, \\ 1 + \sum_{j=0}^{\min\{h, d-1\}-1} T_n(h - j, l - 1, d) & \text{otherwise.} \end{cases}$$

Before solving the above recurrence, we recall that given $n+1$ positive integers $m, k_1, \dots, k_n$ such that $m = k_1 + \dots + k_n$, the multinomial coefficient $\binom{m}{k_1, \dots, k_n}$ is defined as $\frac{m!}{k_1! \cdot k_2! \cdot \dots \cdot k_n!}$.

Lemma 2. *For every h, l, d , $T_n(h, l, d) =$*

$$\sum_{i=0}^l \sum_{j=0}^{h-1} \sum_{\substack{0 \leq k_{d-2} \leq k_{d-3} \leq \dots \leq k_2 \leq k_1 \leq i \\ k_1 + k_2 + \dots + k_{d-2} = j}} \binom{i}{i-k_1, k_1-k_2, \dots, k_{d-3}-k_{d-2}, k_{d-2}}.$$

The following theorem is a direct consequence of Lemma 1, Lemma 2 and Definition 9.

Theorem 3. *For every h, l, d , the maximum number of nodes reachable on a chain network by a $\langle h, l, d \rangle$ -layout is $\mathcal{T}_n(h, l, d) = 1 + \sum_{k=1}^h T_n(k, l - 1, d)$.*

More details will be shown in the full version of the paper.

5 Optimal Layouts for Ring Networks

In this section we provide optimal layouts for ring networks R_n with $V = \{0, 1, \dots, n-1\}$ and $E = \{\{i, (i+1) \bmod n\} | 0 \leq i \leq n-1\}$. Again we assume that the adjacency graph A coincides with R_n and without loss of generality we take $s = 0$ as the source node. Moreover, we let $d > 1$, since as remarked in Section 3, no layout with maximum distance 1 exists for cyclic adjacency graphs.

Notice that in any $\langle h, l, d \rangle$ -layout Ψ for R_n , by the shortest path property, if n is odd the nodes in the subring $[1, \lfloor \frac{n}{2} \rfloor]$ are reached in one direction from the source, say clockwise, while all the remaining ones anti-clockwise. This means that Ψ can be divided into two separated sublayouts Ψ_c and Ψ_a respectively for the subchains of the nodes reached clockwise in Ψ , that is $[0, \lfloor \frac{n}{2} \rfloor]$, and anticlockwise, that is from $\lceil \frac{n}{2} \rceil$ to 0 in clockwise direction, extremes included. However, the results of the previous section for chains do not extend in a trivial way, as a further constraint exists for the final nodes $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$, that are adjacent in A and thus must be at distance at most d in Ψ . A similar observation holds when n is even.

As for chains, let us say that a $\langle h, l, d \rangle$ -layout Ψ for rings is optimal if it reaches the maximum number of nodes. Moreover, let us call Ψ canonic if the clockwise and anticlockwise sublayouts Ψ_c and Ψ_a are both crossing-free and the virtual topologies induced by their VPs are trees. The following lemma is the equivalent of Theorem 2 for rings.

Lemma 3. *For every h, l, d , there exists an optimal $\langle h, l, d \rangle$ -layout for rings that is canonic.*

Starting from Lemma 3, we generalize the ordered tree $\mathcal{T}(h, l, d)$ to $\mathcal{T}(h, l, d, t)$ by adding a further parameter $t \leq h$, which fixes the hop count of the rightmost leaf to t . Roughly speaking, $\mathcal{T}(h, l, d, h) = \mathcal{T}(h, l, d)$ and $\mathcal{T}(h, l, d, d - 1) = T(h, l, d)$. More precisely, $\mathcal{T}(h, l, d, t)$ is defined recursively as the join of the roots of $\min\{h, t\}$ subtrees $T(i, l - 1, d)$ for $h - \min\{h, t\} < i \leq h$ in such a way that for $i < h$ the root of a $T(i, l - 1, d)$ is the rightmost child of the root of a $T(i + 1, l - 1, d)$, plus a final node as rightmost child of $T(h - \min\{h, t\} + 1, l - 1, d)$. Thus, $\mathcal{T}_n(h, l, d, t) = 1 + \sum_{k=h-\min\{h,t\}+1}^h T_n(k, l - 1, d)$.

Lemma 1 extends directly to $\mathcal{T}(h, l, d, t)$, that in turn corresponds to an optimal $\langle h, l, d \rangle$ -layout for a chain with the further property that the rightmost node (opposite of the source) has hop count t . Therefore, it is possible to prove the following theorem.

Theorem 4. *The maximum number of nodes reachable on a ring network by a $\langle h, l, d \rangle$ -layout is $2\mathcal{T}_n(h, l, d, \lfloor \frac{d}{2} \rfloor) - ((d + 1) \bmod 2)$, with $\mathcal{T}_n(h, l, d, \lfloor \frac{d}{2} \rfloor) = 1 + \sum_{k=h-\min\{h, \lfloor \frac{d}{2} \rfloor\}+1}^h T_n(k, l - 1, d)$.*

6 Conclusion

We have extended the basic ATM model presented in [12,7] to cope with QoS and mobility aspect in wireless ATM networks. This is obtained by adding a further measure, the VCs distance, that represents the time needed to reconstruct connecting VCs when handovers occur and must be maintained as low as possible in order to avoid the rerouting mechanism to be appreciated by the mobile users. We have shown that finding suitable trade-offs between the various performance measures is in general an intractable problem, while optimal constructions have been given for chain and ring topologies.

Among the various questions left open, we have the extension of our results to more general topologies. Moreover, another worth investigating issue is the determination of layouts in which the routed paths are not necessarily the shortest ones, but have a fixed stretch factor or even unbounded length. Finally, all the results should be extended to other communication patterns like all-to-all.

References

1. S. Ahn, R.P. Tsang, S.R. Tong, and D.H.C. Du. Virtual path layout design on ATM networks. In *Proceedings of INFOCOM*, pages 192–200, 1994.

2. I. Akyildiz, J. McNair, J. Ho, H. Uzunalioglu, and W. Wang. Mobility management in next-generation wireless systems. In *Proceedings of the IEEE*, volume 87, pages 1347–1384, 1999.
3. Bora A. Akyol and Donald C. Cox. Rerouting for handoff in a wireless ATM network. In *Proceedings of the 5th IEEE International Conference on Universal Personal Communications (ICUPC)*, 1996.
4. L. Becchetti, P. Bertolazzi, C. Gaibisso, and G. Gambosi. On the design of efficient ATM schemes. In *Proceedings of SOFSEM*, volume 1338 of *LNCS*, pages 375–382. Springer-Verlag, 1997.
5. M. Cheng, S. Rajagopalan, L. Chang, G. Pollini, and M. Barton. PCS mobility support over fixed ATM networks. *IEEE Communications Magazine*, 35:82–92, 1997.
6. C. Chrysostomou, A. Pitsillides, and F. Pavlidou. A survey of wireless ATM handover issues. In *Proceedings of the 3th International Symposium of 3G Infrastructure and Services (3GIS)*, pages 34–39, 2001.
7. I. Cidon, O. Gerstel, and S. Zaks. A scalable approach to routing in ATM networks. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, volume 857 of *LNCS*, pages 209–222. Springer-Verlag, 1994.
8. R. Cohen and A. Segall. Connection management and rerouting in ATM networks. In *Proceedings of INFOCOM*, pages 184–191, 1994.
9. T. Eilam, M. Flammini, and S. Zaks. A complete characterization of the path layout construction problem for ATM networks with given hop count and load. In *Proceedings of the 24th Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *LNCS*, pages 527–537. Springer-Verlag, 1997.
10. O. Gerstel, I. Cidon, and S. Zaks. The layout of virtual paths in ATM networks. *IEEE/ACM Transactions on Networking*, 4(6):873–884, 1996.
11. O. Gerstel, A. Wool, and S. Zaks. Optimal layouts on a chain ATM network. In *Proceedings of the 3th Annual European Symposium on Algorithms (ESA)*, volume 979 of *LNCS*, pages 508–522. Springer-Verlag, 1995.
12. O. Gerstel and S. Zaks. The virtual path layout problem in fast networks. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 235–243, 1994.
13. Jerry D. Gibson. *The Mobile Communications Handbook, Second Edition*. CRC Press in cooperation with IEEE Press, 1999.
14. R. Händler and M.N. Huber. *Integrated Broadband Networks: an introduction to ATM-based networks*. Addison-Wesley, 1991.
15. ITU recommendation. I series (B-ISDN), Blue Book, November 1990.
16. E. Kranakis, D. Krizanc, and A. Pelc. Hop-congestion tradeoffs in ATM networks. In *Proceedings of the 9th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 662–668, 1995.
17. G. Parry. Wireless ATM MAC protocols - a literature survey. In *WARP Project - URL <http://vera.ee.und.ac.za/coe/warp>*, 1999.
18. C. Partridge. *Gigabit Networking*. Addison Wesley, 1994.
19. D. Sobirk and J. M. Karlsson. A survey of wireless ATM MAC protocols. In *Proceedings of the International Conference on the Performance and Management of Complex Communication Networks (PMCCN)*. Chapman & Hall, 1997.
20. L. Stacho and I. Vrto. Virtual path layouts for some bounded degree networks. In *Proceedings of the 3rd Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 269–278. Carleton University Press, 1996.
21. S. Zaks. Path layouts in ATM networks. In *Proceedings of SOFSEM*, volume 1338 of *LNCS*, pages 144–160. Springer-Verlag, 1997.

Modeling Context-Aware Behavior by Interpreted ECA Rules

Wolfgang Beer¹, Volker Christian¹, Alois Ferscha¹, and Lars Mehrmann²

¹ Johannes Kepler University Linz, Department for Practical Informatics,
Altenbergerstrasse 69, 4040 Linz Austria
wolfgang.beer@jku.at, {voc, ferscha}@soft.uni-linz.ac.at
<http://www.soft.uni-linz.ac.at>

² Siemens AG, CT SE2, Otto-Hahn-Ring 6, 81730 Munich, Germany
mehrmann@mchp.siemens.de

Abstract. The software architecture of distributed systems is about to change due to new requirements of modern mobile devices. New network techniques, like ad-hoc radio communication or peer-to-peer networks allow mobile devices to sense their environment and to interact with other devices dynamically. This paper presents a cutting-edge way to describe objects and their interaction, also called context, as well as the possibility to configure such interaction scenarios. A lookup mechanism collects information about the environment and a role-based classification is responsible for identifying possible interaction partners. Furthermore the configuration of scenario behavior with context rules is introduced. Finally a comparison with already existing context frameworks is given and a practical emergency scenario configuration is shown.

1 Introduction

More and more radio based wireless networks seem to become the standard for mobile device interaction. For many tasks it is not necessary to be a participant of a global area network, but to connect to an ad hoc network between several mobile devices. Even personal area networks (PANs) get more and more popular, e. g. in an ad hoc communication between a cellular phone, a handheld organizer and a headset. Depending on fixed networks, traditional software architecture for distributed applications contains aspects that harden its use in ad hoc communication environments. Client server architecture depends on a reliable central server that offers services. In ad hoc communication the network structure changes rapidly and normally it is not possible to set up a globally accessible server. Therefore a mobile device has to act as a server and as a client at the same time, to provide services like voice phone, or to use services like calendar function or headset output.

The first question that comes to our mind, when using ad hoc communication, is how to find other entities and how to describe their functionality. Moreover, it is necessary to know how to describe the kind of interaction, which is possible with this new entity

that appeared. A set of devices is possibly able to solve higher-level problems that the single device is not able to fulfill. Therefore, the single device has to be able to get context information about its environment, i.e. to find compatible partners, in order to solve joint problems. Embedded or personal mobile devices have to exchange sensor and actuator information, to model a digital representation of the real world. With this representation, digital devices should get sensitivity for the context of a human user, e. g. the location or the environment noise and the problems the user would like to solve, e. g. to switch to a louder sound profile. The digital devices could analyze the historical development of this context representation, to propose solutions for future problems. So, embedded wireless devices offer the chance to improve the kind of interaction between human users and their digital helpers [10][1].

2 Description of Digital and Non-digital Objects

In traditional applications the programmer is responsible for connecting client and server programs, to solve a certain kind of problem. The programmer is aware of the interfaces, the network addresses, the functionality and maybe the reliability of the different participants.

In modern peer-to-peer, ad-hoc communication environments the programmer does not automatically get this information. If a device appears in another communication range, it is totally unknown, which device it is and what services it offers, or demands. Therefore it is necessary to get a semantic description of the unknown objects attributes and services. Various research projects deal with the problem of semantic information modeling and distribution, often with the goal to extend the possibilities of smart software agents and reasoning mechanisms.

Originally, the XML based W3C standard RDF (Resource Description Framework) [8] and its schema language were developed to meet the requirements of semantic information retrieval on the web. Various research projects deal also with the topic of joint terminology, like the definition of ontology [11].

Generally, we distinguish between two ways of describing and classifying objects:

- **Closed world assumption:** Data and services are completely known at design time. Therefore, it is easy to define a suitable semantic model, in order to handle this information. It is not possible to add any unknown element in a closed world assumption. Moreover, parts of information are explicitly excluded from the context world model, in order to simplify reasoning [13].
- **Open world assumption:** Data and services are not known at design time. Therefore, it is necessary to derive all the information from the semantic description of the participating agents. This issue is hard to realize and a major problem in AI research [9].

Another important aspect, when we are speaking about the description of objects, is the classification of objects. There are two different methods of classifying objects:

- **Static classification:** The class hierarchy is completely known at design time. Once an object appears, its type either is part of the known class hierarchy, or its type is unknown. The class hierarchy cannot change at runtime.
- **Role based classification:** An unknown object is classified by a library of attribute information. The object's set of attributes is compared with attribute sets that are known. This library of attributes and known attribute sets (we call them attribute templates) can be extended at runtime. Naturally an object is able to fulfill more than one attribute template, so it acts in more than one role. Objects can extend their functionality at runtime, i.e. their attribute template is able to change at dynamically.

To support role-based classification with an open world assumption, our framework is based on the work of XEROX Parc in 1995 [9]. Every object, called entity, is a container for a set of attributes. This set of attributes fully describes the entity's functionality. A specific set of attributes is called an attribute template. It is responsible for classifying an entity at runtime. An entity is likely to act in more than one role. As it is shown in Figure 1, an entity, which owns the attributes {Name, Birthday, SocNumber} would be classified as a person in the European union and in the United States.

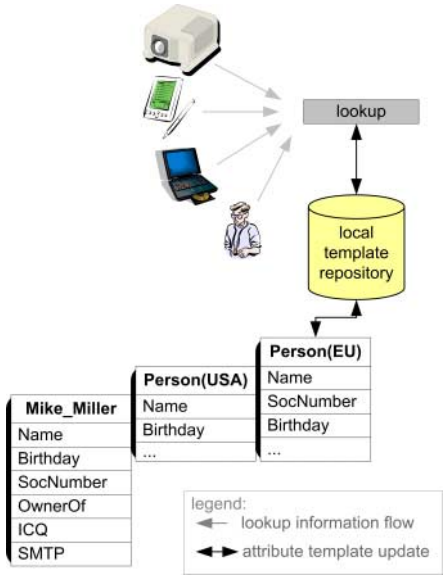


Fig. 1. Entity classification with attribute templates

Without the social security number, the entity would be classified as a person in Europe but not in the United States. So the role based classification of entities is based on the set of attribute templates that are registered locally on the mobile device. The classification of objects is not a task of the context middleware but of the applications that run on it. Imagine a chat application that lists all persons in the local environment. Therefore, the chat application itself has to provide an attribute template that defines how the chat application specifies a person object.

3 Framework Architecture Overview

As already mentioned in section 2, an entity is classified by the set of attributes it provides at runtime. The set of attributes possibly changes at runtime, when an entity loads or unloads attributes over the network or from a local storage. Therefore an entity has to reference a class loader and a transport layer, to load and to deploy new attributes. The class loader itself retrieves class information, either from a local attribute repository or through the transport layer. Because of the recursive nature of the attribute-entity architecture, it is possible for an entity to react as an attribute within another entity, as it is shown in Figure 2.b.).

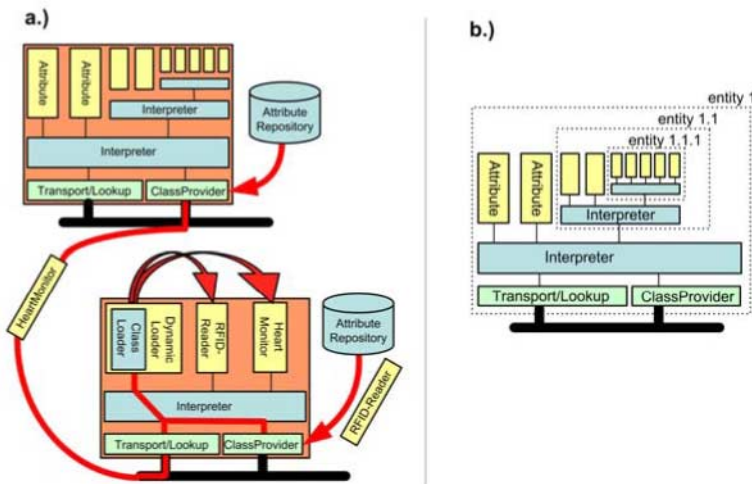


Fig. 2. a.) Shows dynamic attribute loading at run time, while b.) shows a simple recursive attribute-entity architecture

Between every context information flow, including the loading or unloading of attributes, the interpreter checks whether any additional actions should be performed. These additional actions are specified in context rules. *ECA* (Event Condition Action) rules are able to react on certain entity states, where the event specifies the context event an attribute throws [3]. Context rules are described more detailed in Section 4. Figure 2.a.) shows how an entity is able to load an attribute, called Heart Monitor, and how the entity deploys it to another entity over a transport layer. The entities contain a set of attributes, which may be updated at runtime. In this upgrading process special attributes are loaded, which are necessary to fulfill a certain context scenario, see Section 6. The transport layer does not depend on any specific protocol, it can be changed at runtime. At the moment a TCP socket implementation is available as well as a SOAP-HTTP implementation. We plan to implement an additional JXTA transport layer, as far as JXTA provides reasonable performance on mobile devices.

3.1 XML Configuration of Entities

Entities exist in containers, which manage their lookup mechanism, transport and lifecycle. A container has the possibility to host a performance-optimal collection of entities. A desktop computer would probably host a larger set of entities, while a mobile device is limited to a small set. Containers can exchange entities at runtime in order to optimize performance or to minimize network load. To minimize network load, it is possible to host a personal entity on a desktop computer when the user is in the office. When the user leaves his office, his entity description travels with him, hosted on his PDA. At the container startup time, a set of entities is loaded by the *Dynamic Loader* attribute of the container. XML configuration files inform the container, which initial set of entities should be loaded and which initial sets of attributes the entities own. The attribute-specific configuration is also located in separate XML configuration files, which were referenced by the entity configurations. The following figure shows how a simple container configuration is modeled in XML on a mobile device.

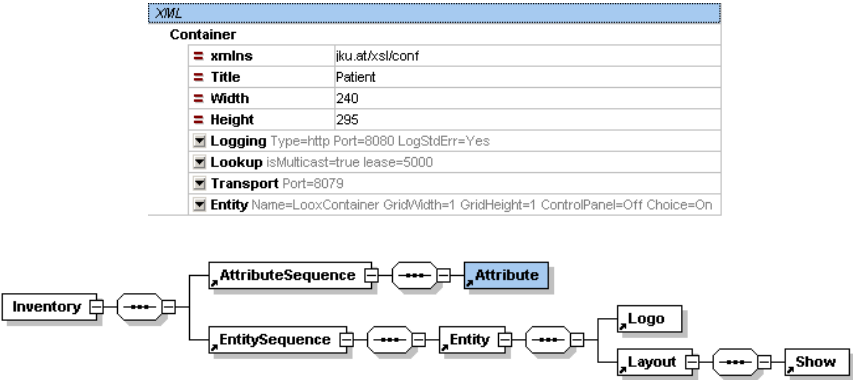


Fig. 3. XML configuration of containers and entities

3.2 Lookup

Communication in ad-hoc mode requires a mechanism that discovers entities, which are inside the communication range of an entity. In order to find a service provider in an ad-hoc network it is necessary to announce the presence of mobile devices in the environment. For a small number of service providers, it is possible to use Broadcast or Multicast IP solutions to announce their presence in the network. Broadcasting of service provider access information is not scalable and therefore not directly used for larger networks. Discovery protocols like SLP (Service Location Protocol) and the discovery mechanism of JINI (Java Intelligent Network Infrastructure) introduce hybrid solutions, where Multicast IP is used for searching in local networks and fixed addresses are used for global service discovery [12].

The lookup mechanism implemented in context framework also uses a hybrid solution where fixed service provider addresses are used for global service discovery. Alternatively, it is possible to use Broadcast or Multicast IP service provider announcement for local area discovery. A context container is responsible for announcing and receiving the presence of entities and their services. Additionally, the context lookup informs about the set of attributes of an entity, which enables the role-based classification. A difference to JINI is, that our solution runs on Java Personal edition version 1.1.8 and is lightweight enough to run on mobile devices. Another difference is, that our lookup solution sends XML based lookup information. The use of XML is a performance drawback, but one of the major disadvantages of JINI is, that only Java based service providers are supported, through the use of Java Object Serialization.

4 Dynamic Interaction of Entities through Context Rules

In any context scenario, it is important that the entities can update or change their relations at runtime. As it is described in Section 2, an entity is able to dynamically classify unknown objects that appear in its environment with attribute templates. In this connection we defined a context rule:

“A context rule specifies the reaction of an entity on a specified context state and defines therefore the event based interaction between entities”

An entity is able to sense its environment with sensors and to change the information in the scenarios world model, as shown in Figure 3.a.). The sensing mechanism could be either event-triggered or time-triggered, as it was already observed in [2]. The framework has to transform the lower level sensor input into information that fits in the framework's context world model.

In the context framework the set of entity attributes contain the high-level context information about an object. So the values of these attributes provide an entity state. A context event is triggered by an attribute, if its state changes and it would like to inform its entity. To react on a specific entity state, it is necessary to define a context rule inside the entity. Context rules define an ECA (Event-Condition-Action) matching service [3], as it is already known from active database research [7].

4.1 Context Rule Syntax

We defined a configuration syntax, which is easy to understand for humans and more compact than XML-coded ECA definitions:

```
Rules = "rules" [ Targets ] "{ " { Rule } " }".
Targets = "for" EntityOrTemplate { ", " EntityOrTemplate }.
Rule = "on" Event [ "if" Condition ] Action
```

The non terminal symbol EntityOrTemplate describes an entity or a template which defines a set of similar entities as target for the rule. Thus it is possible to bind the rule to a specific entity or to a specific role, specified with a template name. Because of

very limited space, the non terminal symbols Event, Condition and Action are not shown in EBNF here. The following example shows a set of rules, that is bound to all entities, which act in the role of a *Person*:

```
rules for <Person> { ... }
```

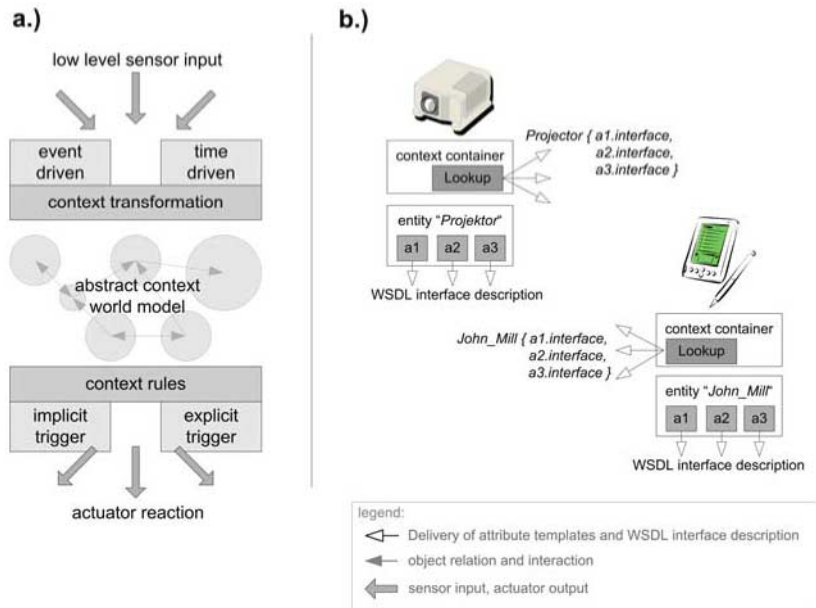


Fig. 4. a.) Shows the context information transformation and event triggering, while b.) shows the lookup and interface WSDL description delivery.

An entity owns a rule interpreter, which is responsible for catching specific events and reacting on them. Also the attribute loading events or the lookup events can be caught by the rule interpreter. The context rule interpreter is able to load an initial set of rules from a file, referenced by the entity's XML configuration. After the container startup, the entity is able to receive new context rules over the transport layer. These new context rules are registered with the interpreter. Considering the easy plain-text syntax of our context rules, it is also possible to register new context rules from mobile devices like cellular phones or PDAs. Distributed rule deployment is a convenient feature to develop a context scenario on a laptop computer and deploy it to different mobile devices.

Attribute Interface Description with WSDL

Distributed scenario development and deployment is also supported by the interface description of attributes with WSDL (Web Service Definition Language). As the lookup service collects information, which set of attributes an unknown entity provides, the entity is also able to deliver WSDL information for its attributes, as it is

shown in Figure 3.b.). With these types of information it is easy to generate context rules automatically or to use a visual builder tool to generate context behavior.

Rule Consistency

One major problem of the context rule management and deployment is to maintain the consistence of rules. This problem statement is already solved by expert systems, or generally by artificial intelligence research. In fact expert system shells like CLIPS or Jess [5] offer the possibility to reason about entities and their relations. For this purpose it is planned to integrate the Jess library into the context framework, in order to manage entity relationship reasoning. At the moment the interpreter itself triggers actions according to incoming events, but it is not able to check the consistence of new rules.

Advantages of Interpreted Context ECA Rules

One of the most important advantages of controlling complex interaction scenarios with interpreted ECA rules is the high degree of flexibility to change the interaction topology at runtime. Therefore, it is possible to change the interaction partners and the interaction itself at runtime. With the use of attribute templates the rules for handling specific events can be routed to a group of entities that fulfill a specific role. The current implementation of our ECA rule syntax enables the dynamic delivery and lifecycle management of rules at runtime. The rules are coded with the use of a proprietary rule syntax, which enables short and concise creation of new rules, also on mobile devices. Another possible module would be a visual builder tool, to support the creation of context rules, or even the creation of whole interaction scenarios without programming. These aspects, of context modeling through ECA rules, allow nontechnical people to control smart environments and devices without knowledge of a complex system, which a smart environment definitely is. As an example for an abstract view on complex scenario modeling, the Washington's Department of Bioengineering developed a tool called Labscape. With Labscape it is possible to model complex scenarios in a laboratory [6].

5 Example Context Scenario

To demonstrate the abilities of the context middleware, a complex emergency scenario simulation was built (aware of the fact that real-time applications normally are not implemented in Java). For an *compact, out of the box demonstrator* we simulated indoor location sensors with RFID transponders and active readers (in a real world outdoor scenario we would use differential GPS and GPRS as transmission protocol). The scenario starts with an automatic emergency call, triggered by a heart patients PDA-HeartMonitor device:

```
rules for <HeartPatients> {
    on HeartMonitor.Alarm() {
        <EmergencyDispatcher>.EmergencyCall:Alarm($Patient);
    }
}
```

The lookup identifies all entities, acting in the role of an *EmergencyDispatcher*, to which the emergency call is delivered to. When the emergency dispatcher receives an emergency call a rule catches this call and delivers some reaction possibilities to the human emergency dispatcher:

```
rules for <EmergencyDispatcher> {
  on EmergencyCall.Alarm(Patient p) {
    Map.ShowPatient(p);
    Map.ShowNearestAmbulances(p.location);
    Map.ShowNearestHospital(p.location);
  }
}
```

When the human emergency dispatcher reacts on the emergency call, the chosen ambulance car is informed:

```
rules for <EmergencyDispatcher> {
  on EmergencyCall.SendAmbulance(Ambulance a, Patient p, Hospital h) {
    a.Alarm:SendTo(p, h);
  }
}
```

The three context rules, that are shown above, are a small and primitive subset of the mass of context rules that form the complete emergency scenario. They should just explain the purpose of such an abstraction layer.



Fig. 5. RFID reader with PDA device and a model emergency car with sensor equipment.

5.1 Emergency Scenario Hardware Setup

The hardware setup is based on three Siemens Loox devices for hosting the patient, the mobile doctors and the ambulance car entities. In order to receive sensor information about the proximity location of entities RFID (Radio Frequency Identification)

readers were used. For network ad-hoc communication purposes, wireless IEEE802.11b compact flash cards are used. The opened device is shown in Figure 4. To get proximity location information, the presentation floor material is tagged with RFID transponders. The devices are now able to find themselves with the lookup mechanism and to communicate over the context containers transport layer. The emergency scenario is modeled with ECA context rules, in order to define the behavior and the interaction between the emergency entities.

6 Conclusion

The goal of this work was, to show how complex ad-hoc scenarios could be modeled by defining ECA rules for each entity, which is relevant for the scenario. A very flexible and stable context middleware software framework was implemented and tested within an example scenario. Many different simulated entities (emergency cars 1-10, mobile doctors 1-5, 1-10 ambulance drivers and nurses) were hosted in mobile context containers, to show the flexibility of our context middleware. The scenario is not bound to any fixed behavior, but could be changed at runtime through distributed ECA rule deployment. Major future issues will be the visual composition of ECA context rules and a lightweight security model to restrict access to the context model.

References

1. A. K. DEY, G. D. ABOWD: Toward a Better Understanding of Context and Context-Awareness. GIT, GVU Technical Report GIT-GVU-99-22, June 1999.
2. A. FERSCHA, S. VOGL, W. BEER: Ubiquitous context sensing in wireless environments, 4th DAPSYS, Kluwer Academic Publishers 2002
3. Diego López de Ipiña: An ECA Rule-Matching Service for Simpler Development of Reactive Applications, Published as a supplement to the Proceedings of Middleware 2001 at IEEE Distributed Systems Online, Vol. 2, No. 7, November 2001
4. J. PASCOE, N. Ryan, D. Morse: Issues in developing context-aware computing. Handheld and Ubiquitous Computing, Nr. 1707 in Lecture Notes in Computer Science, pages 208–221, Heidelberg, Germany, September 1999. Springer-Verlag
5. JESS (Java Expert System Shell), <http://herzberg.ca.sandia.gov/jess/>
6. L. ARNSTEIN et al., „Labscape: A Smart Environment for the Cell Biology Laboratory”, IEEE Pervasive Computing, July-September 2002, pp. 13–21.
7. PATON N.W. and Diaz O.: Active Databases Survey, ACM Computing Surveys, Vol. 31 No. 1, pp. 63–103, March 1999
8. RDF (Resource Description Framework), <http://www.w3.org/RDF/>
9. R. WANT, W. SCHLIT, et. al.: The PARCTAB Ubiquitous Computing Experiment, Technical Report CSL-95-1, Xerox Palo Alto Research Center, March 1995.
10. SENTIENT COMPUTING, AT&T Laboratories, <http://www.uk.research.att.com/spirit/>
11. SEMANTIC WEB Organisation, <http://www.semanticweb.org>
12. SLP (Service Location Protocol), <http://www.ietf.org/rfc/rfc2608.txt>
13. T. KINDBERG, et al.: people, places, things: web presence for the real world

A Coordination Model for ad hoc Mobile Systems

Marco Tulio Valente¹, Fernando Magno Pereira², Roberto da Silva Bigonha²,
and Mariza Andrade da Silva Bigonha²

¹ Department of Computer Science, Catholic University of Minas Gerais, Brazil

² Department of Computer Science, Federal University of Minas Gerais, Brazil
mtov@pucminas.br, {fernandm,bigonha,mariza}@dcc.ufmg.br

Abstract. The growing success of wireless ad hoc networks and portable hardware devices presents many interesting problems to system engineers. Particular, coordination is a challenging task, since ad hoc networks are characterized by very opportunistic connections and rapidly changing topologies. This paper presents a coordination model, called PeerSpaces, designed to overcome the shortcomings of traditional coordination models when used in ad hoc networks.

1 Introduction

With the advent of ad hoc networks, mobile devices can detach completely from the fixed infrastructure and establish transient and opportunistic connections with other devices that are in communication range. Designing applications on these dynamic and fluid networks presents many interesting problems. Particularly, coordination is a challenging task. Since a user may find itself in a different network at any moment, the services available to him change along the time. Thus, computation should not rely on any predefined and well known context. Specifically when operating in ad hoc mode, coordination should not assume the existence of any central authority, since the permanent availability of this node can not be granted. Communication should also be uncoupled in time and space, meaning that two communicating entities do not need to establish a direct connection to exchange data nor must know the identity of each other.

Recently, shared space coordination models, inspired by Linda [4], are being considered for communication, synchronization and service lookup in mobile computing systems. The generative communication paradigm introduced by Linda is based on the abstraction of a tuple space. Processes communicate by inserting, reading and removing ordered sequences of data from this space.

In traditional Linda systems, like TSpaces [8] and JavaSpaces [3], the tuple space is a centralized and global data structure that runs in a pre-defined service provider. In the base station scenario this server can easily be located in the fixed network. However, if operation in ad hoc mode is a requirement, the fixed infrastructure simply does not exist. This suggests that standard client/server implementations of Linda are not suitable to ad hoc scenarios, since they assume a tight coupling between client and servers and the permanent availability of the latter.

This paper formalizes our attempts to customize and adapt shared space coordination models to applications involving mobile devices with ad hoc network capabilities. The model formalized in the paper, called PeerSpaces, has primitives for local and remote communication, process mobility and service lookup. In order to answer the new requirements posed by ad hoc mobile computing systems, PeerSpaces departs from traditional client/server architectures and push towards a completely decentralized one. In the model, each node (or peer) has the same capabilities, acting as client, shared space provider and as router of messages. In order to provide support to operation in ad hoc mode, service lookup is distributed along the network and does not require any previous knowledge about its topology.

The paper is organized as follows. In Section 2 we informally present the PeerSpaces model, including its main design goals, concepts and primitives. In Section 3 we give the formal semantics of the model in terms of a small language derived from the π -calculus. Besides a precise specification of the model, the semantics presented in this section supports formal reasoning about applications built on PeerSpaces. Section 4 compares the model with similar efforts. Finally, Section 5 concludes the paper.

2 The PeerSpaces Model

The main concepts used in PeerSpaces are the following:

Hosts. The model assumes that hosts are mobile devices. Each host has its own local tuple space and a running process. A host is written $h_g[P, T]$, where h is the name of the host, P is the process running in the host, T is its local tuple space, and g is the group of the host.

In PeerSpaces, the name of a host is different from the name of all other hosts. The model also assumes a infinite set H of possible host names.

Groups. Hosts in the model are logically organized in groups. Each group has a name and can also contain subgroups, creating a tree structure. The group of a host is denoted by a tuple $\langle g_1, \dots, g_n \rangle$, that specifies the path from the root group g_1 to the leaf group g_n where the host is located. For example, the tuple $\langle \text{pucminas}, \text{cs}, \text{proglab} \rangle$ denotes the set of hosts in the **proglab** group, which is a subgroup of the group **cs**, which is nested in the root group **pucminas**. Two groups can have the same name, as long they are not subgroups of the same group.

Network. Mobile hosts in the model are connected by a wireless and ad hoc network. As usual in such networks, connectivity is transient and determined by the distance among hosts. Consequently, the topology of the network is continuously changing. In PeerSpaces, a network with hosts $h_1, h_2, \dots, h_n$ is denoted by:

$$h_{1g_1}[P_1, T_1] \mid h_{2g_2}[P_2, T_2] \mid \dots \mid h_{ng_n}[P_n, T_n], E$$

where $g_1, g_2, \dots, g_n$ are the group of the hosts and $E : H \times H$ is a relation representing connections among hosts. The presence of a pair (h_i, h_j) in E , denoted by $h_i \bowtie h_j$, indicates that host h_i is in communication range with host h_j . This relation is in continuous change to reflect reconfigurations in the network.

PeerSpaces also defines a set of primitives to assemble applications using the previous defined concepts. We spend the rest of this section describing such primitives.

Local Primitives. The local tuple space of any host is accessed using the traditional **in**, **rd** and **out** primitives from Linda. Furthermore, there is a **chgrp** g primitive, used to change the group of the current host to the one specified by tuple g .

Process Mobility. Processes in PeerSpaces are mobile in order to model the behavior of mobile agents. A mobile agent is a process that can move among sites carrying computation and accessing resources locally. In wireless environments, agents are a powerful design tool to overcome latency and to embed autonomous computations. In the model, the primitive **move** $h.P$ is used to move a process to node h , where its execution continues as P . If host h is not connected, the operation blocks until a connection to such host is established.

Remote Primitives. Crucial to the scalability and efficiency of any coordination model for mobile computing systems is the design of the remote operations. Thus, from the beginning PeerSpaces departs from the idea of providing seamless access to a global and centralized space. Instead, there are primitives that operate in the remote space of a well-known host h : **out** h, v ; **in** h, p, x and **rd** h, p, x , where v is a tuple, p is a pattern and x is a variable. These operations are merely remote implementations of the traditional Linda primitives and thus do not impact in the overall performance of the system.

As its local version, the remote **out** h, v primitive is asynchronous. The primitive is used when a process wants to leave a information to be consumed later in another host. In order to model its asynchronous behavior, the operation is executed in two steps. In the first step, a tag is added to the tuple v to indicate that it should be transfer as soon as possible to the destination host h . The tagged tuple, denoted by v_h , is then outputted in the local space of the host h' that requested the operation. In the second step, tuple v_h is transferred to the space of host h as soon as it is connected to h' and the tag is removed from the tuple. Since both steps are not atomic, while the tuple is “misplaced” in the source node it can be retrieved by an operation like **in** v_h . For example, this operation can be called by a garbage collector process in charge of reclaim tuples that are waiting for the connection of their destination host for a long time.

Lookup Primitive. Without a lookup primitive the remote operations described above have little use, since a mobile host may not know in advance the

name h of a service provider in its current network. Moreover, since the system is designed to support operation in ad hoc mode, the lookup service must not be centralized in a single host, but must be distributed along the federation of connected devices. In order to accomplish such requirements, there is in PeerSpaces the following primitive: **find** g, p . This primitive queries hosts in group g for tuples matching pattern p in a distributed way. All matching tuples found in group g are copied asynchronously to the local space of the host that has called the operation.

The semantics of PeerSpaces does not assume any specific routing protocol for propagation of lookup queries.

Continuous Queries. Often it is useful to query a group of hosts for a resource and keep the query effective until such resource is available. In this way, a client does not need to periodically send lookup queries to detect new resources that may become available since the last query was issued. In PeerSpaces, lookup queries that remain active after their first execution are called continuous queries.

Continuous queries in PeerSpaces have a lifetime parameter, used to automatically garbage collect the query after its expiration. Continuous lookup queries are issued adding the lifetime t to the **find** primitive: **find** g, p, t . This primitive will search the hosts of group g for all currently available resources matching pattern p and for resources that may become available in t units of time after the query was issued.

3 Formal Semantics

The ultimate goal of our research is to deploy a coordination middleware for ad hoc mobile computing systems. In order to achieve this goal we have initially defined the formal semantics of PeerSpaces.

The formalization presented next uses an operational semantics based on the asynchronous π -calculus [6]. The π -calculus is a good basis as it provides a small, elegant and expressive concurrent programming language. The main departure from π in our semantics is the use of generative communication instead of channel-based communication.

Table 1 summarizes the syntax of our core language. We assume an infinite set H of names, used to name hosts and lookup queries. Meta-variables h and x range over H . Basic values, ranged over by v and g , consist of names and tuples. Tuples are ordered sequences of values $\langle v_1, \dots, v_n \rangle$. A tuple space T is a multiset of tuples. We use the symbol $? \in H$ to denote the distinguished unspecified value.

A program is composed by the network N , the relation E and a global set of names X . The relation $E : H \times H$ represents the connectivity map of the network. The names used over several hosts in the system are recorded in the set X , ensuring their unicity. Each host h is member of a group g and has a running process P and a local tuple space T . Processes are ranged by P and Q . Similar to the π -calculus, the simplest term of our language is the inert process $\mathbf{0}$, which denotes a process with no behavior at all. The term $P \mid Q$ denotes two processes

Table 1. Syntax

$Prog ::= N, E, X$
$N ::= \varepsilon \mid H \mid N$
$H ::= h_g[P, T]$
$P ::= \mathbf{0} \mid P \mid Q \mid !P \mid (\nu x)P \mid \mathbf{out} \ v \mid$
$\quad \mathbf{in} \ v, x.P \mid \mathbf{rd} \ v, x.P \mid \mathbf{find} \ g, p, t \mid$
$\quad \mathbf{chgrp} \ g \mid \mathbf{move} \ h.P$

running in parallel. The term $!P$ denotes a infinite number of copies of P , all running in parallel. The restriction operator $(\nu x)P$ ensures that x is a fresh and unguessable name in the scope of P . Similar to Linda, the primitive operations **out**, **in** and **rd** provide access to the local tuple space. Since the **out** operation is asynchronous it does not have a continuation P . The same happens to the **find** and **chgrp** primitives. We assume that non-continuous lookup queries can be simulated by defining the lifetime equal to zero. Finally, the **move** operation simulates the behavior of single thread mobile agents.

The operational semantics of our calculus is summarized in Tables 2 and 3. The semantics is defined in terms of a reduction relation $\rightarrow$, a structure congruence $\equiv$ between processes and a set of pattern matching rules.

Table 2 summarizes the core language semantics, which is basically Linda with multiple tuple spaces. A reduction $N, E, X \rightarrow N', E', X'$ defines how the configuration N, E, X reduces in a single step computation to N', E', X' . Initially, there are three reduction rules describing the effects on the configuration of each standard Linda primitive. The output operation, **out** v , asynchronously deposits a tuple in the local space (rule L1). The input, **in** $v, x.P$, and read, **rd** $v, x.P$, operations try to locate a tuple v' that matches v (rules L2 and L3). If one is found, free occurrences of x are substituted for v' in P , denoted as $P\{v'/x\}$. In the case of the input, the tuple is removed from the space.

The next set of rules defines a structural congruence relation $\equiv$ between processes (SC1 to SC7) and hosts (SC8 to SC10). As in the π -calculus, such rules define how processes can be syntactically rearranged in order to allow the application of reductions. In such rules, we write $fn(P)$ to denote the set of free names in process P . The definition of pattern matching, written $v \leq v'$, allows for recursive tuple matching. Values match only if they are equal or if the unspecified value occurs on the left hand side.

Table 3 extends the core language with the primitives proposed in PeerSpaces. The **find** g', p, t operation deposits a tuple representing a service lookup query in the local space (rule P1). Such query is a tuple in the format $\langle k, g', p, t, h \rangle$, where k is a fresh name that identifies the query, g' defines the group where the query will be performed, p is a pattern for the desired service, t is the lifetime and h is the name of the current host. The operation **chgrp** g just

Table 2. Core Language Operational Semantics**Reductions****Linda Primitives**

$$h_g[\mathbf{out} \ v \mid P, T] \mid N, E, X \rightarrow h_g[P, v \cup T] \mid N, E, X \quad (\text{L1})$$

$$h_g[\mathbf{in} \ v, x.P \mid Q, v' \cup T] \mid N, E, X \rightarrow h_g[P\{v'/x\} \mid Q, T] \mid N, E, X \quad (\text{L2})$$

$$h_g[\mathbf{rd} \ v, x.P \mid Q, v' \cup T] \mid N, E, X \rightarrow h_g[P\{v'/x\} \mid Q, v' \cup T] \mid N, E, X \quad (\text{L3})$$

The rules are subjected to the following side conditions:

(L2) if $v \leq v'$

(L3) if $v \leq v'$

Structural Congruence Rules

$$P \mid Q \equiv Q \mid P \quad (\text{SC1}) \quad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \quad (\text{SC5})$$

$$!P \equiv P \mid !P \quad (\text{SC2}) \quad P \equiv Q \Rightarrow (\nu x)P \equiv (\nu x)Q \quad (\text{SC6})$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad (\text{SC3}) \quad (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q \quad (\text{SC7})$$

$$P \mid \mathbf{0} \equiv P \quad (\text{SC4})$$

$$P \equiv Q \Rightarrow h_g[P, T] \equiv h_g[Q, T] \quad (\text{SC8})$$

$$h_g[(\nu x)P, T] \mid N, E, X \equiv h_g[P, T] \mid N, E, x \cup X \quad (\text{SC9})$$

$$h_g[P, T] \mid N, E, X \equiv N \mid h_g[P, T], E, X \quad (\text{SC10})$$

The rules are subjected to the following side conditions:

(SC7) if $x \notin fn(P)$

(SC9) if $x \neq h$, $x \notin fn(N)$, $x \notin X$

Pattern Matching Rules

$$v \leq v \quad ? \leq v \quad \frac{v_1 \leq v'_1 \dots v_n \leq v'_n}{\langle v_1 \dots v_n \rangle \leq \langle v'_1 \dots v'_n \rangle}$$

changes the group of the current host to the one specified by tuple g (rule P2). If such group does not exist, it is created. The **move** $h'.P$ operation changes the location of the continuation process P to host h' if this host is connected (rule P3). Otherwise, the operation remains blocked until the engagement of h' .

Reduction rule Q1 defines how lookup queries are propagated in the network. Basically, any host that holds a query $\langle k, g'', p, t, h \rangle$ can propagate it to a connected host h' in group g' , if g'' matches g' and the query is not yet present in h' . If such conditions are satisfied, the query is inserted in the local space of h' and a process P'' is added in parallel with the other processes running in this host. This process continuously read tuples matching the pattern p and then use a remote output operation to send the results to the local space of the host h that has issued the query.

Table 3. PeerSpaces Operational Semantics**Reductions****PeerSpaces Primitives**

$$h_g[\text{find } g', p, t \mid P, T] \mid N, E, X \rightarrow h_g[(\nu k) P, \langle k, g', p, t, h \rangle \cup T] \mid N, E, X \quad (\text{P1})$$

$$h_g[\text{chgrp } g' \mid P, T] \mid N, E, X \rightarrow h_{g'}[P, T] \mid N, E, X \quad (\text{P2})$$

$$h_g[\text{move } h'.P \mid Q, T] \mid h'_{g'}[P', T'] \mid N, E, X \rightarrow h_g[Q, T] \mid h'_{g'}[P \mid P', T'] \mid N, E, X \quad (\text{P3})$$

Query Propagation

$$h_g[P, \langle k, g'', p, t, h \rangle \cup T] \mid h'_{g'}[P', T'] \mid N, E, X \rightarrow h_g[P, \langle k, g'', p, t, h \rangle \cup T] \mid h'_{g'}[P' \mid P'', \langle k, g'', p, t, h \rangle \cup T'] \mid N, E, X \quad (\text{Q1})$$

Network Reconfiguration

$$\frac{E \Rightarrow E'}{N, E, X \rightarrow N, E', X} \quad (\text{N1})$$

The rules are subjected to the following side conditions:

(P3) if $h \bowtie h'$

(Q1) if $(h \bowtie h') \wedge (g'' \preceq g') \wedge (\langle k, g'', p, t, h \rangle \notin T') \wedge P'' = !(\text{rd } p, x.\text{out } h, x)$

Group Matching Rule

$$\frac{g_1 = g'_1 \dots g_n = g'_n}{\langle g_1 \dots g_n \rangle \preceq \langle g'_1 \dots g'_n \dots g'_m \rangle}$$

The last reduction rule introduces a new type of reduction $\Rightarrow$ used to describe reconfigurations in the network and consequently in the connectivity relation E . Basically, this rule dictates that changes in E should be propagated to the current configuration. However, we left $\Rightarrow$ reductions unspecified in the semantics, since they are dependent on the physical location of each host and on technological parameters of the subjacent network.

There is also a special pattern matching rule for group names. Two groups g and g' matches, written $g \preceq g'$ if all subgroups in g are equal to equivalent subgroups in g' , which can also have extra nested subgroups.

4 Related Work

Many characteristics of PeerSpaces have been inspired in file sharing applications popular in the Internet. Particularly, the peer to peer network created by Gnutella [5] over the fixed Internet presents many properties that are interesting in mobile settings, like absence of centralized control, self-organization and adaptation to failures. PeerSpaces is an effort to transport and adapt such

characteristics to mobile computing systems. This explains the choice of Linda shared spaces as the prime coordination infrastructure for PeerSpaces.

Lime [7] introduces the notion of transiently shared data space to Linda. In the model, each mobile host has its own tuple space. The contents of the local spaces of connected hosts are transparently merged by the middleware creating the illusion of a global and virtual data space. Applications in Lime perceive the effects of mobility by atomic changes in the contents of this virtual space. The scalability and performance weakness of Lime have motivated the proposal of CoreLime [2], where in name of simplicity and scalability the idea of transiently shared spaces is restricted to the set of mobile agents running in a host. Another work proposing an alternative semantics to the notion of transiently shared spaces is [1].

5 Conclusions

In this paper we have presented and formalized PeerSpaces, a coordination model for mobile computing systems. The model was designed to overcome the main shortcoming of shared spaces coordination models when used in ad hoc wireless networks – the strict reliance on the traditional client/server architecture – while preserving the main strengths of such models – the asynchronous and uncoupled style of communication. PeerSpaces can be used as the building block of ad hoc mobile systems like file sharing, groupware, mobile commerce and message systems.

Acknowledgments. We would like to thank Jan Vitek and Bogdan Carbutar for the discussions that led to this paper.

References

1. N. Busi and G. Zavattaro. Some thoughts on transiently shared tuple spaces. In *Workshop on Software Engineering and Mobility*, May 2001.
2. B. Carbutar, M. T. Valente, and J. Vitek. Corelime a coordination model for mobile agents. In *International Workshop on Concurrency and Coordination*, volume 54 of *Electronic Notes on Theoretical Computer Science*. Elsevier Science, July 2001.
3. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.
4. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.
5. Gnutella Home Page. <http://gnutella.wego.com>.
6. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
7. A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, May 2001.
8. P. Wycko, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, Aug. 1998.

Making Existing Interactive Applications Context-Aware

Tatsuo Nakajima, Atsushi Hasegawa, Tomoyoshi Akutagawa, Akihiro Ibe, and
Kouji Yamamoto

Department of and Computer Science, Waseda University
3-4-1 Okubo Shinjuku Tokyo 169-8555 JAPAN
`tatsuo@dc1.info.waseda.ac.jp`

Abstract. We propose a new approach to build context-aware interactive applications. Our approach enables us to use existing GUI-based interactive applications although a variety of interaction devices can be used to control them. This means that we can adopt traditional GUI toolkits to build context-aware applications. The paper describes design and implementation of our middleware that enables existing applications to be context-aware, and presents some examples to show the effectiveness of our approach.

1 Introduction

Context-awareness[4,11] will reduce complexities to access our environments since the environments recognize our situation to fill many parameters that are chosen by us in traditional approaches. The parameters are retrieved by various sensors embedded in our environments. However, implementing context-aware applications is hard for usual programmers. Also, we already have many existing GUI-based interactive applications. In the future, these applications will like to be modified to support context-awareness, but the modification is not easy.

In this paper, we propose a new approach to build context-aware applications. Our middleware enables the GUI-based interactive applications to be context-aware without modifying them. Also, programmers can adopt traditional GUI toolkits such as GTK+ and Java Swing to build context-aware applications. Our current work focuses on home computing environments where home appliances such as TV and VCR are connected by a home network. Most of current home appliances do not take into account context-awareness. Also, standard middleware for home computing has adopted traditional GUI toolkits such as Java AWT. Therefore, our middleware is attractive to build context-aware home computing applications in an easy way.

2 Ubiquitous Computing Applications

In ubiquitous computing environments, one of the most important problems is how to interact with a variety of objects embedding computers. The interaction

between us and computers embedded in various objects has been studied. However, it is necessary for us to consider how to manage these interaction devices. Let us consider that a user uses a mobile device to control home appliances or to retrieve interesting information. In this case, applications running on the device need to take into account the mobility of devices. Also, we like to use various display devices to show control panels of home appliances. For example, when a user sits on a sofa, he will use his PDA to control the appliances. The control panel is displayed on the PDA, and he navigates the panel to control appliances. However, if he can find a large display near him, he likes to use it to control the appliances. The control panel should be displayed on the large display, and the panel can be navigated from his PDA. Moreover, if the display is a touch panel, he will navigate the panel by touching it. We believe that it is important to use a variety of interaction devices according to his situation, and one interaction device is not enough to cover various situations. In this case, a user interface middleware to offer context-awareness is necessary to change various interaction devices according to the situations.

There are many researches to develop context-aware applications[6,11]. Also, there is a generic framework to develop context-aware applications[4]. However, we believe that there are two important issues that should be considered to build context-aware applications. The first issue is how to retrieve context information. We need various sensors and common interface to provide higher level abstraction to abstract various sensor information. For example, context toolkit[4] and sentient information framework[5] provide high level abstraction to hide details of context information retrieved from the sensors.

The second issue is a mechanism to adapt software structure according to retrieved context information[9]. Traditionally, most of context-aware applications change their software structures in an ad-hoc way, and it is not easy to take into account new context information. We believe that it is important to adopt a framework to build adaptive applications in a systematic way.

Although there are many researches to attack the problems described above, it is not easy to develop context-aware applications. Because implementing adaptive applications is still ad-hoc, and programmers need to learn advanced programming techniques such as design patterns and aspect-oriented programming to build context-aware applications that can take into account new situations. Also, we need to reimplement existing applications if we like to modify the applications to support context-awareness. For example, if a context-aware application needs to display video streams according to the location of a user. The application should implement the redirection of the video streams to transmit them to computers that is desirable to display them[1].

We believe that we need a simpler approach to build context-aware applications. Especially, it is important to build context-aware applications from existing applications to build ubiquitous computing environments in a practical way. Our middleware based on a thin-client architecture enables us to build context-aware applications in a simple way. Existing applications are completely separated from our middleware components. Our middleware enables us to inte-

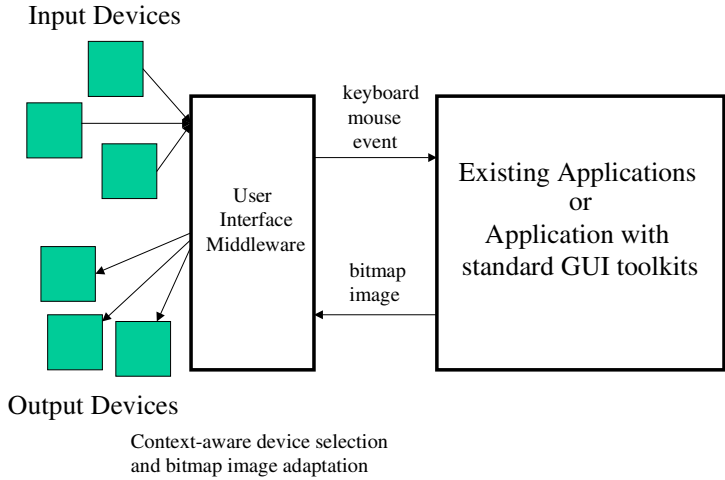


Fig. 1. Basic Architecture

grate new interaction devices and context-awareness without modifying existing applications.

3 Design and Implementation

3.1 Basic Architecture

Fig. 1 shows an overview of our middleware infrastructure. In the architecture, an application generates bitmap images containing information such as control panels, photo images and video images. The approach is simple because popular operating systems provide a mechanism to retrieve bitmap images generated by applications. Also, these applications can receive keyboard and mouse events to be controlled. The user interface middleware receives bitmap images from applications and transmits keyboard and mouse events. The role of the middleware is to select appropriate interaction devices by using context information. Also, input/output events are converted to keyboard/mouse events according to the characteristics of interaction devices.

Our system uses the thin-client system to transfer bitmap images to draw graphical user interface, and to process mouse/keyboard events for inputs. The usual thin-client system such as Citrix Metaframe[3], Microsoft Terminal Server[7], Sun Microsystems Sun Ray[13], and AT&T VNC(Virtual Network Computing) system[10] consists of a viewer and a server. The server is executed on a machine where an application is running. The application implements graphical user interface by using a traditional user interface system such as the X window system. The bitmap images generated by the user interface system are transmitted to a viewer that is usually executed on another machine. On the

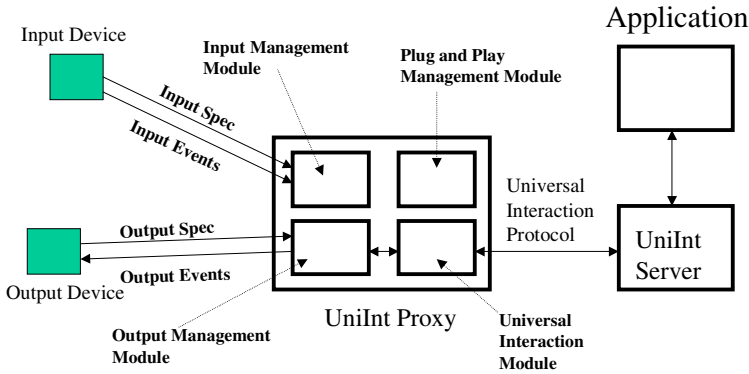


Fig. 2. System Architecture

other hand, mouse and keyboard events captured by the viewer are forwarded to the server. The protocol between the viewer and the server are specified as a universal interaction protocol. The system is usually used to move a user's desktop according to the location of a user[6], or shows multiple desktops on the same display, for instance, both MS-Windows and the X Window system.

In our system, we replace the viewer of a thin-client system to the UniInt(Universal Interaction) proxy that forwards bitmap images received from a UniInt server to an output device. Also, UniInt proxy forwards input events received from an input interaction device to the UniInt server. In our approach, a server of any thin-client systems can be used as the UniInt server.

Our system consists of the following four components as shown in Fig. 2.

- Home Computing Application
- UniInt Server
- UniInt Proxy
- Input/Output Interaction Devices

Home computing applications[12] generate graphical user interface for currently available home appliances to control them. For example, if TV is currently available, the application generates user interface for the TV. On the other hand, the application generates the composed GUI for TV and VCR if both TV and VCR are currently available.

The *UniInt server* transmits bitmap images generated by a window system using the universal interaction protocol to a UniInt proxy. Also, it forwards mouse and keyboard events received from a UniInt proxy to the window system. In our current implementation, we need not to modify existing servers of thin-client systems, and any applications running on window systems supporting a UniInt server can be controlled in our system without modifying them.

The *UniInt proxy* is the most important component in our system. The UniInt proxy converts bitmap images received from a UniInt server according to

the characteristics of output devices. Also, it converts events received from input devices to mouse or keyboard events that are compliant to the universal interaction protocol. The UniInt proxy chooses a currently appropriate input and output interaction devices for controlling appliances. To convert interaction events according to the characteristics of interaction devices, the selected input device transmits an input specification, and the selected output device transmits an output specification to the UniInt proxy. These specifications contain information that allow a UniInt proxy to convert input and output events.

The last component is *input and output interaction devices*. An input device supports the interaction with a user. The role of an input device is to deliver commands issued by a user to control home appliances. An output device has a display device to show graphical user interface to control appliances.

3.2 Implementation of UniInt Proxy

The current version of UniInt proxy is written in Java, and the implementation contains four modules as shown in Fig. 2. The first module is the universal interaction protocol module that executes the universal interaction protocol to communicate with a UniInt server. The replacement of the module enables us to use our system with different thin-client systems. The module can use the same module implemented in a viewer of a thin-client system. The second module is the plug and play management module. The module collects information about currently available interaction devices, and builds a database containing information about respective interaction devices. The third module is the input management module. The module selects a suitable input interaction device by using the database contained in the plug and play management module. The last module is an output management module. The module also selects a suitable output interaction device. Also, the module converts bitmap images received from the universal interaction module according to the output specification of the currently selected output interaction device.

Management of Available Interaction Devices: The plug and play management module detects currently available input and output devices according to context information. The module implements the Universal Plug and Play Protocol to detect currently available interaction devices. In our system, we assume that all interaction devices can be connected to the Internet. An interaction device transmits advertisement messages using the simple service discovery protocol (SSDP). When a UniInt proxy detects the messages, it knows the IP address of the interaction device. Then, the UniInt proxy transmits an HTTP GET request to the interaction device. We assume that each interaction device contains a small Web server, and returns an XML document.

The XML document contains information about the interaction devices. If the interaction device is an input device, the file contains various attributes about the device, which are used for the selection of the most suitable device. For an output device, the file contains information about the display size and the attributes for the device. The plug and play management module maintains a database containing all information about currently detected interaction devices.

Adaptation of Input and Output Events: The role of the input management module and the output management module is to determine the policies for selecting interaction devices. As described in the previous section, all information about currently available interaction devices are stored in a database of the plug and play management module. The database provides a query interface to retrieve information about interaction devices. Each entry in the database contains a pair of an IP address and a list of attributes for each interaction device, then the entry whose attributes are matched to a user's preference provided in a query is returned.

The current implementation of the input management module receives all input events from any currently available input devices. When a new input device is detected, a new thread for receiving input events from the newly detected device is created. All input events are delivered to the universal interaction module, and they are processed by applications eventually.

The output management module converts bitmap images received from the universal interaction module according to the display size of an output device. The size is stored in the database of the plug and play management module. When an output device is selected, the display size is retrieved from the database. The bitmap image is converted according to the retrieved information, then it is transmitted to the selected output device.

3.3 Current Status

Our system have adopted the AT&T VNC system[10] as a thin-client system, and the VNC server can be used as the UniInt server without modifying it. The current prototype in our HAVi-based home computing system[12], where HAVi is a standard specification for digital audio and video, emulates two home appliances. The first one is a DV viewer and the second one is a digital TV emulator. Our application shows a graphical user interface according to currently available appliances as described in the previous section. Also, the cursor on a screen that displays a graphical user interface can be moved from a Compaq iPAQ. However, if the device is turned off, the cursor is controlled by other devices such as a game console. It is also possible to show a graphical user interface on the PDA device according to a user's preference. Also, the current system has integrated cellular phones to control home appliances. NTT Docomo's i-mode phones have Web browsers, and this makes it possible to move a cursor by clicking special links displayed on the cellular phones. In our home computing system, Linux provides an IEEE 1394 device driver and an MPEG2 decoder. Also, IBM JDK1.1.8 for the Java virtual machine is used to execute the HAVi middleware component.

Fig. 3 contains several photos to demonstrate our system. Currently, our home computing applications are executed on HAVi, and a control panel is written by using Java AWT. In the demonstration, if both a DV camera and a digital TV tuner are simultaneously available, the control panels for them are combined as one control panel as shown in the photo(Top-Left). The control panel can be navigated by both a cellular phone(Top-Right) and a game

console(Bottom-Left). Also, the control panel can be displayed and navigated on a PDA(Bottom-Right).

Our middleware proposed in this paper enables us to use various interaction devices and to interact with the appliances in a context-aware way. By integrating home appliances with our middleware, a user is allowed to choose the most suitable interaction device according to his situation.



Fig. 3. Controlling Our HAVi based Home Computing System

4 Scenarios of Our Approach

In this section, we present two scenarios to show the effectiveness of our approach. The first scenario enables us to interact with existing applications running on MS-Windows or Linux in a location-aware fashion. The system has been implemented, and shows the effectiveness of our approach. The second scenario is a location-aware video phone.

4.1 Location-Aware Interaction

The system enables a user to control a home appliance in a location-aware way. In the future, our home will have many display devices to show control panels

for controlling home appliances. A user usually likes to use the nearest display device to him to control a variety of home appliances. For example, if a user sits on a sofa, the control panel of a home application is displayed on his PDA. On the other hand, he is in front of a display device. The control panel is shown on the display device. In this case, if the display is a touch panel, he can access the control panel by touching the display. However, if the display is a normal display, he can navigate the control panel from his PDA or a game console.

4.2 Ubiquitous Video Phones

The second example is a ubiquitous video phone that enables us to use a video phone in various ways. In this example, a user speaks with his friend by using a telephone like a broadband phone developed by AT&T Laboratories, Cambridge[2]. The phone has a receiver like traditional phones, but it also has a small display. When the phone is used as a video phone, the small display renders video streams transmitted from other phones. The display is also able to show various information such as photos, pictures, and HTML files that are shared by speakers. Our user interface system makes the phone more attractive, and we believe that the extension is a useful application in ubiquitous computing environments.

When a user needs to start to make a dinner, he will go to his kitchen, but he likes to keep to talk with his friend. However, a phone is put in a living room, and the traditional phone receiver is not appropriate to continue the conversation with his friend in the kitchen because his both hands may be used for cooking. In this case, we use a microphone and a speaker in the kitchen so that he can use both hands for making the dinner while talking with his friend. In the future, various home appliances such as a refrigerator and a microwave provide displays. Also, a kitchen table may have a display to show a recipe. These displays can be used by the video phone to show a video stream. In a similar way, a video phone can use various interaction devices for interacting with a user. The approach enables us to use a telephone in a more seamless way.

Our system allows us to use a standard VoIP application running on Linux. The application provides a graphical user interface on the X window system. However, our system allows a user to be able to choose various interaction styles according to his situation. Also, if his situation is also changed, the current interaction style is changed according to his preference. Currently, we are incorporating an audio stream in our system, and it is easy to realize the example described in the section by adopting our approach.

5 Conclusion and Future Work

In the paper, we have proposed a new approach to build ubiquitous computing applications. Our middleware enables us to convert existing applications to context-aware applications without modifying existing applications. Therefore, a programmer needs not to learn new APIs to write context-aware applications.

Currently, we are extending our middleware to support video streams and audio streams. Also, we are implementing a location-aware video phone described in this paper. Our system can transmit video streams to any output devices if the bandwidth is enough to process the video streams. We like to incorporate a dynamic QOS control[8] scheme in our system. This means that the quality of video streams is changed according to the processing power of a machine to execute UniInt proxy programs.

To support audio streams, we are working on an audio snoop program in a Linux kernel. The program runs in the kernel, and captures events about audio streams. The program communicates to UniInt proxy, and the UniInt proxy transmits audio streams to appropriate speakers, and receive audio streams from appropriate microphones. The audio streams can be converted according to the capabilities of microphones and speakers similar to other interaction devices.

References

1. J. Bacon, J Bates, and D. Halls, "Location-Oriented Multimedia", IEEE Personal Communications, Vol.4, No.5, 1997.
2. AT&T Laboratories, Cambridge, "Broadband Phone Project", <http://www.uk.research.att.com/bphone/apps.html>.
3. Citrix Systems, "Citrix Metaframe 1.8 Background", Citrix White Paper, 1998.
4. A.K. Dey, D. Salber, and G.D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications", Human-Computer Interactions, Vol.16, No.2-4, 2001.
5. Diego Lopez de Ipina, "Building Components for a Distributed Sentient Framework with Python and CORBA", In Proceedings of the 8th International Python Conference, 2000.
6. Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, Paul Webster, "The Anatomy of a Context-Aware Application", In Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1999.
7. Microsoft Corporation, "Microsoft WIndows NT Server 4.0: Technical Server Edition, An Architecture Overview", Technical White Paper 1998.
8. T. Nakajima "A Dynamic QOS Control based on Optimistic Processor Reservation", In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, 1996.
9. T. Nakajima "Adaptive Continuous Media Applications in Mobile Computing Environment", In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, 1997.
10. T.Richardson, et al., "Virtual Network Computing", IEEE Internet Computing, Vol.2, No.1, 1998.
11. B.N. Schilit, N.Adames, and R. Want, "Context-Aware Computing Applications", In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, 1994.
12. K.Soejima, M.Matsuda, T.Iino, T.Hayashi, and T.Nakajima, "Building Audio and Visual Home Applications on Commodity Software", IEEE Transactions on Consumer Electronics, Vol.47, No.3, 2001.
13. Sun Microsystems, "Sun Ray 1 Enterprise Appliance", <http://www.sun.com/products/sunray1/>.

Benefits and Requirements of Using Multi-agent Systems on Smart Devices

Cosmin Carabelea¹, Olivier Boissier¹, and Fano Ramparany²

¹ SMA/SIMMO/ENS Mines de Saint-Etienne, France
{cosmin.carabelea, olivier.boissier}@emse.fr

² DIH/OCF/France Télécom R&D
Fano.Ramparany@rd.francetelecom.com

Abstract. Due to the emergence of Internet and embedded computing, humans will be more and more faced to an increasing intrusion of computing in their day-to-day life by what it is called now *smart devices*. Agent characteristics like pro-activeness, autonomy and sociability and the inherent distributed nature of multi-agent systems make them a promising tool to use in the smart devices applications. This paper tries to illustrate the benefits of using multi-agent systems on smart communicating objects and discusses the need of using multi-agent platforms. We then present an overview of multi-agent platforms created for use on small devices (i.e. devices with limited computing power) and for each reviewed multi-agent platform we try to identify what are its main characteristics and architectural concepts.

1 Introduction

In the near future it is to be expected that humans will be more and more faced to an increasing intrusion of computing (in the form of *smart devices*) in their day-to-day life. The term *smart device* designates any physical object associated with computing resources and able to communicate with other similar objects via any physical transmission medium, and logical protocol, or with humans via standard user interface. The scale spans from big smart devices such as PDAs, to small ones such as RFID Tags.

We are mainly interested in devices that share the following characteristics: tight memory constraints, limited computing power (due to low power consumption), limited user interface peripheral, embedded in the physical world and exhibiting real time constraints. The increasing number of such devices makes us envision a myriad of applications in which users will shortly need to be intelligently assisted in their interactions with these computing entities. The evolution surely will produce several users in interaction with several devices, so we need to introduce in these intelligent objects some cooperating and social reasoning capabilities.

For many years, multi-agent systems have proposed a new paradigm of computation based on cooperation and autonomy. An intelligent agent is "a computer

system, situated in some environment, that is capable of flexible and autonomous action in order to meet its design objectives" [8]. A multi-agent system is a federation of software agents interacting in a shared environment, that cooperate and coordinate their actions given their own goals and plans. Many properties of this metaphor of computing make them attractive for tackling with the requirements presented above:

- *Pro-activity*: agents can pro-actively assist their users and try to achieve goals given the evolution of the environment; they can help them in discovering information, in interacting.
- *(Adjustable) autonomy*: agents are able to try to achieve their own goals, but are also able to take into-account and adapt their autonomy according to their users' constraints.
- *Self-awareness*: agents are conscious of themselves in the sense that they have a model of their goals, of their own competences, of their state.

Although the multi-agent approach may seem suitable for the smart communicating objects, one has to solve the problems of the actual deployment of agents on those small devices. This paper tries to identify the requirements for executing agents on such devices (operating system, communication facilities, computational power, etc.) and, in the meantime, we try to identify the constraints that the use of small devices is imposing to the multi-agent architectures and services.

The remainder of the paper is structured as follows. In the next section we will present our arguments on why the utilization of multi-agent systems on smart devices is beneficial. Section 3 describes what a multi-agent platform is and what are the FIPA standards for those platforms; we then present some of the main characteristics of several multi-agent platforms for small devices. Finally, in Section 5 the utility of using multi-agent systems and platforms on small devices is discussed and we are drawing some conclusions and tracing directions for future work.

2 Why Use Multi-agent Systems on Smart Devices?

Pervasive and ubiquitous computing will involve more and more objects endowed with computing and communication resources, called smart devices. These smart devices should be able to sense their environment and react to changes in it. In order to alleviate the user information overload, it will be necessary to enable these objects to communicate directly with one another, while in the same time they should be able to communicate with the user.

Due to the high number of these devices, we will have to reduce the number of objects needing explicit control from the user, thus making these devices autonomous. Another way of simplifying the human – smart device interaction is to provide smart devices with knowledge and reasoning capabilities, so that most of the repetitive tasks involved in the interaction could be automated.

An intelligent agent is "a computer system, situated in some environment, that is capable of flexible and autonomous action in order to meet its design objectives" [8]. Beside the (adjustable) autonomy, an agent has other properties. It should be able to

detect and react to changes in its environment. Agents can pro-actively assist their users and try to achieve goals given the evolution of the environment; they can help them in discovering information, in interacting. We can have several agents acting in a common environment, thus obtaining what is called a multi-agent system. From the multi-agent point of view, an agent should be able to interact with other agents, i.e., communicate, cooperate or negotiate with them. Also, an agent should be able to reason about the existing organizational structures in the system.

It is no surprise that the multi-agent applications try to solve the same problems as the applications on smart devices. Some of these problems are concerned on how to realize the interaction between the diverse entities in the system, while others on how the agents or the smart devices should adapt themselves to new situations. The smart devices environment is a very dynamic and open one, devices can enter or leave it at any time. Multi-agent systems research is also interested in the modelization of open systems: what check-in/check-out procedures are needed, how to ensure the service discovery in an open system and so on. The problem of trust and reputation is also tackled in both domains.

It is clear for us that intelligent agents have characteristics desirable for smart devices. Moreover, the multi-agent systems already solve or try to solve similar problems with those of smart devices, so one can benefit from these solutions. Of course, an application for smart devices can be created without the multi-agent technology, but by using it, one can take advantage of the methodologies and solutions provided by the multi-agent paradigm. And we should not forget the reusability: a multi-agent system created for a specific problem can be very easily used to solve another one. In other words, using agents on smart devices is not the only perspective of solving the problem, but it might prove better than others.

3 How to Use Multi-agent Systems on Small Devices?

A *multi-agent platform* is a software infrastructure used as an environment for agents' deployment and execution. It should provide comfortable ways for agent programmers to create and test agents and it can be viewed as a collection of services offered to developers, but also to agents at runtime.

As depicted in the previous characterization, the platform is an execution environment for the agents in the sense that it should allow one to create, execute and delete agents. Secondly, the platform should act as a middleware between the operating system and the applications (agents) running on it, as depicted in Fig.1. The developer will then be able to create agents for a platform and use them on all the systems that support the platform without changing the code.

More than that, a platform should hide from the developer the communication details between the agents. If, for example, there are several communication protocols available on the device (e.g. HTTP, RMI, Bluetooth, etc.), the platform should choose the appropriate protocol for each situation and use it. This will allow one to create agents that are independent of the communication protocols available on the machine, leaving the actual sending of the information as a task for the platform.

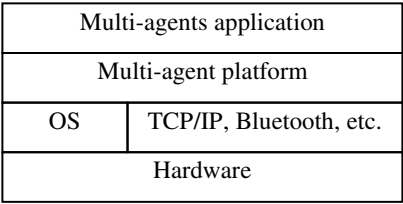


Fig. 1. Coarse-grained view of a device executing an agent platform and an agent application

Because the choice of an agent platform has a great influence on the way the agents are designed and implemented, FIPA¹ has produced standards that concern how an agent platform should be structured [6], i.e. what are the basic services it should offer (e.g. a message transport service, an agent management service, a directory service). These standards exist to ensure a uniform design of the agents, independent of the multi-agent platform. Thanks to it, for instance, at the moment several agent platforms are deployed in several cities of the world on which agents can go and execute and cooperate with other agents [1].

In the next chapter we will give a brief description of several agent platforms designed for small (and wireless) devices, on which the limitations of computing resources have a great influence on the complexity of the applications. For each platform we will try to put in evidence the targeted devices and the wireless technologies it works with, as well as if it is FIPA-compliant or not and what are its most relevant characteristics. All the platforms we have investigated are Java-based, so we will try to specify for each one what is the Java Virtual Machine (VM) it needs.

4 Multi-agent Platforms for Small Devices

Due to the limited resources available, the agent platforms for the small devices need to have a light architecture. Most of these platforms are designed to run completely (i.e. with all the services they provide) on the small device, while others are keeping some of the services (or sometimes all) on a server and they are downloaded on the device only when needed. From this point of view, we can classify existing platforms in three categories replicating the conceptual classification presented in [14]:

- *Portal platforms* that don't execute the agents on the small devices, but on other hosts. The small device is used only as an interface with the user.
- *Embedded platforms* are executed entirely on the small device(s), together with one or several agents.
- *Surrogate platforms* that are executing only partially on the small device, while some part of them is executed on other hosts.

¹ Non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents. <http://www.fipa.org>

In what follows, we will present the basic characteristics of several platforms from all categories.

4.1 Portal Platforms – MobiAgent

The MobiAgent [13] system architecture consists of three main components: a handheld mobile wireless device, an Agent Gateway and the network resources. The Agent Gateway is executing an agent platform and the agents for that platform. When the user wants to delegate a task to an agent, the mobile device connects to the gateway and *downloads an interface* that will configure the agent for her. The agent will perform its task and it will later report the results via the same mechanism.

Note that neither the platform, neither the agents are running on the device, the only application running there is a MIDlet that configures the agent. This approach does not impose anything to the platform or to the agents running on the Agent Gateway; the platform running there may be FIPA-compliant, but this is not mandatory. This infrastructure is available on any device supporting J2ME/CLDC/MIDP, including mobile phones. The connection between the device and the gateway is done via HTTP over a wireless network. Unfortunately, no links for downloading the platform are currently available.

4.2 Surrogate Platforms – kSaci

kSACI [9] is a smaller version of the SACI platform [15] suited to the kVM. SACI is an infrastructure for creating agents that are able to communicate using KQML [10] messages. Each SACI agent has a mailbox to exchange messages with the others. Its architecture also contains a special agent, called the *facilitator*, offering the white- and yellow-pages services of the system. The architecture of the platform is not FIPA-compliant.

The kSACI platform is usable on small devices running the kVM, but the platform is not entirely situated on the small device. In fact, the facilitator is on a server and on every small device there is only one agent running. This agent should contain a mailbox used to exchange messages, but it doesn't know how to pass messages to the others; it communicates via HTTP with a proxy running on a desktop machine that further passes the messages to and from other agents. This solution makes the agent lighter, so this platform can be used with devices as small as a mobile phone or a two-way pager.

4.3 Embedded Platforms

Several multi-agent platforms exist. Among them few are suited to be embedded on small devices. We have investigated four of them, but, due to space limitations, only two are presented here. The interested reader is invited to [4] for more information about the two other platforms, MAE and microFIPA-OS.

4.3.1 AgentLight

AgentLight [2] is an agent platform with the objectives of being capable of deployment on fixed and mobile devices with various operating systems and that can operate over both a fixed and a wireless network. Its authors are using a set of profiles that allows one to configure the platform for several devices and/or operating systems. Also, this platform aims to be FIPA-compliant and operating system agnostic.

The architecture of AgentLight is called “half-daisy” and consists of an *agent container* running on each device on which several agents can be placed. If the two agents on the same device communicate with each other, the agent container does the communication internally, but if the agents are on separate devices, the agent container acts as a proxy.

Each agent is seen as a set of rules and it is using for execution an inference engine provided by the agent container. This architecture is not entirely FIPA-compliant (e.g. the directory service is missing) and it is still work in progress. It can be run on devices with J2ME/CLDC/MIPD and with the kVM and the smallest device targeted is a mobile phone, although for the moment there are not any tests to prove a mobile phone can support the platform, especially the inference engine that can be quite heavy at the execution time.

4.3.2 LEAP

The Lightweight Extensible Agent Platform (LEAP 3.0) is probably the most known agent platform for small devices. It is the result of the LEAP project [12] that had the objective of creating a entirely FIPA compliant platform with similar objectives to the AgentLight Platform: be capable of deployment on fixed and mobile devices with various operating systems and deployed in wired or wireless networks. Since the last version, 3.0, LEAP is an add-on of the JADE platform [7] and it uses a set of profiles that allows one to configure it for execution on various machines, OS and Java VM.

The architecture of the platform is modular and is organized in terms of modules among which some can be mandatory, requested by FIPA specifications, while other are optional. The mandatory ones are the *kernel module* that manages the platform and the lifecycle of agents and the *communication module*, which handles the heterogeneity of communication protocols.

Some of the modules are device-dependent (e.g. the communication module), while others are independent of the characteristics of the device (e.g. a security plug-in for the communication module). The platform is operating system agnostic and it can run on devices ranging from mobile phones and PDAs to workstations. It can work over wireless networks that support TCP/IP, like WLAN or GPRS.

As in AgentLight, the architecture of LEAP splits the platform in several containers, one for every device/workstation used, with one or several agents in each container. These containers are responsible for passing the messages between agents by choosing the appropriate communication protocol available. One of these containers has a special role and it is called *main-container*; it contains two special agents that implement the FIPA specifications to provide the white and yellow pages services. This main-container cannot be executed on a small device, but only on a PC. In order to allow the execution of containers even on very limited devices, LEAP has

two execution modes: stand-alone and split. In the former, the container is completely embedded on the small device, while in the former, a part of the container is executed on a PC.

Table 1. Characteristics of multi-agent platforms for small devices

Platform	MobiAgent	kSACI	AgentLight	LEAP	MAE	micro FIPA-OS
Connection to SD	portal	surrogate	embedded	surrogate / embedded	embedded	embedded
Smallest targeted device	mobile phone	mobile phone	mobile phone	mobile phone	PDA	PocketPC
FIPA-compliant	it can be	no	yes (?)	yes	no	yes
No. of agents on device	0	1	several	several (pref. 1)	several	several (pref. 1)
Available for download	no	yes	yes	yes	no	yes
JavaVM	kVM	kVM	kVM	various	various	PersonalJava

5 Discussion and Conclusions

In this article we have presented some benefits of using multi-agent systems with the smart devices. But this usage of multi-agent systems on small devices has its drawbacks, i.e. the need for an existing infrastructure that will allow the creation and execution of agents and the communications between them: a multi-agent platform. We have presented a set of multi-agent platforms for small devices, underlining some of their major characteristics and architectural concepts. We have used the three main directions proposed in [14] to bind smart devices and multi-agent platforms: portal, surrogate and embedded. It was beyond the scope of this paper to identify one of these platforms as being the best. We believe there doesn't exist such a platform, but each one of them is suited for different applications. Table 1 summarizes the most important characteristics we have identified for each platform.

As future work we intend to set up a benchmark to evaluate these platforms within several scenarios; the results obtained will then allow us to analyse the platforms' performances in different situations and what is the real amount of computational resources (processing power, memory size) needed by each one. We have begun these practical tests by deploying LEAP on a Siemens S55 mobile phone. So far, the results are encouraging: the phone supports the execution of the platform and of an agent, although with a low execution speed.

When using a multi-agent platform on a small device, there is a high risk that the device resources will not be enough for both the platform and the agent(s) running on

it. Although the use of a platform can greatly reduce the developing cost and increase portability, it is clear that a trade-off should be found between the services the platform offers and its size. One might eventually arrive in the situation of using a small device like a mobile phone and a platform designed to work on it (like LEAP) and to discover that the platform is too big for the device's capabilities or there is no room left for any agents [3].

As for the FIPA-compliance, some projects such as Extrovert-Gadgets [5] for instance, question its utility for multi-agent applications on small devices. The advantage of compelling to some standards is clear, but these standards have not been designed for the particular characteristics of the small and wireless devices. A good example is presented in [11] where the LEAP platform is used, but some of its mandatory modules, which make it FIPA-compliant, are useless for some applications (e.g. ad-hoc networks).

References

1. AgentCities: <http://www.agentcities.org>
2. AgentLight – Platform for Lightweight Agents: <http://www.agentlight.org>
3. Berger, M. et al.: Porting Distributed Agent-Middleware to Small Mobile Devices. In Proc. of the Ubiquitous Computing Workshop, Bologna, Italy (2002)
4. Carabelea, C., Boissier, B.: Multi-agent platforms for smart devices: dream or reality?. In Proc. of the Smart Objects Conference (SOC'03), Grenoble, France (2003), p. 126-129.
5. E-Gadgets: <http://www.extrovert-gadgets.net>
6. FIPA Abstract Architecture Spec.: <http://www.fipa.org/repository/architecturespecs.html>
7. JADE – Java Agents DEvelopment Framework: <http://jade.cselt.it>
8. Jennings, N.R., Sycara, K., Wooldridge, M.: A Roadmap of Agent Research and Development. Int. J. of Autonomous Agents and Multi-Agent Systems 1 (1) (1998), p. 7–38
9. kSACI: <http://www.cesar.org.br/~rla2/ksaci/>
10. Labrou, Y., Finin, T.: A Semantics approach for KQML – A General Purpose Communication Language for Software Agents. In Proc. of Int. Conf. on Information and Knowledge Management (1994)
11. Lawrence, J.: LEAP into Ad-Hoc Networks. In Proc. of the Ubiquitous Computing Workshop, Bologna, Italy (2002)
12. LEAP – the Lightweight Extensible Agent Platform: <http://leap.crm-paris.com>
13. Mahmoud, Q.H.: MobiAgent: An Agent-based Approach to Wireless Information Systems. In Proc. of the 3rd Int. Bi-Conference Workshop on Agent-Oriented Information Systems, Montreal, Canada (2001)
14. Ramparano, F., Boissier, O.: Smart Devices Embedding Multi-agent Technologies for a Pro-active World. In Proc. of the Ubiquitous Computing Workshop, Bologna, Italy (2002)
15. SACI – Simple Agent Communication Infrastructure: <http://www.lti.pcs.usp.br/saci/>

Performance Evaluation of Two Congestion Control Mechanisms with On-Demand Distance Vector (AODV) Routing Protocol for Mobile and Wireless Networks^{*}.

Azzedine Boukerche

University of Ottawa, Canada
boukerche@site.uttowa.ca

Abstract. In this paper, we focus upon the congestion control problem in on-demand distance vector routing (AODV) protocol for mobile and wireless ad hoc networks. We propose to study two mechanisms to deal with the congestion problem within AODV routing protocol. First, we investigate a randomized approach of AODV protocol. then we present a preemptive ad hoc on-demand distance vector routing protocol for mobile ad hoc networks. We discuss the implementation of both algorithms, and report on the performance results of simulation of several scenarios using the *ns-2* ad hoc network simulator.

1 Introduction

Ad hoc wireless networks are expected to play an increasingly important role in future civilian and military settings where wireless access to wired backbone is either ineffective or impossible. However, frequent topology changes caused by node mobility make routing in ad hoc wireless networks a challenging problem [3,4,5,6].

In this paper, we are concerned with the congestion problem and packets dropped that are mainly due to the path break in the ad hoc network where the topology changes frequently, and the packet delay due to finding a new path and the transfer time. As a consequence, we present a Randomized AODV protocol (*R-AODV*), and *P_rAODV*, an extension to the AODV routing protocol using a preemptive mechanism. While *R-AODV* uses a randomized approach to divert routing packets from congested paths to reliable and less congested paths, *P_rAODV* scheme a preemptive approach to find a new path before the existing path breaks, thereby switching to this new path. Our main focus in this paper is to study the feasibility and the effectiveness of both congestion control mechanisms in the proposed protocol.

^{*} Part of this research is supported by Canada Research Chair program and Texas Advanced Research Program grant ATP/ARP

2 The Basic AODV Protocol

AODV uses an on-demand based protocol to discover the desired path, while using hello packets to keep track of the current neighbors. Since it is an on demand algorithm, it builds routes between nodes only upon request by the source nodes. It maintains these routes as long as they are needed by the sources. It is loop-free and self-starting. Earlier studies have also shown that AODV protocol scales quite well when we increase the number of mobile nodes [2]. AODV allows mobile nodes to obtain routes quickly for new destinations and does not require mobile nodes to maintain routes to destinations that are not in active communication. It also allows mobile nodes to respond to situations associated with broken link and changes in network topology in a timely manner, as we shall see later.

AODV uses a destination broadcast id number to ensure loop freedom at all times, since the intermediate nodes only forward the first copy of the same request packet. The sequence number is used in each node, and the destination sequence number is created by the destination for any route information it sends to requesting nodes. It also ensures the freshness of each established route. The route is updated if a new reply is received with the greater destination sequence or the same destination sequence number but the route has fewer hops. Therefore, this protocol will select the freshest and shortest path at any time.

- **Path discovery:** When a route is needed, the source invokes a path discovery routine. It has two steps: the source sends out a request and the destination return a reply. In what follows, we will discuss these two steps.

Request phase: When a route to a new destination is needed, the node uses a broadcast RREQ to find a route to that destination. A route can be determined when the RREQ reaches either the destination itself or an intermediate node with a *fresh* route to the destination. The *fresh* route is an unexpired route entry for the destination associated. The important step during the request phase is that a *reverse path* from the destination to the source can be set up. When a source node wants to find a path to a destination, it broadcasts a route request packet to its neighbors. The neighbors update their information for the source node, set up backwards pointers to the source node in their route tables, and rebroadcast the request packet.

Reply phase: When the request arrives at the destination or at an intermediate node that has a path to that destination a reply packet is returned to the source node along the path recorded. While the reply packet propagates back to the source, nodes set up forward pointers to the destination, and set up the *forward path*. Once the source node receives the reply packet, it may begin to forward data packets to the destination.

- **Data Packet Forwarding:** After the reply packet returns from the destination, the source can begin sending out the packet to the destination via the new discovery path, or the node can forward any enqueued packets to a destination if a *reverse path* is set up and the destination of the reverse path is the destination of the path.

A route is considered *active* as long as there are data packets periodically traveling from the source to the destination along that path. Once the source stops sending data packets, the links will time out and eventually be deleted from the intermediate node routing tables.

• **Dealing with the broken links:** If a link breaks while the route is active, the packets that are flowing in that path can be dropped and an error message is sent to the source or a local repair routine will take over. If the node holding the packets is close to the destination, this node invokes a local repair route. It enqueues the packet and finds a new path from this node to the destination. Otherwise, the packets are dropped, and the node upstream of the break propagates a route error message to the source node to inform it of the now unreachable destination(s). After receiving the route error, if the source node still desires the route, it can re-initiate a route discovery mechanism. However, we can add a preemptive protocol to AODV and initiate a rediscovery routine before the current routes goes down. In the next section, we will discuss this preemptive approach in more detail.

3 Randomized AODV (*R-AODV*) Protocol

AODV protocol is extended with a drop factor that induces a randomness feature to result in Randomized Ad-Hoc On-Demand Routing (*R-AODV*) protocol. During the route discovery process, every intermediary or router nodes between the source and the destination nodes makes a decision to either broadcast/forward the RREQ packet further towards the destination or drop it. Before forwarding a RREQ packet, every node computes the drop factor which is a function of the inverse of the number of hop counts traversed by the RREQ packet. This drop factor lies in the range of 0 to 1. Also, the node generates a random number from 0 to 1. If this random number is higher than the drop factor, the node forwards the RREQ packet. Otherwise, the RREQ packet is dropped. Dropping of RREQ packets does not necessarily result in a new route discovery process by the source node. This is due to the fact that the original broadcast by the source node results in multiple RREQ packets via the neighbors and this diffusing wave results quickly in a large number of RREQ packets traversing the network in search of the destination. A major proportion of these packets are redundant due to the fact that in the ideal case, a single RREQ packet can find the best route. Also, a number of these packets diffusing in directions away from the destination shall eventually timeout. Hence, in *R-AODV*, the aim is to minimize on these redundant RREQ packets, or alternatively, drop as much as possible of these redundant RREQ packets. The drop policy is conservative and its value becomes lesser with higher number of hops. As RREQ packets get near the destination node, the chances of survival of RREQ packets is higher. Hence, the first phase of the route discovering process, that is, finding the destination node, is completed as soon as possible and a RREP packet can be transmitted from the destination node back to the source node.

In *R-AODV*, the dropping of redundant RREQ packets reduces a proportion of RREQ packets that shall never reach the destination node, resulting in a decrease of network congestion. Hence, the ratio of the number of packets received by the nodes to the number of packets sent by the nodes, namely, throughput, should be higher in *R-AODV* compared to AODV.

The following algorithm is used in the decision making process of whether to drop the RREQ packets by the intermediary or routing nodes.

```

Step 1: Calculate drop_factor
        drop_factor = (1/(Hop_count_of_RREQ_packet + 1))
Step 2: Calculate a random value in the range of 0 to 1.
Step 3: If (random_value > drop_factor) then broadcast/forward RREQ_packet
        else drop RREQ_packet

```

4 Adaptive Preemptive AODV Protocol

In this section, we introduce the preemptive protocol first, then we discuss how this preemptive protocol is added into the original AODV,

The preemptive protocol initiates a route rediscovery before the existing path breaks. It overlaps the route discovery routine and the use of the current active path, thereby reducing the average delay per packet. During the development of *P_rAODV* development, we have investigated several preemptive mechanisms. In this paper, we will settle on the following two approaches:

(i) *Schedule a rediscovery in advance*: In this approach, when a reply packet returns to the source through an intermediate node, it collects the information of the links. Therefore, when the packet arrives at the source, the information about the condition of all links will be known, including the minimum value of the lifetime of the links. Hence, we can schedule a rediscovery $T_{rediscovery}$ time before the path breaks.

(ii) *Warn the source before the path breaks*: Some mechanisms are needed to take care of finding which path is likely to break. We can monitor the signal power of the arrived packets as follows: when the signal power is below a threshold value, we begin the ping-pong process between this node and its immediate neighbor nodes. This node sends to its neighbors in the upstream a hello packet called *ping*, and the neighboring nodes will respond with a hello packet called *pong*. Such *ping-pong* messages should be monitored carefully. In our approach, when bad packets were received (or we timeout on ping packets) a warning message should be sent back to the source during the monitoring period. Upon receiving a warning message, a path rediscovery routine is invoked.

In our preemptive AODV protocol, we combine the above two preemptive mechanisms, and add them to the original AODV. *P_rAODV* protocol is built based on the following assumptions. Future work will directed to eliminate such assumptions.

- *Congestion is caused only due to the path breaks*. Other factors such as the limited bandwidth available on wireless channels and the intensive use of resources in an ad hoc network cannot affect our protocol. We assume that the network has enough bandwidth available, and each node has enough capacity to hold packets.

- *The route discovery time can be estimated.* We estimate the discovery time by monitoring the request and reply packets, as the sum of the time required to process and monitor these two packets.
- *We can detect when a link breaks* by monitoring the transfer time of packets between the two neighboring nodes.

Based on the above assumptions, we present our P_rAODV as the following:

(i) *Schedule a rediscovery:* We have added an additional field, which we refer to as a lifetime, in the reply packet to store the information of the links along the new discovered path. Each link associates a time-to-live, a parameter that is set up in the request phase. The lifetime field contains the minimum value of the time-to-live of the links along a path. The reply packet also has an additional field, a timestamp, which indicates the time that the request packet was sent out.

Let us now see how we compute RTT , the estimated time needed to discover a new path. When the reply packet arrives at the source node, it computes RTT as $(arrivalTime - timestamp)$, then schedule a rediscovery at time $(lifetime - RTT)$. We use RTT as our estimated value for the time needed to discover a new path.

(ii) *Warning the source:* Recall that a preemptive protocol should monitor the *signal power* of the receiving packet. In our approach, we monitor the transfer time for the hello packets. According to [1], $P_n = \frac{P_0}{r^n}$, where P_0 is a constant for each link and n is also a constant, we can see that the distance, which connects to the transfer time of the packet via each link, has a one-to-one relationship to the signal power. Thus, by monitoring the transfer time of packets via each link, we can identify when the link is going to break.

In our protocol, when a node (destination) receives a real packet other than AODV packets, it sends out a pong packet to its neighbor (source), i.e., where the packet comes from. Note that there is no need to send this pong packet if we can include a field in the real packet to store the current time the packet is sent. However, in the original packet, it doesn't have such a field, hence, we need an additional *pong* packet to get such information. In both ping and pong packets, we include a field containing the current time that the packet is sent. When the source monitoring the transfer time is greater than some specific value c_1 , it begins the *ping-pong* process. If more than two pong packets arrived with the transfer time greater than a constant c_2 , the node will warn the source of the real packet.

Upon receiving the warning message, the source checks if the path is still active. This can easily be done, since each route entry for a destination associates a value, *rt_last_send*, which indicates if the path is in use or not. If the path is still in use, then a rediscovery routine is invoked.

Theoretically, these two cases can take care of finding new paths before current paths break. The first case is quite helpful if the data transmission rate is slow. If only one data packet is sent out per time-expired period, then even the current route is still needed. Note that this entry may not have packets to go through it during the next time period. This can happen if the interval between

two continuous packets is greater than the timeout period of the current route. The second case can be used to find a new path whenever a warning message is generated. However, those two cases are also based on some assumptions, such as the span of a link which can be estimated by the time out of ping packets and the values of those constants are accurate. Once we break those assumptions, the test results will be unexpected.

5 Simulation Experiments

We have implemented both *R-AODV* and *P_r-AODV* protocols using *ns-2, version 2.1b8*. The implementation of both algorithms is based on the CMU Monarch extension. The source-destination pairs are spread randomly over the network. By changing the total amount of traffic sources, we can get scenarios with different traffic loads. The mobility models used in our experiments to investigate the effect of the number of nodes are 500m x 500m field with different number of nodes. Each node starts its journey from a random location to a random destination with a randomly chosen speed uniformly distributed between 0-20m/sec. Once the destination is reached, another random destination is targeted after a pause. Varying the pause time changes the frequency of node movement. In each test, the simulation time is 1000 seconds. The experimental results were obtained by averaging several trial runs.

In order to evaluate the performance of *R-AODV* and *P_r-AODV* ad hoc routing protocols, we chose the following metrics:

- *Routing path optimality*: the difference in the average number of hops between the optimal (shortest) routing path and the real routing path.
- *Throughput*: (i.e., packet delivered ratio) is a ratio of the data packets delivered to the destination to those generated by the CBR sources.
- *Average delay per packet*: this includes all possible delays caused by buffering during route discovery latency, queuing at the interface queue, retransmission delays at the MAC, and propagation and transfer time.
- *Packet arrived*: this is the number of packets that are successfully transferred to the destination.

5.1 Simulation Results

In this section, we report on the performance results of simulation of several workload models of several scenarios we have used to evaluate the performance of both *R-AODV* and the preemptive ad hoc routing protocols.

AODV vs. R-AODV. Recall that the use of the drop factor in *R-AODV* will lead to a drop of some forwarding packets but the drop rate is conservative enough not to let the source node result in a loop of initiation of route discoveries.

The routing path optimality is illustrated in figure 1. As we can see, the routing path optimality is better in *R-AODV* compared to AODV on an average.

We also observe that the routing path optimality is about 20% better in *R-AODV* compared to AODV on an average. Larger percentages were observed with a larger field. This is mainly due to the induction of the dropping factor in *R-AODV* to reduce the node congestion.

Figure 2 shows the comparison of AODV with *R-AODV* on packet overflow due to the limited packet buffer space. It is observed that the number of packets overflowed in *R-AODV* is lower on the average. This is due to the induction of the drop factor in *R-AODV* to decrease node congestion. Recall that the packets overflow is dependent on the dynamic network topology. It decreases with an increase in the pause time of the nodes in both AODV and *R-AODV*. This implies that for more stable networks, wherein the nodes are static for higher quantum of time, the latency in finding a route to a node is relatively less.

The throughput is given in figures 3. As we can see, *R-AODV* exhibits a higher throughput on the average since most of the forwarding packets that were dropped were redundant packets and allows more packets to reach their destination without being dropped due to reduce node congestion. The higher stable network dynamics result in a higher throughput. Overall, since throughput is higher for *R-AODV*, lesser number of packets that were sent out by the nodes in the network were lost during the route discovery process, thus reducing network congestion in *R-AODV* when compared to AODV.

Figure 4 shows improved delay performance in most cases in *R-AODV*. Especially, for larger networks and higher mobility rate, the performance improvement shows a obvious contrast compared to AODV. For high mobility rate, the routes are frequently renewed with the newly discovered neighbors. Consequently the route update rate increases and *R-AODV* has more chances to find alternative paths due to decrease node congestion. The delay drops as the pause time increase. A main reason attributed to this observation is that as node mobility decrease, the route update interval increases since low node mobility results in less route updates. Thus, delay increases depending on node mobility. The overhead is presented in figure 5 which shows similar characteristics for a given pause time in both *R-AODV* and AODV.

AODV vs. P_r AODV. In general, in both AODV and P_r AODV, the number of nodes does affect their performance. We have observed that with the preemptive protocol, P_r AODV can significantly improve the performance of the throughput, the average delay per packet, and the packet delivered. The pause time and the packet rate can also affect its performance. In the course of our experiments, we have observed that P_r AODV has much better performance than AODV when the packet rate is low.

In our experiments, we set the traffic flow to 10-pairs, the interval traffic rate to 4, and the pause time to 600. As illustrated in Figure 6 our results indicate that the number of nodes does not have any effect on both protocols, AODV and P_r AODV. However, the preemptive AODV protocol indicates that it is more stable than AODV. This is mainly due to the fact that P_r AODV balances the traffic quite efficiently to a new path before the current one breaks. The best

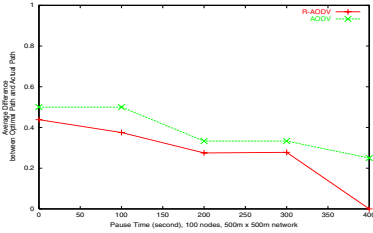


Fig. 1. Average Route Path Optimality – 500m x 500m

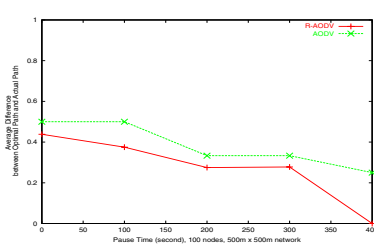


Fig. 2. Overflow – 500m x 500m

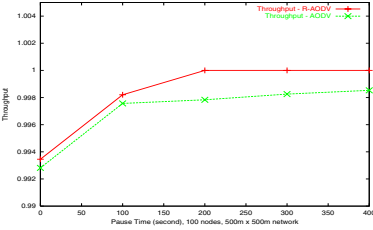


Fig. 3. Throughput – 500m x 500m

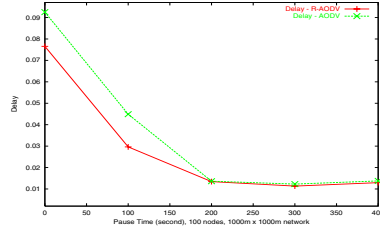


Fig. 4. Delay – 500m x 500m

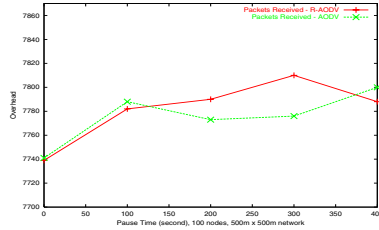


Fig. 5. Overhead – 500m x 500m

performance is observed when the number of nodes is set to 35, as illustrated in Figure 6. In our experiments, we have noticed that this type of behavior is always true with only 5 traffic pairs as well.

In our next set of experiments, we wish to study the effect of the number of traffic on the performance of P_rAODV . Based upon our initial experiments, (see above), we choose a scenarios with 35 nodes and 5 traffic pairs. We also set the pause time at 600 seconds. Our results are as indicated in Figure 7. In both protocols, we observe an increase of the the number of the packet arrivals as we increase amount of traffic. The throughput obtained with P_rAODV is lower when compared to the AODV protocol, and the delay obtained with the preemptive AODV protocol is lower when compared to the original AODV. This is mainly due the preemptive paradigm used in P_rAODV and the number of packets that are dropped during the simulation.

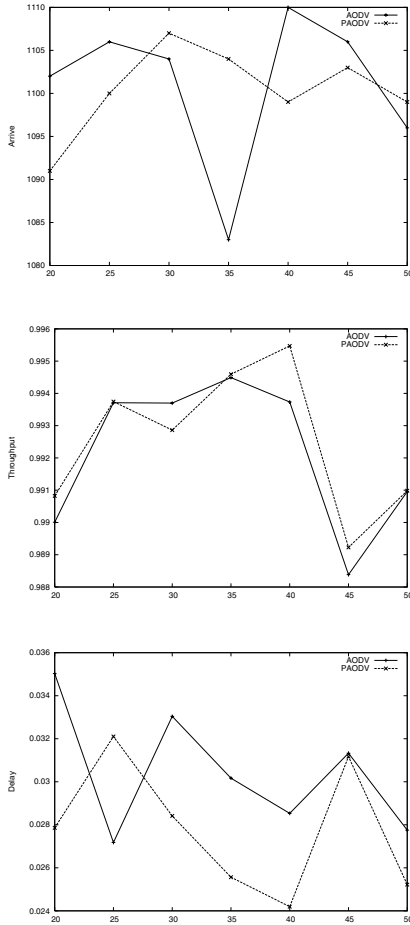


Fig. 6. Effect of the number of nodes

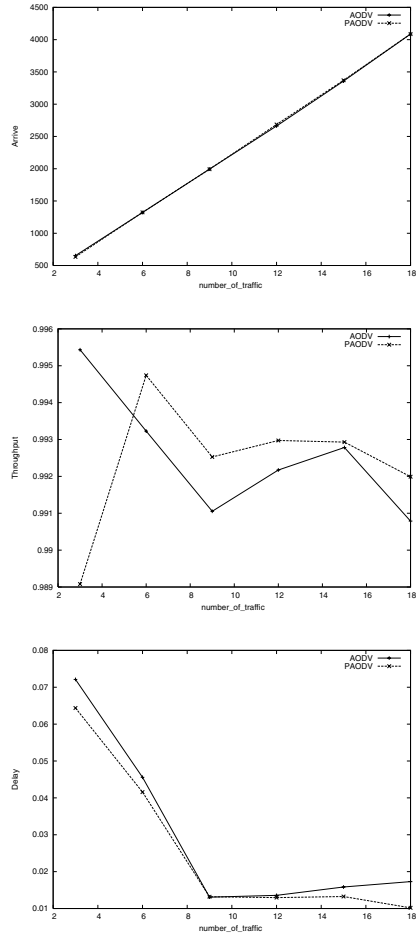


Fig. 7. The effect of the amount of traffic pairs

6 Conclusion

In this paper, we have studied the congestion problem in mobile and wireless networks. and we have presented two congestion control mechanisms for AODV ad hoc routing protocol, which we refer to as, *R-AODV*, a Randomized AODV protocol, and *P_rAODV*, an extension to the AODV routing protocol using a preemptive mechanism. We have discussed both algorithms and their implementations. We have also reported a set of simulation experiments to evaluate their performance when compared to the original AODV on-demand routing protocol.

Our results indicate that both *R-AODV* and *P_rAODV* protocols are easy to scale, the number of mobile nodes does not affect their performance. and the traffic is quite well balanced among the available paths in the network. While our results indicate that both protocols increase the number of arrival packets within

the same simulation time, they decrease the delay per packet when compared to the original AODV scheme.

References

1. Preemptive Routing in Ad Hoc Network URL:
<http://opal.cs.binghamton.edu/nael/research/papers/mobicom01.pdf>
2. A. Boukerche, "Simulation Based Comparative Study of Ad Hoc Routing Protocols" *34th IEEE/ACM/SCS Annual Simulation Symposium*, April 2001, pp. 85–92.
3. A. Boukerche, S. K. Das, and A. Fabbri "Analysis of Randomized Congestion Control with DSDV Routing in Ad Hoc Wireless Networks" *Journal of Parallel and Distributed Computing*, pp. 967–995, Vo. 61, 2001.
4. D.B. Johnson, D.A. Maltz, Dynamic Source Routing in Ad Hoc Wireless Networks, in: *Mobile Computing*, Editors: T. Imielinski and H.F. Korth (Kluwer Academic Pub. 1996), Ch.5, 153–181.
5. C.E. Perkins, Ad Hoc On Demand Distance Vector (AODV) Routing, *IEFT Internet Draft*, available at: <http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-02.txt> (November 1998).
6. C.E. Perkins, P. Bhagwat, Highly Dynamic Destination-Sequenced Distance Vector Routing (DSDV) for Mobile Computers, *Proceedings of ACM SIGCOMM'94*, (1994), 234–244.

Towards an Approach for Mobile Profile Based Distributed Clustering

Christian Seitz and Michael Berger

Siemens AG, Corporate Technology, Information and Communications,
81730 Munich, Germany,
`Christian.Seitz@mchp.siemens.de`
`m.berger@siemens.com`

Abstract. We present a new application for mobile ad hoc networks, which we call *Mobile Profile based Distributed Clustering* (MPDC), which is a combination of mobile clustering and data clustering. In MPDC each mobile host is endowed with a user profile and while the users move around, hosts with similar profiles are to be found and a robust mobile cluster is formed. The participants of a cluster are able to cooperate or attain a goal together. We adapt MPDC to a taxi sharing application, in which people with similar destinations form a cluster and could share a taxi or other public transportation resources.

1 Introduction

Technology developments like mobile devices and mobile communication have formed a new computing environment which is referred as mobile computing, in which an entire new class of distributed applications has been created. An mobile ad hoc network consist of hosts travelling through physical space and communicating in an opportunistic manner via wireless links. In the absence of a fixed network infrastructure, the mobile hosts must discover each other's presence and establish communication patterns dynamically. The structure of an ad hoc mobile network is highly dynamic. In an ad hoc network two hosts that want to communicate may not be within wireless transmission range of each other, but could communicate if other hosts between them are also participating in the ad hoc network and are willing to forward packets for them. The absence of a fixed network infrastructure, frequent and unpredictable disconnections, and power considerations render the development of ad hoc mobile applications a very challenging undertaking.

We present a new mobile ad hoc network application area, which we call *Mobile Profile based Distributed Clustering* (MPDC). In MPDC each mobile host is endowed with a user profile and while the users move around, hosts with similar profiles are to be found and a mobile cluster is formed.

We apply MPDC to a taxi sharing scenario. If a train arrives at a railway station or an airplane at an airport the different passengers may have the same destination e. g. a hotel. This destination address is part of a user profile and

stored on a mobile device. While people are waiting at the baggage terminal the mobile devices exchange the profiles and try to find small groups with similar destinations.

The rest of the paper is organized as follows. Section 2 gives an overview of related work of other clustering problems. In section 3 we outline the architecture of MPDC, define the ad hoc network model and make some assumption. The next section describes the used algorithms in our approach to MPDC and finally, section 5 concludes the paper.

2 Problem Classification and Related Work

Mobile Profile based Distributed Clustering comprises three main problems. The first problem is the dynamic behavior of an ad hoc network, where the number of hosts and communication links permanently changes. Furthermore, an expandable profile has to be defined and a mechanism must be created to compare profile instances. Finally, similar profiles have to be found in the ad hoc network and the corresponding host form a robust cluster, in spite of the dynamic behavior of the ad hoc network.

The term clustering is used in the research areas databases, data mining and in the mobile networks are. Clustering in mobile networks describes the partitioning of a mobile network in several, mostly disjoint, clusters [1,2]. The clustering process comprises the determination of a cluster head in a set of hosts. A cluster is a group of hosts, all able to communicate with the cluster-head. This clustering takes place at the network layer and is used for routing purposes. Clustering in the database or data mining area encompasses the search for similar data sets in huge data bases. In the surveys of Fasulo [4] or Fraley and Raftery [5] an overview of many algorithms for that domain can be found. Maitra [8] and Kolatch [7] examine data-clusters in distributed databases.

In *Mobile Profile based Distributed Clustering* mobile hosts are equipped with wireless transmitters, receivers, and a user profile. They are moving in a geographical area and form an ad hoc network. In this environment hosts with similar profiles have to be found. Therefore, MPDC must combine the two aforementioned clustering approaches to accomplish its objective. The problems, arising by means of the motion of the hosts could be solved by methods used in the mobile network area. Searching for similar profiles is based on algorithms of data clustering. Both methods must be adapted to MPDC, e. g. while in the database area millions of data sets must be scanned, in the MPDC application at the utmost one hundred other hosts are present. In contrast to data sets in databases ad hoc hosts move around and are active, i. e. they can publish their profile by their own.

There is other work that analyzes ad hoc clustering algorithms. Roman *et al.* [10] deals with consistent group membership. They assume that the position of each host is known by other hosts and two hosts do only communicate with each other, if it is guaranteed that during the message exchange the transmission

range will not exceed. In our environment obtaining position information is not possible, because such data is not always available, e. g. inside of buildings.

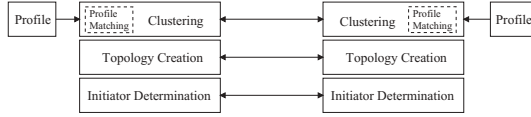


Fig. 1. Architecture

3 Architecture

In this section the architecture of MPDC is presented, which consists of three layers. With each layer at least one agent is associated. Figure 1 envisages the layered architecture of MPDC. The lowest layer is the *Initiator Detection* layer, which assigns the initiator role to some hosts. An initiator is needed in order to guarantee, that the algorithm of the next layer is not started by each host of the network. This layer does not determine one single initiator for the whole ad hoc network. It is sufficient, if the number of initiator nodes is only reduced. The *Virtual Topology* layer is responsible for covering the graph G with another topology, e. g. a tree or a logical ring. This virtual topology is necessary to reduce the number of messages, that are sent by the mobile hosts. First experiences showed, that a tree is the most suitable virtual topology and therefore we will only address the tree approach in this paper. The next layer is the most important one, the *Clustering* layer, which accomplishes both, the local grouping and the decentralized clustering. Local grouping comprises the selection of hosts which are taken into account for global clustering. Decentralized clustering encompasses the exchange of the local groups with the goal to achieve a well defined global cluster. The *Profile Matching* module, depicted with a dashed box in figure 1 is responsible for comparing two profiles.

Below, two definition are given to distinguish local grouping and decentralized clustering.

Definition: A *local group* g_i in a graph $G(V,E)$ is a subset of vertices $V_g^i \subseteq V$ with *similar* profiles according to a node v_i . Furthermore, there is a similarity operator σ , with $\sigma(v_i, V) = V_g^i$. Graph G consists of $|V|$ local groups, which together build a set, denoted with $\mathcal{G}$.

Definition: A *decentralized cluster* $\mathcal{C}$ is the intersection of all local groups g_i in a graph $G(V,E)$. A decentralized cluster $\mathcal{C}$ is obtained, if σ is applied to all V_g^j .

4 Algorithms

In this section the network model is defined and the algorithms for each layer are presented. The used middleware for MPDC is an ad hoc multi agent platform

which is described in Berger and Watzke [3], where each agent platform hosts an ad hoc management agent that makes itself known to its neighbors by generating a beacon at regular intervals and by listening to signals from other hosts around.

4.1 The ad hoc Network Model

An ad hoc network is generally modelled as an undirected graph $G_0 = (V_0, E_0)$ as depicted in figure 2a. The vertices v_i of G_0 represent mobile hosts. If the distance between v_i and v_j is below the transmission range r_t , the two vertices are connected by an edge e_{ij} . Due to the motion of the vertices, a graph G_0 as shown in figure 2a is only a snapshot of an ad hoc network, because in consequence of the mobility of the hosts G_0 will change.

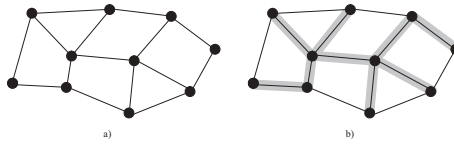


Fig. 2. Graph and a possible Spanning tree

Assumptions on the the mobile nodes and network are:

- Each mobile device has a permanent, constant unique ID.
- The transmission range of *all* hosts is r_t .
- Each host knows all its neighbors and its associated IDs.

4.2 Initiator Determination

At first, initiators must be determined who are allowed to send the first messages. Without initiators all hosts start randomly sending messages with the result that the algorithm in the next layer cannot start. We are not in search of one single initiator, we only want to guarantee, that not all hosts start the initiation.

There are active and a passive methods to determine an initiator. The active approach starts an election algorithm (see Malpani *et al.* [9]). These algorithms are rather complex and a lot of messages are sent. They guarantee that only one leader is elected and in case of link failures that another host takes the initiator role. This is not necessary for MPDC, because the initiator is only needed once and it matters little if more than one initiator is present. Therefore, we decided for a passive determination method, which is similar to Gafni and Bertsekas [6]. By applying the passive method no message is sent to determine an initiator. Since each host has an ID and knows all neighbor IDs, we only allow a host being an initiator, if its ID is larger than all IDs of its neighbors. The initiator is in charge of starting the virtual topology algorithm, described in the next section.

4.3 Virtual Topology Creation

Having confined the number of initiators, the graph G_0 can be covered with a virtual topology (VT). Simulations showed that a *spanning tree* is a promising approach for a VT and therefore we will only describe the spanning tree VT in this paper.

A spanning tree $\text{spT}(G)$ is a connected, acyclic subgraph containing all the vertices of the graph G . Graph theory guarantees, that for every G a $\text{spT}(G)$ exists. Figure 2b shows a graph with one possible spanning tree.

The Algorithm

Each host keeps a spanning tree sender list (STSL). The STSL contains the subset of a host's neighbors belonging to the spanning tree. The initiator, determined in the previous section, sends a **create**-message furnished with its ID to all its neighbors. If a neighbor receives a **create**-message for the first time, this message is forwarded to all neighbors except for the sender of the **create**-message. The host adds each receiver to the STSL. If a host receives a message from a host which is already in the STSL, it is removed from the list. To identify a tree, the ID of the initiator is always added to each message. It may occur that a host already belongs to another tree. Under these circumstances the message is not forwarded any more and the corresponding host belongs to two (more are also possible) trees.

In order to limit the tree size a hop-counter c_h is enclosed to each message and is each time decremented, the message is forwarded. If the counter is equal to zero, the forwarding process stops.

By using a hop-counter it may occur that a single host does not belong to any spanning tree, because all tree around are large enough, i. e. c_h is reached. The affiliation of that host is not possible, because tree nodes do not send messages in case the hop-counter's value is zero. When time elapses and a node does notice it does still not belong to a tree, an initiator determination is started by this host. Two cases must be distinguished. In the first one the host is surrounded only by tree nodes, in the other case a group of isolated hosts are existing. In both cases, the isolated host contacts all its neighbors by sending an **init**-message, and if a neighbor node already belongs to a tree it answers with a **join**-message. If no non-tree nodes are around, the single node chooses arbitrarily one of the neighbors and joins the tree by sending an **join-agree**-message, to the other hosts a **join-refuse**-message is sent. If another isolated host gets the **init**-message, a **init-agree**-message is returned and the host sending the **init**-message becomes the initiator starts creating a new tree.

Evaluation

The reason for creating a virtual spanning tree is the reduction of messages needed to reach an agreement. Let n be the number of vertices and let e be the number of edges in a graph G . If no tree is built each message must be forwarded to all its neighbors, which results in $2e - n + 1$ messages. Overlaying a graph with a virtual spanning tree, the number of forwarded messages is reduced to $n - 1$ plus $2e - n + 1$ messages for tree creation. Determining the factor A ,

when a tree becomes more profitable leads us to $A = \frac{2e-n+1}{2(e-n+1)}$. If on the average $e = 2n$, the amortization A results in $\frac{3n+1}{2n+2}$.

Table 1. Relation of edges and vertices in an arbitrary graph

#vertices n	#edges e	amortization A
9	26	1.22
19	45	1.33
26	71	1.27
50	135	1.28

To confirm this formal work, some arbitrary ad hoc networks were investigated (see table 1). The amortization value A does never exceed 1.4 (theoretical value), that means if each host sends only two messages to all of its neighbors, less messages are sent then without the spanning tree.

In the equation above, the tree maintenance cost are not taken into account. If a new host comes into transmission range or an other host goes away, additional messages must be sent to re-establish the virtual topology.

4.4 Local Grouping: Optimizing the Local View

In this section the subset of neighbor hosts are determined, which initially belong to a host’s local cluster, called a *group*. The algorithms presented in this and the following section depend on the used profile, which on its part depends on the application. We describe the grouping and clustering algorithms using the taxi sharing application with a very simple profile that only consists of a X- and Y-value, representing the destination of a person. These points are plotted in figure 3a in a coordinate system, which is the local view of the black point. The grey points are the X,Y-values of the points the black one is able to communicate with. Besides, all hosts are currently located around the origin of the coordinate system, which can always be achieved by rotating and translating the coordinate system.

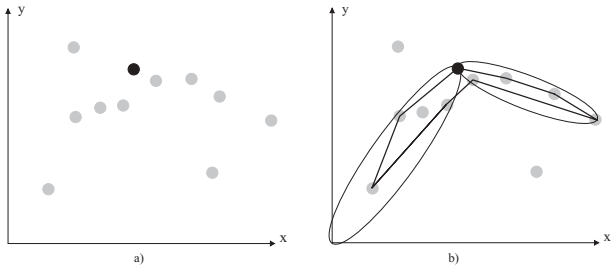


Fig. 3. Local Grouping of a mobile host

The algorithm starts by creating an ellipse from the origin of the coordination system to the local's point destination (P_d), see the left ellipse in figure 3b. An ellipse was chosen because it is a continuous shape, whereas a rectangle is not. The height (the semi minor half-axis) of the ellipse is in fixed relation to the length of the ellipse. All points inside this oval already belong to the local group. In order to interchange local groups, a more compact shape, e. g. polygon, including all points is desirable.

After creating the first oval, the local group has to be enlarged, if other points are still present. Therefore, a new destination point P_d must be found, which acts as new ending point of another oval (see second oval in figure 3b). This point must meet the following requirements:

- P_n must not harm the structure of the existing oval group, i. e. no P_n is allowed which result in a cycle, loop or a helix.
- To build the local group with as little ovals as possible, P_n must include as many other points as possible.

To guarantee the first requirement, the angle between the first oval and the potential next one has to be considered and may not exceed a specified value. In order to satisfy the second requirement, for each point P , which is not yet in an oval, it is calculated, how many other points would be in that new oval if it may become the next P_d and how many points are excluded in becoming further P_d points. The point with the largest difference becomes the new P_d .

This polygon has to be merged with the polygon of the first oval, because for the clustering we need one single polygon, that contains the whole group. Merging polygons is a tricky undertaking, because the merged polygon should not be larger than the ovals. But simply connecting two polygons does not meet this requirement. Therefore, additionally virtual profile points P_v must be inserted to reduce the size of a merged polygon.

Having determined a local group it is easy to proceed, when a host with a new profile appears. If the new point is inside the polygon, no changes are to be done. If it is outside the polygon and could become a new P_d the polygon is adjusted. In all other cases the point does not become a member of the group.

4.5 Decentralized Clustering: Achieving the Global View

In the previous section each host has identified its neighbor hosts that belong to its local group g_i . These local groups must be exchanged and a global cluster has to be achieved.

The algorithm presupposes no special initiator role. Each host may start the algorithm and it can even be initiated by more than one host contemporaneously. Initially, each host sends a **cluster**-message with its group-polygon enclosed to its neighbors which are element of the spanning tree. If a message arrives, the enclosed polygon is taken and it is intersected with its current local view of the host to get a new local view. This new local view is forwarded to all neighbors except for the sender of the received message. If a node has no other outgoing

edges and the algorithm has not terminated, the message is sent back to the sender. If node receives two messages from different hosts, only one message is forwarded in order to reduce the number of messages. If the algorithm has terminated, each host has the same lokal view, i. e. the global is achieved.

A critical point is to determine the termination of the clustering process. The algorithm terminates in at most $2 \cdot d_G = 4 \cdot c_h$ steps. If a host receives this amount of messages, the clustering is finished. But, if the tree is smaller or larger than it is supposed to be, waiting until $2 \cdot d_G$ are received is no termination criteria. For that reason, the real hop counter must be enclosed to a cluster message. If isolated points are adopted, c_h increases and the new c_h value must be announced.

5 Conclusion

In this paper we presented a new ad hoc applications called *Mobile Profile based Distributed Clustering* (MPDC). Each mobile host is endowed with its user's profile and while the user walks around clusters are to be found, which are composed of hosts with similar user profiles. The ad hoc network is covered with a virtual topology in order to reduce the number of messages. Each host determines a set of hosts, that belong to its local group. Finally, the local groups are exchanged and a global cluster is achieved.

We are simulating MPDC by means of a taxi sharing application and analyzing MPDC with respect to performance issues. But currently, the grouping and clustering depends on the used profile. One major goal will be to find a more generic solution, for other domains with more complex profiles.

References

1. S. Banerjee and S. Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. Technical report, Univ. of Maryland at College Park, 2000.
2. S. Basagni. Distributed clustering for ad hoc networks. In *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, Perth., pages 310–315, 1999.
3. M. Berger, M. Watzke, and H. Helin. Towards a FIPA approach for mobile ad hoc environments. In *Proceedings of the 8th International Conference on Intelligence in next generation Networks (ICIN)*, Bordeaux, 2003.
4. D. Fasulo. An analysis of recent work on clustering algorithms. Technical report, University of Washington, 1999.
5. C. Fraley and A. E. Raftery. How many clusters? Which clustering method? Answers via model-based cluster analysis. *The Computer Journal*, 41(8):578–588, 1998.
6. E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, COM-29(1):11–18, January 1981.
7. E. Kolatch. Clustering algorithms for spatial databases: A survey. Technical report, Department of Computer Science, University of Maryland, College Park, 2001.

8. R. Maitra. Clustering massive datasets. In statistical computing at the 1998 joint statistical meetings., 1998.
9. N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proc. of the Fourth Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 96–103, 2000.
10. G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Int. Conference on Software Engineering*, pages 381–388, 2001.

Simulating Demand-Driven Server and Service Location in Third Generation Mobile Networks

Geraldo Robson Mateus, Olga Goussevskaia, and Antonio A.F. Loureiro

Department of Computer Science, Federal University of Minas Gerais, Brazil
{mateus,olga,loureiro}@dcc.ufmg.br

Abstract. In this work we present a model for a 3G mobile network, emphasizing some of its main characteristics, such as provision of various types of services, including Voice, Video and Web; classification of users according to their mobility behavior in different geographical areas and during different times of day; varying demand generation patterns for each class of user; and exploration of the knowledge about the geographical location of users in order to improve the resource and information distribution over the network. A mobility simulator was implemented to generate a 24-hour demand in a typical metropolitan area and an integer-programming problem was proposed to optimize the server and service allocation in a dynamic fashion. Since the problem turned out to be NP-Hard, a heuristic method was developed in order to solve it. The obtained results were near optimal. The proposed approach allows a personalized and demand driven server and service location in 3G networks.

1 Introduction

Third-generation mobile telecommunication systems and technologies are being actively researched worldwide. Due to their "mass market nature", they should be designed as high-capacity systems, able to cope with the envisaged overwhelming traffic demands. The influence of mobility on the network performance is expected to be strengthened, mainly due to the huge number of mobile users. Another important characteristic, that is going to be part of 3G mobile systems, is the ability to determine the geographic location of mobile units [6]. This property will originate a whole new class of services that are based on client's location [9]. Among them are emergency services, traffic information and fleet management. Moreover, the ability to locate mobile units (MUs) will enable more sophisticated network management techniques and more precise resource and information distribution over the network. The growth of the competition in this sector is notable as well, making the ability to reduce system costs, in order to offer more competitive prices, critical in this scenario.

A typical mobile network environment consists of mobile hosts, fixed hosts and certain access points. The fixed hosts are all connected to a backbone (e.g., the Internet). MUs usually do not contact them directly, but use hosts closely located as access points to the backbone in order to minimize the distance which has to be bridged by a mobile connection line. Inserting such an environment into

a metropolitan area brings out some difficulties to the problem of where to locate the access points of the network and how to make the association between these points and the MUs. Such difficulties arise due to the variety of users' behaviors in metropolitan regions, as well as due to the dynamic demand distribution. For example, business regions present a higher concentration of people during working hours, whereas residential regions have quite an opposite behavior. Two questions that arise in this context are: (i) How to locate the servers over a geographic area in such a way that their capacity will not be wasted while the demand is concentrated somewhere else?, and (ii) How to locate the servers in such a way that location-dependent information will be delivered efficiently?

In this work we formulate and solve the problem of how to automatically distribute network resources according to a user demand, which may vary with time and space, so that the equipment maintenance and service provision costs are minimized. It means that, given a system composed of a set of servers, a set of services, and a set of mobile units, the attendance of the demand is made by those servers that are most closely located to the mobile users and that have the lowest maintenance cost at that moment. Such subsets of active servers may change from time to time, according to the users' mobility behavior and changes in demand distribution. The result is a sequence of activations and deactivations of servers so that the demand is satisfied and there is no capacity waste. The proposed approach gives a personalized, demand driven information supply.

The problem was treated as follows. A mobility simulator was implemented to represent a hypothetical city, a typical contemporary metropolitan area. The city was divided into geographical zones, such as a City Center and some suburbs, and populated with different groups of MUs, whose mobility behavior, as well as demand for different kinds of services was generated in order to simulate a typical 24-hour day input data for the model. Given the generated demand at a certain period of time, a server allocation was made in order to attend this demand. The allocation was achieved by modeling the system state as an integer-programming problem and by, afterwards, running an optimization algorithm on it. The output from the optimization process was used to reconfigure the system by activating or deactivating the selected servers and attending the current demand. The resulting integer programming problem turned out to be NP-Hard [5], so a heuristic method was necessary in order to solve it. Lagrangean Relaxation technique [3,10], integrated to Subgradient Optimization [10], was chosen for this task, and the obtained solution was quite close to the optimum.

The performance of the proposed approach was measured relatively to what we call a "Common" allocation approach, which was also implemented for the sake of comparison. The Common approach is based on the way server allocation is made in regular cellular systems. The performed experiments demonstrated the system's sensitivity to different kinds of parameters, such as user mobility, demand distribution over time and space, time of day, as well as the area type where events occur. The results show a significant system cost reduction.

The rest of this paper is organized as follows. Section 2 presents the developed mobility simulator. Section 3 contains a detailed description of the optimization

process, including the formulated integer programming model, and the heuristic technique applied. Section 4 provides some experimental results and their analysis. Finally, Section 5 discusses our conclusions and future work.

2 Mobility Simulator

Geographical Area. Transportation theory [1] divides the geographic area under study into area zones. The division is based on criteria related to population density and natural limits (e.g., rivers, highways). Considering mobile telecommunication requirements, it seems reasonable to assume that an area zone equals to a network area (e.g., macrocell, local exchange area). The area zones are connected via high-capacity routes, which represent the most frequently selected streets for movement support. It is worth noting that areas outside the city can also be modeled as area zones which attract a relatively low number of trips.

Based on the above observations, we chose to model the simulated geographic area as a twenty-kilometer-radius radial city, composed of four area types: city center, urban, suburban, and rural. Figure 1 is a representation of such a city. The model consists of 32 area zones (eight per city area type), four peripheral (one per area type) and four radial high-capacity routes.

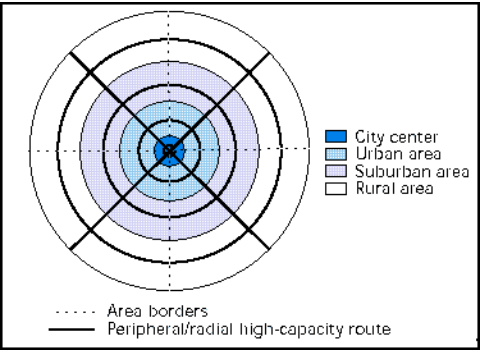


Fig. 1. The city area model consisting of area zones connected via high-capacity routes.

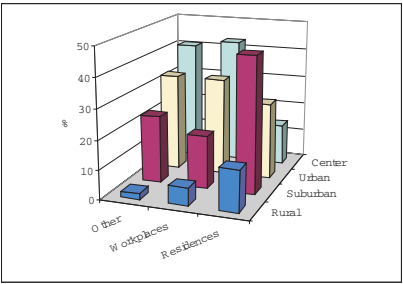


Fig. 2. The distribution of movement attraction points over the city area.

Movement attraction points represent locations that attract population movements and at which people spend considerable time periods. Examples are workplaces, residences, and malls. Each movement attraction point characterizes the population group it attracts. In our simulator, we consider movement attraction points to be residences, workplaces, and other points (e.g., shopping centers, parks). Figure 2 presents the assumed distribution of movement attraction points over the whole city area. Note that within a certain area type (e.g., urban, suburban) the movement attraction points are uniformly distributed.

Population. The population is divided into mobile unit groups according to the mobility characteristics of the individuals and on the kind of demand they generate [11]. The groups created are 24-hour delivery boy, common worker, housekeeper, and taxi driver. A movement table was associated to each of these groups in order to determine their mobility behavior. This was done dividing the day into time periods, and associating a probability to a group of being at a certain location at a given moment.

A call distribution and a set of requested services are also associated to each group. Some examples are: *24h Delivery Boy*: Call Distribution: Poisson with mean equal to 10 min; Service: Voice with duration exponentially distributed with mean equal to 80 s. *Common Worker*: Call Distribution: Poisson with mean equal to 14 min; Services: Voice with duration exponentially distributed with mean equal to 80 s, Web with duration exponentially distributed with mean equal to 180 s, Video with duration normally distributed with mean equal to 1 h.

A speed variation is associated to each group. It is assumed that the city has a pre-defined average speed that serves as reference. For example, a delivery boy moves 10 km/h faster than the city average speed, whereas a common worker moves exactly at the average speed.

The way MUs move depends on each group movement table, movement attraction point distribution and topology. The choice of a new destination is made by first choosing the destination area (using the group movement table, and the attraction point distribution), and then choosing a specific area zone and a particular coordinate at that area zone. These procedures are performed considering a uniform distributed probability. The route toward the destination is selected using the shortest path between the current and the destination positions as a sequence of radial and peripheral high-capacity route passages.

In the beginning of the simulation, all mobile units are assigned a home place and a working location. Home locations are uniformly distributed among those area types with greater residential attraction point concentration, whereas workplaces are chosen inside those area types with greater workplace concentration.

Cost definition is a very important part of the simulation process. Costs may vary in both time and space. It means that if we are interested in differentiate servers by their location, their use may be minimized or even avoided. In a similar way, if costs are strictly dependent on a given time period (day, year), their work can be reduced during those periods. Connection (or service attendance) costs can also be differentiated according to various factors. Among them are the physical distance between the server and the mobile unit, the number of channels required by a particular kind of service (Voice: 1 channel, Web: 3 channels, Video: 2 channels), time of the day, or even a priority associated to a particular user.

3 Optimization Process

Modeling. In order to build a mathematical model for a mobile network that supports multi-service provision in a demand driven way, we combined some well-known combinatorial optimization models, such as Incapacitated Location

Problem and P-Median Location Problem, which have been widely studied [4,8,10], and then add some new characteristics to them.

The following notation is used: T : set of service classes; I : set of servers; U : set of active mobile units; d_j^t : binary parameter, expressing whether the mobile unit $j \in U$ is demanding a service $t \in T$ or not; p^t : number of servers to be selected for every service $t \in T$; c_{ij}^t : variable location dependent cost of attending the demand d_j^t by server $i \in I$; f_i^t : cost of activating server $i \in I$ to provide service $t \in T$; x_{ij}^t : Boolean variable, which is set to 1, when the mobile unit $j \in U$, requiring service $t \in T$, is attended by the facility $i \in I$ and set to 0, otherwise; y_i^t : Boolean variable, which is set to 1, when server $i \in I$ is active to provide service $t \in T$, and set to 0, otherwise;

The problem is formulated as follows: (M) :

$$\min \sum_{t \in T} \sum_{i \in I} \sum_{j \in U} c_{ij}^t x_{ij}^t + \sum_{t \in T} \sum_{i \in I} f_i^t y_i^t \quad (1)$$

subject to:

$$\sum_{i \in I} x_{ij}^t = d_j^t, \quad \forall j \in U, \forall t \in T, \quad (2)$$

$$\sum_{i \in I} y_i^t \geq p^t, \quad \forall t \in T, \quad (3)$$

$$x_{ij}^t \leq y_i^t, \quad \forall t \in T, \forall i \in I, \forall j \in U, \quad (4)$$

$$x_{ij}^t \in \{0, 1\}, \quad \forall t, \forall i \in I, \forall j \in U, \quad (5)$$

$$y_i^t \in \{0, 1\}, \quad \forall t \in T, \forall i \in I. \quad (6)$$

The objective function (1) minimizes the total cost of activating the selected servers and attending the active MUs at each period. The set of constraints (2) ensures that each active MU demand for each class of service is attended by a unique server at each period. The set of constraints (3) imposes a minimum number of servers that must be selected for each class of service. The set of constraints (4) guarantees that no MU will be allocated to a non-selected server. Finally, sets (5) and (6) of constraints guarantee the integrality of variables.

Solution. As long as the formulated problem turned out to be NP-Hard [5], what we propose in this work is a method to obtain a tight bound, or an approximate solution, to the mixed integer programming problem M described earlier. The developed strategy makes use of the Subgradient Optimization technique [10] to successively maximize the lower bounds obtained for the problem using the Lagrangean Relaxation [2,3,4,10]. A combination of these two techniques allows us to determine the quality of the solution obtained.

We have tried several relaxations, and the best results have been obtained by choosing the set of constraints (2), disregarding the fact that for every active mobile unit $j \in U$ and for every class of service $t \in T$, the demand d_j^t must be attended by one and exactly one server. Associating a set of multipliers $\lambda_j^t, \forall j \in U, t \in T$ to the relaxed set of constraints (2) and using the same notation as in (M) , the following Lagrangean problem (LP) is formulated:

$$(LP) : \quad \min \sum_{t \in T} \sum_{i \in I} \sum_{j \in U} (c_{ij}^t - \lambda_j^t) x_{ij}^t + \sum_{t \in T} \sum_{i \in I} f_i^t y_i^t - \sum_{t \in T} \sum_{j \in U} \lambda_j^t d_j^t \quad (7)$$

Subject to (3), (4), (5) and (6)

The lower bound for M is obtained by resolving the LP associated to it using the current values for the Lagrangean multipliers. The following algorithm was used to find this solution:

Suppose a facility y_i^t is selected, then:

$$x_{ij}^t = \begin{cases} 1, & \text{if } (c_{ij}^t - \lambda_j^t) \leq 0 \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

It follows that the contribution of facility y_i^t in (7) is measured by:

$$\beta_i^t = f_i^t + \sum_{j \in U} \min(0, c_{ij}^t - \lambda_j^t) \quad (9)$$

Using the formulation above, the LP can be rewritten as:

$$(LP) : \quad \min \sum_{t \in T} \sum_{i \in I} \sum_{j \in U} \beta_i^t y_i^t + \sum_{t \in T} \sum_{j \in U} \lambda_j^t d_j^t \quad (10)$$

Subject to (3), (4), (5) and (6).

The optimal solution to the relaxed problem LP is obtained by the following algorithm:

1. Order the β_i^t parameter in increasing order, $\forall i \in I, t \in T$;
2. Make $y_i^t = 1$ if $\beta_i^t \leq 0$, $\forall i \in I, t \in T$;
3. If $\sum_{i \in I} y_i^t < p^t$, make $y_i^t = 1$ in increasing order of beta parameter until the required minimum number p^t of servers is selected, $\forall t \in T$;
4. Make $x_{ij}^t = 1$ if $y_i^t = 1$ and $(c_{ij}^t - \lambda_j^t) \leq 0$, and $x_{ij}^t = 0$ otherwise.

The value of the lower bound Z_{LB} is obtained by replacing the values of the obtained variables x and y in (7).

The initial upper bound $Z_{UB} = \infty$. As far as feasible solutions for the proposed problem are obtained, the upper bound is renovated by the lowest value found up to that moment for the objective function.

A solution is obtained from the previous lower bound solution by setting:

$$x_{ij}^t = \begin{cases} 1, & \text{if } \beta_i^t \text{ is minimal } \forall i \in I, t \in T \\ 0, & \text{otherwise} \end{cases} \quad (11)$$

In this way the set of constraints (2) is satisfied and the resulting solution is feasible. The upper bound Z_{UP} is obtained by replacing the new variables X in the objective function (7).

As the lower and upper bounds are determined, their values are used in the subgradient optimization iterative process. The total cost of the procedure is polynomial, and the output is the resultant duality gap ($\min Z_{UB} - \max Z_{LB}$) or the optimal solution itself. In any case, a quality guarantee can be associated to the obtained solution.

4 Experiments

In order to evaluate the performance of the proposed architecture, three different approaches were simulated, and their performances compared.

Common Allocation Approach. It is a simulation of the way demand attendance is performed in today's traditional cellular networks. The geographic area is divided into cells and usually one server (Radio Base Station) is responsible for service provision to all mobile units physically inside the cell region. Each area zone of the simulated city area was treated as a separate cell.

Online Allocation Approach (OnAA). In this work, we propose the On-line Allocation Approach as a better alternative to the Common Approach. Its main characteristic is a periodic reconfiguration of the system's resources (a sequence of activations and deactivations of servers) based on the current demand distribution, and the current location of mobile units.

Every fixed amount of time, all information regarding the system's current state (i.e., which services are being requested, and from which locations) is collected and an integer programming problem is built. The Lagrangean Relaxation technique is then applied to find the closest solution to the optimum that was pre-defined as acceptable. The solution is then used to reconfigure the system's state, what means the demand is attended with the lowest cost at that instant. While mobile units move across the city area, mobile network resources try to go along with the generated demand.

Offline Allocation Approach (OffAA). It is similar to the OnAA, except for the fact that the integer programming problem is built for more than one time instance once and pre-solved off-line. The simulation is repeated twice using the same seed for random sequences. During the first time, all information regarding user mobility and demand generation is collected and periodically saved into a file. The integer programming problem is then built with a small difference—one new dimension, namely time, is added to the problem, and its solution is calculated for all simulated periods. The resulting problem presents a slightly higher computational complexity than the first one. However, because the computation was performed off-line, good results were obtained. During the second time the simulation was run, the pre-calculated solution was used to attend the demand.

The purpose of this approach was to be able to compare the overall performance of a real-time system reconfiguration (OnAA) to the performance of an off-line system reconfiguration, under the supposition that user mobility and demand generation could be perfectly predicted.

As we shall see in the next section, interesting results were obtained using this approach, what strongly motivates the study of mobility and demand prediction methods in mobile telecommunication systems.

Lagrangean Relaxation Performance Evaluation. The parameters that we found interesting to evaluate regarding Lagrangean Relaxation performance are the number of subgradient optimization iterations and the number of optimal solutions obtained. The proximity of non-optimal solutions to the optimum was not analyzed in this work but, according to [7], for all cases analyzed here, the solution is less than 10% from the optimum.

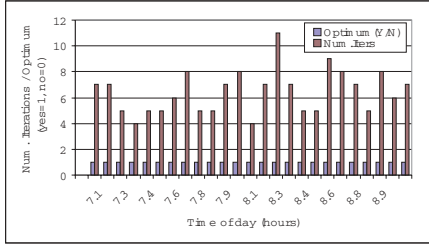


Fig. 3. Lagrangean Relaxation Evaluation for Online Approach (50 MUs).

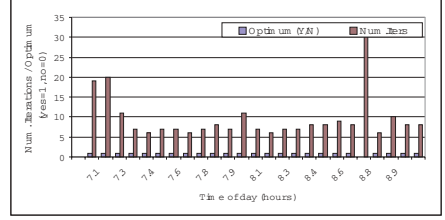


Fig. 4. Lagrangean Relaxation Evaluation for Online Approach (200 MUs).

Figures 3 and 4 show the number of iterations of the subgradient optimization loop performed to obtain solutions to the integer programming problem M during the period from 7:00 am to 9:00 am. For 50 MUs, in all instances, optimal solutions were obtained in less than 20 iterations. For 200 MUs, however, we can observe one case, near 8.48 am (8.8 h), for which the optimum was not found in 300 iterations (the value of 30, in the figure, was put not to distort the proportion of other values). In this case an approximate solution was used.

To measure the performance of Lagrangean Relaxation for the OffAA, experiments considering 50, 100, 200 and 500 users were performed. For less than 500 MUs, the optimum solution is found in less than 30 iterations. For 500 users, however, only an approximate solution could be achieved.

System Costs. The evaluation of system costs was performed through the values of the objective function of the integer programming problem. It means that all costs involved in server activation (f_i^t), as well as all costs involved in demand attendance (c_{ij}^t), were taken into account.

Figures 5 to 8 show a comparison between the costs of the three server allocation approaches. Results for different numbers of simulated mobile units, and different cost proportions are analyzed. In Figures 5 and 6 variable costs (c_{ij}^t) are, in average, 10 times higher than fixed costs (f_i^t). This proportion makes the system configuration more sensible to the distances between mobile units and servers. In Figures 7 and 8, on the other hand, fixed costs are, in average, 10 times higher than variable costs, what makes the system more sensible to server activation/maintenance costs.

In all four figures, the predominance of Common Approach costs over other approaches, and the predominance of Online over Offline Approach costs, can be seen. This result could be predicted since the Common Approach does not consider any cost evaluation in its demand attendance process, whereas Online and Offline approaches do so, with the following difference between them: the Online Approach optimizes system costs based on one period information about the system's state, and does it in real time. The Offline Approach has a "broader" view of the system's state, as long as it operates over several periods of time at once, what allows it to do a better resource allocation, achieving lower system

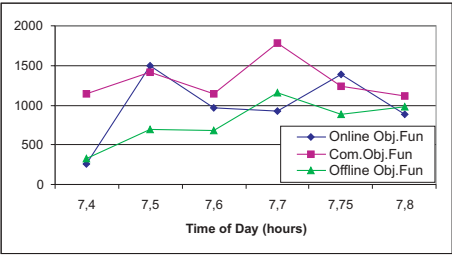


Fig. 5. System costs (50 users, variable cost predomination).

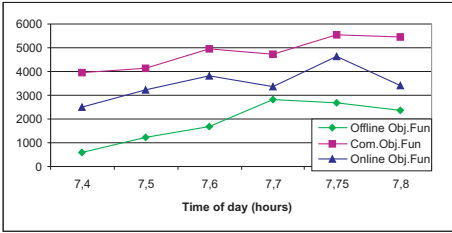


Fig. 6. System costs (200 users, variable cost predomination).

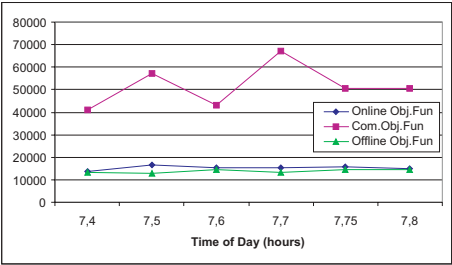


Fig. 7. System costs (50 users, fixed cost predomination).

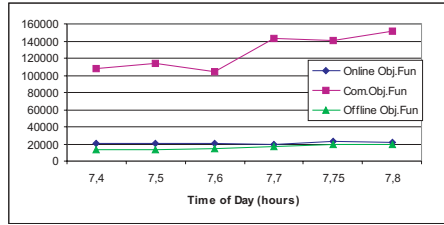


Fig. 8. System costs (200 users, fixed cost predomination).

costs. It is worth noting that such allocation approach could be applied in a real system only if a very powerful prediction mechanism was available.

Comparing Figures 5 and 6, it can be seen that, as the number of MUs grows, the relation among costs achieved by the three approaches remains and becomes even more explicit. The same fact can be observed comparing Figures 7 and 8.

One more aspect to be analyzed is the influence of cost proportion over the results. When fixed costs are expressively lower then variable costs, it is expected that more facilities are selected, in order to minimize the distances between servers and users. On the other hand, when the proportion is inverted, a minimum number of facilities is going to be selected and, if capacity is not treated, a concentration of users is going to occur on those servers of lowest maintenance cost. This fact is reflected in Figures 7 and 8, which present a much more expressive difference between the optimization and non-optimization approaches. What happens in these cases is a selection of a very low number of servers, more likely in the suburban areas, what makes the overall system costs become much lower. However, this kind of solution is not very realistic, since capacity issues are bound to arise in this scenario. The purpose of these tests, though, is to show how sensitive the proposed architecture is to such parameters.

5 Conclusions and Future Work

In this work we have formulated and solved the problem of how to dynamically distribute network resources in third-generation mobile telecommunication systems, aiming to minimize the equipment maintenance and service provision costs. A mobility simulator was implemented in order to represent a hypothetical city, which was populated with different groups of users, whose mobility behavior and demand for different kinds of services was generated in order to simulate a typical 24-hour day behavior. Given the generated demand at a certain period of time, a server allocation was made in order to attend it. This was done by modeling the system as an integer-programming problem and by, afterwards, running an optimization algorithm on it. The output from the optimization process was used to reconfigure the system by activating the selected servers and deactivating the others. The problem turned out to be NP-Hard, so a heuristic method was necessary in order to solve it. We chose the Lagrangean Relaxation technique to do the task, and obtained good results in terms of the solution proximity to the optimum.

The performed experiments demonstrated the system's sensitivity to different kinds of parameters, such as user mobility, demand distribution over time and over space, time of day, type of area where the events occur, and the relation between fixed and variable costs of the system. A significant system cost reduction was also achieved.

One of the limitations of the proposed model is the lack of capacity treatment, which is something we intend to improve. Some new constraints can be added to the proposed model in order to make it able to cope with capacity issues. Equations limiting the number of mobile units attended by each server and the radius covered by them are going to be elaborated. Some new features will probably be necessary in the developed mobility simulator in order to adapt it to this new scenario.

References

1. K.G. Abakoukmin. Design of transportation systems. 1986.
2. M.L. Fisher. The lagrangean relaxation for solving integer programming problems. *Management Science*, 27:1–18, 1981.
3. M.L. Fisher. An application oriented guide to lagrangean relaxation. *Interfaces*, 15(2):10–21, 1985.
4. R.D. Galvão and E.R. Santibañez-Gonzales. A lagrangean heuristic for the p_k -median dynamic location problem. *European Journal of Operational Research*, 58:250–262, 1992.
5. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman W.H. and Company, 1979.
6. J. Hightower and G. Borriello. A survey and taxonomy of location systems for ubiquitous computing. *Univ. of Washington, Comp. Sci. and Eng.*, 2001.

7. G.R. Mateus, O. Goussevskaia, and A.A.F. Loureiro. The server and service location problem in the third generation mobile communication systems: Modelling and algorithms. *Workshop on Mobile Internet Based Services and Information Logistics, Austria*, pages 242–247, 2001.
8. P.B. Mirchandani and R.L. Francis. *Discrete Location Theory*. Wiley, 1990.
9. Prasad Ranjee. Overview of wireless personal communications: Microwave perspectives. *IEEE Communications Magazine*, pages 104–108, 1997.
10. C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell, 1993.
11. M.N. Rocha, G.R. Mateus, and S.L. Silva. Traffic simulation and the location of mobile units in wireless communication systems. *First Latin American Network Operations and Management Symposium*, pages 309–319, 1999.

Designing Mobile Games for a Challenging Experience of the Urban Heritage

Francesco Bellotti, Riccardo Berta, Alessandro De Gloria, Edmondo Ferretti, and
Massimiliano Margarone

¹ DIBE – Dep.t of Biophysical and Electronic Engineering of the University of Genoa, Via
dell’Opera Pia 11/a, 16145 Genova, Italy
{franz, berta, adg, ed, marga}@dibe.unige.it
<http://www.elios.dibe.unige.it/>

Abstract. Ubiquitous gaming is a new emerging research area, that is gaining focus from industry and academy as smart wireless devices are becoming ever more widespread. We have explored this field implementing a treasure hunt game aimed at enhancing the visitor’s experience of the heritage in a city area, such as Venice. The game relies on standard, commercially available hardware and middleware systems. The paper describes the pedagogical principles that inspire the game and the corresponding structure of VeGame. We also discuss how combining mobile gaming and an educational focus may deliver to the player a challenging and engaging experience of the heritage, as shown by early user tests. Extensive qualitative and quantitative evaluation will start in June 2003.

1 Introduction

Mobile computing can support the user in the exact moment of the requirement – during her/his normal daily-life activities –, without the need for relying on a wired desktop computer. Our research group has explored such issues in the fields of tourism and culture, designing and developing multimedia mobile guides on palmtop computers for tourists [1, 2]. In this context, mobile services support visitors in the most important moment of their education experience: when they are up close to the subject, whether they’re viewing a painting in a museum, a monument in a park, or an animal in the zoo.

We are now interested in developing further this research field, exploring how mobile gaming can help tourists enhance their experience of art and history, favoring a pleasant and challenging interaction with documents and artifacts in a real urban context.

1.1 Designing Games to Support Acquisition of Knowledge

In this article we describe VeGame (Venice Game, Fig. 1) – a team-game played along the Venice’s narrow streets to discover the art and the history of the city – and

present design choices made by the interdisciplinary project team in an iterative process exploiting feedback from early usability tests.

VeGame has been designed and developed by the ELIOS lab (Electronics for the Information Society) of the University of Genoa in collaboration with the Future Center of Telecom Italia Lab of Venice.



Fig. 1. VeGame's initial page. The game uses a PocketPC, with Human-Computer Interaction software developed by the ELIOS lab of the University of Genoa.

The focus of our work is on the end-user [3] and, thus, in content and Human-Computer Interaction (HCI) design: how mobile services can deliver a useful and pleasant experience to the tourist, who is a special kind of general user - typically computer-illiterate - and cannot dedicate much mental effort to learn interaction with the computer. Her/his attention, in fact, is mainly dedicated to the surrounding environment (e.g. a picture, a church, a palace) which she/he is visiting.

Thus, the main contribution of this paper consists in presenting our design experience and discussing how mobile games may improve the quality of the tourist visit, by providing a challenging and compelling experience of the surrounding heritage and by favoring development and acquisition of knowledge directly in the moment of the visit.

We are thus interested in exploring pedagogic principles able to support knowledge acquisition in the field and studying how technology can systematically exploit them [4]. In particular, we focus on constructivistic learning philosophies [5, 6], which rely on construction of knowledge by situating cognitive experiences in authentic activities [7], and are the current subject of the ubiquitous computing research community [4,8].

According to these principles, VeGame features three main learning modalities:

- **Learn by doing** (or playing) , inviting players to operate in the field, exploring the territory and observing attentively the documents, such as pictures, palaces, bridges, etc.
- **Learn by thinking**, stimulating reflection on what has been seen and told, critical reasoning and concatenation of experiences and clues.
- **Learn through social interaction**, inviting players – both from the same and other teams – to play together for cooperative purposes and to comment on their

experience during the game. Moreover, we suggest players to interact with local citizens to answer specific questions on the local dialect and activities.

1.2 Challenges

Designing an edutainment experience in a city of art is a complex and challenging task, presenting issues that involve various technical and humanistic fields such as computer science, HCI, multimedia content design, game design, art, history, psychology and pedagogy. In order to face these issues we built a multidisciplinary team which has worked on the project since the early stages of understandings of settings, user needs capture, specification definition and feasibility study.

Defining the specifications, the design team analyzed several issues and questions including how to deliver a convincing and compelling edutainment experience (i.e. entertaining and valid from an educational point of view), stimulate and enhance interaction of visitors with the artifacts and other elements of heritage available in the territory, deliver contents from various fields (e.g. sculpture, painting, history, geography, poetry, architecture, local dialect, religion, etc.), favor social interaction of players among them and with other tourists and citizens.

In particular, we have analyzed how mobile computing technology can systematically contribute to achieve these targets.

2 VeGame

Relying on the above mentioned pedagogic principles, VeGame fosters an active exploration of the territory (streets, channels, squares, churches, palaces, etc.), its people and their activities (e.g. the famous Murano's glaziers), in order to have a better understanding and appreciation of the city's cultural heritage.

The educational experience is delivered through a game frame, since we consider it particularly suited to capture and appeal a wide audience, and allows us to exploit specific aspects of computer games able to support education and knowledge acquisition in a constructivistic way [9].

Players are grouped in teams – every team is assigned a HP IPAQ loaded with the VeGame software. The suggested team size is limited to 2 players, so that every person can actively participate to the games, in particular considering the limited screen size of palmtop computers.

The game can be played by several teams simultaneously (e.g. on adequately advertised special events, on school trips, etc.), or independently by tourist who want to visit Venice. VeGame challenges players and stimulates competition by assigning score as a reward for the players' activity. There is a session standings (for special events) and a hall of fame for all participants.

We have designed a game structure which closely resembles a treasure hunt game. The participating teams are invited to go through a sequence of stages, which are points of interest distributed in the city, such as important churches, palaces and squares (e.g. Fig. 2a).



Fig. 2. (a) The map of the urban area covered by VeGame, showing the stages and the path done until current stage, represented as a solid line. (b) The stage-menu with 5 games (in this case: HistoricalQuiz, RightPlace, VisualQuiz, Couples and VideoGame) selectable by the player.

At each stage, the player is presented with a stage-menu, which leads to the five games available at that stage (Fig. 2b). The games are instances of 13 predefined typologies. The game typologies were designed by our multidisciplinary team (partially inspired by common real-world games), and then refined according to early tests with users and experts.

We broadly divide games in three categories, according to the cognitive skills they mostly involve: observation games, reflection games, and action videogames.

- **Observation games.** These games privilege the sight as a sense to investigate and explore the surrounding environment. In general, these games tend to exploit the “knowledge in the world” in order to develop the cognition activity [7, 10]. They aim to stimulate spatial processing skills. Such skills are important in cognitive development since they allow an individual to create meaning by manipulating visual images [7,8].
- **Reflection games.** These games tend to favor reflection, discussion among team members, analysis of questions and possible answers considering clues available in the neighborhood and concepts learned previously during the game.
- **Action videogames.** These games stimulate similar skills as observation games. Their specificity lies in the animated graphics and engaging interaction, which helps to create a convincing and pleasant experience. They stimulate fantasy and evoke images and atmospheres that can be used to convey educational messages which are easily memorized by players.

We will now describe 3 different kinds of games (one per each of the above mentioned categories), showing our design choices and implementations and briefly discussing how technology can be exploited to support knowledge acquisition in-the-field by the general public and in a pleasant and entertaining fashion.

2.1 RightPlace

This game involves dragging some objects - represented by small icons – onto their right position on a map (Fig. 3a). Samples of objects can be details of a picture, prod-



Fig. 3. Sample games. (a) RightPlace: put the details in the right place on the picture. The picture represents the XVth century triptych of Bartolomeo Vivarini at Santa Maria Formosa. (b) HistoricalQuiz: What is the name of the store palaces that used to host the communities of foreign merchants in Venice? (c) Casanova. The lady is trying to grab the rose cast by Casanova.

ucts, names of cities, etc. The map could be a real geographic map – for instance, the player is asked to associate the names of XIV century’s Venetian naval basis to their right position in the Mediterranean area – or a picture, a palace’s façade, etc.

Depending on the actual implementations, the RightPlace game typology solicits different types of skills. In general, it aims at supporting creation of a mental model of the surrounding environment (e.g. the neighborhood of a church or of a square), possibly favoring visualization and localization of historical events and situations (e.g. historical maps), and stimulates the ability to understand a map and find correspondences between its parts.

In the case of pictures and other elements of the heritage, RightPlace favors an attentive analysis of the details and the relationships among them and with the bigger picture (e.g. a painting, a façade, etc.).

2.2 HistoricalQuiz

HistoricalQuiz is typically a question concerning the story of Venice (Fig. 3b). The question is generally tied with the place where it is made. For instance, the question at the Arsenal concerns interpretation of a Dante’s piece of the *Divina Commedia* where the work at the Venice’s Arsenal is described.

This game tends to stimulate critical reasoning and evaluation of alternatives (the HistoricalQuiz is often a multiple choice question). Typically, the quiz does not rely on previous knowledge, but it stimulates reasoning on clues available in the territory and/or on concepts addressed in the introduction and/or other previous games. The first user tests have shown that players appreciate this approach since it does not rely on scholastic knowledge, and rewards the actual field-activity of the tourist.

2.3 Casanova

This game can be played by two teams connected through a real-time Bluetooth channel. One team is Casanova – the famous libertine gentleman of the XVIII century – who has to collect roses and throw them to his lover. The second team plays the lady's role, who lies in a gondola and has to catch the roses (Fig. 3c). In the single player version (in case there is no team available in the neighborhood) the lady's role is played by the computer.

The Casanova game evokes the customs and the atmosphere of Venice in the XVIII century. In general, videogames are important to create interest and deliver images useful to represent meaning and content which can be recalled with ease. From early tests with users and experts we discovered that it is important to complement such images with explicit verbal knowledge in order to convey an educational message clearly understandable by the players. For instance, Casanova evokes the XVIII century Venice's atmosphere, which is really different from XIII - XVI centuries – introduced in the previous stage at the Arsenal – when Venice was a marine super-power. The Casanova game's conclusion text explicates the concept by saying that Casanova's Venice was important from the cultural viewpoint, but had limited international power.

Casanova explicitly fosters cooperation between teams: the total score, in fact, is shared among the two players. This favors interaction between teams and fun, since team players typically communicate among each other in order to help the partner to take the right decision on when to cast the rose or how to grab it.

We observed in field tests that Casanova – as the other videogames – is played by just one player at a time, for every team. Anyway, the others are not excluded: they gather around the team-mate supporting her/him and giving suggestions.

2.4 Inter-team Communication

According to the educational principles of VeGame – to acquire cultural knowledge and have an entertaining experience of the heritage – inter-team communication should not disturb the player visitors from their main educational task. Thus, inter-team communication involves (i) simple two-player games, where two teams have to meet in a physical space (the Bluetooth range is about 10 meters) and agree to play together, and (ii) a chat and messaging system – managed by a server connected via the GPRS cellular network. These two systems do not require real-time interaction and are not invasive nor strictly necessary to complete the game. But they support an important type of interaction: teams can freely help each other, exchange opinions to improve their field experience, comment on their activity, exchange personal information, and use these communication channels with extreme freedom.

3 Related Works

Recent research has combined wireless and ubiquitous computing technologies to implement a transparent linkage between the physical world and the resources available on the web [11, 12], enhancing the quantity and quality of information available to visitors and field workers. In fact, this model has been successfully implemented to enhance the tourist's experience in contexts such as museums and parks [1, 12, 13]. Recent studies have also highlighted the suitability of handheld devices for education - in particular for children from the ages 5 to 18 [14, 15].

In parallel, Ubiquitous Computing Game (UCG) research [16] has implemented games which incorporate live data from the real world and process players' location and movement. Our work does not involve innovative sensing technologies. Instead, we rely on standard, commercial hardware and middleware systems to study how mobile gaming can enhance the visitor's experience and support field learning. The UCG research also involves games with have an educational focus. For instance, Environmental Detectives [17] implements scenarios - ranging from airborne particles to radioactive spills - which allow educators to investigate a wide range of environmental issues, leveraging their pedagogical and entertainment potential.

Inspiring to tour guides, VeGame features an in-depth linkage of the game with the territory - which is an indispensable context, rich of local information and people. VeGame exploits mobile computing technologies in order to provide users two major added-values: the possibility of providing an enjoying experience - similar to that of a compelling videogame - and the support to explore autonomously the territory in order to accomplish challenging tasks. These factors are important to support learning, in particular attracting a category of public which is less prone to cultural activities.

4 Conclusions

In the VeGame project we are exploring how mobile technologies can enable new services and applications in nontraditional settings to enhance people's education and understanding of their cultural heritage.

Despite the proposed games typologies are relatively simple - they are to be understood and played with ease by the general public - we believe that these games are good samples of ubiquitous computing games. They exploit the computer as a constant presence [18] that accompanies the user as a friendly travel-mate, favoring cognition development, and stimulating interaction with the real-world, in particular with the heritage, the other players and the local people (e.g. the Dialect game makes questions about the meaning of Venetian words and phrases).

While we still miss quantitative data from extensive field user tests - they will start in June 2003 - it is important that our initial field tests show an enthusiastic user acceptance, essentially because the system is able to assist the visit privileging cognitive approaches based on field observation, social interaction and fun, which give the visitor a challenging and engaging experience of the heritage.

Acknowledgments. We would like to thank the Telecom Italia Lab, the Future Center, and, in particular, Roberto Saracco and Fabio Carati for their precious contribution to VeGame project.

We also thank the anonymous reviewers for their insightful and constructive comments, which have helped us to improve the quality and readability of the paper.

References

1. Bellotti F., Berta R., De Gloria A., and Margarone M., User Testing a Hypermedia Tour Guide, *IEEE Pervasive Computing*, Volume 1, Issue 2, April-June 2002.
2. Bellotti F., Berta R., De Gloria A., Gabrieli A. and Margarone M., E-Tour: Multimedia Mobile Guides to Enhance Fruition of the Heritage, in *E-work and E-commerce*, ed. Brian Stanford-Smith and Enrica Chiozza, IOS Press, 2001.
3. Weiser M., The computer for the 21st century, *Scientific Am.*, September 1991, pp. 94-104.
4. Mühlhäuser, M., Trompler, Ch.: Learning in the Digital Age: Paving a Smooth Path with Digital Lecture Halls. *Proc. Hicss 35th Hawaii Intl. Conference*, Waikola, HI, Jan 7-12, 2001. IEEE CS press, Los Alamitos, CA, 2001.
5. T.M. Duffy and D. H. Jonassen, *Constructivism and the Technology of Instruction: A Conversation*. Lawrence Erlbaum, Hillsdale, New Jersey, 1992.
6. B. Wilson (Ed.): *Constructivist Learning Environments: Case Studies in Instructional Design*. Englewood Cliffs, New Jersey. Educational Technology Publications, Inc., 1996.
7. Gardener H., *Multiple intelligences, the theory in practice*. N.Y.: Basic Books, 1993.
8. Abowd G. D., Mynatt E. D., and Rodden T., The Human Experience, *IEEE Pervasive Computing*, January-March 2002, Vol. 1, No. 1, pp. 48-57.
9. Natale M. J., The Effect of a Male-Oriented Computer Gaming Culture on Careers in The Computer Industry, *Computers and Society*, June 2002, pp. 24-31.
10. Hutchins E., *Cognition in the Wild*, MIT Press, Cambridge, Mass., 1995.
11. S. Pradhan, C. Brignone, J-H. Cui, A. McReynolds, M. T. Smith, "Websigns: Hypelinking Physical Locations to the Web", *IEEE Computer*, August 2001, pp. 42-48.
12. N. Davies, K. Cheverst, K. Mitchell, A. Efrat, "Using and Determining Location in a Context-Sensitive Tour Guide", *IEEE Computer*, August 2001, pp. 35-41.
13. F. Kusunoku, "Toward an Interactive Museum Guide System with Sensing and Wireless Network Technologies", *IEEE Int'l Workshop on Wireless and Mobile Technologies in Education (WMTE)*, August 2002, Vaexjoe, Sweden.
14. E. Soloway, C. Norris, P. Blumenfeld, B. Fishman, J. Krajcik, and R. Marx, *Handheld Devices are Ready-at-Hand*, *Comm. of the ACM*, June 2001, Vol. 44, No. 6, pp. 15-20.
15. C. Norris, E. Soloway, and T. Sullivan, *Examining 25 Years of Technology in Education*, *Communications of the ACM*, August 2002, Vol. 45, No. 8, pp. 15-18.
16. S. Björk, J. Holopainen, P. Ljungstrand, and R. Mandryk, Editors, Special Issue on "Ubiquitous Gaming", *Personal and Ubiquitous Computing*, December 2002, Vol. 6.
17. E. Klopfer, K. Squire, H. Jenkins, "Environmental Detectives: PDAs as a Window into a Virtual Simulated World", *IEEE Int'l Workshop on Wireless and Mobile Technologies in Education (WMTE)*, August 2002, Vaexjoe, Sweden.
18. Abowd G. D., Mynatt E. D., *Charting Past, Present, and Future Research in Ubiquitous Computing*, *ACM Transaction on Computer-Human Interaction*, Vol. 7, No. 1, March 2000, pp. 29-58.

QoS Provision in IP Based Mobile Networks

Vilmos Simon, Árpád Huszák, Sándor Szabó, and Sándor Imre

Budapest University of Technology and Economics, Department of Telecommunications
Mobile Communications and Computing Laboratory
Magyar Tudósok krt.2, H-1117. Budapest, HUNGARY
svilmos@hotmail.com, huszak@freemail.hu

Abstract. While IP is declared as the key technology of the future's wired and mobile communication, the currently used version of IP, IPv4 itself is not suitable to be used in mobile scenarios. Next generation mobile users require special support to provide connectivity, although they change their place of attachment to the network frequently [1]. In our work, we have created a network design algorithm and an agent (GMA/MAP) router selection algorithm in Regional Registration and Hierarchical Mobile IPv6 to optimise the handover management in IP based next generation mobile networks. Our research is supported by ETIK (Inter-University Center for Telecommunications and Informatics).

1 Introduction

Reduced radio cell sizes (increasing the number of handovers) and more complex protocols and services in next generation mobile networks increase the signalling overhead, causing significant signalling delay. This is critical in the case of timing-sensitive real-time media applications that call for mobile QoS [2].

Mobile IPv6 [3] is not capable of supporting real-time handovers. A solution is to make Mobile IPv6 responsible for macro-mobility, and to have a separate protocol to manage local handovers inside micro-mobility domains. Other solutions are Hierarchical Mobile IPv6 [4], or Regional Registration [5]. The basic idea of these hierarchical approaches is to use domains organised in the hierarchical architecture with a mobility agent on the top of the domain hierarchy.

The standard does not address the realisation considerations in detail of the hierarchical tree during the network design. The implementation details of the hierarchy are entrusted to the engineer. In our work we present a method, showing how to configure these hierarchy levels in order to reduce signalling traffic. We propose a novel graph-theory algorithm, which takes into consideration the mobile node's mobility model.

2 Protocol Optimisation

2.1 The Domain Forming Algorithm

The main concept of the algorithm is to connect adjacent radio cells with high mutual handover probabilities to the same access router. As a result, most handovers will take place between base stations belonging to the same access router, so the number of care-of

address changes – and the amount of related signalling messages, handover delays, etc. – are reduced. The algorithm is described mathematically as follows: in a graph-modelled network, the cells are the nodes of the graph (C) and the possible directions of the cell boundary crossing between adjacent cells will be the edges (E). c_1 and c_2 are adjacent points (cells), if $\{c_1, c_2\} \in E$. We must divide the $G = (C, E)$ graph into $G' = (C', E')$ subgraphs, so that the subgraphs contain the maximal weight spanning tree. We define weights to the graph's edges, not negative real numbers in the range $[0,1]$, based on the direction probability vector (the handover vector), equivalent to the probabilities of the movement directions.

We choose the edges one by one in the following manner: first we choose one of the edges with the biggest weight: $e_{\max} = p_{\max}$. The two nodes (cell), which are connected to this edge, $\{c_1, c_2\} \in e_{\max}$, can be joined, and after that we manage them as if they are one, $C' = \{c_1, c_2\}$. From the six edges outgoing from the two joined nodes, we must choose one with the biggest weight value, $\max(E')$, and the belonging node becomes the member of the common set, $C' = \{c_1, c_2, c_3\}$. From the eight edges belonging to the nodes, which are in the common set, we choose the next one so as to make a circle in the graph, namely if we have an $(c_1, e_1, c_2, e_2, \dots, c_k, e_k, c_{k+1})$ edge series, then $c_1 = c_{k+1}$. In this way, we can avoid the domains becoming too entangled and far-reaching. We can continue this algorithm, until the element number of C' reaches M , $N(C') \leq M$, so one domain will consist of M cells. When we cover the entire G graph with not-connected $G'_1, G'_2, \dots, G'_N$ subgraphs, the algorithm of forming domains is finished.

2.2 Relation between the Mobile Node's Speed and the Hierarchical Levels

There are several potential GMA (Gateway Mobility Agent) routers inside a HMIPv6 capable domain. The MN chooses one GMA from the list attached to the Router Advertisement. In the list there are Regional Care-of-Addresses. While the MN is moving in the domain area, it can choose another GMA router. Every GMA change must be reported to the Home Agent. For the best result the selection of the GMA must depend on the movement speed of the mobile node.

If the MN often changes its Access Router, a high level GMA must be chosen while a GMA on low hierarchical levels advertises Binding Updates too often.

For a slow moving MN the situation is the opposite. It is advisable to choose a GMA near the Access Router. Because of the slow moving, the change of the Regional CoA is rare, and the incoming packets get to the MN in a shorter way.

We give an algorithm, which selects the optimal Regional CoA (GMA) from the list.

It is noticeable that increasing the movement speed, the probability of the GMA changing is increasing too, if the MN selects the GMA randomly. The effect is the same, if the number of the hierarchy levels is increasing. Using the algorithm, in a router hierarchy, which has three levels, up to 50% gain can be achieved (section 3.2.).

3 Analytical Examination

3.1 Analytical Examination of Domain Forming Algorithm

We have examined analytically, how the number of signalling messages changes using a randomly formed lowest hierarchy level and the one, which is formed by our algorithm, to see if the algorithm really decreases the administrative overload generated by the handover.

In our examination we have modelled the cover of a busy avenue and its surroundings. We adjusted probabilities to the handover directions, modelling a realistic traffic on the given system of roads. The probabilities gave us the information in which direction (north/south/west/east) and with what relative frequency the mobile users will cross the cell boundary, based on online measurements.

The required bandwidth, if the mobile node changes a domain, is:

$$B = K \cdot N \cdot L + 2 \cdot L = L \cdot (K \cdot N + 2), \quad (1)$$

where N is the number of the foreign Correspondent Nodes, L is the size of the Binding Update message, and K is the number of Binding Updates which was sent to one foreign Correspondent Node.

In the first case, we made a random arrangement of four cell groups and we calculated how many domain handovers it causes when the mobile node is travelling on significant routes on our system of roads. One change of domain requires $K \cdot N + 2$ number of Binding Updates, and the bandwidth is defined by (1).

In the second case we used our domain-forming algorithm to see whether it really reduces the number of domain changes for the same significant routes. The results are given in Table I.

Table 1. Number of domain changes

<i>Route</i>	<i>Random</i>	<i>Algorithmic</i>
<i>(a)</i>	$5 \cdot L \cdot (K \cdot N + 2)$	$3 \cdot L \cdot (K \cdot N + 2)$
<i>(b)</i>	$3 \cdot L \cdot (K \cdot N + 2)$	$2 \cdot L \cdot (K \cdot N + 2)$

Based on the results obtained with the help of the domain forming algorithm the handover signalling load can be reduced by 30-40% on average (see Table I.), which makes it possible to improve the QoS parameters (for example in real time applications). The domain changing update traffic was calculated for other significant routes, too.

3.2 Analytical Examination of GMA/MAP Selection

We have examined how the number of re-registrations at the Home Agent changes using a random GMA/MAP selection and our selection algorithm. For simplicity in the three-layer hierarchy network, all of the routers are GMA/MAP capable routers. In this network, the MN can always choose from three different GMA/MAP routers. In this case, the MN receives a Regional Care-of-Address Extension message. In this message there are three different GMA/MAP router addresses.

In this sample the whole speed interval is partitioned in four smaller intervals (v_a , v_b , v_c , v_d). If we know in which interval the speed-rate is, we can choose the optimal

GMA/MAP router. We have calculated the probability of GMA/MAP changes in case of random GMA/MAP selection. As the results show, the probabilities that the MN changes its GMA/MAP router once, twice, and so on are given in Table II.

Table 2. Probability of GMA changes in random selection

GMA change	V _a	V _b	V _c	V _d
1	1	2/3	1/3	1/3
2	-	1/3	4/9	1/3
3	-	-	2/9	7/27
4	-	-	-	2/27
Σ	1	1	1	1
<i>M</i> (ξ)	1	1,33	1,88	2,07

The expected number of GMA/MAP changes can be calculated with (2) using the results from Table II:

$$M(\xi) = \sum_{i=1}^n p_i \cdot x_i \tag{2}$$

It is noticeable that in case of random agent selection, as the speed of the terminals increases (or when the hierarchical tree has more levels), the number of GMA/MAP changes also rises. As a result, the amount of signalling overhead is increasing significantly compared to the optimised scenario.

4 Conclusions

Comparing the signalling traffic load of a random network and the optimised one, the results show a significant reduce of the signalling traffic. An agent selection algorithm is also proposed, and its advantages are studied in comparison with a random agent choosing method. These results help to support global QoS in next generation networks.

Our future plan is to use computer simulations (like NS2 or OmNet++) to analyse the algorithms, which will help us to get more results. Even if further studies necessary, next generation IP based mobile networks will surely benefit of the enhanced IP mobility management solutions.

References

[1] I.F. Akyildiz, J. McNair, J. Ho, H. Uzunalioglu, W. Wang, "Mobility Management in Current and Future Communications Networks", IEEE Network, July/August 1998

[2] Hemant Chaskar, " Requirements of a QoS Solution for Mobile IP", draft-ietf-mobileip-qos-requirements-02.txt, 10 February 2002

[3] S. Deering, R. Hinden, "Internet Protocol, Version 6 Specification", RFC 2460, 1998

[4] Claude Castelluccia, K.Malki, Hesham Soliman, Ludovic Bellier, "Hierarchical Mobile MIPv6 mobility management", draft-ietf-mobileip-hmipv6-05.txt, 2001

[5] J. Malinen, C. Perkins, "Mobile IPv6 Regional Registrations", draft-malinen-mobileip-regreg6-00.txt

Design of a Management System for Wireless Home Area Networking

Tapio Rantanen¹, Janne Sikiö¹, Marko Hännikäinen¹, Timo Vanhatupa¹
Olavi Karasti², and Timo Hämäläinen¹

¹ Tampere University of Technology Institute of Digital and Computer Systems
Korkeakoulunkatu 1, FIN-33720 Tampere, Finland
{tapio.a.rantanen, marko.hannikainen}@tut.fi

² Elisa Corporation, POB 81, FIN-33211 Tampere, FINLAND
olavi.karasti@elisa.fi

Abstract. A management system for home area networks has been developed. The main design targets have been to support heterogeneous network technologies, automate the network configuration and management, and to enable application based Quality of Service support. The resulting system architecture has a five-layer functional architecture and a centralised topology with a management server. Also, the support of new proprietary management functionality has been designed for wireless access points and client terminals. A wireless LAN and Java based prototype of the management system is being implemented and its architecture presented.

1 Introduction

Both the number and variety of home applications and projected network technologies for home environment are growing. While office type of data transfer is still required, interactive gaming and consumer electronics are placing new demands on network services [1]. Broadband wireless technologies can replace wired networks because of the comfort and mobility provided [2]. Wireless Local Area Networks (WLAN) and Wireless Personal Area Networks (WPAN) are the main wireless technology category for home area. Currently, the most important standard WLAN technology used in home networking is the IEEE 802.11 [3].

No single management protocol has been established for the management of various equipment of the home network environment. Therefore, flexible management systems are required with the support for multiple management protocols. The Simple Network Management Protocol (SNMP) standard has been most widely adopted by network device manufacturers [4]. It uses a Management Information Base (MIB) to define attributes managed in a network device.

Quality of Service (QoS) support for enabling the different applications to operate and coexists is seen as a key functional requirement. The application QoS distribution can be divided into two main implementation approaches: bandwidth reservation and differentiated traffic handling. Bandwidth reservation systems use control messages to

allocate bandwidth for a flow before the actual data transfer. This approach is taken by the Internet Engineering Task Force (IETF) for Integrated Services (IntServ) [5].

Few new management specifications targeting at management of heterogeneous networks are emerging. IETF SNMPConf working group has been developing methods for using SNMP framework for policy management [6]. The term policy management means the practise of applying management operations globally on all managed elements with similar characteristics. The Distributed Management Task Force (DMTF) is developing the Web-Based Enterprise Management (WBEM) initiative with high level of interoperability support with existing management standards [7]. Sun Microsystems has extended the Java 2 platform with a Federated Management Architecture (FMA) standard. The FMA implementation is called Jiro, which enables interoperability of different management systems, and provides tools for easy management application programming. [8]

The home area network management is a new challenge. The management challenges in home environment have been identified and a high level architecture proposed in [9]. However, no technical architecture has been provided and the operation of the system is left open. Home network service and management architecture is provided in [10]. The architecture is flexible and takes advance of mobile code. However, it requires that the agents situated in home network appliances to be capable of running the mobile code. In heterogeneous network environment not all devices support running environment required for mobile code.

In this paper, an architecture and design for a QoS management system for home area wireless networks are presented. The system is targeted for managing of heterogeneous networks and to provide abstraction and automation to the home management tasks. Section 2 presents the architecture and functionality of the designed management system, while Section 3 gives an overview of the prototype implementation. Conclusions and future work is discussed in the final section.

2 Wireless Home Area Management System (WHAMS)

The conceptual layer architecture of the system is depicted in Fig. 1. First, physical *devices* represent the actual managed network devices, such as APs, LAN bridges, and terminals. The devices contain a set of managed attributes (e.g. variables, management procedures), which are organised into a MIB.

Attribute adaptors have been defined for hiding the details of management access protocols and the physical device parameters from the higher layers of WHAMS. Thus, the adaptors are a uniform interface layer to the needed attributes of devices. Table 1 presents the designed attribute adaptors.

Functions use the attribute adaptors to perform automated management tasks. Functions can be adjusted to be in observation or automation states. In an automation state, a function observes, generates notifications, and automatically reacts to changes according to the operation and configuration of the function. The target is to improve the network operation and performance that is related with the function.

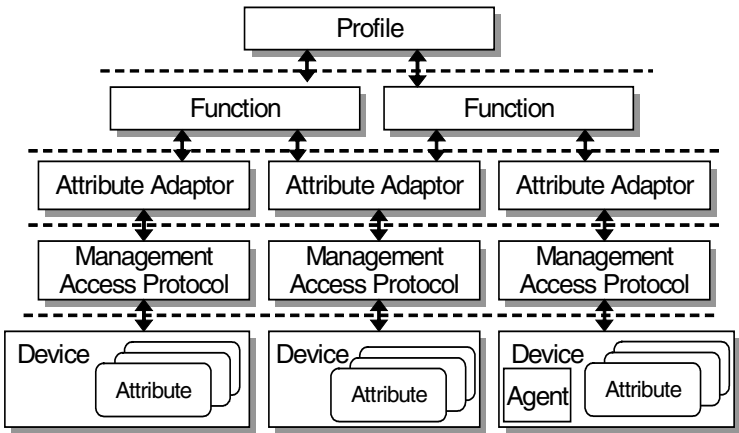


Fig. 1. WHAMS concept architecture with different layers

The following functions are defined: traffic function, frequency function, media function, security function, and auto configuration function. The traffic function observes traffic on all managed sub-networks, and balances the traffic loading of wireless networks by assigning client terminals to less loaded APs. The function provides a view of network traffic loading to a network manager.

The frequency function observes or automates frequency allocations using radio attribute adaptors. When automated this function tries to balance used frequencies so that interference from other networks is minimal.

The media function observes or automates bandwidth allocations of traffic flows. The function uses media connection attribute adaptors. When automated this function provides media connections with the requested bandwidth according to the connection priorities. This is established by limiting other traffic in the network when required. The security function manages the security policy of the whole network. When automated this function observes changes in the wireless networks (new devices), and ensures that the requested security level is achieved at all times.

The auto configuration function enables automatic configuration of devices in the network. The configuration requires that the client is running auto configuration software that performs the configuration of the wireless client.

A *profile* defines the network management configuration, defining the operation environment and its characteristics. Thus, a profile contains all parameters that are required by the functions for automated network management. A common policy is needed, as the operation of the different functions can be contradictory. A profile can emphasise the importance of certain applications and devices. In the current design, the profile rules are stated as priorities assigned for functions, terminals, and applications.

The resulting system architecture consists of a centralised management server, wireless APs and optional client terminals. The server will contain the management functionality. A client terminal can have added WHAMS specific functions for

Table 1. Attribute adaptors of the WHAMS system

Adaptor	Examples of attributes	Description
Status	Connection status, data transmission rate	Generic adaptor that can be used to any type of device
Radio	Radio type, used frequency, signal strength, radio usage	Abstracts the management of wireless media.
Security	Encryption, authentication, failed authentications	Abstracts the security issues of a device.
Media connection	Start address, end address, bandwidth control, delay, jitter	Abstracts the QoS management of an application flow.
Access point	Number of connected devices, loading	Abstracts the management of a access point.
Internet gateway	Loading, configuration, access type	Abstracts the management of Internet gateway.
Traffic control	Number of retries, number of errors, duplicate count	Abstracts traffic control management

enabling more efficient management. Each device is connected to the WHAMS management system via a management access protocol. As discussed, there are several different standard protocols available, while proprietary protocols can also be supported.

The efficient and full scale implementation of advanced management functions defined by WHAMS, such as application based QoS management, requires means to measure and analyse application traffic in network nodes. Adding of special measurement *agents* (WHAMS APs) can fulfil these functional requirements [11].

Still, the implantation of application QoS can be partly supported at the endpoints of a flow, where WHAMS specific functionality is easier to add. WHAMS is not platform specific but it can adapt to provide functionality that is possible with current technology.

3 WHAMS Prototype Implementation

A prototype of the WHAMS is being implemented for verifying the functionality. The prototype is presented in Fig. 2. The server software contains implementations of layer architecture components: management protocols, attribute adaptors, functions, and profiles. The WHAMS server is implemented as Java. The server also contains a WWW-server for loading the user interface implemented as Java applet. The server platform is PC computer with Windows 2000 operating system.

The WHAMS AP is capable of measuring flows and flow attributes such as throughput, delay, and delay variance. It provides bandwidth control in form of bandwidth reservation by queuing and access control for defined application traffic flows.

The WHAMS AP software can be divided into four modules: bandwidth control, measurements, packet analyser, and management access. The management access module contains sniffer functionality. The packet analyser examines packet headers

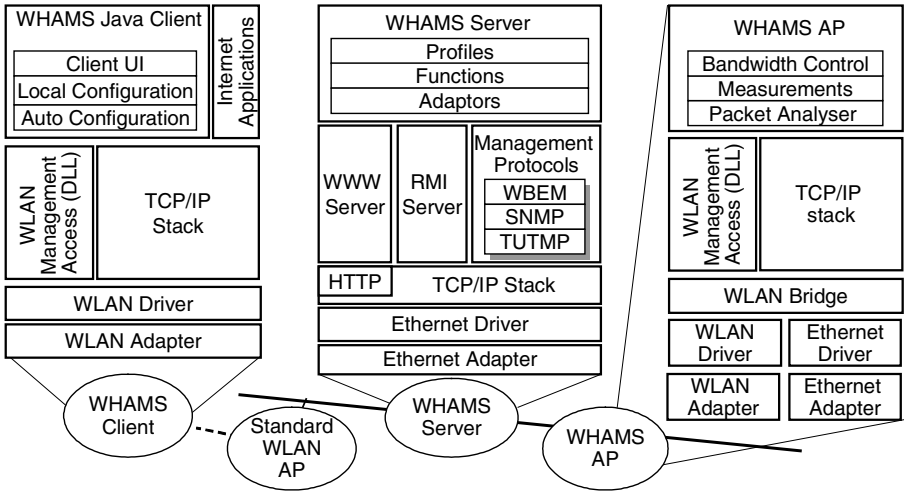


Fig. 2. WHAMS prototype topology and node architectures

for identifying flows. The measurements function performs actual flow measurements and stores the acquired data. The bandwidth control performs bandwidth reservations and control. It follows that the current reservations are fulfilled, and monitors the amount of unreserved bandwidth.

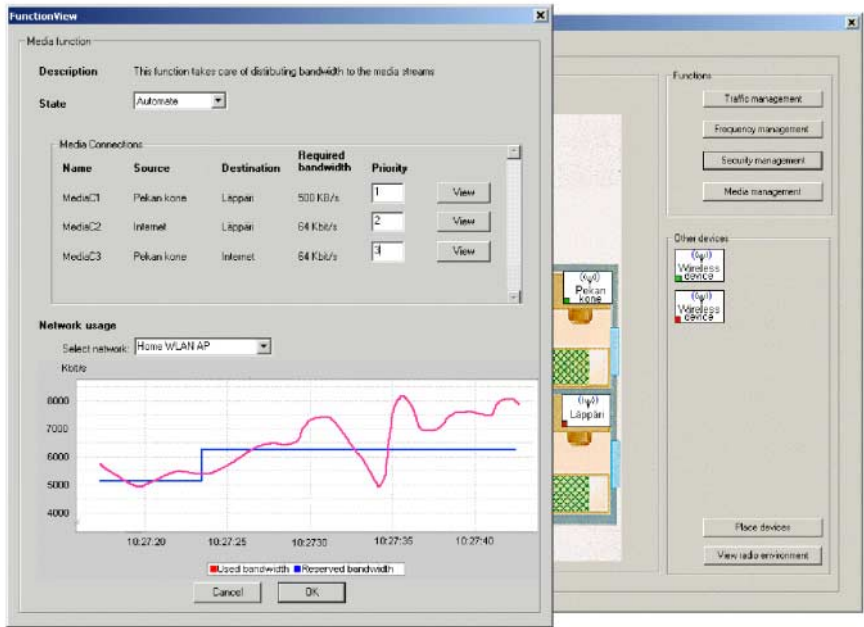


Fig. 3. Main view with floor plan (right) and the media function view (left) of the prototype

The user interface applet loads automatically to an Internet browser and establishes a communication to the WHAMS Remote Method Invocation (RMI) server. The manager UI consists of system views, function views, and adaptor views. Each function and adaptor has its own view displaying specific attributes and measurement results in graphics.

The main window of the manager client shows the floor plan and all known devices and gives an access to managing devices, radio environment, function views, and placing new devices on the floor plan. The device window shows device properties and available adaptor views. The main windows and the media function views are presented in Fig. 3.

In prototype implementation, the basic functions and adaptors use SNMP queries to several different standard and commercial MIBs. The media connection and adaptor implementations use a custom protocol to connect to a WHAMS AP and to obtain/adjust application QoS measurements and settings. The auto configuration function in WHAMS client listens to a selected TCP port and adjusts the client networks settings according to the configuration received from WHAMS server.

4 Conclusions and Future Work

The WHAMS is being developed for the management of heterogeneous wireless and wired networks in home environments. The design targets have been to facilitate the co-existence of wireless networks, support QoS for application flows, and enable more automatic management. The resulting architecture has been a centralised management server, and add-on functionality for client terminals and APs. The hierarchical functional architecture enables flexibility. A prototype system based on Java implementation is currently under construction, and will be used to evaluate and verify the performance of the developed management system.

References

1. EURESCOM: LUPA: Local provision of 3G and 3G+ services, available at <http://www.eurescom.de/> (May 2003)
2. M. Hännikäinen et al.: Trends in personal wireless data communications, *Computer Communications* 25 (2002) 84 – 99
3. IEEE Std 802.11: Wireless Medium Access Control (MAC) and Physical layer (PHY) specifications (1999)
4. J.D. Case, et al.: Simple Network Management Protocol (SNMP), IETF RFC 1157 (1990)
5. P. P. White: RSVP and Integrated Services in the Internet: A Tutorial, *IEEE Communications*, (1997) 100-106
6. S. Waldbusser et al.: Policy Based Management MIB, IETF SNMPConf working group, Internet-draft (2002)
7. Homepage of DMTF WBEM: http://www.dmtf.org/standards/standard_wbem.php (May 2002)
8. Sun Microsystems Inc.: Federated Management Architecture (FMA) specification version 1.0 revision 0.4, available at <http://jcp.org/en/home/index> (May 2003)

9. S. Moyer, et al.: Home Network Configuration Management & Service Assurance, IEEE 4th International Workshop on Networked Appliances, Gaithersburg, USA (2002) 77–86
10. C. Tsai, et al.: MASSIHN: a Multi-Agent Architecture for Intelligent Home Network Service, IEEE Transactions on Consumer Electronics 48, issue3 (2002) 509–514
11. T Rantanen et. al.: Design of a Quality of Service Management System for Wireless Local Access Networks, The International Conference on Telecommunications (ICT'2001), Bucharest, Romania (2001) Vol.3 107–114

Short Message Service in a Grid-Enabled Computing Environment

Fenglian Xu, Hakki Eres, and Simon Cox

School of Engineering Sciences,
University of Southampton,
Highfield, Southampton, SO17 1BJ, UK
{F.Xu, Hakki.Eres, sjc}@soton.ac.uk
<http://www.geodise.org/>

Abstract. Mobile computing devices such as mobile phones together with a land-based and wireless communication network infrastructure are the existing technical prerequisites for continuous access to networked services. The security of the system is a high concern in this environment, as well as its usability. This paper presents a network infrastructure for using the Short Message Service (SMS) to communicate with mobile phones via a grid-enabled service. The network system includes: a messenger server, a messenger client, Globus Servers and the Short Message Service Centre (SMSC) tied together with XML-RPC, TCP/IP and GRAM protocols for communication. A Matlab tool to use the grid-enabled SMS has been implemented and we demonstrate its use in a grid-enabled application.

1 Introduction

Pervasive computing can deliver access to information with no limits. The Geodise project [1] aims to aid engineers in the design process by making available a suite of design optimisation and search tools and Computational Fluent Dynamics (CFD) analysis packages integrated with distributed grid-enabled computing, data, and knowledge resources. Here we use the mobile phone short message service (SMS) to communicate to a service or an application in a grid-enabled environment results of, for example, progress in an Engineering Design Calculation. The SMS should provide a two-way channel to schedule messages crossing most mobile network operators immediately and it also should be robust, reusable and platform independent.

This paper demonstrates a case of how to use a mobile text message function in a grid-enabled computing environment [2] with a simple usage. The process is done remotely via Globus 2.2 [3] middleware software integrated into Matlab [4] which we use as a scripting/hosting environment. A full lifecycle of the SMS service software from design, implementation and test is presented. A usage of a tool is demonstrated to show how the SMS service is used in the entire grid-enabled environment [5]. Related work is discussed in Sect. 2. The architecture and the implementation of the message system are described in Sect. 3. An

exemplar of using the SMS is depicted in section Sect. 4. Conclusions and future work are summarised in Sect. 5.

2 Related Work

Several recent efforts have been proposed to address the issue of incorporating SMS capabilities for the remote monitoring and control of automation systems. The OPC-SMS gateway [6] is a service which enables mobile users to send a SMS message to request specific information from OPC servers and to be notified by OPC servers. A similar mechanism has been used for sending notifications to a cellular phone in a grid environment [7]. IBM has focused its resource on new technologies that connect to any application, from any device over any network, using any style of interface [8]. With recent advances in mobile computing technology, it is possible to support wireless messaging services which enable mobile users to access their messages at anytime and anywhere [9]. However, they all have a complicated usages to send/access the messages and the mobile phones play roles same as the computers.

Several commercial SMS service providers allow users to send messages to mobiles via either a web application or their own applications which use APIs from the provider. Redcoal SMS [10] uses Simple Mail Transfer Protocol (SMTP) protocol so that users have to access the Internet to read and send messages. However, this service only works with Vodafone networks. Lucin SMS [11] has a web service with an engine to enable users to send text messages, ringtones, logos and retrieve a list of messages. Level 9 [12] provides a Short Message Service Centre (SMSC) which enables a two-way communication at a low cost. They also provide APIs in Perl, C/C++ and Java programming environment. Java programming is object-oriented and platform independent which meets the system requirements mentioned earlier.

3 System Architecture and Implementation

The infrastructure of the SMS system is shown in Fig. 1. The Messenger Service (MS) is the main entity which enables computer applications to send/receive messages to/from mobile phones. The MS includes a stack of services: Globus 2.2 Server, Messenger Client and Messenger Server. They provide the functions of security, sending and receiving the messages over the Internet crossing firewalls. The Globus server enables the message client to be run remotely and safely in a grid-enabled environment using the Globus grid computing toolkit. The protocol between the Globus server and the remote application is via Globus Resource Allocation Management (GRAM) and the protocol between the messenger client and the messenger server is TCP/IP. The messenger server waits for connections from clients and it then forwards the messages from the clients to the SMSC provided by Level 9. The SMSC enables communications between GSM networks and the Internet for sending and delivering text messages/binary images to current/3G mobile phones. The protocol between the messenger server

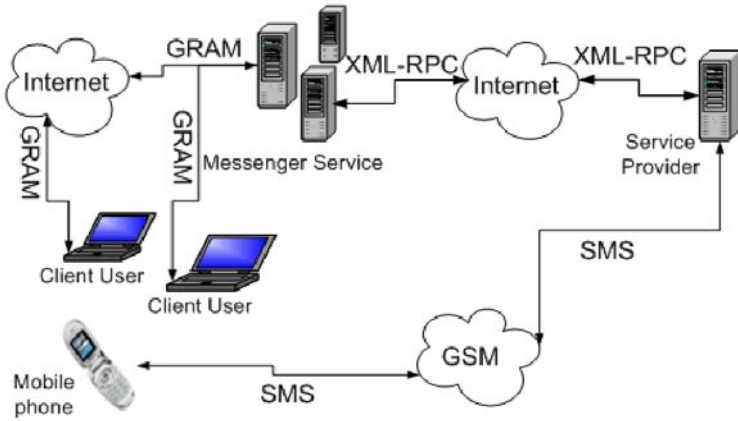


Fig. 1. System Architecture

and the SMSC software is via XML-RPC [13] over the Internet. XML-RPC is a simple, portable way to make remote procedure calls over HTTP. We have integrated the functionality into Matlab which is used as a hosting/scripting environment in Geodise.

Security is an important issue in a network system due to the data transfer over the Internet and over the wireless network. A combination of authentication and authorisation is used for controlling who can use the software and what permissions are granted to the users. Globus Toolkit 2.2 Grid Security Infrastructure (GSI) [14] is used for enabling authentication and communication over an open network. Users can benefit from a single sign-on and delegation to run a job remotely via GRAM with a trusted authority, such as the Globus organisation or the UK-eScience Certificate Authority [15].

The SMS system is based on the client-server design pattern and is implemented with Java socket programming. The messenger client is responsible for requesting a connection to the messenger server, sending a short text message and closing the connection. The messenger server creates a server socket for incoming connections and listens for clients constantly for accepting a new client connection and starting a message received thread. The interface between the Matlab tool and the messenger client is via Resource Specification Language (RSL) [16] which is understood by GRAM. The message server acts as an SMSC client to connect with the SMSC and makes a remote procedure call with arguments which include the following fields: user name, password, mobile number and text message.

4 Application Exemplar

Design optimisation and search is a long and repetitive process. The process begins with a problem definition or a tool to generate a meshed geometry; a

CFD analysis is then performed from which an objective function is calculated. Design Search is the process by which the geometry is systematically modified to improve the value of the chosen objective function. This process often takes a long time (up to weeks) and may be fraught with problems caused by the compute infrastructure, meshing, or the actual CFD runs, or with the optimisation failing to make much progress in improving the design. It is therefore important to monitor the ongoing design run to ensure that it is progressing and hasn't stalled or terminated anomalously. This section demonstrates how to use the `gd_sendtext` tool to send messages to mobile users in the Matlab environment. This tool checks the security before sending a message and returns a job status to see if the message is sent successfully. The Geodise toolkit [17] provides a set of Matlab tools, such as `gd_proxyquery`, `gd_createproxy` and `gd_jobsubmit`, to perform these tasks. The job is submitted by giving an RSL string and a remote host name as arguments. The RSL string combines the remote directory, the executable program name and the arguments to execute the program. An example of using `gd_sendtext` is shown in Fig. 2. As it can be seen from Fig. 2(a), a message is sent by the tool `gd_sendtext` in the Matlab environment and the job handle can be obtained for checking the job status by using `gd_jobstatus`. Fig. 2(b) shows that the message is received from a mobile phone in a second.



(a) Using `gd_sendtext` to send a message (b) Message received from `gd_sendtext`

Fig. 2. Example of using `gd_sendtext`

5 Conclusions and Future Work

The power of this application has enabled users to be notified beyond geographic boundaries in a grid-enabled environment. The messenger service can be plugged into any grid-enabled environment and it is easy to use. We have demonstrated how the short message service has been integrated with a grid-enabled computation of design search and optimisation system to allow monitoring of a job.

The advantage of using the short message service instead of the email protocol SMTP is that it does not require an Internet connection and is fast and easy to use.

We have also developed a two way process in which a message from a phone can be used to steer or terminate calculations. Future work will include use of third generation (3G) technology to send output, e.g. graphics, from a code to a 3G mobile. We will also move to using the Open Grid Services Architecture [18] as reference implementations become available as an open standard for connecting our service to grid-enabled applications.

Acknowledgements. We are grateful to Level 9 networks for providing SMS service capability for this project and EPSRC grant GR/R67705/01.

References

1. The Geodise Project. (2002) <http://www.geodise.org/>
2. Global Grid Forum. (2002) <http://www.gridforum.org/>
3. The Globus Project. (2002) <http://www.globus.org/>
4. Matlab 6.5. (2002) <http://www.mathworks.com/>
5. Pound, G.E., Eres, M.H., Wason, J.L. and Jiao, Z., Keane, A.J., Cox, S.J.: A Grid-Enabled Problem Solving Environment (PSE) for Design Optimisation within Matlab. Proceedings of 17th IPDPS (2003)
6. Kapsalis, V., Koubias, S., Papadopoulos, G.: OPC-SMS: a Wireless Gateway to OPC-based Data Sources. Computer Standards & Interfaces **24** (2002) 437–451
7. Kranzlmüller, D.: Dewiz – Event-based Debugging on the Grid. EUROMICRO-PDP’02, IEEE (2002)
8. IBM Resource. (2002)
<http://www.research.ibm.com/thinkresearch/pervasive.shtml>
9. Tan, D.H.M., Hui, S.C., Lau, C.T.: Wireless Messaging Services for Mobile Users. Journal of Network and Computer Applications **24** (2001) 151–166
10. (Redcoal Website) <http://www.redcoal.com>
11. (Lucin Website) <http://www.soapengine.com/lucin/soapenginex/smsx.asmx>
12. (Level9 Website) <http://www.level9.net:2048/RPC2>
13. (XMLRPC Website)
<http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html>
14. Foster, I., Kesselman, C., Tsudik, G., Tuecke, S.: A Security Architecture for Computational Grids. In: 5th ACM Conference on Computers and Communications Security, San Francisco, California (1998)
15. UK Grid-Support Centre. (2002) <http://www.grid-support.ac.uk/>
16. The Globus Resource Specification Language (RSL) v1.0. (2002)
http://www-fp.globus.org/gram/rsl_spec1.html
17. Eres, M.H., Pound, G.E., Jiao, Z., Wason, J., Xu, F., Keane, A.J., Cox, S.J.: Implementation of a grid-enabled Problem Solving Environment in Matlab. Accepted by the Workshop on Complex Problem-Solving Environments for Grid Computing (Held in conjunction with the ICCS) (2003)
18. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: The Physiology of the Grid - an open grid services architecture for distributed systems integration.
<http://www.globus.org/ogsa/> (2002)

Service Migration Mechanism Using Mobile Sensor Network*

Kyungsoo Lim, Woojin Park, Sinam Woo, Sunshin An

Department of Electronics Engineering, Korea University, 1, 5-Ga, Anam-dong Sungbuk-ku,
Seoul, 136-701 Republic of Korea

{angus, progress, niceguy, sunshin}@dsys.korea.ac.kr

Abstract. The recent advancement of wireless communication technology provides us with new opportunities, but it initiates new challenges and demands as well. Especially, there is an increasing demand for supporting user mobility and service mobility transparently. In this paper, we propose a new mechanism for service migration based on sensor networks of wireless sensor nodes. This mechanism is composed of a component-based server-side model for service mobility and Mobile IP technology for supporting user mobility among sensor networks.

1 Introduction

The technologies of wireless communications and electronics have made a remarkable progress recently. These technological advancements have enabled us to develop a new mechanism based on the sensor networks. A sensor network consists of a large number of sensor nodes which collect information in their coverage regions, and a sink node[1]. In this paper, we propose a new mechanism for service migration using the networks of wireless sensor nodes[2]. This mechanism can support user mobility and service mobility independent of the current location of users, and is composed of two parts: a component-based server-side model for service mobility[3,4] and Mobile IP technology for supporting user mobility among sensor networks.

The remainder of this paper is organized as follows. In section 2, we illustrate our service migration model using mobile sensor network to support user mobility. Section 3 describes the service management server mechanism to provide service migration transparently. Also, in section 4, the overall scenario is proposed. Finally, section 5 summarizes our work.

* An unabridged version of this paper is available at the following URL :
“http://nlab.korea.ac.kr/~angus/euro-par2003_paper.pdf”.

2 Service Migration Model Using Mobile Sensor Network

The term “service” means the set of information that is provided from a computer user’s application programs. Also, the term “mobile sensor node” means the sensor node that provides the mobile node’s function in Mobile IP. The users are identified and authenticated by using the mobile sensor node. Service migration means that the working environment of a computer user moving into other networks is supported and maintained transparently by using a mobile sensor node[2]. In this paper, we propose a new mechanism to support service migration effectively in wireless communication networks. To configure a sensor network, sensor detection, sensor auto-configuration, and sensor handoff are required. Also, we utilize Mobile IP for constituting these sensor networks. Mobile IP is the technology that can maintain its connection even though the mobile host is moving. Mobile IP has route discovery, tunneling, and handoff mechanism for mobility support, and it is used for the global network[5]. By utilizing these characteristics of Mobile IP, sensor nodes can be automatically detected and configured. Therefore, sensor mobility is ensured. In addition, the architecture of a sensor network has the advantages of flexibility and extensibility that both mobile devices and sensors can be applied and integrated in the same way. A sensor node is configured by Mobile IP and ad-hoc routing protocol[2], and a SMS maintains and manages the service profile and authentication information for service migration.

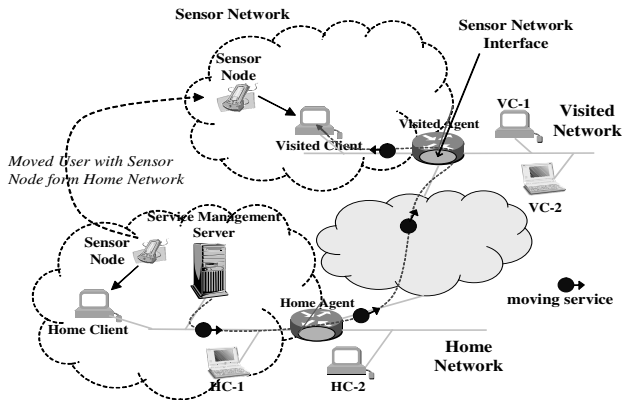


Fig. 1. Service migration model

Figure 1 shows the overall model for service migration. It illustrates the conceptual model of a sensor network for providing service migration effectively. A sensor network is composed of sensor nodes and a sink node. A sink node is the external interface to communicate with agents and SMS(Service Management Server). Also, home agent and foreign agent in Mobile IP scheme are executed in the sink node to communicate with other sensor networks and external networks. Also, sensor nodes can be intermediate nodes to configure the sensor networks and the routing mechanism necessary for mobility support. In this paper, an agent operates as a sink

node, manages its sensor network, and configures sensor nodes automatically. Because the communication range of a sensor node is limited, it can function as a router for neighbor nodes to interact with its agent. Agents are entities that process the transmission of messages between a server and the clients. A visited client has only to send request messages to receive a service object regardless of its server location and its current network. This provides the transparency of request messages.

A VC(Visited Client) can communicate with a SMS in a home network through VA(Visited Agent) only. Because the secure tunneling mechanism of Mobile IP is used, service object has the characteristics of the integrity and the safety. A HA(Home Agent) sends messages to a VA by encapsulating the messages and the VA decapsulates them. Then, the VA transfers the messages to the visited client based on its sensor ID and its IP address.

3 Service Management Server

In this paper, we propose a component-based model that processes and transports service objects to support sensor-based service mobility. A SMS consists of a server-side component, which processes the request of users and stores its result. Also, SMS administers and manages the user passport, the client information, and the application information. The server-side component is composed of a session bean and three entity beans[6] as shown in figure 2.

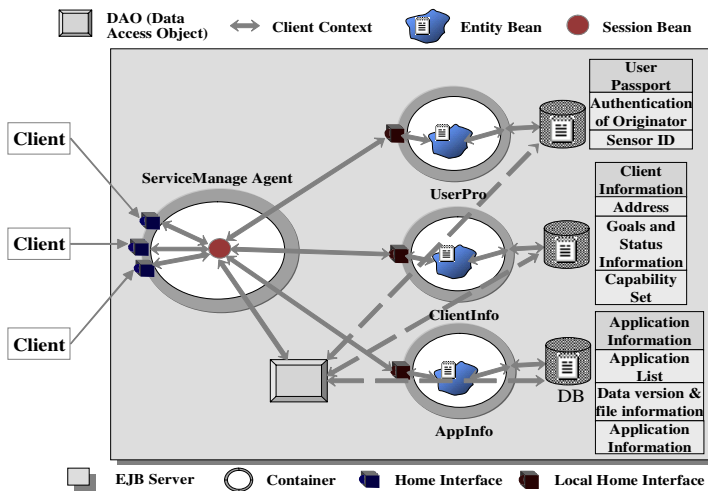


Fig. 2. System architecture of a server

A service management agent(SMA) bean takes charge of overall service logic of a server. This bean operates as follows. A SMA is a session bean, which loads the overall service logic and processes messages from a HA. When a SMA receives the home client's request message from a HA, it processes and classifies the request, and

then sends its result to a HA. Entity beans are object expressions of a persistence data process[7], and they each have their own database and manage it. These entities are made up of UserPro, ClientInfo, and AppInfo. Messages are classified according to characteristics of each entity bean, and it is stored in a pertinent table of each database[7]. UserPro, ClientInfo, and AppInfo bean are mapped to User Passport, Client Information, and Application Information in order. When a SMA receives the request related to service object from a HA, it collects the stored information of the user in the database through DAO(Data Access Object) and send it to a HA. Agents send the requests of the clients to a SMS and the responses of the SMS to the clients. Therefore, agents are entities that process the transmission of messages between a server and the clients.

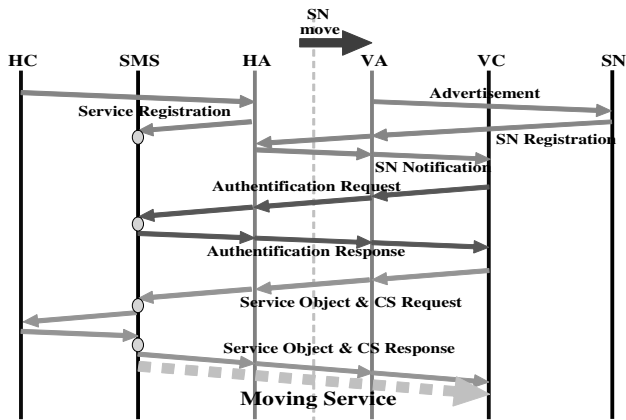


Fig. 3. Service scenario

4 Service Migration Mechanism

Figure 3 shows the overall Moving service scenario flow. A sensor node has only to communicate with an agent of its current network. Also, a visited client has only to communicate with its SMS through agents. Authorization and information for service migration are provided through a SMS.

When a sensor node moves into a VN, it is configured by its VA. The VA broadcasts the sensor node's registration information and then this information is displayed in its local hosts. When the user confirms the information, its client application requests its authentication and waits for its response to be received. After that, the visited client sends the user's service object request and capability set to its VA. The VA sends these messages to the sensor node's HA and the HA forwards the information to a SMS. The SMS reply by sending the service object and some additional information to the HA. Finally, the HA transmits the above messages to the visited client. This scenario ensures our sensor-based service migration effectively.

The method to maintain the user's working environment should be considered. There are two ways to maintain the user's working environment in our network model that supports sensor-based service migration. First, the user stores its working environment information and registers to its SMS through home client application. The second, if the user moves into another VN without storing its information, a HA can conceive the sensor node's mobility through sensor node detection process. In the home network, a HA asks home client application to store and register the user's working environment in its SMS.

As a sensor node moves into the other VN, the above two methods are repeated. Although a sensor node roams around the world, service migration can be provided through its VA and HA in our proposed model. Our component-based SMS has advantages in interoperability and compatibility. Also, it can back-up service profile. These service profiles in a SMS are maintained and managed by individual version information.

5 Conclusion and Future Work

In this paper, we have proposed a new mechanism to support service migration transparently in wireless sensor networks. This mechanism consists of a component-based server-side model to accomplish service mobility and Mobile IP technology between sensor networks to ensure user mobility. With these two new features, service migration can be guaranteed. Our model is interoperable and compatible with any server in other networks, and this system can be implemented on any ad hoc network.

We have showed that the proposed mechanism can be applied to a wireless sensor network, and that it can be a key technology of future wireless sensor networks for supporting user mobility and service mobility.

Future work of this research includes implementation and experimentation of our proposed model. In particular, we will focus on the following:

- Proposing a method for a sensor node to find and select the optimal client that satisfies CS(capability-set)[8] in mobile sensor networks.
- Resolution of the problem of interference between adjacent sensor nodes.
- Reduction of the useless packets transmitted when a handoff is occurred.
- Providing an enhanced model to support service mobility and user mobility in various network environments.

References

1. Ian F. Akyildiz, Weilian su, Yogesh Sankarasubramaniam, and Erdal Cayirci "A Survey on Sensor Networks", IEEE Communications Magazine, August, 2002.
2. Jin Zhu, Symeon Papavassiliou, and Sheng Xu, "Modeling and analyzing the dynamics of mobile wireless sensor networking infrastructures" Vehicular Technology Conference (VTC 2002-Fall), IEEE, 2002 56th.

3. C. BRENT HIRSCHMAN, "Service Mobility/Transparency For Personal Communications", Universal Personal Communication (ICUPC '92) 1992.
4. Rong N. Chang, "Realizing Service Mobility for Personal Communications Applications", Universal Personal Communications 1993 'Personal Communication : Gateway to the 21st Century(ICUPC'93), 1993.
5. James D. Solomon "Mobile IP", PRENTICE HALL 1998 .
6. Richard Monson-Haefel, "Enterprise Java Beans", O'REILLY, 1999.
7. Ted Neward, "Server-Based Java Programming", MANNING, 2000.
8. Pearce, P.R. "CS-2 enhancements for user interaction", Telecommunications, 1998. 6th IEE Conference on (Conf. Publ. No. 451) , Mar-1 1998.
9. Sasha Slijepcevic, Vlasios Tsiatsis, Scott Zimbeck, "On Communication Security in Wireless Ad-Hoc Sensor Networks", International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises(WETICE'02), 2002.
10. Charles. E. Perkins, "Mobile IP", International Journal of communication system, May, 1998.

Topic 16

Distributed Systems and Distributed Multimedia

Fernando Pereira, Abdulmotaleb El Saddik, Roy Friedman, and
László Böszörményi
Topic Chairs

Topic 16 (*Distributed Systems and Distributed Multimedia*) has received 23 papers. 9 of them were redirected to Topic 12 (*Architectures and Algorithms for Multimedia Applications*). From the remaining 14 papers, the program committee has accepted 7 regular and 1 short papers.

The majority of the papers in this chapter cover issues of distributed systems in general. Two papers deal with questions of distributed multimedia systems. The chapter contains papers on

- Cluster computing;
- Mobile agent systems;
- Distributed shared memory systems;
- Dynamic node placement in adaptive video servers;
- Multimedia bitstream description in the context of the emerging MPEG-21 standard.

Nswap: A Network Swapping Module for Linux Clusters

Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel

Swarthmore College, Swarthmore, PA 10981, USA

Abstract. Cluster applications that process large amounts of data, such as parallel scientific or multimedia applications, are likely to cause swapping on individual cluster nodes. These applications will perform better on clusters with network swapping support. Network swapping allows any cluster node with over-committed memory to use idle memory of a remote node as its backing store and to “swap” its pages over the network. As the disparity between network speeds and disk speeds continues to grow, network swapping will be faster than traditional swapping to local disk. We present Nswap, a network swapping system for heterogeneous Linux clusters and networks of Linux machines. Nswap is implemented as a loadable kernel module for version 2.4 of the Linux kernel. It is a space-efficient and time-efficient implementation that transparently performs network swapping. Nswap scales to larger clusters, supports migration of remotely swapped pages, and supports dynamic growing and shrinking of Nswap cache (the amount of RAM available to store remote pages) in response to a node’s local memory needs. Results comparing Nswap running on an eight node Linux cluster with 100BaseT Ethernet interconnect and faster disk show that Nswap is comparable to swapping to local, faster disk; depending on the workload, Nswap’s performance is up to 1.7 times faster than disk to between 1.3 and 4.6 times slower than disk for most workloads. We show that with faster networking technology, Nswap will outperform swapping to disk.

1 Introduction

Using remote idle memory as backing store for networked and cluster systems is motivated by the observation that network speeds are getting faster more quickly than are disk speeds [9]. In addition, because disk speeds are limited by mechanical disk arm movements and rotational latencies, this disparity will likely grow. As a result, swapping to local disk will be slower than using remote idle memory as a “swap device” and transferring pages over the faster network. Further motivation for network swapping is supported by several studies [2,4,10] showing that large amounts of idle cluster memory are almost always available for remote swapping.

We present Nswap, a network swapping system for heterogeneous Linux clusters and networks of Linux machines. Nswap transparently provides network swapping to cluster applications. Nswap is implemented as a loadable kernel

module that is easily added as a swap device to cluster nodes and runs entirely in kernel space on an unmodified ¹ Linux kernel; applications can take advantage of network swapping without having to re-compile or link with special libraries. Nswap is designed to scale to large clusters using an approach similar to Mosix's design for scalability [1]. In Nswap there is no centralized server that chooses a remote node to which to swap. In addition, Nswap does not rely on global state, nor does it rely on complete or completely accurate information about the state of all cluster nodes to make swapping decisions. Thus, Nswap will scale to larger clusters because each node independently can make swapping decisions based on partial and not necessarily accurate information about the state of the cluster.

Nswap supports dynamic growing and shrinking of each node's Nswap cache in response to the node's local memory use, allowing a node to reclaim some of its Nswap cache space for local paging or file I/O when it needs it, and allowing Nswap to reclaim some local memory for Nswap cache when the memory is no longer needed for local processing. Growing and shrinking of the Nswap cache is done in large chunks of memory pages to better match the bursty behavior of memory usage on a node [2]. Another feature of Nswap is that it avoids issuing writes to disk when swapping; it does not write-through to disk on swap-outs, and it supports migration of remotely swapped pages between the cluster nodes that cache them in response to changes in a remote node's local memory use. Only when there is no available idle memory in the cluster for remote swapping does Nswap revert to swapping to disk. Because Nswap does not do write-through to disk, swapping activity on a node does not interfere with simultaneous file system disk I/O on the node. In addition, as long as there is idle remote memory in the cluster, Nswap can be used to provide disk-less cluster nodes the ability to swap.

In section 2 we discuss related work in network swapping. In section 3 we present the details of Nswap's implementation, including a presentation of the swapping protocols. In section 4 we present the results of measurements of Nswap running on an eight node cluster. Our results show that with 100 BaseT technology we are comparable to high-end workstation disks, but that with faster interconnect, Nswap will outperform swapping to disk. In section 5 we discuss future directions for our work.

2 Related Work

There have been several previous projects that examine using remote idle memory as backing store for nodes in networks of workstations [3,6,10,8,5,11,7]. For example, Feeley et. al.[6] implement a network cache for swapped pages by modifying the memory management system of the DEC OSF/1 kernel. Their system views remote memory as a cache of network swapped pages that also are written through to disk; remote servers only cache clean pages and can arbitrarily drop

¹ Currently, we require a re-compile of the 2.4.18 kernel to export two kernel symbols to our module, but we have not modified Linux kernel code in any way.

a page when their memory resources become scarce. Each node's memory is partitioned into local space and network cache space. When a page fault occurs, a node's local space may grow by one page at the expense of a page of its network cache space. Our growing and shrinking policies do something similar, but we grow and shrink in larger units to better match bursty memory use patterns. In addition, our system does not do a write-through to disk on every swap-out. Thus, our system will not be slowed down by disk writes and will not interfere with a node's local file I/O.

Markatos and Dramitinos [10] describe reliability schemes for a remote memory pager for the DEC OSF/1 operating system. Their system is implemented as a client block device driver, and a user-level server for storing pages from remote nodes. Both dirty and clean pages can be stored at remote servers. When a server is full, or when it needs more memory for local processes, remote pages are written to the server's disk, resulting in a significant slow down to a subsequent swap-in of the page. Our system avoids writing pages to disk in similar circumstances by migrating the page to another server. In addition, our system allows a server's cache size to change, and it allows nodes to dynamically change roles between clients and servers based on their local memory use.

Bernard and Hamma's work [5] focuses on policies for balancing a node's resource usage between remote paging servers and local processes in a network of workstations. Their policy uses local and remote paging activity and local memory use to determine when to enable or disable remote paging servers on a node. Our growing and shrinking policies for Nswap cache sizes have to solve a similar problem. However, on a cluster system, where there is no notion of a user "owning" an individual cluster node, so we expect that we can keep Nswap cache space on nodes with under-committed memory but higher cpu loads.

3 Nswap Implementation

Nswap consists of two components running entirely in kernel space (shown in Figure 1). The first is a multi-threaded client that receives swap-in and swap-out requests from the kernel. The second is a multi-threaded server that manages a node's Nswap cache and handles swap-in and swap-out requests from remote clients. Each node in the Nswap cluster runs an Nswap client and an Nswap server. At any given time, a cluster node is acting either as a client or a server, but typically not as both simultaneously; a node adjusts its role based on its local memory use. The goals of Nswap's design are to provide efficient, transparent network swapping to cluster applications, to dynamically adjust to individual node's memory needs, and to scale to large, heterogeneous clusters.

3.1 Nswap Client and Server

The Nswap client is implemented as a block pseudo-device. This abstraction allows an easy and portable interface to the Linux kernel's swapping mechanism, which communicates with Nswap exactly as it would any physical block device. Idle memory across the cluster is accessed through read and write requests from

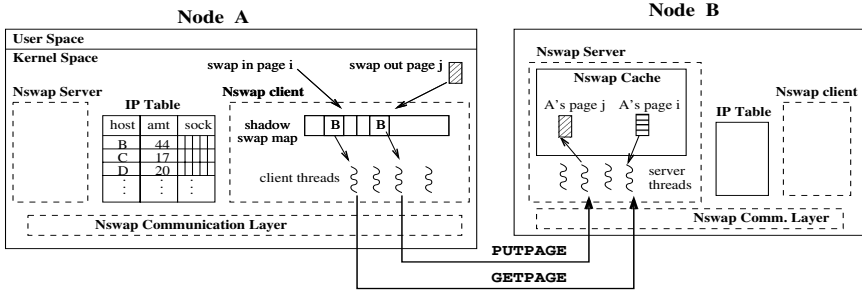


Fig. 1. Nswap System Architecture. Node A shows the details of the client including the shadow slot map used to store information about which remote servers store A's pages. Node B shows the details of the server including the Nswap cache of remotely swapped pages. In response to the kernel swapping in (out) a page to our Nswap device, a client thread issues a PUTPAGE (GETPAGE) to write (read) the page from a remote server.

the kernel to the Nswap client device. The client uses multiple kernel threads to simultaneously service several swap-in or swap-out requests.

The Nswap client needs to add additional state to the kernel's swap map so it can keep track of the location of its remotely swapped pages. We add a shadow swap map (Figure 1) that stores the following information (in 16 bits) for each slot in the kernel's swap map of our device: the `serverID`, designating which remote server stores the page; the `hop_count`, counting the number of instances a page has been migrated (used to avoid race-conditions that can occur during page migration and to limit the number of times a page can be migrated); the `time_stamp`, identifying an instance of the slot being used by the kernel (used to identify "dead" pages in the system that can be dropped by the server caching them); and the `in_use` bit to control conflicting simultaneous operations to the same swap slot. Additionally, these fields allow for a communication protocol that does not require the client's state to be synchronously updated during page migrations.

The Nswap server is implemented as a kernel-level daemon that makes local memory available for Nswap clients. When the Nswap module is loaded, the server starts three kernel threads: a listener thread, a memory thread, and a status thread. The listener thread listens for connections from Nswap clients, and starts new threads to handle the communication. The memory thread monitors the local load, communicating its load to other hosts, and if necessary, triggering growing and shrinking its Nswap cache. The status thread accepts UDP broadcast messages from memory threads on other servers. These messages contain changes in a server's available Nswap cache size. The status thread updates its IPTable with this information.

The IPTable on each node contains a potentially incomplete list of other server's state. Each entry contains an Nswap server's IP, the amount of Nswap cache it has available, and a cache of open sockets to the server so that new

connections do not have to be created on every page transfer. The Nswap client uses information in the IPTable to select a remote server to send its pages, and to obtain connections to remote servers. The information in the IPTable about each server's Nswap cache availability does not have to be accurate, nor does there need to be an IPTable entry for every node in the cluster, for a client to choose a remote server on a swap-out. This design will help Nswap scale better to larger clusters, but at the expense of clients perhaps not making the best possible server choice. Typically, the information is accurate enough to make a good server choice. Nswap recovers from a bad server choice by migrating pages to a better server.

When the client makes a request for a page, it sends page meta-data (`clientID`, `slot_number`, `hop_count`, `timestamp`) that the server uses to locate the page in its Nswap cache. Page meta-data is also used to determine when cached pages are no longer needed. Since the Linux kernel does not inform a swap device when it no longer needs a page, "dead" pages can accumulate. A garbage collector thread in the Nswap client periodically runs when the client has not swapped recently. It finds and cleans dead slots in the swap slot map, sending the server a message indicating that it can drop the "dead" page from its cache. In addition, "dead" pages are identified and dropped during page migration, and when a client re-uses an old slot during a swap-out operation.

A predictive monitoring policy is used to dynamically grow or shrink the size of Nswap cache. A status thread on each node periodically polls the behavior of the Linux memory management subsystem. If the machine is showing symptoms of high memory usage, Nswap cache will shrink, possibly triggering migration of some of the remote pages it caches to other Nswap servers. When local memory is underutilized, Nswap cache size is increased.

3.2 Nswap Communication Protocol

The communication protocol between different nodes in an Nswap cluster is defined by five types of requests: `PUTPAGE`, `GETPAGE`, `PUNTPAGE`, `UPDATE` and `INVALIDATE`. When Nswap is used as a swap device, the kernel writes pages to it. The client receives a write request and initiates a `PUTPAGE` to send the page to a remote server. At some later time the kernel may require the data on the page and issue a read request to the Nswap device. The client uses the `GETPAGE` request to retrieve the page (see Figure 1). If the workload distribution changes, it may become necessary for a server to reduce its Nswap cache size, using the `PUNTPAGE` request to offload the page to another Nswap server. Moving a page from one server to another involves an `UPDATE` request to alert the client to the new location of the page and an `INVALIDATE` request from the client to the old server to inform the old server that it can drop its copy of the page (see Figure 2).

`GETPAGE` and `PUTPAGE` are designed to be as fast as possible for the client who is currently swapping. If a client makes a bad server choice for a `PUTPAGE`, the Nswap servers handle it through page migration rather than forcing the client to make a better server choice, which would slow down the client's swap-outs. In addition, the protocol is designed to limit synchronous activities. As a result,

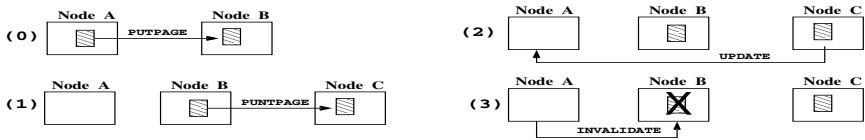


Fig. 2. Page Migration in Nswap. Node A acts as an Nswap client, and Nodes B and C act as servers. A **PUTPAGE** from server B to server C (1), triggers an **UPDATE** from server C to client A (2), which in turn triggers an **INVALIDATE** from client A to server C (3). At this point, B can drop its copy of A's page.

extra state (page meta-data) is passed with requests so that a receiver can detect out-of-order and old requests and handle them appropriately. The following is an overview of the communication protocol:

PUTPAGE is used to ask a remote host to store a local page. The client picks a remote server using its IPTable information, and sends the server a **PUTPAGE** command and the page's meta-data. The server almost always responds with an **OK_PUTPAGE**, and then the client sends the page data. Even a server with a full Nswap cache typically will accept the page from the client. In these cases, the server starts to remove some of its least recently cached pages by issuing **PUNTPAGES** after completing the **PUTPAGE** transaction.

GETPAGE is a request to retrieve a page that is stored remotely. The client sends the **GETPAGE** command and the page meta-data. The server uses the meta-data to find the page in its Nswap cache, and sends it to the client.

INVALIDATE is used by a client to inform a server that it may drop a page from its Nswap cache. An **INVALIDATE** can result from page migration (see **PUNTPAGE** below) or from garbage collection on the client. The client sends an **INVALIDATE** command and the page meta-data. The server compares all the meta-data of the page and frees the page if these match. Otherwise, the **INVALIDATE** request is for an old page and is ignored.

PUNTPAGE is used to implement page migration. When a server becomes over-committed, it attempts to get rid of the least recently cached foreign pages by punting them to other servers. The over-committed server sends a **PUNTPAGE** command and the page data to another remote server. The original server cannot drop the page until it receives an **INVALIDATE** from the client as described above. Once the page has been transferred to the new server, the new server initiates an **UPDATE** to the owner of the page. If there is no available Nswap cache space in the cluster, the server punts the page back to its owner who writes it to its local disk.

UPDATE is used to inform a client that one of its pages has moved. The new server sends an **UPDATE** command to the client with the page meta-data. The page meta-data is used to detect whether the **UPDATE** is for an old page, in which case the client sends an **INVALIDATE** to the sender of the **UPDATE**; otherwise, the client sends an **INVALIDATE** to the previous server caching the page so that it can drop its copy of the page.

4 Results

We present results comparing swapping to disk and Nswap for several workloads. Our experiments were run on a cluster of 8 nodes² running version 2.4.18 of the Linux kernel connected by 100BaseT Ethernet. Four of the nodes have Intel Pentium II processors, 128 MB of RAM, and Maxtor DiamondMax 4320 disks with a data rate of up to 176 Mb/sec. The other machines have Pentium III processors, 512 MB of RAM, and IBM Deskstar disk with a sustained data rate of 167-326 Mb/sec and a max rate of 494 Mb/sec. In both machine types, disk transfer rates are faster than network transfer rates, so we expect to be slower than swapping to disk. However, our results show that for several workloads swapping across a slower network is faster than swapping to faster local disk. In addition, we calculate that on 1 and 10 Gbit Ethernet, Nswap will outperform swapping to disk for almost all workloads.

4.1 Workload Results

Table 1 shows run times of several workloads comparing Nswap with swapping to disk. On the PIII's we ran two versions of Nswap, one using TCP/IP to transfer pages (column 5), the other using UDP/IP (column 6). We ran one and four process versions of each workload with one node acting as the client and three nodes acting as servers. We also disabled Nswap cache growing and shrinking to reduce the amount of variation between timed runs.

Workload 1 consists of a process that performs a large sequential write to memory followed by a large sequential read. It is designed to be the best case for swapping to disk because there will be a minimal amount of disk head movement when swapping due to the way in which Linux allocates space on the swap partition. Workload 2 consists of a process that performs random writes followed by random reads to a large chunk of memory. This workload stresses disk head movement within the swap partition only. Workload 3 consists of two processes. The first runs a Workload 1 application and the second is a process that performs a large sequential write to a file. Workload 3 further stresses disk head movement when swapping to disk because file I/O and swap I/O are concurrently taking place in different disk partitions. Workload 4 consists of two processes; the first runs the Workload 2 application and the second runs the large sequential file write application. The four process versions of each workload are designed to represent a more realistic cluster workload. The size of each Workload varied from platform to platform based on the size of RAM on each machine, and they varied between the Random and Sequential tests. As a result, for each Workload (row), only the values in columns 2 and 3 can be compared, and the values in columns 4, 5, and 6 can be compared.

We expect that disk will perform much better than Nswap on the single process versions of Workload 1 and Workload 2 because the disk arm only moves

² Although our cluster currently consists of only Pentium architectures, Nswap will run on any architecture supported by Debian Linux.

Table 1. Swapping to fast disk vs. TCP Nswap and UDP Nswap on 100BaseT for PII and PIII nodes. The rows are 1 and 4 process runs of each of the four Workloads. Time is measured in seconds and the values are the mean of 5 runs of each benchmark. Bold values indicate runs for which Nswap is faster than disk.

(Workload) # procs	DISK (PII)	NSWAP (PII, TCP)	DISK (PIII)	NSWAP (PIII, TCP)	NSWAP (PIII, UDP)
(1), 1	98.0	450.9 (4.6x slower)	13.1	154.3 (11.7x slow)	61.3 (4.6x slow)
(1), 4	652.9	630.6 (1.04x fast)	551.4	1429.7 (2.6x slow)	614.4 (1.1x slow)
(2), 1	874.6	1937.0 (2.2x slow)	266.8	1071.8 (4.0x slow)	153.5 (1.7x fast)
(2), 4	996.7	617.0 (1.6x fast)	68.6	189.3 (2.8x slow)	50.3 (1.4x fast)
(3), 1	632.9	737.7 (1.7x slow)	770.2	1111.1 (1.4x slow)	811.0 (1.1x slow)
(3), 4	1312.1	1127.2 (1.2x fast)	727.1	1430.5 (1.9x slow)	619.5 (1.2x fast)
(4), 1	1971.2	2111.0 (1.07x slow)	923.9	1529.3 (1.7x slow)	821.7 (1.1x fast)
(4), 4	1453.0	1094.6 (1.3x fast)	502.5	498.7 (1.01x fast)	429.2 (1.2x fast)

within the swap partition when these workloads run, and our results confirm this; the worst slow downs (4.6 and 11.7) are for the single process runs of Workload 1. The differences between the performance of Workloads 1 and 2 show how disk head movement within the same disk partition affects the performance of swapping to disk. For the four process runs of Workloads 1 and 2, Nswap is much closer in performance to disk, and is faster in some cases; the TCP version on the PIIIs is 2.6 and 2.8 times slower than disk, the UDP version on the PIIIs is 1.4 times faster for Workload 2, and on the PIIIs both workloads are faster than disk (1.04 and 1.6 times faster). This is due to a slow down in disk swapping caused by an increase in disk arm movement because multiple processes are simultaneously swapping to disk. The results from Workloads 3 and 4, where more than one process is running and where there is concurrent file I/O with swapping, show further potential advantages of Nswap; Nswap is faster than disk for the four process version of all Workloads on the PIIIs, for three of the four Workloads under UDP Nswap, and for Workload 4 under TCP Nswap on the PIIIs. The differences in TCP and UDP Nswap results indicate that TCP latency is preventing us from getting better network swapping performance. Nswap performs better for Workloads 3 and 4 because network swapping doesn't interfere with file system I/O, and because there is no increase in per-swap overhead when multiple process are simultaneously swapping.

4.2 Results for Nswap on Faster Networks

Currently we do not have access to faster network technology than 100BaseT Ethernet, so we are unable to run experiments of Nswap on a cluster with a network that is actually faster than our cluster's disks. However, based on measurements of our workloads running Nswap with 10BaseT and Nswap with 100BaseT Ethernet, we estimate run times on faster networks.

The rows of Table 2 show execution times of workloads run on 100BaseT and 10BaseT, and our estimates of their run times on 1 and 10 Gigabit Ethernet.

Table 2. Time (and Speedups) for TCP & UDP Nswap on faster networks. All Workloads were run on the PIIIs. Bold values indicate cases when Nswap is faster than disk. The rows show for each workload calculated 1 and 10 Gbit run times (col. 5 & 6) based on measured speedup values between runs on 10 and 100 BaseT (col. 3 & 4).

(Workload)	Disk	10BaseT	100BaseT	1 Gbit	10 Gbit
(1) TCP	580.10	5719.00	1518.34 (speedup 3.8)	1075.00 (5.3)	1034.17 (5.5)
(1) UDP	12.27	306.69	56.80 (speedup 5.4)	28.90 (10.6)	26.30 (11.6)
(2) UDP	226.79	847.74	153.54 (speedup 5.5)	77.30 (10.9)	70.30 (12.1)
(4) UDP	6265.39	9605.91	1733.93 (speedup 5.54)	866.18 (11.1)	786.72 (12.2)

We use the speedup results from our 10 and 100 BaseT measurements of each benchmark and apply Amdahl’s Law to get estimates of speedups for 1 and 10 Gigabit Ethernet using the following equation:

$$TotalSpeedup = \frac{1}{1 - FractionBandwidth + FractionBandwidth / SpeedupBandwidth}.$$

For each Workload, we compute *FractionBandwidth* based on *TotalSpeedup* values from measured 10 and 100 BaseT runs. Using this value, we compute *TotalSpeedup* values for 1 and 10 Gbit Ethernet (columns 5 and 6 in Table 2).

The measured speedups between 10 and 100 BaseT of TCP version of Nswap are lower than we expected (one example is shown in row 1 of Table 2); timed runs of these workloads when they completely fit into memory compared to when they don’t, show that over 99% of their total execution time is due to swapping overhead. We also measured our **GETPAGE** and **PUTPAGE** implementations and found that over 99% of their time is due to transferring page data and metadata. However, the UDP version of Nswap results in speedup values closer to what we expected. In fact, UDP Nswap on Gigabit Ethernet outperforms swapping to disk for all Workloads except the single process version of Workload 1 on the PIIIs. Our speedup results further indicate that TCP latency is preventing Nswap from taking full advantage of improvements in network bandwidth.

5 Conclusions and Future Work

Nswap is a network swapping system for heterogeneous Linux clusters. Because Nswap is implemented as a loadable kernel module that runs entirely in kernel space, it efficiently and transparently provides network swapping to cluster applications. Results from experiments on our initial implementation of Nswap on an eight node cluster with 100BaseT Ethernet show that Nswap is comparable to swapping to faster disk. Furthermore, since it is likely that network technology will continue to get faster more quickly than disk technology, Nswap will be an even better alternative to disk swapping in the future.

One area of future work involves examining reliability schemes for Nswap. We are also investigating reliable UDP implementations for the Nswap communication layer so that we can avoid the latency of TCP to better take advantage

of faster networks, but to still get reliable page transfers over the network. Other areas for future work involve testing Nswap on faster networks and larger clusters, and further investigating predictive schemes for determining when to grow or shrink Nswap caches sizes. In addition, we may want to examine implementing an adaptive scheme into Nswap. Based on the results from the single process version of Workload 1 in Table 1, there may be some workloads for which swapping to disk will be faster. Nswap could be designed to identify these cases, and switch to disk swapping while the workload favors it.

Acknowledgments. We thank Matti Klock, Gabriel Rosenkoetter, and Rafael Hinojosa for their participation in Nswap's early development.

References

1. Barak A., La'adan O., and Shiloh A. Scalable cluster computing with MOSIX for Linux. In *Proceedings of Linux Expo '99*, pages 95–100, Raleigh, N.C., May 1999.
2. Anurag Acharya and Sanjeev Setia. Availability and Utility of Idle Memory on Workstation Clusters. In *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 35–46, May 1999.
3. T. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A case for NOW (Networks of Workstations). *IEEE Micro*, February 1999.
4. Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 267–278, 1995.
5. G. Bernard and S. Hamma. Remote Memory Paging in Networks of Workstations. In *SUUG'94 Conference*, April 1994.
6. Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *15th ACM Symposium on Operating Systems Principles*, December 1995.
7. Michail D. Flouris and Evangelos P. Markatos. *Network RAM, in High Performance Cluster Computing: Architectures and Systems, Chapt. 16*. Prentice Hall, 1999.
8. Liviu Iftode, Karin Petersen, and Kai Li. Memory Servers for Multicomputers. In *IEEE COMPCON'93 Conference*, February 1993.
9. John L. Hennessy and David A. Patterson. *Computer Architectures A Quantitative Approach, 3rd Edition*. Morgan Kaufman, 2002.
10. Evangelos P. Markatos and George Dramitinos. Implementation of a Reliable Remote Memory Pager. In *USENIX 1996 Annual Technical Conference*, 1996.
11. Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht. Incorporating Job Migration and Network RAM to Share Cluster Memory Resources. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.

Low Overhead Agent Replication for the Reliable Mobile Agent System

Taeseon Park and Ilsoo Byun

Department of Computer Engineering
Sejong University
Seoul 143-737 KOREA
{tspark,widewis}@sejong.ac.kr

Abstract. Fault-tolerance is an important design issue to build a mobile agent system satisfying the exactly-once execution property. Two fault tolerance schemes have been proposed. The replication scheme provides a higher degree of fault-tolerance, while the checkpointing scheme requires the lower overhead. In this paper, we analyze the performance of three optimization approaches to the agent replication and compare their overhead with the one of the checkpointing scheme. For the performance evaluation, optimized replication schemes and the checkpointing scheme have been implemented on top of the Aglet system. The experimental results show that the optimization can achieve up to 37% reduction on the agent migration time.

1 Introduction

A mobile agent moves over the system sites connected by a network, in order to perform an assigned task. During the execution or the migration, the agent may get lost if the execution site or the migration site fails. Also, it may get duplicated, if the related sites are not careful in recovery from the failure. To guarantee the correct execution of a mobile agent, it should not be lost or duplicated. For the exactly-once execution [7] of a mobile agent, many fault-tolerance schemes have been proposed.

One approach is to use checkpointing [1,6,8]. On arrival of a new site, the agent saves its current states into a stable storage. In case of a site failure, the agent is re-activated from the checkpointed states, as if it just arrives to the site. The checkpointing scheme is usually implemented with two-phased agent migration scheme [3] to properly handle the failure during migration. The checkpointing scheme provides the fault-tolerance with a low cost. However, it may suffer from the long blocking time, since in this scheme, an agent can resume the execution only after the site recovers from the failure.

To cope with the blocking problem, the replication scheme [2,5,7] is used. A mobile agent in this scheme is replicated and sent to $2k + 1$ sites to tolerate k failures. Among $2k + 1$ agents, one called a *primary* is responsible for the initial execution. When the primary fails, one of the other replicas takes over the execution. To detect failures of other replicas and prevent duplicate execution,

the replicas perform the consensus at the end of each site execution. When the majority of the replicas agree on an execution site, one stage of the execution is successfully completed.

Considering the degree of fault-tolerance, the replication scheme is more desirable. However, the cost of agent replication and consensus is considerable compared to the checkpointing cost. The performance of two fault-tolerance schemes have been compared in [4,5,6]. Especially, in [4], we have suggested two optimization approaches to the agent replication and compared their performance with that of checkpointing. One approach is the asynchronous agent replication, in which the replicas are migrated to the designated sites in an asynchronous manner so that the primary can begin execution without waiting for the migration of other replicas. The other optimization is the usage of a consensus agent. In this approach, to tolerate k failures, $k + 1$ agent replicas for the possible execution and k consensus agent for the agreement are used.

In this paper, we further analyze the effect of each optimization approach to the agent replication. We also suggest another optimization approach, the pipelined scheme, in which the replica migration sites are selected among the ones previously visited by the agent. Most of the agent replication time is taken by the agent migration time, which includes the time to move agent's current states and the time to move agent's class codes. By reusing the previously visited sites, migration time for class codes can be eliminated. To measure the effect of various optimization schemes, we implement the schemes on top of the mobile agent system, Aglets [3]. The performance results show that the various optimization schemes together can achieve up to 37% reduction on the replication overhead.

2 The Aglets System

The *Aglets* [3] is a Java-based mobile agent system. To support execution, migration and communication of agents, the system provides the *AgletContext* environment. The agents in the Aglets system is created inheriting properties and methods from the *AgletClass* and perform the event-driven activities. For the inter-agent communication, the message-passing mechanism is provided and the *AgletProxy* is used to support the location transparency of the mobile agent. The *AgletProxy* is an interface to the *Aglet* object and every message is sent to the *Aglet* object through the *AgletProxy*, regardless of its location.

Two types of agent migration can be considered. One is the *strong migration*, in which agent codes, intermediate data and agent execution states are migrated. The other is the *weak migration*, in which only the agent codes and the intermediate data are migrated. As most of the Java-based mobile agent systems, the Aglets system supports the weak migration. For the migration, an agent is first serialized and then transferred. On the destination site, the agent is de-serialized and then activated. For the performance reason, the entire class codes for the agent are not migrated with the agent and the class codes are transferred only when the destination site does not have them.

For the reliable agent migration, *ATP* (*Agent Transfer Protocol*) is provided. The ATP is for the two-phased agent migration. The current execution site in the first phase sends the *ATP request* to the destination site, which sends back the *ATP reply* to notify the acceptance of the agent migration. The agent is then migrated within the *ATP transfer* message in the second phase. The destination site notifies the successful agent migration by sending the *ATP Ack*. The fail-stop failure model is assumed; that is, once a system component fails, it stops its execution and does not perform any malicious actions.

3 The Experimental System

3.1 Agent Replication and Consensus

An agent in the mobile agent system migrates from a site to another site, in order to perform a task assigned by a user. The execution of a subtask at one system site is called a *stage*. In other words, the mobile agent execution consists of a sequence of stages and the agent migrates between two stages. In the replication scheme, an agent is replicated and the replicas are migrated to the $2k + 1$ sites to execute one stage. With the $2k + 1$ replicas, the system can tolerate up to k failures.

Among the $2k + 1$ replicas, one called a *primary* is responsible for the initial execution of a stage. The others are called *observers*, which may take over the stage execution in case that the primary fails. Fig. 1 describes the agent replication process when three replicas are used ($k = 1$). As shown in the figure, a primary residing in a site P_{i-1} first sends the observers to the designated sites, $R_{i,1}$ and $R_{i,2}$; and then migrates to the execution site, P_i . In our experimental system, ATP protocol is used for the reliable agent migration. The *ATP_Begin* and *ATP_Complete* in the figure denote the initiation and the completion of the ATP protocol.

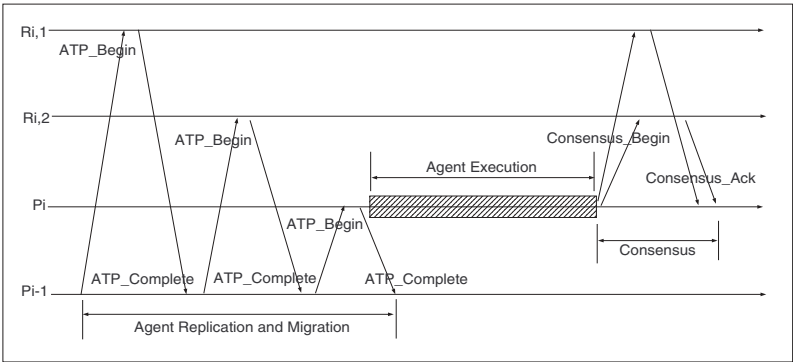


Fig. 1. Agent Replication and Consensus

On the completion of the execution, the primary at the site P_i begins the consensus process by sending the *Consensus_Begin* messages to the observers. Each observer replies with the *Consensus_Ack* message. The primary decides the successful completion when it receives the *Consensus_Ack* messages from k observers. It then prepares for the agent replication for the next stage. When the primary fails, one of the observers takes over the execution. To decide which observer takes over the execution first, our experimental system follows the priority-based scheme proposed in [5]. In this scheme, the priority of each observer is predetermined and an observer may begin the execution only when all of the observers with the higher priority are known to fail.

3.2 Optimized Agent Replication

The first approach to optimize agent replication is the use of asynchronous agent replication. Fig. 2 describes the asynchronous agent replication process for three replicas. As shown in the figure, the primary migrates to the next site, P_i , first. And then, the other replicas are sent to the sites, $R_{i,1}$ and $R_{i,2}$, by the previous primary at site P_{i-1} . The new primary begins execution as soon as it arrives at P_i , and hence the agent execution at P_i is overlapped with the migration of the observers. Therefore, the migration time of observers can be compensated, if the agent execution time at P_i is long enough.

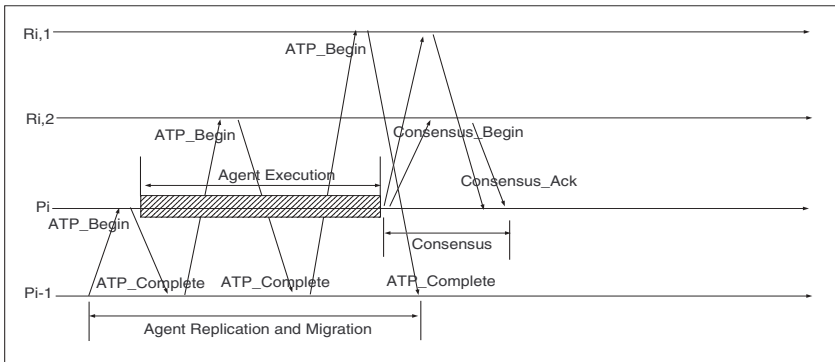


Fig. 2. Asynchronous Agent Replication

The implementation of asynchronous replication is however somewhat different from the one of synchronous replication. In the synchronous replication, the primary at P_{i-1} first makes replicas by using the *clone()* method and sends them to the sites $R_{i,1}$ and $R_{i,2}$. During this transfer, the primary obtains the *AgletProxys* for the replicas so that the primary can use them for the consensus, after it migrates to P_i and completes the execution.

However, if the primary migrates to P_i first in the asynchronous replication, all the threads for the primary at P_{i-1} should be terminated and hence the transfer of the other replicas becomes impossible. Therefore, in the asynchronous replication, the first replica sent by the primary at P_{i-1} becomes a new primary at P_i . The previous primary at P_{i-1} continues copying two more replicas. After successfully transferring them, the primary at P_{i-1} terminates itself. One more possible problem in the asynchronous scheme is that the new primary does not have **AgletProxys** for the observers. Hence, the previous primary has to transfer the **AgletProxys** of the observers to the new primary at P_i , before its termination.

Another approach to optimize the agent replication is to use the consensus agent instead of copying the primary agent. In the replication scheme, $2k + 1$ replicas are used to tolerate up to k failures. Among these, $k + 1$ replicas are for the possible execution which includes the alternative agent execution or the exception handling routines. However, the other k replicas are for attending the consensus only. Therefore, instead of copying $2k + 1$ replicas, using the k agents carrying the small consensus codes can reduce the migration time, especially when the size of the original agent is large. The consensus agents have the same agent identifier and the same stage number as the other replicas, so that they can be treated as the replicas of the same agent.

The pipelined replication is another approach to optimize the agent replication. Usually, the migration of an agent includes copying and transferring of the source codes of the agent and its intermediate data states. However, if an agent revisits a site which has been visited before, the site may have the source codes for the agent and hence it is enough to transfer the current data states. Using this property, in the pipelined replication, the set of migration sites for the observers are selected from the $2k$ recently visited sites. As a result, replication time and migration time for the source codes of an agent can partially be eliminated.

To compare the optimized replication schemes with the checkpointing scheme, our experimental system also implements the checkpointing. To checkpoint the intermediate states of a primary agent, the **snapshot()** method is called when a primary arrives on a new site. The **snapshot()** method is to save the current states of an agent into the secondary storage. If the system detects the abnormal termination of an agent, the agent image saved in the secondary storage is reactivated. The saved snapshot is safely eliminated when the agent is successfully migrated to the next site or completed by the user.

4 Performance Study

4.1 Experimental Setup

To evaluate each of the optimization approaches, replication schemes and the checkpointing scheme have been implemented on top of the Aglets system. For our experimental system, *Aglet DSK 1.1b2* was used. For the agents to access

the appropriate replication methods, the **Replication** class was created under the **AgletClass**. Three replication classes were created under the **Replication** class. One is the **AsynRep** class to support the asynchronous agent replication. Another is the **ConsRep** class under which the consensus agents can be used. The **PipeRep** class is used for the pipelined selection of the execution sites of the observers.

A cluster of six Pentium IV 1 GHz PCs connected through a 100 Mbps Ethernet was used for the experiments. Each machine supported an AgletContext and an agent traversed the sites in a predetermined order. The sites to execute the observers are randomly selected unless the pipelined replication is used. A task assigned to an agent consists of twenty stages and at each stage, the agent sleeps for one second instead of performing any action. For the stable performance, ten runs of the agent task were measured and then eight measured values were averaged out, excluding the lowest and the highest values.

Also, to observe the influence of the agent size, we implemented a **Sizer** class, which iteratively inserts garbage values in the Vector class so that the size of the serialized agent object can be controlled.

4.2 Experimental Results

Fig. 3 first shows the portion of replication time and the consensus time out of the total execution time. This performance was obtained when no optimization was applied. In the figure, the replication time and the consensus time are denoted by *Replication* and *Consensus*, respectively. The actual execution time for the task is denoted by *Sleep*. Also, the notation, $\alpha P\beta k$, denotes the agent execution with α replicas when the size of the agent is $\beta KBytes$.

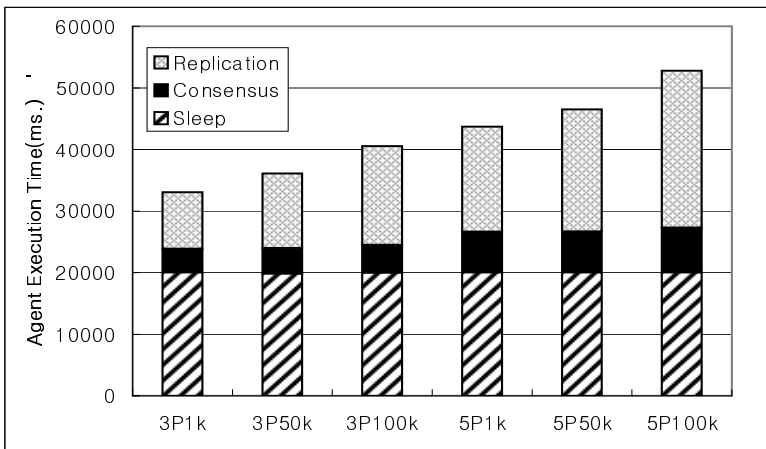


Fig. 3. Replication Time vs. Consensus Time

As shown in the figure, the consensus time depends on the number of replicas. Because the consensus time consists of the time for broadcast the consensus message and the time for obtaining the majority of the replies, it is obvious that a large number replicas impose more consensus time. It is also obvious that there is not much changes in the consensus time, as the agent size increases. However, the replication time is greatly influenced by the number of replicas as well as the size of the agent. Since for the synchronous replication, the primary begins a new stage after the whole replica migration has been completed, the waiting time of the primary should be longer as the number of replicas or the agent size increases. As it is also noticed from the figure, the replication time takes larger portion of the total execution time compared to the consensus time. This can be the main motivation of the replication optimization.

Fig. 4 shows the effect of each optimization approach to the agent replication. Fig. 4.(a) first shows the migration time of the agent when the asynchronous replication was applied. Besides the asynchronous replication, no other optimization was applied. The asynchronous replication achieves 16%–24% reduction of the migration time compared with the synchronous replication, when the number of replicas are three. In this case, the amount of reduction becomes larger as the agent size is increased. The possible reason is that most of the agent migration time can be overlapped with the execution time of the next stage. However, when the number of replicas is five, at most 12% reduction was achieved for the small agent. The reduction of the migration time for the large agent is negligible. Maybe, the agent migration in this case cannot be fully overlapped with the next stage execution since the number of agent and the agent size are too large.

Such undesirable performance of the "5P100k" case can be solved with the consensus agent. Fig. 4.(b) shows the effect of consensus agents. Besides the use of consensus agents, no other optimization was applied. Basically, using the consensus agent reduces the size of k replicas among $2k + 1$ replicas. Hence, the performance gain must be larger as the value of k increases and also the agent size becomes increased. As it is expected, we can observe up to 29% reduction of the migration time when the number of replicas is five and the agent size is 100 KBytes. For the five replicas, two replicas are substituted by the consensus agents, which causes the large reduction. However, for the three replicas, there can be a reduction of one replica. As a result, there was not much benefit for the case "3P1k", especially when the agent size is small. For this case, the reduction is only 14%, which is even smaller than that of the asynchronous replication.

Fig. 4.(c) shows another effect of the pipelined selection of the observer sites. Besides the use of pipelined optimization, no other optimization was applied. By choosing the previously visited sites for the observers, the migration time of class codes for the observer can be eliminated. Hence, the pipelined approach can benefit the agent which is computation intensive or carries small data. However, in our experimental setup, we intentionally inserted more data to increase the agent size and hence the large-sized agents can have only marginal benefits from this approach. The small agent, a large portion of which consists of source codes, can have the benefit from this approach. Fig. 4.(c) confirms this logic. As shown

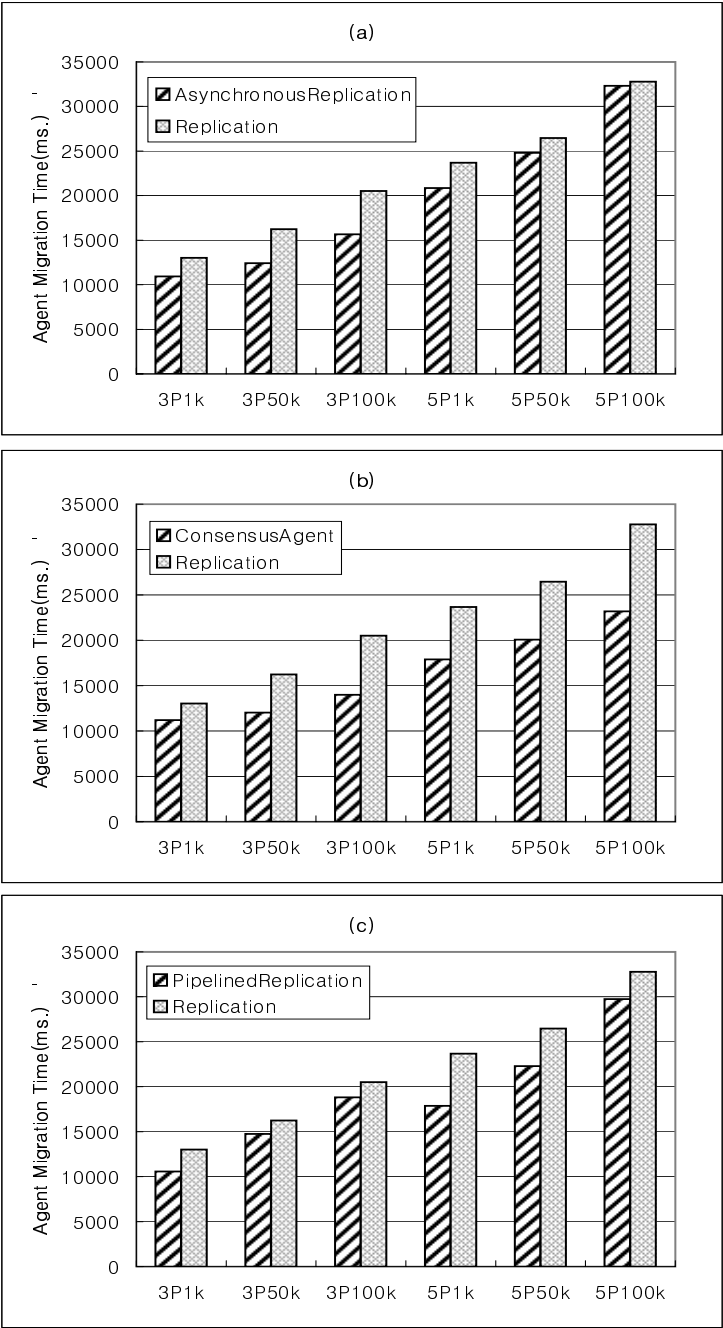


Fig. 4. The Effect of Optimization Approach

in the figure, the 100KBytes-sized agents achieve only 8% and 9% reduction of the migration time when the number of replicas is three and five, respectively. However, for the 1KBytes agents, the reduction of the migration time becomes 19% and 25% when the number of replicas is three and five, respectively. This is the most reduction for the 1KByte agent.

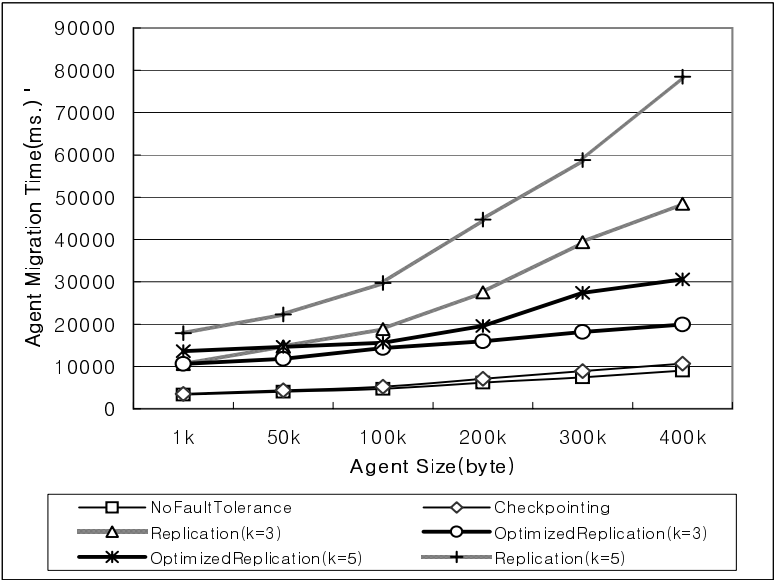


Fig. 5. The Performance of Optimized Replication

Now, Fig. 5 shows the effects of three optimization approaches. When all three optimization schemes were implemented, the system at least achieved 19% and 24% reduction of the migration time for three and five replicas, respectively. The reduction becomes larger as the size of the agent is increased. For 100KBytes agents, 30% and 37% reduction of migration time was achieved for three and five replicas. When the agent size becomes 400KBytes, 59% and 61% of migration time reduction was achieved. By applying three optimization approaches together, the system becomes possible to have the best performance provided by each approach.

However, as it is noticed in the figure, there is still the performance gap between the replication scheme and the checkpointing scheme. Consider the case that agent size is 1KByte and three replicas are used. The difference in the migration time of two fault-tolerance schemes is 7103 ms. However, among this time, the inter-agent consensus takes 3805 ms. In other words, the replication itself incurs only 3298 ms. extra time, which is almost 64% reduction on the replication time. The reduction becomes larger as the number of replicas and

the agent size are increased. Therefore, we can conclude that using three optimization approaches, the system may achieve possibly the best replication time.

5 Conclusions

In this paper, we have presented three optimization approaches to the agent replication. In the presented schemes, the migration and the execution of the primary agent proceeds asynchronously with the agent replication. Also, to reduce the replica migration time, the use of small-sized consensus agents was considered and the pipelined selection of the observer sites was used to reduce the time for the source code migration. To evaluate the performance of the optimization approaches, we have implemented the various replication schemes on top of the Aglets system and measured the performance. The performance results show that the optimized replication achieves 37% reduction on the replication overhead, when the agent size was 100KBytes and five replicas were used. Also, the cost reduction can be enlarged as the agent size and the number of replicas are increased.

Acknowledgments. This work was supported by grant No. (R04-2002-000-20102-0) from the Basic Research Program of the Korea Science & Engineering Foundation.

References

1. Gendelman, E., Bic, L.F., Dillencourt, M.B.: An Application-Transparent, Platform-Independent Approach to Rollback-recovery for Mobile Agent Systems. Proc. of the 20th Int'l Conf. on Distributed Computing Systems (2000).
2. Johansen, D., Marzullo, K., Schneider, F.B., Jacobsen, K.: NAP: Practical Fault-Tolerance for Itinerant Computations. Proc. of the 10th Int'l Conf. on Distributed Computing Systems (1999).
3. Karjoth, G., Lange, D.B., Oshima, M.: A Security Model for Aglets. IEEE Internet Computing (1997).
4. Park, T., Byun, I., Kim, H., Yeom, H.Y.: The Performance of Checkpointing and Replication Schemes for Fault Tolerant Mobile Agent Systems. Proc. of the 21st Symp. on Reliable Distributed Systems (2002) 256–261.
5. Pleisch, S., Schiper, A.: FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach. Proc. of the Int'l Conf. on Dependable Systems and Networks (2001) 215–224.
6. Silva, L., Batista, V., Silva, J.G.: Fault-Tolerant Execution of Mobile Agents. Proc. of the Int'l Conf. on Dependable Systems and Networks (2000).
7. Strasser, M., Rothermel, K.: Reliability Concepts for Mobile Agents. International Journal of Cooperative Information Systems, Vol. 7, No. 4 (1998) 355–382.
8. Strasser, M., Rothermel, K.: System Mechanism for Partial Rollback of Mobile Agent Execution. Proc. of the 20th Int'l Conf. on Distributed Computing Systems (2000).

A Transparent Software Distributed Shared Memory

Emil-Dan Kohn and Assaf Schuster

Computer Science Department, Technion
{emild,assaf}@cs.technion.ac.il

Abstract. Software Distributed Shared Memory (SDSM) systems use clusters to provide yet another level of scalability to multi-threaded shared-memory applications. However, linking with SDSM libraries usually requires adaptation of the program's system calls to the SDSM specific APIs, alignment of program variables to page boundary, in-depth verification of the program against the SDSM memory model, transforming global variables to be dynamically allocated, and more of the like. In this work we present the transparent SDSM - an SDSM that can efficiently execute any multi-threaded program (given in binary compiled form). The memory model of the transparent SDSM correctly supports any shared-memory application, whether programmed using relaxed or strict consistency in mind. By presenting a prototype and measurements, we show that the performance of the transparent SDSM is not compromised, essentially matching that of a non-transparent high-performance SDSM.

1 Introduction

The programming interface offered by most software distributed shared memory (SDSM) systems consists of a header file and a library. In order to use the services provided by a standard SDSM system, an application must link with the SDSM system library. At runtime, the application calls a special function which initializes the SDSM system. After that, the application accesses the services offered by the SDSM system via the application programming interface (API) exported by the SDSM system library.

Unfortunately, the APIs provided by SDSM systems are not standard. Many SDSM systems present incompatible APIs and different models for the behavior of the shared memory. Porting an existing application to an SDSM system, or from one SDSM system to another is thus a tedious and error-prone process. Moreover, such an approach assumes that the source code of the application is available, which is not always the case.

On the other hand many modern operating systems support the *multi-threading* programming paradigm. A multi-threaded application consists of several execution threads which run in parallel and are scheduled independently on the available processors. Each thread has its own stack and CPU register set, but threads belonging to the same process share a common address space.

Thus several threads from the same process can communicate efficiently through global variables. The multi-threaded programming model is intuitive and easy to use. Multi-threading is available on most operating systems via portable, standardized APIs (POSIX on UNIX-like systems, Win32 on Windows systems) or is directly supported by modern high-level languages such as Java.

The standard execution environments for parallel multi-threaded applications are symmetric multi-processor (SMP) machines. The major disadvantage of SMP machines is that they are not easily extendible, and that their maximum number of processors is fairly limited. This poses a serious limitation on the maximum scalability that can be achieved by these machines. On the other hand SDSM systems and computing clusters in general can be easily extended, and are far less limited in the maximum number of computing nodes than SMP machines. Thus, their scalability can be much better than that of the SMP machines.

In this paper we show the design and implementation of an SDSM system which combines the convenience of the multi-threaded programming model with the extra level of scalability offered by computing clusters. Our SDSM system allows running standard multi-threaded applications on a computing cluster and lets them take advantage of the computation power of all the nodes in the cluster. We assume that the applications are available only in compiled (binary) form; no assumptions whatsoever are made about the availability of the applications source code. The SDSM system is completely transparent to the applications.

In order to achieve transparency, the SDSM system must emulate the execution environment of standard multi-threaded applications. Our approach for emulating the execution environment of multi-threaded applications is based on a fine granularity page-based SDSM system, which uses strict consistency. We use *binary interception* for intercepting the calls the application makes to the operating system/C standard library and redirecting them to the SDSM system library.

However, intercepting OS calls is not sufficient to achieve full transparency. First, multi-threaded applications share data via global variables, and thus the global variables must be managed by the SDSM as well. Second, while for most SDSM systems initialization is a trivial task which is accomplished by calling a special function at the start of the application execution, this simple solution cannot be used for a transparent SDSM system running standard multi-threaded binary applications. The initialization code must be injected somehow into the application after it has been loaded. Finally, memory allocation, commonly optimized for on-the-fly calls in multi-threaded applications, is traditionally triggered at the beginning of execution (out of the critical path for “speedup measurements”) for SDSM benchmarks. Clearly, the transparent SDSM must match the light-weight allocation capabilities assumed by multi-threaded applications.

It turned out not easy for a base SDSM system to meet the performance, correctness and portability attributes challenged by the transparent SDSM: it must tolerate false sharing, it must support the standard multi-threaded memory model, and it must allow for efficient intra-node multi-threading. Many SDSM systems do not have good answers to this combination: page-based systems tend

to use relaxed consistency to avoid false sharing, and relaxed consistency does not preserve the multi-threading memory model; page-based systems must share protection between threads thus corrupting complex, carefully-optimized SDSM protocols; non page-based systems, on the other hand, do not fully avoid false sharing, impose instrumentation overhead, and tend to be machine-architecture dependent.

We have thus implemented a prototype of the transparent SDSM system, by adapting the Millipede 4.0 SDSM [6,8]. Millipede 4.0 can share memory in application-tailored granules, thus efficiently avoiding false sharing. In this way, Millipede 4.0 matches the performance of relaxed consistency SDSMs, despite using the OS protection mechanism, and despite providing strict consistency with an exceptionally thin-protocol. The results show that the transparent SDSM matches the performance of Millipede 4.0, with a negligible (non-measurable) overhead.

2 Intercepting the Operating System API

One of the key requirements for making our system fully transparent is the ability to intercept all the relevant calls the user application makes to the operating system and/or to the standard C library and redirect these calls to the SDSM system library. Unlike many SDSM systems, we assume nothing about the availability of the source code of the original application. Therefore we have to intercept the calls made by the application at the binary code level.

Binary Interception. We start by setting up a common terminology: **Target:** the original operating system call/library function that we want to intercept; **Detour:** The function that should be invoked whenever the application attempts to invoke the *target*. **Trampoline:** All the techniques described below allow calling the original system call/C library function. Often the *detour* invokes the original *target* as a subroutine and performs some extra processing. The mechanism which makes this possible is called a *trampoline*. The main difference between the different interception techniques is in the mechanism to construct the the trampoline.

The idea behind the technique that we use, called Target Function Rewriting [5], is to replace the first bytes of the *target* by a *jump* instruction to the *detour*. This way whenever the application code tries to invoke the *target*, control will be transferred to the *detour* because of the *jump* instruction. Creating the *trampoline* is a little bit more complicated. It involves saving aside the bytes that are going to be overwritten by the *jump* instruction and appending a new *jump* instruction to the rest of the original *target* code.

This technique has several advantages over other techniques. First, it does not assume that the *target* resides in a DLL (as do IAT-rewrite techniques). Second, it does not require knowing all functions in the target DLL (as do proxy DLL techniques) Third, it is portable, as it has no special requirements from the underlying platform (such as import tables). Finally, the required disassembler

is very simple (much simpler than that required by naive code instrumentation techniques).

The Intercepted Operating System API Subset. The operating system API subset that is intercepted includes *thread management*, *dynamic memory management*, *synchronization*, and *I/O*. Using binary interception, these functions are replaced by their counterparts from the SDSM system library. Threads are created using the SDSM system thread creation API. The SDSM system makes sure that a handle to a thread object is valid on all the nodes of the SDSM system. Dynamic memory is allocated from the SDSM area. A pointer returned by the memory allocation function is valid on all the nodes. The synchronization primitives are also global in that they involve the threads on all the nodes of the SDSM system, not just the threads on a single node. For example an SDSM lock guarantees mutual exclusion for all the threads running in the SDSM system, regardless of their location.

Virtual Handles. When a program running on a SDSM system requests a resource, ultimately the request will propagate via the SDSM system library to the local operating system running on a certain node, which will return a handle to that resource. If we aim for full transparency, this resource should be usable – via its handle – by threads running on nodes other than the node on which the resource has been allocated, in the same way as a resource allocated by one thread in a multi-threaded environment can be used by all the other threads belonging to the same process. However, the standard resource handle returned by the operating system is valid only on the node that made the request. Moreover, it is possible for two handles to two different resources which have been allocated on two different nodes to have an identical bit pattern, because each node runs its own copy of operating system. In this case, there is no way for the SDSM system to distinguish between these two resources.

In order to overcome these problems, we have devised the virtual handles mechanism. Resource handles returned by the SDSM system library functions (which replace the operating system API functions using binary interception) have a special bit pattern which makes them distinct from standard operating system resource handles, and are guaranteed to be unique across all the SDSM nodes. These handles are called *virtual handles*. The bit pattern of the virtual handles is chosen such that it is straightforward for functions from the SDSM system to locate the actual operating system resource handle, as well as the node on which the handle is valid. Since the application programs do not care about the bit pattern of the resource handles, the virtual handles mechanism is completely transparent to them.

3 Moving the Global Variables to the SDSM

Most SDSM systems provide shared memory to applications only via a `malloc()`-like API function. However, for full transparency, this is not sufficient. The threads of an ordinary multi-threaded application can share data through global variables, whereas typical SDSM systems allow sharing only via dynami-

cally allocated variables. Therefore we have to provide a means for sharing the global variables among the nodes of our system. Since we want to run our applications without any re-compilation, the only reasonable solution is to map the global variables to the shared memory.

The Executable Loading Process. Most modern operating systems use memory mapping for loading executable files into memory. The executable file mirrors the in-memory image of a process. An executable file consists of several parts, called *sections*. When loading an executable, the operating system loader maps each section separately and sets the protection attributes accordingly. For example the `.text` section (also known as `.code` section on other systems) which contains the code of the application is typically mapped with *read+execute* permissions. The sections which are of more interest to us are `.data`, which contains all the initialized global variables, and `.bss` which contains all the uninitialized global variables. Both sections are mapped with *read+write* permission by the loader. Typically, the `.bss` section does not occupy space in the executable. The loader allocates the necessary space only when the executable file is loaded.

Moving the Global Variables to the Shared Memory. The SDSM system on which we base our implementation uses the MULTIVIEW technique in order to achieve high performance and eliminate false sharing [6]. A brief introduction to MULTIVIEW can be found in section 6. In MULTIVIEW, each memory page is mapped several times to the address space of the process using it. Each such mapping is called a *view*. The SDSM system uses one special view, called the *privileged view* for updating the contents of the shared memory. All the other views, called *user views* are used by the application. The SDSM system adjusts the access permissions of the user views according to the consistency protocol.

The crux in the implementation is to exploit the fact that both the operating system loader and the MULTIVIEW technique use memory mapping. The memory mappings performed by the operating system on the `.data` and `.bss` sections of the executable are used as user views. The privileged view is allocated by the SDSM system at initialization.

Unfortunately, because of the Copy On Write mechanism (COW), which is applied by the OS on the global variable sections (`.bss` and `.data`), mapping the privileged view is not easy. Simply setting up a privileged view, and using it for accessing the global variables will result in their copy to a different location, which would invalidate the OS-created view. Our solution is to copy the sections containing global variables to a location that is not COW-protected, and then remap the OS view. The details of this mechanism can be found in the full version.

4 System Initialization

Most SDSM systems require some sort of initialization to be performed before a user application can use their services. These SDSM systems provide an initialization function in their API. User programs have to call this function before

calling any other function from the SDSM system API. Typically the call to the SDSM initialization function is placed at the beginning of the `main()` function in the application source.

Unfortunately this simple initialization procedure cannot be used by fully transparent SDSM systems, because these systems run ordinary multi-threaded applications that have already been compiled for the underlying platform. These applications are aware neither of the SDSM system nor of its initialization function, because these applications have not been linked against the SDSM system library. Therefore we have to devise a more sophisticated technique for the transparent SDSM system initialization, which is described in this section.

For all C/C++ programs, `main()` is considered the program entry point. However, `main()` is *not* the function that is invoked immediately after the operating system loader has loaded the executable file into memory. Before `main()` gets invoked, the program has to perform several initialization tasks. After an executable has been loaded into memory, control is passed to the *entry point* of the executable. The address of the entry point function is written in the header of the executable file, so it can be easily located by the loader.

The entry point performs the following tasks: It initializes the C/C++ runtime library. This includes opening the standard I/O streams and initializing the dynamic memory management data structures. It retrieves the command line arguments, parses them if necessary and invokes the `main()` function. It invokes the `exit()` function, passing it the return value of `main()` as a parameter.

The code of the entry point and the initialization function resides in the C/C++ runtime library which any C/C++ program is linked with. Fortunately, the code between the entry point and the invocation of `main()` is fixed and does not depend at all on the application being executed. While the address of `main()` may change, the relative address of the *jump to subroutine* instruction which invokes `main()` is fixed. In other words, the instruction invoking `main()` lies at a fixed offset after the program entry point.

Based on this observation, we can devise the following procedure for the transparent SDSM system initialization: **(1)** Using a platform specific mechanism, we arrange that the program main execution thread gets suspended immediately after the executable has been loaded into memory. **(2)** Using a platform specific mechanism, we “forcefully” link the executable with the DLL that contains the code of the SDSM system library. This will cause the entry point of the DLL to be executed, which will perform the following actions: **(2.a)** Obtains the address of the entry point by examining the executable header. **(2.b)** From the address of the entry point, it obtains the address of the *jump to subroutine* instruction invoking `main()`. This address is located at a fixed offset from the address of the entry point. The address of the `main()` function is also obtained during this step from the target of the *jump to subroutine* instruction. **(2.c)** Overwrites the target address of the *jump to subroutine* instruction invoking `main()` with the address of a new function which does the following: **(2.c.i)** Invokes the SDSM initialization function. **(2.c.ii)** If the id of the current node is zero, creates a thread using the SDSM system API with `main()` being its

start function; otherwise, the current thread is put to sleep. **(3)** We resume the execution of the main thread that has been suspended during step (1) again by using a platform specific mechanism. The platform specific mechanisms will be detailed in the full version.

After Step (3) completes the application will execute the code from sub-steps (2.c.i) and (2.c.ii) instead of the usual `main()`. The SDSM system initialization function will be invoked on each node.

This procedure performs the interception of the Win32/standard C library calls, in addition to the usual initialization tasks of a standard SDSM system. On node zero, a thread will be created starting at the `main()` function. Note that this thread is started on node zero only, and not on all nodes as with non-transparent SDSM systems, since the transparent SDSM must emulate the behavior of standard multi-threaded applications that begin with only a single main thread.

The main thread must be created with the SDSM system thread creation API and not with the operating system API, because we want the main thread to behave like any other thread from the SDSM system (for example, its page fault handlers must be designated the SDSM fault handlers). In turn, the threads created from within the main thread will be created using the SDSM system thread creation API, due to the interception of the operating system thread creation API during the initialization of the SDSM system.

5 Optimizing Memory Allocation for Small Objects

Many SDSM systems use simple, sometimes even stop-the-world like memory allocators. Such allocators are easier to design, but their performance can be very poor. A typical simple memory allocator works like this: When a node wants to allocate memory, it sends a message to the memory allocation server, indicating the size of the memory area it wishes to allocate. The memory allocation server performs the allocation locally and broadcasts the result of the allocation (start address and size) to all the nodes in the SDSM system. Upon receiving the message from the allocation server, each node updates its memory management data structures accordingly and acks. In addition to this, the node that requested the allocation also returns the result to the caller.

This kind of solution is unacceptable for a transparent SDSM system. While the benchmarks for most SDSM systems are written such that they allocate the whole memory at the beginning of their execution and distribute it to all nodes, typical multi-threaded object-oriented programs perform many relatively small-sized allocations throughout their execution. Since transparent SDSM systems do not run applications specifically written for them, but rather they run ordinary multi-threaded programs, the memory allocator should be adapted to the behavior of such programs.

In order to satisfy the performance demands of a transparent SDSM system, we have devised a new memory allocator which is better suited to the memory behavior of real object-oriented programs. This new allocator minimizes the

communication that takes place during memory allocation, thereby significantly improving its performance. In fact, no communication takes place at all during memory allocation.

While the implementation of this scheme seems straightforward on other SDSM systems, this is not the case for MULTIVIEW. An initial allocation of views should be computed, and the virtual address space should be split according to the allocation. However, because this complication is specific to the MULTIVIEW method, its details are out of the scope of this paper. The interested reader can find them in [4].

6 Performance Evaluation

In this section we give an overall evaluation of the transparent SDSM system runtime performance. Our test-bed consists of a cluster of Pentium PCs, interconnected by a Myrinet LAN. Each node runs Windows NT 4.0 as the underlying operating system. We have based the implementation on the Millipede 4.0 SDSM system, using the MULTIVIEW technique.

The MultiView Technique and the Millipede 4.0 SDSM System.

MULTIVIEW [6] is a technique to eliminate false sharing completely, despite using the memory page protection mechanism. The basic idea is to use the underlying operating system's virtual address mapping mechanism to map each shared variable to its own virtual page. This way, the access permissions of a variable do not affect the access permissions of other variables.

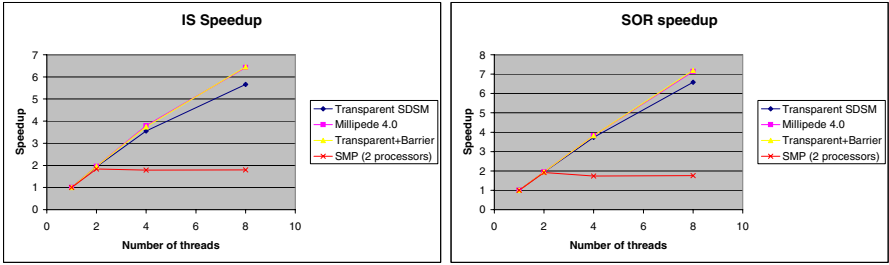
A physical page is mapped at several different locations in the virtual address space, one for each variable. These mappings are called *views*. User programs access each variable v through its separate view, using a designated ptr_v pointer which points to the corresponding view. The SDSM system can set the protection of each view according to the consistency protocol, without affecting the protection of other views.

Using this technique, Millipede 4.0 can achieve runtime performance comparable to systems employing relaxed consistency, but without posing any restrictions to the application programmer.

Performance Evaluation. In order to run the benchmarks on the transparent SDSM, we had to modify their code slightly. We had to completely decouple the application code from the SDSM library with whom the original benchmarks had to be linked, and convert the benchmarks into ordinary multi-threaded Win32 applications. This involves the following main steps: **(1)** Calls to `dsm_malloc()` – the shared memory allocation function – are replaced with calls to the standard C `malloc()` function. **(2)** SDSM locks are replaced by Win32 `CRITICAL_SECTION` synchronization primitives. **(3)** Since the barrier synchronization primitive has no direct Win32 equivalent, we had to emulate it using Win32 API functions. We have experimented with two implementations, one based on a combination of Win32 events and critical sections, and another one based on a server thread and Win32 messages. **(4)** The calls to the `dsm_distribute()` function, which used to scatter data to all the nodes of

the Millipede 4.0 system, are no longer necessary, because global variables are shared among all the nodes, so these calls are removed. (5) The explicit call to `dsm_init()` – Millipede 4.0’s initialization function – is no longer necessary, because the transparent SDSM initializes itself automatically before the execution of `main()` as described in section 4. After these transformations, the benchmarks become stand-alone Win32 multi-threaded applications which no longer require linking with the Millipede 4.0 library. The results are shown in figure 1.

Executing the stand-alone Win32 versions of the benchmarks on the transparent SDSM system yields speed-ups comparable to those obtained by the original Millipede 4.0 system. The differences in speed-ups between the transparent SDSM system and the classical Millipede 4.0 SDSM system are less than 10%.



(a) IS speed-up comparison

(b) SOR speed-up comparison

Fig. 1. Speed-up comparisons. Other benchmarks show similar behavior (see full version).

The main sources of overhead are the implementation of the barriers and the extra code executed by the API interception layer, though the latter is negligible. Since the barriers are implemented using ordinary Win32 function calls (Win32 has no native support for barrier synchronization primitives), the number of messages passed around is much higher than in the Millipede 4.0 implementation. In order to illustrate this, we have written a version of the barriers library which has the same interface as the library used by the transparent SDSM system, but instead of being based on the Win32 API only, invokes the original Millipede 4.0 implementation behind the scenes.

The results of running the benchmarks with this version of the barrier library are shown by the “Transparent+Barrier” graphs in figure 1. We notice that the run-time performance of this version is extremely close to the original Millipede 4.0.

7 Related Work

Relaxed consistency SDSM systems, such as Munin[1,3], TreadMarks[7], and Midway[2], place restrictions on application programs to ensure their correct

execution (e.g., no intentional *data races*). In contrast, our transparent SDSM supports a strict memory model. Thus, the transparent SDSM can execute correctly any program, regardless of the memory model that the programmer had in mind. In particular, any multithreaded program can be executed correctly.

Instrumentation-based SDSMs, such as Blizzard[10] and Shasta[9], modify the executable by wrapping each `load` and `store` instruction with code that implements the consistency protocol. This approach has two main disadvantages. First, the overhead incurred by the extra code to check all memory accesses can be quite significant, and requires aggressive optimization techniques for its minimization. Second, the technique is architecture-specific and thus not portable. In contrast, our page-based SDSM system achieves fine sharing granularity without the overhead associated with code instrumentation. Furthermore, despite using instrumentation, our use of the target rewriting technique is relatively portable. As proven by our results, applying it on the operating system calls incurs only negligible overhead.

References

1. J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 90–99, June 1997.
2. B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed system. In *Proc. of 38th IEEE Int'l Computer Conf. (COMPCON Spring '93)*, pages 528–537, February 1993.
3. J. B. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995.
4. E.-D. Kohn. A Transparent Software Distributed Shared Memory, 2002. M.Sc. thesis, Computer Science Department, Technion.
5. G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *3rd USENIX Windows NT Symposium*. Microsoft Research, July 1999.
6. A. Itzkovitz and A. Schuster. Multiview and millipage – fine-grain sharing in page-based dsms. In *Proc. of the 1999 Intl' Conf. on Parallel Processing (ICPP '99)*, pages 220–227, September 1999.
7. P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.
8. N. Nitzann and A. Schuster. Transparent Adaptation of Sharing Granularity in MultiView-based DSM Systems. *Software, Practice & Experience*, 31:1439–1459, October 2001. (Prelim. version (Best Paper Award) in proc. IPDPS, San Francisco, April 2001).
9. D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 174–185, October 1996.
10. I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

On the Characterization of Distributed Virtual Environment Systems^{*}

Pedro Morillo¹, Juan M. Orduña¹, M. Fernández¹, and J. Duato²

¹ Departamento de Informática. Universidad de Valencia. SPAIN

² DISCA. Universidad Politécnica de Valencia. SPAIN

{Pedro.Morillo,Juan.Orduna}@uv.es, jduato@gap.upv.es

Abstract. Distributed Virtual Environment systems have experienced a spectacular growth last years. One of the key issues in the design of scalable and cost-effective DVE systems is the partitioning problem. This problem consists of efficiently assigning clients (3-D avatars) to the servers in the system, and some techniques have been already proposed for solving it.

In this paper, we propose the correlation of the quality function proposed in the literature for solving the partitioning problem with the performance of DVE systems. Since the results show an absence of correlation, we also propose the experimental characterization of DVE systems. The results show that the reason for that absence of correlation is the non-linear behavior of DVE systems with the number of avatars in the system. Also, the results show that workload balancing mainly has an effect on system throughput, while minimizing the amount of inter-server messages mainly has an effect on system latency.

1 Introduction

Professional high performance graphic cards currently offer a very good frame-rate for rendering complex 3D scenes in real time. On other hand, fast Internet connections have become worldwide available at a relatively low cost. These two factors have made possible the current growth of Distributed Virtual Environment(DVE) Systems. These systems allow multiple users, working on different computers that are interconnected through different networks (and even through Internet) to interact in a shared virtual world. This is achieved by rendering images of the environment as if they were perceived by the user. Each user is represented in the shared virtual environment by an entity called *avatar*, whose state is controlled by the user input. Since DVE systems support visual interactions between multiple avatars, every change in each avatar must be propagated to the rest of the avatars in the shared virtual environment. DVE systems are currently used in many different applications, such as collaborative design, civil and military distributed training, e-learning or multi-player games.

One of the key issues in the design of a scalable DVE system is the *partitioning problem* [6]. It consists of efficiently assigning the workload (avatars) among different servers in the system. The partitioning problem determines the overall performance of the DVE system, since it has an effect not only on the workload each server in the

^{*} Supported by the Spanish MCYT under Grant TIC2000-1151-C07-04

system is assigned to, but also on the inter-server communications (and therefore on the network traffic). Some methods for solving the partitioning problem have been already proposed [6,7]. However, there are still some features in the proposed methods that can be improved. For example, the quality function proposed in the literature must be correlated with system performance, in order to design actually scalable and efficient partitioning strategies.

In this paper, we present the experimental correlation of the quality function proposed in the literature with the performance of DVE systems. Since the results show an absence of correlation, we also propose the experimental characterization of DVE systems, in order to analyze the reasons for that absence of correlation. This characterization study measures the impact of different parameters of the quality function on the performance of DVE systems, and it shows that the behavior of DVE systems is non-linear with the number of avatars in the system. Therefore, in order to design an actually scalable partitioning method for DVE systems, a different (non-linear) partitioning strategy must be proposed.

The rest of the paper is organized as follows: Section 2 describes the partitioning problem and the existing proposals for solving it. Section 3 details the proposed characterization setup that allows to experimentally study the behavior DVE systems. Next, Section 4 presents the correlation and evaluation results. Finally, Section 5 presents some concluding remarks and future work to be done.

2 The Partitioning Problem in DVE Systems

Architectures based on networked servers are becoming a de-facto standard for DVE systems [7,6]. In these architectures, the control of the simulation relies on several interconnected servers. Multi-platform client computers join the DVE system when they are connected to one of these servers. When a client modifies an avatar, it also sends an updating message to its server, that in turn must propagate this message to other servers and clients. Servers must render different 3D models, perform positional updates of avatars and transfer control information among different clients. Thus, each new avatar represents an increasing in both the computational requirements of the application and also in the amount of network traffic. When the number of connected clients increases, the number of updating messages must be limited in order to avoid a message outburst. In this sense, concepts like areas of influence (AOI) [7], locales [1] or auras [4] have been proposed for limiting the number of neighboring avatars that a given avatar must communicate with.

Lui and Chan have shown the key role of finding a good assignment of clients to servers in order to ensure both a good frame rate and a minimum network traffic in DVE systems [6]. They propose a quality function, denoted as C_p , for evaluating each assignment of clients to servers. This quality function takes into account two parameters. One of them consists of the computing workload generated by clients in the DVE system, denoted as C_p^W . In order to minimize this parameter, the computing workload should be proportionally shared among all the servers in the DVE system, according to the computing resources of each server. The other parameter of the quality function consists of the overall inter-server communication requirements, denoted as C_p^L . In order to

minimize this parameter, avatars sharing the same AOI should be assigned to the same server. Quality function C_p is defined as

$$C_p = W_1 C_p^W + W_2 C_p^L \quad (1)$$

where $W_1 + W_2 = 1$. W_1 and W_2 are two coefficients that weight the relative importance of the computational and communication workload, respectively. These coefficients should be tuned according to the specific features of each DVE system. Using this quality function (and assuming $W_1 = W_2 = 0.5$) Lui and Chan propose a partitioning algorithm that re-assigns clients to servers [6]. The partitioning algorithm should be periodically executed for adapting the partition to the current state of the DVE system (avatars can join or leave the DVE system at any time, and they can also move everywhere within the simulated virtual world). Lui and Chan also have proposed a testing platform for the performance evaluation of DVE systems, as well as a parallelization of the partitioning algorithm [6].

The partitioning method proposed by Lui and Chan currently provides the best results for DVE systems. However, the correlation of the quality function C_p with DVE system performance should be studied, and parameters W_1 and W_2 must be properly tuned. In this sense, the characterization of DVE systems is crucial in order to design partitioning strategies that actually improves the scalability of DVE systems.

3 Characterization Setup

We propose the characterization of generic DVE systems by simulation. The evaluation methodology used is based on the main standards for modeling collaborative virtual environments, FIPA [3], DIS [2] and HLA [5]. We have developed a simulation tool (a program written in C++) that models the behavior of a generic DVE system with a network-server architecture. Concretely, we have implemented a set of multi-threaded servers. Each thread in a server uses blocking sockets for communicating with a client. Each client simulates the behavior of a single avatar, and it is also implemented as a multi-threaded application. One of the threads of the client manages the communication with the server assigned to that client, and another thread manages user information (current position, network latency, etc.).

Our simulator model is composed of a set of S interconnected servers and n avatars. Following the approach specified in FIPA and HLA standards, one of the servers acts as the main server (called *Agent Name Service* [3] or *Federation Manager* [5]) and manages the whole system. The main server also maintains a partitioning file for assigning a given server to each new avatar. Avatars can join the simulation through this main server, that assigns each new avatar to one of the servers in the system. In each simulation, all avatars sharing the same AOI must communicate between them for notifying their position in the 3D virtual world. The message structure used for notifying avatar movements is the *Avatar Data Unit (ADU)* specified by DIS [2].

A simulation consists of each avatar performing 100 movements, at a rate of one movement every 2 seconds. Each time an avatar performs a movement, he notifies that movement to his server by sending a message with a timestamp. That server must then notify that movement to all the avatars in the same AOI of the sender avatar. When

that notification arrives to these avatars, they return an ACK message to the server, that in turn propagates that ACK messages to the sender avatar. When an ACK message arrives, the sender avatar computes the round-trip delay for communicating with each neighbor avatar. We have denoted this round-trip delay (measured in real-time) as the *system response*. When a simulation ends, each avatar has computed the average system response for the avatars in its AOI. At this point, all avatars send these average system responses to their respective servers, and each server then computes its own average system response. Finally, starting from the average value provided by each server the main server computes the average system response for the entire system, denoted as the *average system response* (ASR). An ASR value is computed after each simulation. An actually scalable DVE system must keep this measure as low as possible as the number of avatars in the system increases. On other hand, the system throughput is given by the maximum number of avatars that the system can manage while keeping the ASR below a certain threshold. Therefore, we have considered ASR as the main performance measure for characterizing the behavior of DVE systems.

In order to evaluate the performance of each partitioning method, usually 3 different distributions of avatars in the virtual world are proposed in the literature: uniform, skewed and clustered distributions of avatars. However, both the movement rate and also the AOI of avatars can also be adjusted in order to make that workload to be independent of the distribution of avatars in the virtual world. Therefore, for the sake of simplicity we have only considered a uniform distribution of avatars, with the same AOI and the rate of movements for all the avatars in the system.

4 Simulation and Correlation Results

In this section, we present the correlation and simulation results obtained for the DVE model described in the previous section. We have tested a great number of different DVE configurations, ranging from small virtual worlds (3 servers and 180 avatars) to large virtual worlds (900 avatars and 6 servers). Since we have obtained very similar results in both of them, we present the results for a small DVE configuration.

Figure 1 shows the performance results for a small DVE system composed of 3 servers and 180 avatars when different partitions (showing different values of the quality function C_p) are simulated. This Figure shows on the X-axis the values of C_p obtained for different partitions (assignments of avatars to the servers in the system). The Y-axis shows ASR values for the simulations performed with these partitions. Each point in the plot represents the average value of the ASR obtained after 30 simulations of the same DVE system. The standard deviation for any of the points shown in the plot was not higher than 25 ms. in any case.

This Figure clearly shows that C_p does not correlate with DVE system performance. ASR ranges in a 5% of its initial value (sometimes decreasing) as C_p values are greatly increased. Thus, for example, the ASR value obtained for a partition with a value of C_p equal to 800 (the worst value of C_p) is even lower (better) than the ASR values obtained for partitions with C_p values ranging from 400 to 700. There not exists a linear correspondence between the values of the quality function C_p and ASR. Therefore, a characterization study is needed in order to capture the behavior of DVE systems.

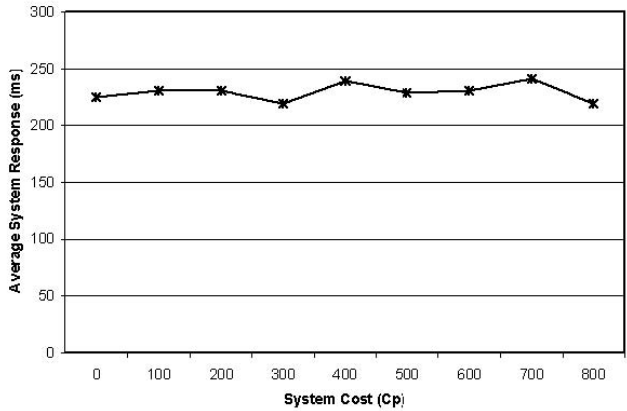


Fig. 1. Correlation of quality function C_p with average system response

In order to make a methodical characterization study, we have studied the behavior of DVE systems as the two terms of the sum in equation 1 vary. Given a DVE system composed of 3 servers, first we have optimized the value of C_p^W (workload balancing) and we have simulated both a partition with optimum (minimized) value of C_p^L (inter-server communication requirements) and also a partition with the worst value of C_p^L as possible. Also, for the same DVE system we have unbalanced the partitions to obtain the worst value of C_p^W as possible, performing the same simulations with the partitions showing both the best and the worst values of the term C_p^L .

Figure 2 shows the obtained ASR values for a DVE configuration of 3 servers as the number of avatars in the system increases. The value of C_p associated to the simulated partitions was zero in all the cases (perfect workload balancing and absence of inter-server messages, that is, C_p^L and C_p^W equal to zero). ASR seems to be invariant with the number of avatars (ASR plot has the shape of a flat line) until it reaches a saturation point. From that saturation point, ASR values greatly increase as new avatars are added to the simulation. These results clearly show that the behavior of DVE system is non linear with the number of avatars in the system. Since C_p is defined (equation 1) as a linear function of both workload balancing and inter-server communications, this non-linear behavior can explain the absence of correlation shown in Figure 1.

In order to determine the reason for the non-linear behavior of DVE systems, Table 1 shows the CPU utilization and the average system response in milliseconds (SR-Sx) obtained for each server corresponding to the simulation results shown in Figure 2, as well as the corresponding global ASR value. This table shows that the DVE system reaches the saturation point when *any* of the servers in the system reaches a CPU utilization of 100%. All the SR-Sx values increase as the respective CPU utilization so does, and there are not significant differences between the three SR-Sx values while the CPU utilization remains under 99% (simulation with 500 avatars). When CPU reaches more than 99% in any of the servers then the system response for that server greatly increases,

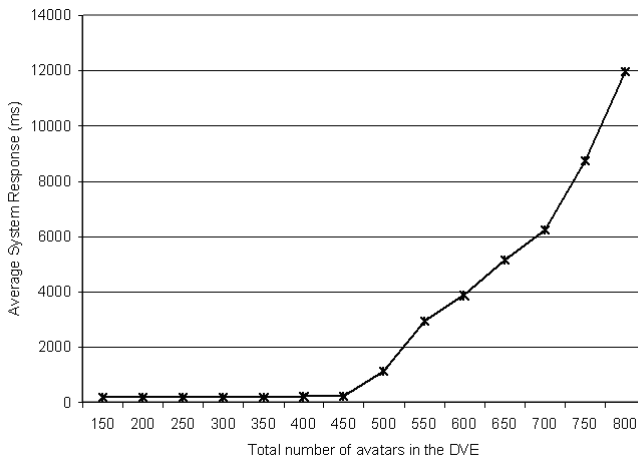


Fig. 2. Average system response as the number of avatars per server increases

also increasing global ASR accordingly. Therefore, we can conclude that the non-linear behavior of DVE systems shown in Figure 2 is due to the limit of 100% in any CPU utilization. Since quality function C_p does not take into account CPU utilization in order to measure the quality of a partition, it cannot take into account the non-linear behavior of DVE systems as the number of avatar increases.

Table 1. Average system responses and server CPU utilization in simulations shown in Figure 2

No. of avatars	%CPU1	%CPU2	%CPU3	SR-S1	SR-S2	SR-S3	Average SR
350 (116,117,117)	51	54	52	193.21	191.81	196.43	193.82
400 (133,133,134)	67	69	71	249.29	245.60	247.36	247.17
450 (150,150,150)	78	76	78	279.43	281.12	280.91	280.49
500 (166,167,167)	96	99	100	545.73	892.54	1775.85	1138.04
550 (183,183,184)	100	100	100	3021.44	2811.45	2982.12	2938.34
600 (200,200,200)	100	100	100	4356.91	3645.21	3409.20	3803.77
650 (216,217,217)	100	100	100	5320.65	5114.32	5112.22	5182.40

These results means that although a partition can provide perfect workload balancing and minimum inter-server communication requirements (thus providing a optimum value of C_p , as the partitions in Figure 2 do), ASR will be very poor if the amount of avatars assigned to any server requires a CPU utilization of 99% or more. Also, given a DVE system and given a certain amount of avatars in the system, the performance of that DVE system will remain practically invariant with C_p if none of the servers reaches a CPU utilization of 99%. This is the case for the simulations whose results are shown in Figure 1, where only 90% of CPU utilization was reached in the worst case.

Figure 2 shows the behavior of the DVE with the best partitions as possible. Table 2 shows the same measurements of Table 1 taken on the same DVE when simulating partitions with the worst C_p^L as possible (the maximum number of inter-server messages). When comparing these two tables in a line-by-line analysis, we can see that a reduction in the amount of inter-server messages (Table 1) results in a fixed reduction in ASR while the system is not saturated (the difference between ASR values of both tables in the first 3 lines remains constant, about 40 ms.). However, the effect of inter-server messages in ASR becomes more important as servers approach saturation. Also, we can see that for the same number of avatars Table 2 shows higher CPU utilization. Therefore, in order to maximize the throughput of a DVE system (the amount of avatars a given DVE system can manage without enter saturation) inter-server messages must also be minimized.

Table 2. Average system responses and server CPU utilization in simulations shown in Figure 2

No. of avatars	%CPU1	%CPU2	%CPU3	SR-S1	SR-S2	SR-S3	Average SR
350 (116,117,117)	61	64	65	221.45	214.62	217.81	217.96
400 (133,133,134)	76	75	79	281.41	291.31	289.91	287.94
450 (150,150,150)	80	82	83	350.13	333.73	342.39	342.08
500 (166,167,167)	98	100	100	942.45	1792.54	1675.85	1470.28
550 (183,183,184)	100	100	100	3674.56	3844.17	3364.22	3627.68
600 (200,200,200)	100	100	100	4991.34	4565.71	4398.83	4651.96
650 (216,217,217)	100	100	100	6541.34	6199.45	5853.37	6198.05

Additionally, Tables 3 and 4 show CPU utilization and server system responses for the same DVE system (composed of 3 servers) when the term C_p^W is maximized (worsened). The partitioning strategy whose results are shown in Table 3 minimizes inter-server communication requirements (the term C_p^L), while the partition whose results are shown in Table 4 maximizes this term. In a line-by-line comparison of both tables, we can see that CPU utilization are very similar in both tables. This result shows that the term C_p^L does not have any effect on the system throughput when the term C_p^W is worsened (the system reaches saturation with 425 avatars in both tables). However, the system responses obtained for each server greatly differs between the two tables.

Table 3. Average system response and CPU utilization for each server when C_p^L is minimized

No. of avatars	%CPU1	%CPU2	%CPU3	SR-S1	SR-S2	SR-S3	Average SR
375 (125,125,125)	72	74	71	221.23	233.57	225.71	226.84
400 (125,150,125)	71	81	73	219.36	285.21	218.77	241.11
425 (125,175,125)	73	100	70	228.74	1725.84	235.54	730.04
450 (125,200,125)	72	100	69	225.67	6308.87	218.99	2251.18
475 (125,225,125)	70	100	72	226.54	11554.86	221.42	4000.94

A comparison of Tables 3 and 4 with Tables 1 and 2 shows that minimizing the term C_p^W results in improving the throughput (the scalability) of DVE systems. Effectively, the same DVE system reaches saturation with 500 avatars in Tables 1 and 2, while reaching saturation with only 425 avatars in Tables 3 and 4. However, there not exists any significant difference in ASR values between Tables 1 and Tables 3 in the first two lines, neither between Tables 2 and 4 on the first two lines. These results show that the term C_p^W does not have an important effect on ASR. On the contrary, the term C_p^L mainly affects to ASR values (system latency), and this effect is amplified when the term C_p^W is worsened. However, if the term C_p^W is optimized then the impact of C_p^L in ASR is minimized.

Table 4. Average system response and CPU utilization for each server when C_p^L is maximized

No. of avatars	%CPU1	%CPU2	%CPU3	SR-S1	SR-S2	SR-S3	Average SR
375 (125,125,125)	69	72	73	217.89	241.13	233.45	230.82
400 (125,150,125)	71	83	72	251.41	323.22	218.31	264.31
425 (125,175,125)	69	100	73	1290.21	2992.85	1432.80	1905.29
450 (125,200,125)	72	100	70	5678.51	8861.91	6007.82	6849.41
475 (125,225,125)	71	100	72	9998.83	16873.21	10254.31	12375.45

5 Conclusions

In this paper, we have proposed the experimental correlation of the quality function proposed in the literature for solving the partitioning problem with the performance of DVE systems. Since results show an absence of correlation, we have also proposed a characterization study of DVE systems.

DVE systems show a non-linear behavior with the number of avatars in the system. Average system response (round trip-delay of messages notifying movements of avatars) remains practically invariant with the number of avatars in the system until the DVE system reaches a saturation point. This saturation point is given by the limit of a CPU utilization of 100% in any of the servers. When this limit is reached, then average system response greatly increases as new avatars are added to the system. We have also studied the effects of the two terms of the quality function proposed in the literature (workload balancing and the amount of inter-server messages) on the performance of DVE systems. The results show that workload balancing mainly has an effect on system throughput, while the amount of inter-server messages mainly has an effect on system latency. However, if none of the servers reaches saturation then the amount of inter-server messages does not have any significant effect on system latency, regardless of the amount of inter-server messages. Therefore, in order to design an efficient and scalable DVE system, the partitioning method should be targeted to balance the workload among the servers in the system in such a way that none of them reaches 100% of CPU utilization.

References

1. D.B.Anderson, J.W.Barrus, J.H.Howard, "Building multi-user interactive multimedia environments at MERL", in *IEEE Multimedia*, 2(4), pp. 77–82, Winter 1995.
2. DIS. 1278.1 IEEE Standard for Distributed Interactive Simulation-Application Protocols (ANSI). DMSO. DoD High Level Architecture. 1997.
3. FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000. Available at <http://www.fipa.org/specs/fipa00023/>
4. J.C.Hu, I.Pyarali, D.C.Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks", *Proc. of the 2nd. IEEE Global Internet Conference*, November.1997.
5. Kuhl, F., Weatherly, R, Dahmann, J., "Creating Computer Simulation Systems: An Introduction to the High Level Architecture", Prentice-Hall PTR, Upper Saddle River, NJ, 1999.
6. Jonh C.S. Lui, M.F. Chan, "An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems", *IEEE Trans. Parallel and Distributed Systems*, Vol. 13, March 2002
7. S.Singhal, and M.Zyda, "Networked Virtual Environments", *ACM Press, New York, 1999*.

A Proxy Placement Algorithm for the Adaptive Multimedia Server^{*}

Balázs Goldschmidt and Zoltán László

Budapest University of Technology and Economics
`{balage|laszlo}@iit.bme.hu`

Abstract. Multimedia services are becoming widespread, while the network capacity can not keep up with the growth of user demands. Usually proxies are used to overcome the QoS degradation. One of the fundamental problems of applying proxies is to recognize the relevant set of proxy placement criteria, and to find a good algorithm. In this paper we propose a greedy algorithm variant, *ams-greedy*, that aims at taking into account the different property preferences of the proxy-server and the proxy-client links. We also analyse the consequences of the initial results.

Keywords: Greedy algorithm, optimization, proxies, distributed multimedia.

1 Introduction

There is an increasing demand for on-line multimedia services, like video-on-demand. But as the number of the clients grows, and their distance from the servers is growing, the QoS (bitrate, jitter, etc.) of the video streaming decreases. The Adaptive Multimedia Server (AMS [1]) uses proxies, or data collectors, that collect the video from servers and stream them to the clients. The novelty is that the proxies can be dynamically migrated to new hosts where the provided QoS can be better. This kind of placement of the proxies is a special, dynamic case of the classical load balancing problem, and greedy algorithms have already proved to be a good candidate in similar optimization problems. In this paper we suggest a greedy algorithm variant and examine the experiences.

The rest of the paper is organized as follows. In section 2 a short introduction is given into the Adaptive Multimedia Server and the Host Recommender it relies upon, in section 3 our greedy algorithm variant is described, in section 4 we introduce the test environment and analyse the results, and finally in section 5 we conclude our work and propose future steps.

^{*} Work partially supported by the European Community under the IST programme – Future and Emerging Technologies, contract IST-1999-14191 – EASYCOMP. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

2 The AMS and the Host Recommender

The Adaptive Multimedia Server (AMS) is a part of a major project at the University Klagenfurt, that has the aim of creating a distributed virtual video-on-demand service that supports proactive adaptation techniques. [2]

The AMS comprises cluster manager, data manager (DM), data collector (DC), and client. When a client wants to see a movie, it contacts the cluster manager, and asks, if the video is available and serviceable to him. If the demand is admitted, the client is redirected to a data collector that will serve him using video streaming. The video is initially stored on the data managers, as stripe units, thus providing a better load balance. The task of the data collector is to collect the stripe units and weave them together. Usually the connection to the data managers is TCP-based, thus correct delivery of data is guaranteed. When the video is collected, the streaming to the clients uses the UDP-based RTP protocol, that does not guarantee correct data delivery. However, for a video stream, proper timing is usually more important than error-free delivery.

The novel idea was to place the data collectors dynamically according to (future) client needs. For example when a semester starts the students at the dormitory might want to see some videos displaying courses they want to attend. During this period it would be useful to have the data collectors (or even the data managers) near the dormitory. In other times of the year, however, the data collectors might be put somewhere else, and thus the nodes are available for other tasks.

The means of dynamic placement is a mobile agent based infrastructure, Vagabond2 [3,1], that comprises two major modules. One of them is the application module, that enables the migration of CORBA applications written in Java. The other module is the host recommender. The host recommender uses statistical information about the nodes and the network, and also hints on what the client demands will be. Using this information it recommends hosts that can serve as data collectors. The host recommender has to implement such a data collector placement algorithm, that can give nearly optimal results for different user requirements. These requirements, like low network load and low jitter, are usually contradictory. When testing the algorithm we had to check how the QoS varies in different conditions.

3 The *ams-greedy* Algorithm

Proxy and cache placement is a heavily researched topic with a large number of papers published [4,5,6]. We can not use these results without modifications, because they only investigate static placement. Moreover, the goals of a web proxy and our data collector are also different. In case of the web proxy or a general data replicator the clients download lots of small files, and the aim is to provide the most pages without contacting the web server, and with guaranteed data delivery. In case of the data collector usually one huge media file is transmitted to the clients, while the jitter, network load, etc. has to be minimized. There is usually no harm in losing some smaller parts.

The basic problem is the following: we have several clients, several data managers, and we have to place the data collectors (the ‘proxies’) in a way that satisfies clients’ demand, such as bitrate’s and jitter’s remaining inside an allowable range. It’s an extra advantage, if we can keep the network load low. Phrasing the problem that way it is obvious, that it is a special type of the NP-hard facility location and k-median problems, a well studied area where lots of papers have been published [7,8,9]. Most of the papers propose some variation of the greedy algorithm [10]. We have developed a new, specialized variant, called *ams-greedy*.

Greedy algorithms usually have a cost function, and a set of elements. The goal is to build such a subset that has enough elements to cover some problem. When building this subset, the greedy algorithm selects the next element with the lowest cost from the original set. When the subset gives a complete cover, the algorithm stops.

The cost function of *ams-greedy* is the sum of the latencies between each client and the DC closest to it (weighted by w_{cl}), plus the sum of the latencies between each DC and the DM closest to it (weighted by w_{dm}). We decided to use the latency of the network links, because this is the most easily measurable property, and this property can be modelled the easiest way. Based on the evaluation of the test results, we plan to incorporate other link properties, like jitter and bandwidth, as well. In the cost function the effective parameter is the ratio of the above weights. This parameter is a consequence of the asymmetric kind of the problem: the connection of the DCs to the DMs is TCP-based, the packages can not be lost, while the DCs’ connection to the clients is UDP-based, thus the packages are sent only once, and may be lost.

The algorithm starts with an empty set of data collector nodes, with infinite cost. In every iteration the set is extended with that node that would provide the smallest cost if added to the set. The nodes that are not needed by any client are removed from the set. If the new set costs less than the one the iteration started with, a next iteration starts. Otherwise the result is the final set of the previous iteration. (For a formal description see table 1.)

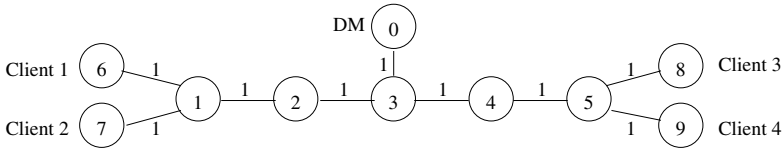


Fig. 1. An example network topology

On figure 1 a small example is shown. On node 0 there is a data manager, on nodes 6, 7, 8, and 9 there are clients. The latency on every link is 1. Let us follow the algorithm choosing node sets for the data collectors. The weights w_{cl} and w_{dm} are both set to 1.

Table 1. The *ams-greedy* algorithm

```

// nodes is the set of nodes; lat is the latency between any two nodes
// clients is the set of clients; dms is the set of DMs, dcs is the set of DCs
// wcl is the weight of the latency between DC and client
// wdm is the weight of the latency between DC and DM

function cost(nodes, lat, clients, dms, dcs, wcl, wdm)
  if (dcs == {}) return ∞ fi
  ret := 0
  foreach c ∈ clients do ret += wcl * mind ∈ dcs(lat(c, d)) od
  foreach m ∈ dms do ret += wdm * mind ∈ dcs(lat(m, d)) od
  return ret
end

function select_next(nodes, lat, clients, dms, dcs, wcl, wdm)
  cand := nodes \ (clients ∪ dms ∪ dcs)
  let d ∈ cand : /∃ c ∈ cand : (cost(nodes, lat, clients, dms, dcs ∪ {c})
    < cost(nodes, lat, clients, dms, dcs ∪ {d}))
  return dcs ∪ {d}
end

algorithm ams-greedy(nodes, lat, clients, dms, dcs, wcl, wdm)
  dcs := {}
  repeat
    dcs' := select_next(nodes, lat, clients, dms, dcs, wcl, wdm)
    dcs'' := {d ∈ dcs' | ∃ c ∈ clients /∃ d' ∈ dcs' : (lat(d', c) < lat(d, c))}
    if (cost(nodes, lat, clients, dms, dcs) > cost(nodes, lat, clients, dms, dcs''))
      dcs := dcs''
    fi
  until dcs unchanged
end

```

In the first iteration the set of data collectors is empty. The algorithm computes for every free node (1-5) the cost of placing the data collector there. The costs are

$$\text{cost}(\{1\}) = w_{cl} \times (\text{lat}(1, 6) + \text{lat}(1, 7) + \text{lat}(1, 8) + \text{lat}(1, 9)) + w_{dm} \times \text{lat}(1, 0) = 15$$

$$\text{cost}(\{2\}) = 14; \quad \text{cost}(\{3\}) = 13; \quad \text{cost}(\{4\}) = 14; \quad \text{cost}(\{5\}) = 15$$

With the smallest cost, set {3} is selected. In the second iteration the initial set of data collectors is {3}, the free nodes are 1, 2, 4, and 5. The costs of the possible new sets are:

$$\text{cost}(\{1, 3\}) = 12; \quad \text{cost}(\{2, 3\}) = 13; \quad \text{cost}(\{3, 4\}) = 13; \quad \text{cost}(\{3, 5\}) = 12$$

The first and last set have the smallest cost, now the first is selected. Using this set, client 1 and 2 are connected to node 1, client 3 and 4 to node 3, no node

is removed from the set. The set of DCs at the end of this iteration is $\{1, 3\}$, because it costs less than the previous set, $\{1\}$. In the third iteration the initial set is $\{1, 3\}$. The costs of the possible extensions are

$$\text{cost}(\{1, 2, 3\}) = 14; \quad \text{cost}(\{1, 3, 4\}) = 12; \quad \text{cost}(\{1, 3, 5\}) = 11$$

Using the last set client 1 and 2 are connected to node 1, client 3 and 4 to node 5. Node 3 is not used by any client, therefore it is removed. The new set is $\{1, 5\}$, with cost 10. In the fourth iteration the initial set is $\{1, 5\}$. The costs of the possible extended sets are

$$\text{cost}(\{1, 2, 5\}) = 12; \quad \text{cost}(\{1, 3, 5\}) = 11; \quad \text{cost}(\{1, 4, 5\}) = 12$$

The one with the smallest cost (11) is $(\{1, 3, 5\})$, same as above. But in this set node 3 is deleted again, the result is once more $\{1, 5\}$. The cost did not change, thus the algorithm ends. The result is that we have two data collectors: client 1 and 2 are served by a collector on node 1, client 3 and 4 by another on node 5.

4 The Test Results

We have decided to use a network simulator to test our algorithm. We have chosen the freely available NS-2 [11], that provides substantial support for simulation of TCP, UDP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

In the simulator we implemented the following model: the system consists of data managers, data collectors, and clients. Every client is connected to exactly one DC, and every DC is connected to exactly one DM. When started, the collector using a TCP connection requests data from the DM it is connected to, and fills up its own buffer. When the buffer is full, it starts to send UDP packets to the clients periodically. When the transmission ends, the result is stored in a log file, and using this log we can analyse the data.

The test session contained 10 runs, each with a new network topology that was created randomly using the GT-ITM network topology generator [12]. GT-ITM created a network of 500 nodes in a transit-stub graph, where the link latencies were also chosen by the generator. Every link had 0.5Mb/sec bandwidth. In each run we had randomly chosen a node for the DM and 20 other nodes for clients. Each run consisted of eight turns; in the first seven the ams-greedy was used with the w_{dm}/w_{cl} ratio ranging from 1/8 to 8. In the eighth turn 20 (the same number as for the clients) random nodes had been chosen for the DCs, and each client had been assigned to the DC closest to it.

In each turn the data manager sent a 1 Mbyte video to the clients via the data collectors. When the buffer of the collector had at most 250 kbyte, or the bitrate of the TCP packets from the DM reached 64 kbit/sec, the DC had started to stream at 64 kbit/sec to the clients assigned to it.

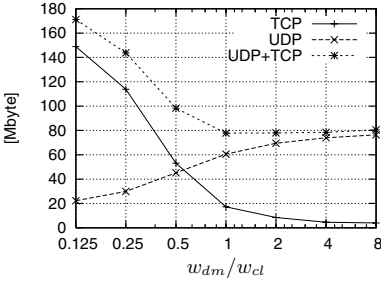
We measured the number of collectors chosen by the greedy algorithm (table 2), the number of data bytes carried by TCP packets, by UDP packets and by

both in the network (fig. 2(a)), and we also measured the standard deviation and the maximum of the delay jitter on the clients (fig. 2(b)). The figures present the average of the result values of turns with the same ratio. In table 2 the improvement of ams-greedy compared to the random algorithm can be seen in percents. Values less than zero mean deterioration.

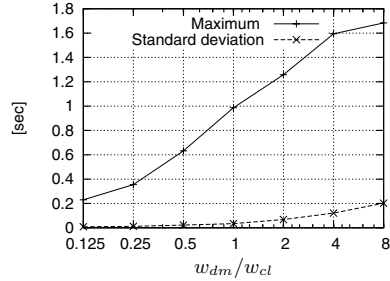
Comparing the ams-greedy algorithm to the random one we can see that when the number of active DCs are near equal (about 6.1) there is a significant improvement both in the jitter and the number of packets used.

Table 2. The average number of active data collectors and the improvement of jitter and network load

w_{dm}/w_{cl}	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	1	2	4	8
No. of DCs (greedy)	17.1	14.1	8.5	4.3	2.5	1.5	1.2
No. of DCs (random)	6.1						
jitter (std dev) [%]	89.42	83.33	70.9	54.69	9.39	-60.27	-167.75
jitter (max) [%]	83.90	75.22	55.79	31.07	12.19	-11.34	-17.59
tcp+udp packets [%]	-23.1	-3.38	29.31	44.05	43.9	43.55	42.11



(a) Cumulated number of data bytes in different kinds of packets



(b) Delay jitter

Fig. 2. Measurement results

The w_{dm}/w_{cl} ratio can be interpreted as a force that moves the DCs towards the data managers (when the ratio is more than one) or towards the clients (when it is less than one). The average number of the DCs as a function of the ratio can be seen on table 2: ams-greedy can place more DCs near the clients than around a DM. The maximum and the deviation of the delay jitter is getting smaller as we get closer to the clients: the route of the UDP streams is shorter, the delay is smaller, and thus the jitter is decreasing.

The most important result is the amount of the network packets (network load) in the function of the ratio. As the ratio decreases, the number of TCP packets increases, because the collectors are not only getting closer to the clients, but we have also placed more of them. The inverse function can be seen in the case of UDP packets, although the change is less drastical, because the number of packets is only effected by the length of the DC-client connections.

Analysing the total number of packets in the network, we can assume that the optimum is about where the ratio equals one. Getting the DCs closer to the clients the growth of the network load is fast, but moving the DCs to the DM the increase is very moderate. It implies, that if the network load should be low, the ratio has to remain near one, i.e. we have to weight the DC-DM and the DC-client latencies similarly.

These result show, that the fine-tuning of the parameters of the algorithm is necessary, and it can lead to significantly reduced network load and delay jitter. Further examinations are needed, however, to provide a more relevant set of network properties that can make the algorithm more effective.

5 Conclusion and Future Plans

In this paper we introduced a greedy-variant algorithm for the host recommender of the Adaptive Multimedia Server. With a simple simulation model we proved that proper parametrization of the algorithm is very important, it may result in such a placement of the data collectors that highly reduces network load and delay jitter.

In the future we plan to improve both the algorithm and the model based on the above results. First, the model should be tested with more than one DMs, and we have to introduce network noise, so that the algorithm could be tested in a more realistic network environment. The model should be made ready to handle multicast RTP streaming, and then its results should be compared to those we gathered by unicast streaming. Later the model ought to be tested also in a real network environment.

Second, the algorithm has to be improved to support not only latency, but other network link parameters as well, like jitter, bandwidth, or even statistical values like average and maximum jitter, average bandwidth, etc. The algorithm should also be improved to take startup latencies into account. Later on it could support even the migration of the data managers.

Acknowledgements. The work related to this paper has been performed in the IKTA-00026/2001, Dynamic broker service for improving quality of database access and resource usage project, which is supported by the Hungarian Ministry of Education in the tender for Information and Communication Technology Applications.

The work was partially supported by the European Community under the IST programme – Future and Emerging Technologies, contract IST-1999-14191 – EASYCOMP.

The work was also partially supported by the Research and Development Division of the Ministry of Education, Hungary and its Austrian partner the Bundesministerium für Auswärtige Angelegenheiten.

References

1. Goldschmidt, B., Tusch, R., Böszörményi, L.: A mobile agent-based infrastructure for an adaptive multimedia server. In: 4th DAPSYS (Austrian-Hungarian Workshop on Distributed and Parallel Systems), Kluwer Academic Publishers (2002) 141–148
2. Böszörményi, L., Döller, M., Hellwagner, H., Kosch, H., Libsie, M., Schojer, P.: Comprehensive treatment of adaptation in distributed multimedia systems in the admits project. In: Proceedings of the 10th ACM International Conference on Multimedia, ACM Press (2002)
3. Goldschmidt, B.: Vagabond2: an infrastructure for the adaptive multimedia server. Technical Report TR/ITEC/02/2.11, Institute of Information Technology University Klagenfurt (2002)
4. Qiu, L., Padmanabhan, V.N., Voelker, G.M.: On the placement of web server replicas. In: INFOCOM. (2001) 1587–1596
5. Barford, P., Cai, J.Y., Gast, J.: Cache placement methods based on client demand clustering. Technical Report TR1437, UW (2001)
6. Xu, J., Li, B., Lee, D.L.: Placement problems for transparent data replication proxy services. *IEEE Journal on Selected Areas in Communications* **20**, No. 7 (2002)
7. Mirchandani, P.B., Francis, R.L., eds.: *Discrete Location Theory*. John Wiley and Sons (1990)
8. Drezner, Z., ed.: *Facility Location: a Survey of Applications and Methods*. Springer Verlag (1996)
9. Chudak, F.A., Williamson, D.P.: Improved approximation algorithms for capacitated facility location problems. *Lecture Notes in Computer Science* **1610** (1999) 99–104
10. Guha, Khuller: Greedy strikes back: Improved facility location algorithms. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms). (1998)
11. <http://www.isi.edu/nsnam/ns>: The network simulator – ns-2 (2003)
12. Zegura, E.: <http://www.cc.gatech.edu/fac/ellen.zegura/graphs.html> (2003)

A New Distributed JVM for Cluster Computing

Marcelo Lobosco¹, Anderson Silva¹, Orlando Loques², and Claudio L. de Amorim¹

¹ Laboratório de Computação Paralela, PESC, COPPE, UFRJ
Bloco I-2000, Centro de Tecnologia, Cidade Universitária, Rio de Janeiro, Brazil
{ lobosco, faustino, amorim }@cos.ufrj.br
² Instituto de Computação, Universidade Federal Fluminense
Rua Passo da Pátria, 156, Bloco E, 3ª Andar, Boa Viagem, Niterói, Brazil
loques@ic.uff.br

Abstract. In this work, we introduce CoJVM, a new distributed Java run-time system that enables concurrent Java programs to efficiently execute on clusters of personal computers or workstations. CoJVM implements Java's shared memory model by enabling multiple standard JVMs to work cooperatively and transparently to support a single distributed shared-memory across the cluster's nodes. CoJVM requires no change to applications written in standard Java. Our experimental results using several Java benchmarks show that CoJVM performance is considerable with speed-ups ranging from 6.1 to 7.8 for an 8-node cluster.

1 Introduction

One of the most interesting features of Java [1] is its embedded support for concurrent programming. Java provides a native parallel programming model that includes support for multithreading and defines a common memory area, called the heap, which is shared among all threads that the program creates. To treat race conditions during concurrent accesses to the shared memory, Java offers to the programmer a set of synchronization primitives, which are based on an adaptation of the classic monitor model. The development of parallel applications using Java's concurrency model is restricted to shared-memory computers, which are often expensive and do not scale easily. A compromise solution is the use of clusters of personal computers or workstations. In this case, the programmer has to ignore Java's support for concurrent programming and instead use a message-passing protocol to establish communication between threads. However, changing to message-passing programming is often less convenient and even more complex to code development and maintenance. To address this problem, new distributed Java environments have been proposed. In common, the basic idea is to extend the Java heap among the nodes of the cluster, using a distributed shared-memory approach. So far, only few proposals have been implemented, and even less are compliant with the Java Virtual Language Specification [2]. Yet, very few reported good performance results [9] and presented detailed performance analysis [17].

In this paper, we introduce the design and present performance results of a new Java environment for high-performance computing, which we called the **CoJVM**

(Cooperative Java Virtual Machine) [3]. CoJVM's main objective is to speed up Java applications executing on homogeneous computer clusters. CoJVM relies on two key features to improve application performance: 1) the HLRC software Distributed Shared Memory (DSM) protocol [4] and 2) a new instrumentation mechanism [3] to the Java Virtual Machine (JVM) that enables new latency-tolerance techniques to exploit the application run-time behavior. Most importantly, the syntax and the semantics of the Java language are preserved, allowing programmers to write applications in the same way they write concurrent programs for the single standard JVM. In this work, we evaluate CoJVM performance for five parallel applications: Matrix Multiplication (MM), SOR, LU, FFT, and Radix. The connected figures show that all benchmarks we tested achieved good speedups, which demonstrate CoJVM effectiveness. Our main contributions are: a) to show that CoJVM is an effective alternative to improve performance of parallel Java applications for cluster computing, b) to demonstrate that scalable performance of Java applications for clusters can be achieved without any syntax or semantic change in the language; and c) to present detailed performance analysis of CoJVM for five parallel benchmarks.

The remainder of this paper is organized as follows. Section 2 presents the HLRC software DSM system. Section 3 describes Java support for multithreading and synchronization, and the Java memory model. In section 4, we review some key design concepts of CoJVM. In section 5, we analyze performance results of five Java parallel applications executed under CoJVM. In section 6, we describe some related works. Finally, in section 7, we draw our conclusions and outline ongoing works.

2 Software DSM

Software DSM systems provide the shared memory abstraction on a cluster of physically distributed computers. This illusion is often achieved through the use of the virtual memory protection mechanism [5]. However, using the virtual memory mechanism has two main shortcomings: (a) occurrence of false sharing and fragmentation due to the use of the large virtual page as the unit of coherence, which leads to unnecessary communication traffic; and (b) high OS costs of treating page faults and crossing protection boundaries. Several relaxed memory models, such as LRC [6], have been proposed to alleviate false sharing. In LRC, shared pages are write-protected so that when a processor attempts to write to a shared page an interrupt will occur and a clean copy of the page, called the twin, is built and the page is released to write. In this way, modifications to the page, called *diffs*, can be obtained at any time by comparing current copy with its twin. LRC imposes to the programmer the use of two explicit synchronization primitives: acquire and release. In LRC, coherence messages are delayed until an acquire is performed by a processor. When an acquire operation is executed the acquirer receives from the last acquirer all the write-notices, which correspond to modifications made to the pages that the acquirer has not seen according to the happen-before-1 partial order [6]. HLRC introduced the concept of home node, in which each node is responsible for maintaining an up-to-date copy of its owned pages; then, the acquirer can request copies of modified pages from their home nodes. At release points, *diffs* are computed and sent to the page's home node, which reduces memory requirements in home-based DSM protocols and contributes to the scalability of the HLRC protocol.

3 Java

In Java, threads programming is simplified since it provides a parallel programming model that includes support for multithreading. The package `java.lang` offers the `Thread` class that supports methods to initiate, to execute, to stop, and to verify the state of a running thread. In addition, Java also includes a set of synchronization primitives and the standard semantics of Java allow the methods of a class to execute concurrently. The `synchronized` reserved word, when associated with methods, specifies that they can only execute in a mutual-exclusion mode. The JVM specifies the interaction model between threads and the main memory, by defining an abstract memory system, a set of memory operations, and a set of rules for these operations [2]. The main memory stores all program variables and is shared by the JVM threads. Each thread operates strictly on its local memory, so that variables have to be copied first from main memory to the thread's local memory before any computation can be carried out. Similarly, local results become accessible to other threads only after they are copied back to main memory. Variables are referred to as master or working copy depending on whether they are located in main or local memory, respectively. The copying between main and local memory, and vice-versa, adds a specific overhead to thread operation. The replication of variables in local memories introduces a potential memory coherence hazard since different threads can observe different values for the same variable. The JVM offers two synchronization primitives, called `monitorenter` and `monitorexit`, to enforce memory consistency. In brief, the model requires that upon a `monitorexit` operation, the running thread updates the master copies with corresponding working copy values that the thread has modified. After executing a `monitorenter` operation, a thread should either initialize its work copies or assign the master values to them. The only exceptions are variables declared as `volatile`, to which JVM imposes the sequential consistency model. The memory management model is transparent to the programmer and is implemented by the compiler, which automatically generates the code that transfers data values between main memory and thread local memory.

4 The Cooperative Java Virtual Machine

CoJVM [3] is a DSM implementation of the standard JVM and was designed to efficiently execute parallel Java programs in clusters. In CoJVM, the declaration and synchronization of objects follow the Java model, in which the main memory is shared among all threads running in the JVM [2]. Therefore in CoJVM all declared objects are implicitly and automatically allocated into the Java heap, which is implemented in the DSM space. Our DSM implementation uses the HLRC protocol for two main reasons. First, it tends to consume less memory and its scalable performance is competitive with that of homeless LRC implementations [4]. Second, the HLRC implementation [7] already supports the VIA [8], a high-performance user-level communication protocol. Surdeanu and Moldovan [9] have shown that the LRC model is compliant with the Java Memory Model for data-race-free programs. Although HLRC adopts the page as the unit of granularity, we are not bound to that specific unit.

CoJVM benefits from the fact that Java already provides synchronization primitives: `synchronized`, `wait`, `notify` and `notifyAll`. The programmer with the use of these primitives can easily define a barrier or other synchronization constructs, or invoke a native routine. CoJVM supports only Java standard features, and adds no extra synchronization primitive. Since the declaration and synchronization of objects in the DSM follow the Java model and no extra synchronization primitive is added in our environment, a standard concurrent application can run without any code change. Threads created by the application are automatically moved to a remote host, and data sharing among them are treated following the language specification in a transparent way. Currently, CoJVM allows one single thread per node due to HLRC's restriction but we plan to solve this problem soon.

5 Performance Evaluation

Our hardware platform consists of a cluster of eight 650 MHZ Pentium III PCs running Linux 2.2.14-5.0. Each processor has a 256 KB L2 cache and each node has 512 MB of main memory. Each node has a Giganet cLAN NIC connected to a Giganet cLAN 5300 switch. This switch uses a thin tree topology and has an aggregate throughput of 1.25 Gbps. The point-to-point bandwidth to send 32 KB is 101 MB/s and the latency to send 1 byte is 7.9 μ s.

Table 1. Sequential times of applications, in seconds

Program	Program Size	Seq. Time - JVM	Seq. Time - CoJVM	Overhead
MM	1000x1000	485.77	490.48	0.97 %
LU	2048x2048	2,345.88	2,308.62	0 %
Radix	8388608 Keys	24.27	24.26	0 %
FFT	4194304 Complex	227.21	228.21	0.44 %
	Doubles			
SOR	2500x2500	304.75	306.46	0.56 %

Table 2. Performance of sequential applications: JVM (first figure) X CoJVM (second figure)

Counter	SOR	LU	Radix	FFT
% Load/Store Inst	75.3 / 73.8	86.8 / 90.1	87.4 / 86.0	91.2 / 91.2
% Miss Rate	1.9 / 2.3	1.2 / 6.1	1.2 / 1.2	1.7 / 1.7
CPI	1.83 / 1.83	2.78 / 2.73	1.76 / 1.76	1.85 / 1.85

To quantify the overhead that CoJVM imposes due to modifications of the standard JVM, we ported four benchmarks from the SPLASH-2 benchmark suite [10]: SOR, LU, FFT and Radix, and developed one, MM. The sequential execution times for each application are shown in Table 1. Each application executed 5 times, and the average execution time is presented. The standard deviation for all applications is less than 0,01%. For all applications, the overheads are less than 1%. We also instrumented both machines with the performance counter library PCL [11], which collects at processor level, run-time events of applications executing on commercial microprocessors. Table 2 presents the CPI (cycles per instruction) of sequential executions of SOR, LU, Radix, and FFT on the Sun JDK 1.2.2 and on CoJVM (which

is based on Sun JDK 1.2.2 implementation). The table also presents the percentage of load/store instructions and data cache level 1 miss rates. Table 2 shows that the percentage of load/store instructions does not change for FFT, but increases for LU and decreases slightly for SOR and Radix. However, the increase in the number of memory accesses in LU does not have a negative impact on miss rate. Indeed, the miss rate for CoJVM is half of the miss for the standard JVM. CoJVM modifications to the standard JVM resulted in lower CPI rate for LU while for the other applications, both CPI were equal.

Table 3 shows application speedups. We can observe that MM achieved an almost linear speedup while the other applications attained good speedups. Next, we analyze application performance in detail. In particular, we will compare CoJVM scalable performance against that of HLRC running the C version of the same benchmarks.

Table 3. Speedups

Program	2 Nodes	4 Nodes	8 Nodes
MM	2.0	4.0	7.8
LU	2.0	3.8	7.0
Radix	1.8	3.5	6.3
FFT	1.8	3.5	6.1
SOR	1.8	3.3	6.8

MM is a coarse grain application that performs a multiplication operation of two square matrixes of size D . Since there is practically no synchronization between threads, MM achieved speedup of 7.8 on 8 nodes. MM spent 99.2% of its execution time doing useful computation and 0.7% waiting for remote pages on page misses.

LU is a single-writer application with coarse-grain access that performs blocked LU factorization of a dense matrix. LU sent a total of 83,826 data messages and 164,923 control messages. Data message is related to the information needed during the execution of the application, such as pages that are transmitted during the execution are counted as data messages. Control is related to the information needed for the correct work of the software DSM protocol, such as page invalidations. Compared with the C version, Java sent 28 times more control messages and 20 times more data messages. LU required almost 1 MB/s of bandwidth per CPU and achieved speedup of 7 on 8 nodes. Two components contribute to LU' slowdown: page and barrier. Page access time increased approximately 23 times, when we compare the execution on 8 nodes with the execution on 2 nodes. Page misses contributed to 3.6% of the total execution time. In the C version of LU, page misses contributed to 2.7% of the execution time. This happened because the number of page misses in Java is 19 times that of C version. Although barrier time increased less than page miss time (11%), it corresponded to 9.6% of the total execution time, against that of 6.4% in C.

Radix is a multiple-writer application with coarse-grain access that implements an integer sort algorithm. It sent a total of 1,883 data messages and 2,941 control messages. Compared with the C version, Java sent 1.9 times less control messages and 2.9 times less data message. Radix required 1.9 MB/s of bandwidth per CPU and achieved a good speedup of 6.3 on 8 nodes. Three components contributed to slow down this benchmark: page, barrier and handler. Page misses contributed to 9% of the total execution time, barrier to 6.5% and handler to 4.2%. In the C version of the algorithm, page misses contributed to 8% of the execution, handler to 12.3% and

barrier to 45%. We observed that in C implementation there were 4.7 times more *diffs* created and applied than that of the Java version. The smaller number of *diffs* also had impact on the total volume of bytes transferred and on the miss rate of cache level 1. The C version transferred almost 2 times more bytes than that of Java, and its cache level 1 miss rate was almost 10 times higher than that of Java.

FFT implements the Fast Fourier Transform algorithm. Communication occurs in transpose steps, which require all-to-all thread communication. FFT is a single-writer application with fine-grained access. It sent a total of 68 MB of data, which is 4.7 times more than that of the C version. FFT required 1.8 MB/s of bandwidth per CPU and achieved speedup of 6.1 on 8 nodes. Page misses and barrier contributed to slow down this application. Page misses contributed to 15.8% of the total execution time, while barrier contributed with almost 10.7%. In the C version of the algorithm, page misses contributed to 18% of the execution and barrier to 3.8%. The miss rate on the level 1 cache is almost 7 times higher in C than in Java.

Table 4. HLRC statistics on 8 nodes: CoJVM (first figure) versus C (second figure)

Statistic	SOR	LU	Radix	FFT
Page faults	21,652 / 1,056	172,101 / 4,348	3,818 / 5,169	20,110 / 9,481
Page misses	690 / 532	81,976 / 4,135	1,842 / 958	15,263 / 3,591
Lock acquired	4 / 0	39 / 0	34 / 24	40 / 0
Lock misses	4 / 0	6 / 0	1 / 24	11 / 0
Diff created	20,654 / 0	79,338 / 0	946 / 4,529	1840 / 0
Diff applied	20,654 / 0	79,337 / 0	946 / 4,529	1840 / 0
Barrier	150 / 150	257 / 257	11 / 11	6 / 6

SOR uses the red-black successive over-relaxation method for solving partial differential equations. Communication occurs across the boundary rows between bands and is synchronized with barriers. This explains why the barrier time was responsible for 13% of execution. In the C version of SOR, barrier time was responsible for just 2.5% of the execution time. The difference between Java and C versions is due to an imbalance in computation caused by the high amount of *diffs* that Java creates. Because of its optimizations, the C version did not create *diffs*. Java created more than 20,000 *diffs*, which caused an imbalance due the deliver of write notices at the barrier. *Diff* is also responsible for 94% of the total data traffic. Data traffic in Java is 21 times more than in the C version. SOR required 1 MB/s of bandwidth per CPU and achieved speedup of 6.8 on 8 nodes. We can observe in Table 3 that the speedup obtained from 2 and 4 nodes are worse than those obtained with 4 and 8 nodes, respectively. Further investigation revealed that the larger caches were responsible for improving speedups.

The above results show that two components are the main responsible for slowing down the benchmarks: page miss and barrier time. The barrier synchronization time is mostly caused by an imbalance in execution times between processors. This imbalance seems to stem mainly from inadequate distribution of pages among nodes, although false sharing could also affect performance. This imbalanced distribution of pages among home nodes occurs because usually one thread is responsible for the initialization of internal objects and fields used during computation. Indeed, the JVM Specification establishes that all fields of an object must be initialized with their initial or default values during the object's instantiation. In our implementation,

however, whenever a thread writes to a page for the first time the thread's node becomes the page's home for the entire execution, according to the "first touch" rule of the HLRC protocol. This explains the imbalance we observed. One immediate solution is to perform a distributed initialization, which may not be so simple. Actually, we are studying techniques to fix this problem in a transparent fashion. The imbalance in the distribution of home nodes impacts also page faults, page misses, and the creation of *diffs* - and consequently, the total amount of control and data messages that CoJVM transfers. This is evident when we compare Java against the C implementation of the same algorithm (see Table 4). SOR, LU and FFT did not create *diffs* in C, while Java created *diffs* in large amount. Radix is the only exception: CoJVM created 4.7 times less *diffs* than C. Finally, we verify the small impact of the internal CoJVM synchronization on application performance. In SOR, this overhead was equal to 2.4 ms, i.e., the cost of 4 lock misses. Lock misses occur when the thread needs a lock (due to a monitor enter operation), but the lock must be acquired remotely. FFT presented the highest overhead, with 11 locks misses. Surprisingly, although in Radix CoJVM requested more locks than that of C version, in CoJVM just 1 lock resulted in overhead, against 24 locks in C. However, in Radix lock requests did not have any impact on performance.

6 Related Work

In this section we describe some distributed Java systems with implementations based on software DSM. A detailed survey on Java for high performance computing, including systems that adopt message-passing approaches, can be found in [12].

Java/DSM [13] was the first proposal of shared-memory abstraction on top of a heterogeneous network of workstations. In Java/DSM, the heap is allocated in the shared memory area, which is created with the use of TreadMarks [14], a homeless LRC protocol, and classes read by the JVM are allocated automatically into the shared memory. In this regard, CoJVM adopts a similar to approach, but using a home-based protocol. Java/DSM seems to be discontinued, and did not report any experimental result. cJVM [15] supports the idea of single system image (SSI) using the proxy design pattern [16], in contrast to our approach that adopts a software DSM protocol. In cJVM a new object is always created in the node where the request was executed first. Every object has one master copy that is located in the node where the object is created; objects from the other nodes that access this object use a proxy. If an object is heavily accessed, the node where the master copy is located becomes potentially a bottleneck. DISK [9] adopts an update-based, object-based, multiple-writer memory consistency protocol for a distributed JVM. CoJVM differs from DISK in two aspects: a) CoJVM adopts an invalidate-based approach, and b) we currently adopt a page-based approach, although our implementation is sufficient flexible to adopt an object-based or even a word-based approach. DISK detects which objects must be shared by the protocol and uses this information to reduce consistency overheads. DISK presents the speedups of two benchmarks, however without analyzing the results. JESSICA [18] adopts a home-based, object-based, invalidation-based, multiple-writer memory consistency protocol for a distributed JVM. In JESSICA, the node that started the application, called console node, performs all the synchronization operations, which impose a severe performance degradation: for

SOR, synchronization is responsible for 68% of the execution time, and the application achieves a speedup of 3.4 on 8 nodes [18]. In CoJVM the lock ownership is distributed equally among all the nodes participating of the computation, which contributed to the better performance achieved by our environment. The modifications made by Jessica in the original JVM impose a great performance slowdown in sequential application: for SOR, this slowdown is equal to 131%, while CoJVM almost do not impose any overhead.

7 Conclusion and Ongoing Works

In this work, we introduced and evaluated CoJVM, a cooperative JVM that addresses in a novel way several performance aspects related to the implementation of DSM in Java. CoJVM complies with the Java language specification while supporting the shared memory abstraction as implemented by our customized version of HLRC, a home-based software DSM protocol. Moreover, CoJVM uses VIA as its communication protocol aiming to improve Java application performance even further.

Using several benchmarks we showed that CoJVM achieved speedups, ranging from 6.1 to 7.8 on 8 nodes. However, we believe that CoJVM can further improve application speedups. In particular, we noticed that a shared data distribution imbalance can significantly impact barrier times, page faults, page misses and the creation of *diffs* – and consequently, the total amount of control and data messages that CoJVM transfers unnecessarily. We are studying new solutions to overcome this imbalance while refining CoJVM to take advantage of the application behavior, extracted from the JVM during run-time, in order to reduce the overheads of the coherence protocol. More specifically, a specialized run-time JVM machinery is being developed to create *diffs* dynamically, to allow the use of smaller units of coherence, and to detect automatically reads and writes to the shared memory, without using the time-expensive virtual-memory protection mechanism.

References

1. Arnold, K; Gosling, J. The Java Programming Language. Addison-Wesley, 1996.
2. Lindholm, T, Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1999.
3. Lobosco, M, Amorim, C, Loques, O. A Java Environment for High-Performance Computing. 13th Symp. on Computer Architecture and High-Performance Computing, Sep 2001.
4. Zhou, Y, et alli. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. OSDI, Oct 1996.
5. Li, K, Hudak, P. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321–359, Nov 1989.
6. Keleher, P, Cox, A, Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. Int. Symp. on Computer Architecture, pp. 13–21, May 1992.
7. Rangarajan M, Iftode L. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. ALS, Oct 2000.
8. VIA. *VIA Specification, Version 1.0*. <http://www.viarch.org>. Accessed on Jan, 29.

9. Surdeanu, M, Moldovan, D. Design and Performance Analysis of a Distributed Java Virtual Machine. *IEEE Trans. on Parallel and Distributed Systems*, Vol. 13, No. 6, Jun 2002.
10. Woo, S, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Int. Symp. on Computer Architecture*, pp. 24–36, Jun 1995.
11. Performance Counter Library. <http://www.fz-juelich.de/zam/PCL/>. Accessed on January, 29
12. Lobosco, M, Amorim, C, Loques, O. Java for High-Performance Network-Based Computing: a Survey. *Conc. and Computation: Practice and Experience: 2002(14)*, pp 1–31.
13. Yu, W.; Cox, A. Java/DSM: a Platform for Heterogeneous Computing. *ACM Workshop on Java for Science and Engineering Computation*, Jun 1997.
14. Keleher, P, et alli. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Winter Usenix Conference*, pp.115–131, Jan 1994.
15. Aridor, Y, Factor, M, Teperman, A. cJVM: a Single System Image of a JVM on a Cluster. *Int. Conf. on Parallel Processing*, Sep 1999.
16. Gamma, E, et alli. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
17. Fang, W, et alli. Efficient Global Object Space Support for Distributed JVM on Cluster. *Int. Conf. on Parallel Processing*, Aug 2002.
18. Ma, M, Wang, C, Lau, F. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing* (60), pp. 1194–1222, 2000.

An Extension of BSDL for Multimedia Bitstream Syntax Description*

Sylvain Devillers

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
Sylvain.Devillers@imec.be

Abstract. In previous works, a generic framework for multimedia content adaptation has been introduced, where XML is used to describe the high-level structure of a bitstream and the resulting description is first transformed by an XSLT style sheet, and then processed to generate an adapted bitstream. In order to provide full interoperability, a new language named Bitstream Syntax Description Language (BSDL) is built on top of W3C XML Schema for the purpose of this generation process. A schema designed in this language and specific to a given coding format allows a generic processor to parse a description and generate the corresponding adapted bitstream. This paper describes an extension of BSDL to provide the new functionality corresponding to the reverse operation, i.e. allowing a generic software module to parse a bitstream conforming to a given coding format described by a schema, and generate the corresponding description. For this, BSDL introduces a number of language mechanisms on top of XML Schema. This paper details these language extensions and reviews the strengths and limits of this approach.

1 Introduction

The multiplication of new devices gaining access to the Internet makes crucial to be able to propose different versions of the same multimedia content (audio, video or still image) adapted to the client resources in terms of bandwidth, display or computing capabilities. On the other hand, the use of scalable media allows to retrieve different versions of a content from a single file by simple editions and hence saves the burden of generating and handling one file for each required version. However, since each multimedia coding format defines its own data structure, a dedicated software module is required for each offered format in order to properly edit the bitstream and produce the adapted version.

To solve this issue, an original and flexible framework was proposed in [1]. In this method, XML [2] is used to describe the high-level structure of a bitstream; the resulting XML document is called a bitstream description. This description is not meant to replace the original binary format, but acts as an additional layer, similar to metadata. In most cases, it does not describe the bitstream on a bit-per-bit basis, but

* This paper was written while the author was with Philips Digital Systems Laboratories, 51, rue Carnot – B.P. 301 92156 Suresnes Cedex, France.

rather addresses its high-level structure, e.g. how the bitstream is organized in layers or packets of data. Furthermore, the bitstream description is itself scalable, which means it may describe the bitstream at different syntactic layers, e.g. finer or coarser levels of detail, depending on the application. With such a description, it is then possible for a generic software to transform the XML document, for example with an XSLT style sheet [3], and then generate back an adapted bitstream. This method is directly inspired by the web publishing framework that has been developed to dynamically adapt XML content with XSLT style sheets.

It is important to note that while a coding format may be specified as a "flat" structure, i.e. as a sequence of binary symbols, these symbols may be grouped in a hierarchical manner in the description, since the inherent tree structure of an XML document has no impact on the bitstream generation process. Furthermore, the choice of tag names is fully transparent. The design of a bitstream schema for a given coding format is therefore not unique and is fully application dependent.

In order to provide full interoperability, it is then necessary that a processor that is not aware of the specific coding format can nevertheless be used to produce a bitstream from its description. For this, a new language based on W3C XML Schema [4] and called Bitstream Syntax Description Language (BSDL) was introduced in [5]. With this language, it is then possible to design specific bitstream schemas describing the syntax of a particular coding format. These schemas can then be used by a generic processor to parse a description and generate the corresponding bitstream.

A complementary method named gBSDL was also proposed for the case of highly constrained devices where the specific syntax schema is not available. A combined approach is currently being standardized within the MPEG-21 Digital Item Adaptation framework [6] and is described in [7].

This paper takes the BSDL approach a step further by building a new challenging functionality. In order to achieve a fully generic framework, it is necessary for a format-unaware, generic software module to be able to parse a bitstream based on the information provided by its schema and generate its description. In this respect, BSDL comes close to other formal syntax description languages such as ASN.1 [8] or Flavor [9], also known as Syntactical Description Language (SDL) in the context of MPEG-4 [10].

The section 2 presents a brief overview of BSDL, then section 3 details the language features specific to the bitstream parsing process. Lastly, the strengths and limitations of our approach are discussed in section 4.

2 BSDL Overview

This section briefly describes how the Bitstream Syntax Description Language (BSDL) introduced above is built on top of XML Schema.

The primary role of a schema is to validate an XML document, i.e. to check that it follows a set of constraints on its structure and datatypes. On top of this, BSDL adds a twofold, new functionality, which is to specify how to generate an adapted bitstream from its description and vice-versa. For this, a number of restrictions and extensions over XML Schema are required.

In the following, we name BSDtoBin Parser the generic software parsing the bitstream syntax description and generating the bitstream, and BintoBSD the software

performing the reverse operation. BSDtoBin uses a set of XML Schema components for which semantics may be assigned in the context of the bitstream generation. Some components should therefore be ignored because they have no meaning in this context, while other constructs are excluded.

Similarly, BintoBSD uses an additional set of extensions and restrictions. In order to remain compatible with XML Schema, these extensions are introduced as application-specific annotations, namely by using the `xsd:appinfo` schema component and adding attributes with non-schema namespace. In order to distinguish both levels of extensions and restrictions, we name BSDL-1 the set of language features corresponding to the bitstream generation process, and BSDL-2 the set required for the description generation, where BSDL-1 is a subset of BSDL-2. The sets of extensions are declared in two schemas respectively named BSDL-1.xsd and BSDL-2.xsd, the latter importing the previous one. The BSDL-1 specification was described in [5] and will not be repeated here. On the other hand, BSDL-2 has not been published yet and is the subject of this paper. The following section details the restrictions and extensions of BSDL-2 in terms of structural and datatypes aspects, as well as their semantics.

In this paper, the "xsd:" prefix is used as a convention to refer to the XML Schema namespace, while "bsdl:" refers to the BSDL extensions.

3 BSDL-2: Bitstream Parsing

3.1 Structural Aspects of BSDL-2

A schema or a DTD define a set of constraints on XML documents. In XML Schema, most constraints are static, i.e. they are fully specified by the schema. For example, the minimum and maximum number of occurrences of a particle (`xsd:minOccurs` and `xsd:maxOccurs`) can be specified as a constant only and it is not possible to define a number of occurrences depending on the value of some other elements or attributes of the instance. On the other hand, this feature is required for parsing a bitstream since the length of a given field or its number of occurrences may be specified by a parameter found upstream.

BSDL-2 thus needs to introduce new dynamic constraints on top of XML Schema, in particular to specify the number of occurrences of a particle or its conditional occurrence. While BSDL-1 declares types and attributes that carry specific semantics in the context of bitstream generation, but are still validated by XML Schema, these constraints are actually new language extensions. In order to remain fully compatible with XML Schema, these new constructs are added as annotations, namely as attributes with non-schema namespace. The three attributes introduced below are used to characterize a particle as an additional property.

Firstly, BSDL-2 introduces a new attribute named `bsdl:nOccurs` to specify a variable number of occurrences. It is declared in the BSDL namespace and is used to characterize a particle in the same way as `xsd:minOccurs` and `xsd:maxOccurs`. It contains an XPath [11] expression indicating the relevant value as explained below. While parsing a bitstream, BintoBSD progressively instantiates the output description in the form of a DOM tree [12]. At a given stage in the bitstream parsing process, any parameter found upstream has thus already been instantiated as an element node of

the DOM tree. XPath 1.0 expressions are then used to locate this element relatively to the current element or with an absolute addressing in the DOM tree.

Similarly to `bsdl:nOccurs`, BSDL-2 introduces an attribute named `bsdl:if` to specify the conditional occurrence of a particle. It contains an XPath expression that should be evaluated as a boolean value.

Note that since `bsdl:nOccurs` is ignored by XML Schema, the allowed range declared by `xsd:minOccurs` and `xsd:maxOccurs` should cope with the expected value. Similarly, if a `bsdl:if` is specified, then `xsd:minOccurs` should be set to zero in case the condition is evaluated as false.

Unlike previous attributes that specify constraints depending on data found upstream, a third attribute named `bsdl:ifNext` specifies a constraint on data found immediately downstream. It contains an hexadecimal string that should be compared to the next bytes in the stream. The particle is parsed if the sequences of bytes are identical. The syntax of `bsdl:ifNext` also allows an array of two hexadecimal strings separated by a dash character to specify a range of allowed values. This allows to test a number of bits other than multiple of eight. For example, the element `<xsd:element name="elt1" bsdl:ifNext="08-0F"/>` is to be parsed if the next four bits in the bitstream are set to zero and the fifth to one.

3.2 Datatypes Aspects of BSDL-2

BSDL-2 uses the same datatype model than BSDL-1, but with a set of additional constraints, in particular on types relating to sequence of bytes.

While for some XML Schema built-in datatypes such as `xsd:short` or `xsd:int`, the encoding length is known from the schema (here, respectively 2 and 4 bytes), the length of `xsd:hexBinary`, `xsd:base64` and `bsdl:byteRange` can only be determined from their lexical representation in the instance. It is therefore necessary to constrain their length in the schema so that BintoBSD knows how many bytes should be read. The `xsd:length` facet is used to constrain the first two types for this purpose. However, it cannot be used for `bsdl:byteRange` since this type is considered by XML Schema as an array of integers. The `xsd:length` facet then constrains the number of items and not the segment length.

Furthermore, a new mechanism is required to constrain the length when it is not constant. BSDL-2 introduces a new facet named `bsdl:length` and declared in the BSDL namespace, which allows to distinguish it from the XML Schema facet. Since XML Schema does not currently allow a user to add his/her own facets, it is added to the `xsd:restriction` component via the annotation mechanism, i.e. the `xsd:annotation/xsd:appinfo` combination, and processed by BintoBSD as a regular facet. It contains an XPath expression that should be evaluated as an integer against the instantiated DOM tree, and gives the length in bytes of the data segment.

The Document 1 gives a snippet of a Bitstream Syntax Schema for the JPEG2000 coding format showing the use of a variable value defining the length of the `jp2:MarkerData` element content.

Document 1. Example of JPEG2000 Bitstream Syntax Schema

```
<xsd:element name="SIZ">
  <xsd:complexType>
```

```

<xsd:sequence>
  <xsd:element ref="jp2:Marker"/>
  <xsd:element name="LMarker" type="xsd:unsignedShort"/>
  <xsd:element name="MarkerData">
    <xsd:simpleType>
      <xsd:restriction base="bsdl:byteRange">
        <xsd:annotation><xsd:appinfo>
          <bsdl:length value="../jp2:LMarker - 2"/>
        </xsd:appinfo></xsd:annotation>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

```

Lastly, several coding formats use start codes to delimit packets of data, which allows to efficiently locate different packets without having to decode their inner structure. In this case, the packet of data may be read as an `xsd:hexBinary`, `xsd:base64` or `bsdl:byteRange` following the application needs, but since the length is unknown, a mechanism is required to indicate that the bitstream should be read until a given start code is found. For this purpose, we introduce a new BSDL facet named `bsdl:startCode` which contains the given sequence of bytes written in hexadecimal format. The BintoBSD parser thus reads the bitstream until the given sequence of bytes is found.

4 Discussion

We have introduced in this paper a language for describing the syntax of a multimedia bitstream. Among the similar languages proposed in the literature, Flavor [9] is probably the most accomplished attempt. This section discusses the strengths and limits of our approach, in particular in comparison to Flavor.

4.1 BSDL and XML Schema

The main characteristic and challenge of our work is that we chose not to develop "yet another language", but rather to build it on top of an existing one. Furthermore, we chose to base it on a declarative language, the function of which is to define constraints on XML documents.

We are conscious that, by doing so, we are using XML Schema for a purpose which is not its initial object. However, several of the introduced features define dynamic constraints on XML documents and are thus still relevant to XML Schema. The `bsdl:if` attribute states a condition on the occurrence of a particle and `bsdl:nOccurs` specifies a variable number of occurrences. Lastly, the `bsdl:length` facet provides the same functionality than `xsd:length` but with a variable value. In our application, XML Schema and BSDL extensions are first used to validate the description. Then, in a second step, we assign specific semantics to the schema

components in the form of a processing model for the description and bitstream generation process, not detailed here. Some other constructs such as the `bsdl:ifNext` attribute or the `bsdl:startCode` facet are specific to the bitstream parsing process and irrelevant to XML Schema. However, in order to simplify the syntax, we chose to define the two different types of extensions in a single specification and namespace.

4.2 Use of BSDL for Bitstream Parsing

It should be noted that BSDL does not and cannot have the ambition to parse any bitstream at any level of detail. Most coding formats have been specified without the use of a formal language (except notably for MPEG-4 Systems written in Flavor/SDL) and do not follow any constraints. In particular, the main part of a bitstream - qualified hereafter as the "payload" - is usually the output of an encoding process such as entropy coding, wavelet or Discrete Cosine Transform. Such sequence of bits can only be decoded via an algorithm implemented in a programming language. It is obviously not the ambition of BSDL or Flavor to natively decode such data.

In order to handle specific encoding schemes, it is possible to extend the BSDL parser. For this, the user-specific datatype should be defined in a schema with an attribute (not described here) indicating that it is hard-coded. The type definition in the schema is used for validating the lexical representation in the description, but not by the BintoBSD parser. The encoding and decoding methods are then implemented following a provided interface and appended to the system classpath (for a Java implementation). While reading the schema, the BintoBSD parser then dynamically instantiates the corresponding class and runs the encoding or decoding method.

On the other hand, we believe that using a formal language to specify a bitstream syntax helps checking its consistency and separating the parsing and decoding processes. While designing a schema for the MPEG-4 Video Object Layer (VOL) format [13] and testing it, we found some inconsistency in the spelling of the mnemonics and some ambiguity on default values of some parameters. Such imprecision is prone to incorrect interpretations and discrepancies in software implementations. The syntax description of MPEG-4 visual part (without the semantics) represents around seventy pages of tables. It is thus a very difficult task, if not impossible, to exclude any error from the text. Writing directly the syntax in a formal language would allow to test and validate its consistency, and track the modifications changes during the specification process.

Furthermore, specifying a bitstream syntax usually consists in two different tasks, firstly in formatting the output of an encoding method as seen above, and secondly in organizing the parameters and packets of data as a stream of symbols. Even though the compactness of the global result is a strong requirement, it is vital to be able to efficiently parse the bitstream to access the different segments of data. Typically, for parallelizing some tasks, the parsing process should be achievable at low cost in order to distribute the data segments to be decoded to dedicated processors. For these reasons, using a formal language implicitly sets some constraints on the bitstream syntax and hence benefits the complexity of the parsing process.

Another main feature of our approach is that it does not only provide a grammar for parsing the bitstream, but also a persistent representation of the bitstream content in the form of an XML document - the description. In comparison, the principle of

Flavor is to generate C++ or Java code from the bitstream syntax description, which can be interfaced with the application. In this case, the memory representation of the bitstream content is a hierarchy of C++/Java objects and is thus language dependent. More recently, an extension named XFlavor [14] has been developed to provide an XML representation of the data. On the other hand, BSDL provides both a grammar to parse the bitstream - the schema - and a persistent representation of the data - the description. Having the data in XML format gives access to the extensive family of XML-based languages and tools (XSLT, XPath...). Furthermore, before generating the XML document, the BSDL parser first instantiates a DOM tree, which is a standard, language-independent memory representation of XML data and can thus be easily interfaced with the application.

Lastly, a limit of our approach is the inherent verbosity of XML and XML Schema. Several solutions have been proposed to solve this issue. In particular, MPEG-7 BiM [15] is a very efficient method for binarizing an XML document by using its schema. Since BSDL remains compatible with XML Schema, BiM can be used to binarize the bitstream description, which is particularly useful when descriptions grow large, notably for video. As for XML Schema, a number of commercial products now provide graphical interfaces to design schemas in a user-friendly way. Lastly, it should be reminded that BSDL was primarily introduced for the problem of content adaptation, and thus is meant to describe bitstreams mainly at a high syntactical level, which should limit the verbosity of the descriptions.

5 Conclusion

In this paper, we extend the scope of the Bitstream Syntax Description Language (BSDL), which was primarily introduced for the problem of content adaptation. By adding new language mechanisms and defining specific semantics in the context of bitstream parsing, it is possible for a generic processor to use the schema to parse a bitstream. With this respect, BSDL comes close to other description languages such as Flavor. It is interesting to note that both languages tend to the same functionality while BSDL is based on XML Schema and Flavor, though presented as a declarative language, on a C++ like syntax. By using XML Schema beyond its original purpose, we also demonstrate the expressive power of this language. Lastly, we introduce new dynamic constraints on XML documents.

Tests have been successfully performed on several multimedia coding formats, including JPEG2000 images, MPEG-4 Video Elementary Streams and MPEG-4 Visual Texture Coding. Other ongoing tests include MPEG-4 Audio AAC and JPEG.

Although a generic software cannot pretend to be as efficient as an ad-hoc, specific software, the BSDL method significantly facilitates the process of accessing and reading structured, binary data by providing a declarative approach to bitstream parsing.

Acknowledgement. Part of this work was funded by the OZONE research project (IST-2000-30026).

References

1. Amielh M. and Devillers S.: Multimedia Content Adaptation with XML, 8th International Conference on Multimedia Modeling MMM'2001 (Amsterdam, The Netherlands, November 5-7, 2001), Paper Proceedings pp. 127–145.
2. XML Extensible Markup Language 1.0 (Second Edition), W3C Recommendation, October 6, 2000.
3. XSL Transformations Version 1.0, W3C Recommendation, November 16, 1999.
4. XML Schema Part 0: Primer, Part 1: Structures and Part 2: Datatypes, W3C Recommendation, May 2, 2001.
5. Amielh M. and Devillers S.: Bitstream Syntax Description Language: Application of XML Schema to Multimedia Content Adaptation, 11th International World Wide Web Conference, WWW2002 (Honolulu, May 6-11, 2002), Alternate Paper Tracks.
6. ISO/IEC JTC 1/SC 29/WG 11/N5353 MPEG-21 Digital Item Adaptation CD, December 2002, Awaji Island, Japan.
7. Panis G., Hutter A., Heuer J., Hellwagner H., Kosch H., Timmerer C., Devillers S. and Amielh M.: Bitstream Syntax Description: A Tool for Multimedia Resource Adaptation within MPEG-21, Accepted to Signal Processing: Image Communication, special issue on Multimedia Adaptation.
8. ASN.1 (ISO Standards 8824 and 8825).
9. Eleftheriadis A.: Flavor: A Language for Media Representation, ACM Multimedia '97 Conference, (Seattle, WA, November 1997), Proceedings pp. 1–9.
10. ISO/IEC 14496-1:2001 Information Technology – Coding of Audio-Visual Objects – Part 1: Systems.
11. XML Path Language, W3C Recommendation, 16 November 1999.
12. Document Object Model (DOM) Level 2 Core Specification, Version 1.0, W3C Recommendation 13 November, 2000.
13. ISO/IEC 14496-2:2001 Information Technology – Coding of Audio-Visual Objects – Part 2: Visual.
14. Hong D. and Eleftheriadis A.: XFlavor: Bridging Bits and Objects in Media Representation, IEEE Int'l Conf. on Multimedia and Expo ICME (Lausanne, Switzerland, August 2002).
15. ISO/IEC 15938-1:2002 MPEG-7: Multimedia content description interface – Part 1 : Systems. See also <http://www.expway.tv> for BiM.

Fast Construction, Easy Configuration, and Flexible Management of a Cluster System

Ha Yoon Song, Han-gyoo Kim, and Kee Cheol Lee

College of Information and Computer Engineering, Hongik University, Seoul, Korea
{song, hkim, lee}@cs.hongik.ac.kr

Abstract. New requirements of cluster systems have arisen. In this paper, we introduce a new clustering scheme for the easy construction and maintenance of a server using the Java Management Extension. Our experiments show that our system and its related applications are easy to construct, expand, and manage with guaranteed performance.

1 Introduction

The internal structure of an application server tends to be a cluster system since it might be impossible for one node server to process huge amounts of services. Regarding the recent trends of non-academic applications, several additions to the traditional requirements exist: The server systems must be fast to construct, easy to expand and configure, and flexible to manage. Additionally, most service handlers of applications are being developed in Java based frameworks.

Coping with the current requirements, a JMX (Java Management Extensions) based cluster system is presented. Our experiences show that the cluster systems built in JMX meet all the traditional and new requirements. For example, two man-months were consumed for the construction of a server system, and its performance remained satisfactory as users were added. The rest of the paper is structured as follows. Section 2 will discuss the construction and configuration issues. Section 3 will show the management details, and section 4 will show the performance verifications of our cluster system. We will conclude this paper in the final section.

2 Cluster System Construction and Configuration

We used JMX [1][2] as a core middleware for our cluster system construction. Figure 1 shows the overall structure of the cluster system in the simplest configuration. There are two nodes based on EJB and JMX, and one Manager which manages the whole system. Alternative configurations can be constructed without any modification of cluster components.

The manageable object for JMX is an application program based on the JAVA technologies such as EJB (Enterprise Java Beans). Each application program will be mapped on an MBean, and each MBean will be registered on the

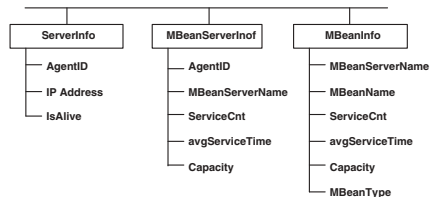
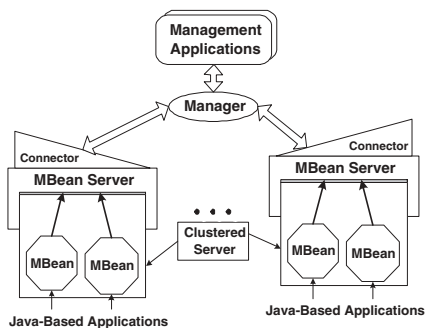


Fig. 1. Conceptual structure of a cluster. **Fig. 2.** Structure of management objects.

MBean server that resides on the same JAVA virtual machine to the applications'. The Monitor of this MBean server (one of the Agent Services) manages the applications and their status information. The Manager system can recognize and resolve the erroneous situation by the reception of events from Monitor whenever Monitor senses exceptional situations. The communication between Manager and Agents can be made by SNMP or JMX [3]. Alternatively, communication can be made by the connector server on Agent side and the connector client on the Manager system, which is used for the actual system implementation in this paper.

2.1 Managed Objects

Manager manages three sorts of objects: ServerInfo, MBeanServerInfo, and MBeanInfo. ServerInfo has the information of a node where Agent resides. MBeanServerInfo has the information of the MBean server created by the Agent. MBeanInfo stands for the management information of each MBean registered on the MBean server. Figure 2 shows the structure of the Managed object and their attributes.

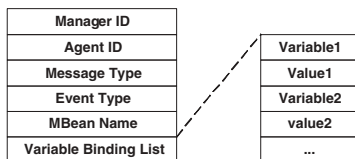


Fig. 3. Message formats.

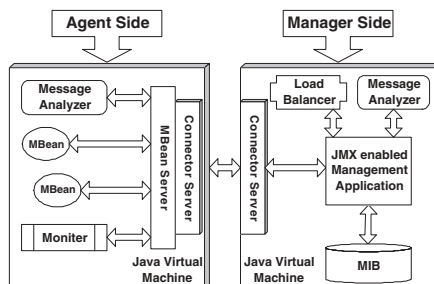


Fig. 4. Implementation details.

2.2 Message Formats and Types

Message format is designed to provide management communication between Manager and Agents as shown in figure 3. A message consists of *MessageID*, *AgentID*, *MessageType* with six subfields for six message types, *EventType* which can be thrown by MBean or MBean server, and *VariableBindingList* which contains MBean or MBean server information. Message types are *MBeanInfoRequest* from Manager to Agent, *MBeanInfoTransfer* from Agent replying Manager, *EventTransfer* from MBean or MBean server to Manager, *MBeanServerStatusRequest* from Manager to Agent, *MBeanServerStatusTransfer* from Agent, and *IsAlive* from Manager to check if a node is alive.

2.3 Implementation

The implemented testbed is composed of normal computers (nodes) without Manager and nodes with Manager over heterogeneous processors and operating systems. The Manager node has a connector client for the communication with Agent. The detailed component diagram is shown in figure 4.

Each application is mapped on an MBean and Agent manages it. The MBean server registers MBeans and Agent services. For this implementation, each node has one MBean server. Whenever each application starts a new service for a client, a new MBean will be created and registered to the MBean server. Among various Agent services, Monitor plays the key role to sense erroneous situations. Monitor checks the number of MBeans on the server and other related information specific to MBeans, and throws a situation-related event so that the Manager recognizes the error occurrence.

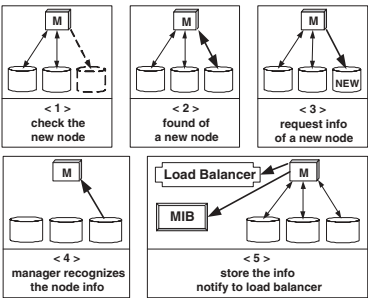


Fig. 5. Addition of a new node.

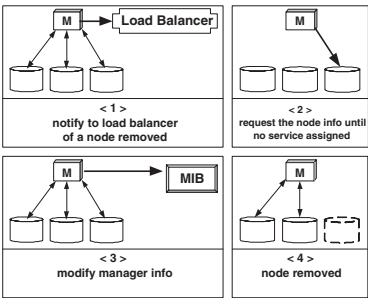


Fig. 6. Removal of an existing node.

3 Cluster Management

Our cluster system can easily cope with various situations of management. The most representative situations and solutions can be distinguished as follows.

- Addition of a new node: Figure 5 shows the detailed node addition procedure. An M inside a node box stands for Manager. Manager checks a new node addition, assigns a new Agent on the node and notifies to Load Balancer.
- Removal of an existing Node: Figure 6 shows the removal procedure. Manager notifies the removal of a node to Load Balancer and updates MIB after all the jobs on the node are finished.
- Management of an overloaded node (load balancing): Figure 7 shows the load balancing procedure. Monitor recognizes the overload of a node and reports it. Manager notifies to Load Balancer and requests ServiceCnt information to check if other nodes can process more jobs. Each node has its own avgServiceTime, and must check if it can reduce the avgServiceTime. Manager informs Load Balancer of the nodes whose ServiceCnt are less than their capacity.
- Management of a node failure: Figure 8 is for a node failure. Monitor's notification to Manager about a dead node starts the management procedure.

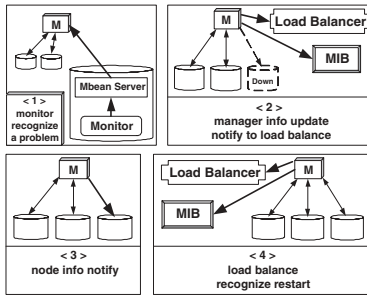


Fig. 7. Overloaded node processing

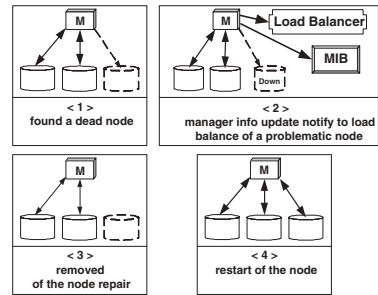


Fig. 8. Node failure processing

4 Load Balancing Experiments

Several exceptional situations are designed and experienced. The graphs of time versus the accumulated number of allotted jobs show the load balancing results.

Figure 9 is for the case of two nodes added and one node removed. The processing capacities of each node were purposely set to a small number. Two nodes were added between minutes 3 and 5. For the next 3 minutes, those two nodes started to be allotted but almost no jobs were allotted to the other nodes. At minute 10, one node was ordered to be removed, and no new jobs were allotted to it. After it was completely removed at minute 13, new job allotments to each node were delayed for 2 or 3 minutes because of limited node capacities and a small number of active nodes.

Figure 10 shows another scenario: three nodes got down, and recovered and added back to the cluster. Additionally two nodes were added at minutes 3 and

17, respectively, and they two started to be allotted 2 or 3 minutes later, limiting the allotments to the other nodes for the time being. Three nodes got down at minutes 9, 11, and 14, and added back to the cluster at minutes 10, 12, and 15, respectively. As a node got down, more jobs were allotted to the other nodes. As a node was newly added or added back, the job allotments to the other nodes were shown to be delayed.

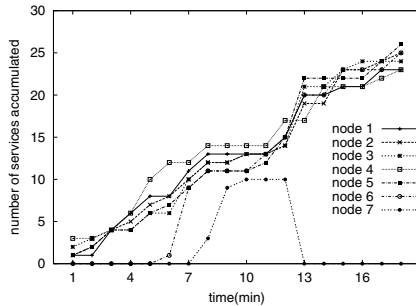


Fig. 9. Two nodes added and one re-moved.

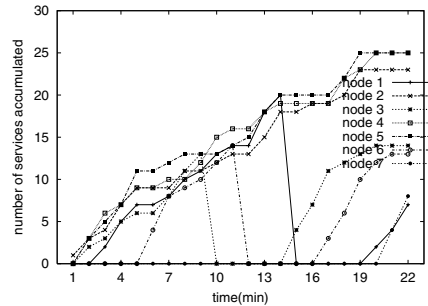


Fig. 10. 3 nodes down, recovered, and 2 added.

5 Conclusion and Future Works

In this paper we demonstrated the fast construction, the flexible configuration, and the easy management of a cluster system. In our system, the application-based management can be transparently separated from the server-based management. It took two man-months to construct our cluster system. Our system can also dynamically cope with the various situations of a cluster management for the high management ability and availability with reasonable performance.

Our cluster system has some JMX-related limitations. For example, one Agent cannot have multiple MBean servers. The next version of JMX might be improved so that more dedicated management can be available by multiple MBean servers on one Agent for better clustering managements.

References

1. H. Kreger. Java management extensions for application management. *IBM Systems Journal*, 40(1), 2001.
2. Sun-Microsystems. Java management extensions instrumentation and agent specifications, v1.1. 2002.
3. Sun-Microsystems. Java management extensions SNMP manager APIs. 1999.

Topic 17

Peer-to-Peer Computing

Luc Bougé, Franck Cappello, Omer Rana, and Bernard Traversat
Topic Chairs

This Topic is devoted to *High-Performance Computing* on widely distributed systems utilizing *Peer-to-Peer technologies*. In contrast with the usual Client-Server organization scheme, these systems are (or attempt at being) highly decentralized, self-organizing with respect to the external context, and with a notion of balanced resource trading. Hence, each node within such a system can provide both Client and Server capability. In the long term, participating nodes may expect to be granted as much resources as they provide.

Because of size, autonomy and high volatility of their resources, P2P platforms provide the opportunity for researchers to re-evaluate many fields of Distributed Computing, such as protocols, infrastructures, security, certification, fault tolerance, scheduling, performance, etc.

A number of 15 submissions have been received, 5 of which have been selected for regular presentation. They address various aspects of P2P systems: managing hierarchical P2P architectures, revisiting classical distributed computing techniques in a P2P context, instilling P2P techniques in Grid Computing, devising abstract models for the global behavior of P2P systems. One additional research note has also been selected because of its original insights in the context of P2P systems: it is concerned with combatting *freeriders*, users consuming resources without contributing in a commensurate way.

The overall appreciation of the Program Committee is that many submissions describe interesting ideas, but that these ideas are not sufficiently backed up by scientific evidence. In fact, the experimental evaluation is often weak, or sometimes even void, so that it is difficult to decide upon the actual impact of the idea on scientific grounds.

This aspect is obviously a major challenge, as evaluating the actual impact of such techniques requires managing and monitoring a large-scale configuration of volatile and faulty agents. Or, probably more realistically, to simulate such a configuration: the recent advent of large simulation equipments (e.g., the *Netbed/Emulab* Platform in the US <www.emulab.net>, or the *Grid Explorer Project* in France) opens new opportunities in this direction. The Program Committee would like to strongly encourage the authors to include a detailed evaluation of their ideas in their future papers, together with an extensive discussion of their expected impact.

Finally, we would like to thank the Euro-Par 2003 organizers for giving us the opportunity to launch this new Topic, and to all submitters for their contribution.

Hierarchical Peer-to-Peer Systems

L. Garcés-Erice¹, E.W. Biersack¹, P.A. Felber¹, K.W. Ross², and G. Urvoy-Keller¹

¹ Institut EURECOM, 06904 Sophia Antipolis, France
{garces|erbi|felber|urvoy}@eurecom.fr

² Polytechnic University, Brooklyn, NY 11201, USA
ross@poly.edu

Abstract. Structured peer-to-peer (P2P) lookup services organize peers into a flat overlay network and offer distributed hash table (DHT) functionality. Data is associated with keys and each peer is responsible for a subset of the keys. In hierarchical DHTs, peers are organized into groups, and each group has its autonomous intra-group overlay network and lookup service. Groups are organized in a top-level overlay network. To find a peer that is responsible for a key, the top-level overlay first determines the group responsible for the key; the responsible group then uses its intra-group overlay to determine the specific peer that is responsible for the key. We provide a general framework and a scalable hierarchical overlay management. We study a two-tier hierarchy using Chord for the top level. Our analysis shows that by using the most reliable peers in the top level, the hierarchical design significantly reduces the expected number of hops.

1 Introduction

Peer-to-peer (P2P) systems are gaining increased popularity, as they make it possible to harness the resources of large populations of networked computers in a cost-effective manner. A central problem of P2P systems is to assign and locate resources among peers. This task is achieved by a P2P *lookup service*. Several important proposals have been recently put forth for implementing distributed P2P lookup services, including Chord [1], CAN [2], Pastry [3] and Tapestry [4]. In these lookup services, each key for a data item is assigned to the live peer whose node identifier is “closest” to the key (according to some metric). The lookup service determines the peer that is responsible for a given key. The lookup service is implemented by organizing the peers in a structured overlay network, and routing a message through the overlay to the responsible peer. The efficiency of a lookup service is generally measured as a function of the number of peer hops needed to route a message to the responsible peer, as well as the size of the routing table maintained by each peer. For example, Chord requires $O(\log N)$ peer hops and $O(\log N)$ routing table entries when there are N peers in the overlay. Implementations of the distributed lookup service are often referred to as **Distributed Hash Tables (DHTs)**.

Chord, CAN, Pastry and Tapestry are all flat DHT designs without hierarchical routing. Each peer is indistinguishable from another in the sense that all peers use the same rules for determining the routes for lookup messages. This approach is strikingly different from routing in the Internet, which uses hierarchical routing. Hierarchical routing in the Internet offers several benefits over non-hierarchical routing, including scalability and administrative autonomy.

Inspired by hierarchical routing in the Internet, we examine two-tier DHTs in which (i) peers are organized in disjoint groups, and (ii) lookup messages are first routed to the destination group using an inter-group overlay, and then routed to the destination peer using an intra-group overlay.

We present a general framework for hierarchical DHTs. Each group maintains its own overlay network and intra-group lookup service. A top-level overlay is defined among the groups. Within each group, a subset of peers are labeled as “superpeers”. Superpeers, which are analogous to gateway routers in hierarchical IP networks, are used by the top-level overlay to route messages among groups. We consider designs for which peers in the same group are locally close. We describe a cooperative caching scheme that can significantly reduce average data transfer delays. Finally, we also provide a scalable algorithm for assigning peers to groups, identifying superpeers, and maintaining the overlays.

After presenting the general framework, we explore in detail a particular instantiation in which Chord is used for the top-level overlay. Using a novel analytical model, we analyze the expected number of peer hops that are required for a lookup in the hierarchical Chord instantiation. Our model explicitly captures inaccuracies in the routing tables due to peer failures.

The paper is organized as follows: We first discuss related work in Section 2. We then present the general framework for hierarchical DHT’s in Section 3. We discuss the particular case of a two-tier Chord instantiation in Section 4, and we quantify the improvement of lookup latency due to the hierarchical organization of the peers.

2 Related Work

P2P networks can be classified as being either unstructured or structured. Chord [1], CAN [2], Pastry [3], Tapestry [4], and P-Grid [5], which use highly structured overlays and use hashing for targeted data placement, are examples of structured P2P networks. These P2P networks are all flat designs (P-Grid uses a virtual distributed search tree only for routing purposes). Gnutella [6] and KaZaA [7], whose overlays grow organically and use random data placement, are examples of unstructured P2P networks. Ratnasamy et al. [8] explore using landmark nodes to bin peers into groups. The basic idea is for each peer to measure its round-trip time (RTT) to M landmarks, order the resulting RTTs, and then assign itself to one of $M!$ groups. Our hierarchical DHT schemes bear little resemblance to the scheme in [8]. Although in [8] the peers are organized in groups according to locality, the lookup algorithm applies only to CAN, does not use superpeers, and is not a multi-level hierarchical algorithm.

Our approach has been influenced by KaZaA, an enormously successful unstructured P2P file sharing service. KaZaA designates the more available and powerful peers as *supernodes*. In KaZaA, when a new peer wants to join, it bins itself with the existing supernodes, and establishes an overlay connection with the supernode that has the shortest RTT. The supernodes are connected through a top-level overlay network. A similar architecture has been proposed in CAP [9], a two-tier unstructured P2P network. Our design is a blend of the supernode/hierarchy/heterogeneity of KaZaA with the lookup services in the structured DHTs.

Brocade [10] proposes to organize the peers in a two-level overlay. All peers form a *single* overlay O_L . Supernodes are typically well connected and situated near network access points, forming another overlay O_H . Brocade is not truly hierarchical since *all* peers are part of O_L .

Finally, Castro et al. present in [11] a topology-aware version of Pastry [3]. At each hop Pastry presents multiple equivalent choices to route a request. By choosing the closest (smallest network delay) peer at each hop, they try to minimize network delay. However, at each step the possibilities decrease exponentially, so delay is mainly determined by the last hop, usually the longest. We propose large hops to first get to a group, and then shorter local hops inside the group, leading to a more natural caching scheme, as shown later in section 3.4.

3 Hierarchical Framework

We begin by presenting a general framework for a hierarchical DHT. Although we focus on a two-tier hierarchy, the framework can be extended to a general tier hierarchy. Let P denote the set of peers participating in the system. Each peer has a node id. Each peer also has an IP address (dynamic or not). The peers are interconnected through a network of links and switching equipment (routers, bridges, etc.) The peers send lookup query messages to each other using a hierarchical overlay network, as described below.

The peers are organized into groups (see group management in Section 3.3). The groups may or may not be such that the peers in the same group are topologically close to each other, depending on the application needs. Each group has a unique group id. Let I be the number of groups, G_i the peers in group i , and g_i the id for group i .

The groups are organized into a *top-level overlay network* defined by a directed graph (X, U) , where $X = \{g_1, \dots, g_I\}$ is the set of all the groups and U is a given set of virtual edges between the nodes (that is, groups) in X . The graph (X, U) is required to be connected, that is, between any two nodes g and g' in X there is a directed path from g to g' that uses the edges in U . It is important to note that this overlay network defines directed edges among groups and not among specific peers in the groups.

Each group is required to have one or more *superpeers*. Let $S_i \subseteq G_i$ be the set of superpeers in group i . Our architecture allows for $S_i = G_i$ for all $i = 1, \dots, I$, in which case all peers are superpeers. We refer to architectures for which all peers are superpeers as the *symmetric design*. Our architecture also allows $|S_i| = 1$ for all $i = 1, \dots, I$, in which case each group has exactly one superpeer. Let $R_i = G_i - S_i$ be the set of all “regular peers” in group g_i . For non-symmetric designs ($S_i \neq G_i$), an attempt is made to designate the more powerful peers as superpeers. By “more powerful,” we primarily mean the peers that are up and connected the most (and secondarily, those with high CPU power and/or network connection bandwidth). The superpeers are gateways between the groups: they are used for inter-group query propagation. To this end, we require that if s_i is a superpeer in G_i , and (g_i, g_j) is an edge in the top-level overlay network (X, U) , then s_i knows the name and the current IP address of at least one superpeer $s_j \in S_j$. With this knowledge, s_i can send query messages to s_j . If p is a regular peer, then p can only reach other groups through superpeers. Figure 1 (left) shows a top-level overlay network and possible communication relationships between the corresponding

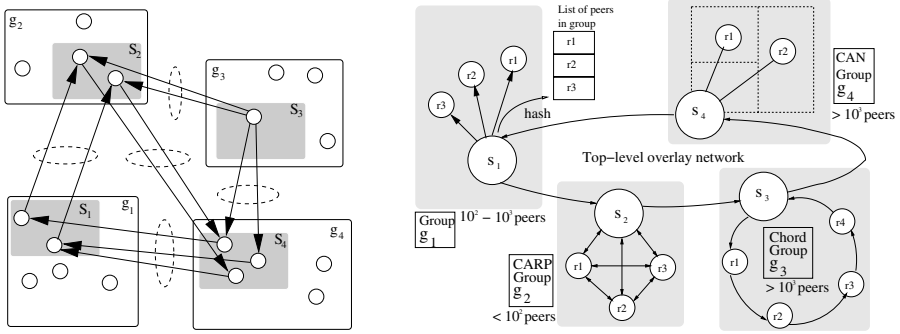


Fig. 1. Communication relationships between groups in the overlay network and superpeers in neighboring groups (left). On the right, a ring-like overlay network with a single superpeer per group. Intra-group lookup is implemented using different lookup services (CARP, Chord, CAN).

superpeers. Figure 1 (right) shows an example for which there is one superpeer in each group and the top-level overlay network is a ring. Within each group there is also an overlay network among the peers in the group.

3.1 Hierarchical Lookup Service

Consider a two-level lookup service. Given a key k , we say that group g_j is responsible for k if g_j is the “closest” group to k among all the groups. Here “closest” is defined by the specific top-level lookup service (e.g., Chord, CAN, Pastry, or Tapestry).

Our two-tier DHT operates as follows. Suppose a peer $p_i \in G_i$ wants to determine the peer that is responsible for a key k .

1. Peer p_i sends a query message to one of the superpeers in S_i .
2. Once the query reaches a superpeer, the top-level lookup service routes the query through (X, U) to the group G_j that is responsible for the key k . During this phase, the query only passes through superpeers, hopping from one group to the next. Eventually, the query message arrives at some superpeer $s_j \in G_j$.
3. Using the overlay network in group j , the superpeer s_j routes the query to the peer $p_j \in G_j$ that is responsible for the key k .

This approach can be generalized to an arbitrary number of levels. A request is first routed through the top-most overlay network to some superpeer at the next level below, which in turn routes the request through its “local” overlay network, and so on until the request finally reaches some peer node at the bottom-most level.

The hierarchical architecture has several important advantages when compared to the flat overlay networks.

- *Exploiting heterogeneous peers:* By designating as superpeers the peers that are “up” the most, the top-level overlay network will be more stable than the corresponding flat overlay network.
- *Transparency:* When a key is moved from one peer to another within a group, the search for the peer holding the key is completely transparent to the top-level

algorithm. Similarly, if a group changes its intra-group lookup algorithm, the change is completely transparent to the other groups and to the top-level lookup algorithm. Also, the failure of a regular peer $r_i \in G_i$ (or the appearance of a new peer) will be *local* to G_i ; routing tables in peers outside of G_i are not effected.

- *Faster lookup time*: Because the number of groups will be typically orders of magnitude smaller than the total number of peers, queries travel over fewer hops.
- *Less messages in the wide-area*: If the most stable peers form the top-level DHT, most overlay reconstruction messages happen inside groups, which gather peers that are topologically close. Less hops per lookup means also less messages exchanged. Finally, content caching inside groups can further reduce the number of messages that need to get out of the group.

3.2 Intra-group Lookup

The framework we just described is quite flexible, allowing for different independent intra-group overlays:

If a group has a small number of peers (say, in the tens), each peer could track all the other peers in its group (their ids and IP addresses); CARP [12] or consistent hashing [13] could be used to assign and locate keys within the group. The number of steps to perform such an intra-group lookup in the destination group is $O(1)$ (g_2 in Figure 1, right).

If the group is a little larger (say, in the hundreds), then the superpeers could track all the peers in the group. In this case, by forwarding a query to a local superpeer, a peer can do a local lookup in $O(1)$ steps (g_1 in Figure 1, right).

Finally, for larger groups, a DHT such as Chord, CAN, Pastry, or Tapestry can be used within the group (g_3 and g_4 in Figure 1, right side). A local lookup takes $O(\log M)$ hops, for M peers in the group.

3.3 Hierarchy and Group Management

We now briefly describe the protocols used to manage groups: consider peer p joining the hierarchical DHT. We assume that p is able to get the id g of the group it belongs to (e.g., g may correspond to the name of p 's ISP or university campus). First, p contacts and asks another peer p' already part of the P2P network to look up key g . Following the first step of the hierarchical lookup, p' locates and returns the IP address of the superpeer(s) of the responsible group. If the group id of the returned superpeer(s) is precisely g , then p joins the group using the regular join mechanisms of the underlying intra-group DHT; additionally, p notifies the superpeer(s) of its CPU and bandwidth resources. If the group id is not g , then a new group is created with id g and p as only (super)peer.

In a network with m superpeers per group, the first m peers to join a group g become the superpeers of that group. Because superpeers are expected to be the most stable nodes, we let superpeers monitor the peers that join a group and present “good” characteristics. Superpeers keep an ordered list of the superpeer candidates: the longer a peer remains connected and the higher its resources, the better a superpeer candidate it becomes. This list is sent periodically to the regular peers of the group. When a superpeer fails or disconnects, the first regular peer in the list becomes superpeer and joins the top-level

overlay. It informs all peers in its group, as well as the superpeers of the neighboring groups.

We are thus able to provide stability to the top-level overlay using multiple superpeers, promoting the most stable peers as superpeers, and rapidly repairing the infrequent failures or departures of superpeers.

3.4 Content Caching

In many P2P applications, once a peer p determines the peer p' that is responsible for a key, p then asks p' for the file associated with the key. If the path from p' to p traverses a congested or low-speed link, the file transfer delay will be long.

In many hierarchical DHT setups, we expect the peers in a same group to be topologically close and to be interconnected by high-speed links (corporate or university campus). By frequently confining file transfers to intra-group transfers, we reduce traffic loads on the access links between the groups and higher-tier ISPs.

Such hierarchical setups can be naturally extended to implement cooperative caching: when a peer $p \in G_i$ wants to obtain the file associated with some key k , it first uses group G_i 's intra-lookup algorithm to find the peer $p' \in G_i$ that would be responsible for k if G_i were the entire set of peers. If p' has a local copy of the file associated with k , it returns the file to p ; otherwise, p' obtains the file (using the hierarchical DHT), caches a copy, and forwards the file to p . Files are cached in the groups where they have been previously requested. Standard analytical techniques to quantify the reduction in average file transfer time and load on access links can be found in [14].

4 Chord Instantiation

For the remainder of this paper we focus on a specific top-level DHT, namely, Chord. In Chord, each peer and each key has a m -bit id. Ids are ordered on a circle modulo 2^m (see Figure 2, left). Key k is assigned to the first peer whose identifier is equal to or follows k in the identifier space. This peer is called the successor of key k . Each peer tracks its successor and predecessor peer in the ring. In addition, each peer tracks m other peers, called *fingers*; specifically, a peer with id p tracks all the successors of the ids $p + 2^{j-1}$ for each $j = 1, \dots, m$ (note that p 's first finger is in fact its successor). The successor, predecessor, and fingers make up the Chord routing table.

During a lookup, a peer forwards a query to the finger with the largest id that precedes the key value. The process is repeated from peer to peer until the peer preceding the key is reached, which is the "closest" peer to the key. When there are P peers, the average number of hops needed to reach the destination is $O(\log P)$ [1].

4.1 Inter-group Chord Ring

In the top-level overlay network, each "node" is actually a group of peers. This implies that the top-level lookup system must manage an overlay of groups, each of which is represented by a set of superpeers. Chord requires some adaptations to manage groups instead of nodes. We will refer to the modified version of Chord as "top-level Chord".

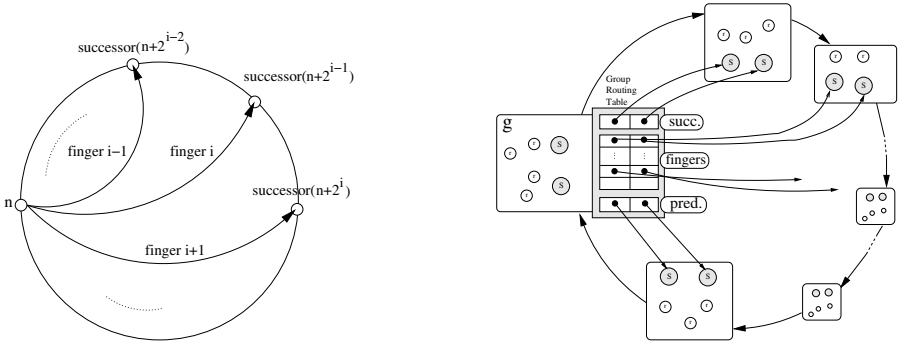


Fig. 2. Normal Chord routing (left) and hierarchical Chord routing (right).

Each node in top-level Chord has a predecessor and successor *vector*, holding the IP addresses of the superpeers of the predecessor and successor group in the ring, respectively. Each finger is also a vector. The routing table of a top-level Chord with two superpeers per group is shown in Figure 2 (right).

The population of groups in the top-level overlay network is expected to be rather stable. However, individual superpeers may fail and disconnect the top-level Chord ring. When the identity of the superpeers S_i of a group g_i changes, the new superpeers eagerly update the vectors of the predecessor and successor groups. This guarantees that each group has an up-to-date view of its neighboring groups and that the ring is never disconnected. Fingers improve the lookup performance, but are not necessary for successfully routing requests. We lazily update the finger tables when we detect that they contain invalid references (similarly to the lazy update of the fingers in regular Chord rings [1]). It is worth noticing that the regular Chord must perform a lookup operation to find a lost finger. Due to the redundancy that our multiple superpeer approach provides, we can choose without delay another superpeer in the finger vector for the same group.

To route a request to a group pointed to by a vector (successor or finger), we choose a random IP address from the vector and forward the request to that superpeer, thus balancing load among superpeers.

4.2 Lookup Latency with Hierarchical Chord

In this section, we quantify the improvement of lookup latency due to the hierarchical organization of the peers. To this end, we compare the lookup performance of the flat Chord and a two-tier hierarchical DHT in which Chord is used for the top level overlay, and arbitrary DHTs are used for the bottom level overlays. For each bottom level group, we only suppose that the peers in the group are topologically close so that intra-group lookup delays are negligible. In order to make a fair comparison, we suppose that both the flat and hierarchical DHTs have the same number of peers, denoted by P . Let I be the number of groups in the hierarchical design. Because peers are joining and leaving the ring, the finger entries in the peers will not all be accurate. This is more than probable, since fingers are updated lazily. To capture the heterogeneity of the peers, we suppose that there are two categories of peers: *Stable peers*, for which each peer is down with probability p_s . *Instable peers*, for which each peer is down with probability p_r , with

$p_r \gg p_s$. We suppose that the vast majority of the peers are instable peers. In real P2P networks, like Gnutella, most peers just remain connected the time of getting data from other peers [15]. For the hierarchical organization, we select superpeers from the set of stable peers, and we suppose there is at least one stable peer in each group. Because there are many more instable peers than stable peers, the probability that a randomly chosen Chord node is down in the flat DHT is approximately p_r . In the hierarchical system, as all the superpeers are stable peers, the probability that a Chord node is down is p_s .

To compare the lookup delay for flat and hierarchical DHTs, we thus only need to consider a Chord ring with N peers, with each peer having the same probability p of being down. The flat DHT corresponds to $(N, p) = (P, p_r)$ and the hierarchical DHT corresponds to $(N, p) = (I, p_s)$. We now proceed to analyze the lookup of the Chord ring (N, p) . To simplify the analysis, we assume the N peers are equally spaced on the ring, i.e., the distance between two adjacent peers is $\frac{2^m}{N}$. Our model implies that when a peer attempts to contact another in its finger table, the peer in the finger table will be down with probability p , except if this is the successor peer, for which we suppose that the finger entry is always correct (i.e., the successor is up or the peer is able to find the new successor. This assures the correct routing of lookup queries).

Given an initial peer and a randomly generated key, let the random variable H denote the number of Chord hops needed to reach the target peer, that is, to reach the peer responsible for the key. Let T be the random variable that is the clockwise distance in number of peers from the initial peer to the target peer. We want to compute the expectation $E[H]$. Clearly

$$E[H] = \sum_{n=0}^{N-1} P(T=n)E[H|T=n] = \frac{1}{N} \sum_{n=0}^{N-1} E[H|T=n] \quad (1)$$

From (1), it suffices to calculate $E[H|T=n]$ to compute $E[H]$. Let $h(n) = E[H|T=n]$. Note that $h(0) = 0$ and $h(1) = 1$. Let $j_n = \max\{j : 2^j \leq \frac{2^m n}{N}\}$. The value j_n represents the number of finger entries that precede the target peer, excluding finger 0, the successor. For each of the finger entries, the probability that the corresponding peer is up is p .

Starting at the initial peer, when hopping to the next peer, the query will advance $\lceil \frac{2^{j_n}}{2^m/N} \rceil$ peers if the j_n th finger peer is up; if this peer is down but the $(j_n - 1)$ th finger peer is up, the query will advance $\lceil \frac{2^{j_n-1}}{2^m/N} \rceil$; and so on. Let $q_n(i)$ denote the probability that the i th finger is used. We therefore have

$$h(n) = 1 + \sum_{i=0}^{j_n} q_n(i)h\left(n - \left\lceil \frac{2^i}{2^m/N} \right\rceil\right) \quad (2)$$

The probability that the i th finger is used is given by $q_n(i) = p^{j_n-i}(1-p)$ for $i = 1, \dots, j_n$, and by $q_n(0) = p^{j_n}$. Combining Equation 2 with the above expression for $q_n(i)$ we obtain

$$h(n) = 1 + p^{j_n}h(n-1) + (1-p) \sum_{i=1}^{j_n} p^{j_n-i}h\left(n - \left\lceil \frac{2^i}{2^m/N} \right\rceil\right) \quad (3)$$

Using this recursion, we can calculate all the $h(n)$'s beginning at $h(0) = 0$. We then use Equation 1 to obtain the expected number of hops, $E[H]$.

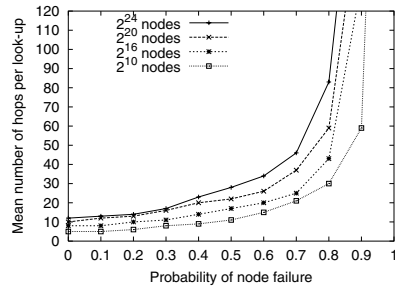


Fig. 3. Nb. of hops per lookup in Chord.

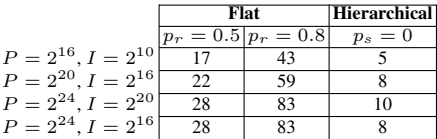


Fig. 4. Flat vs. hierarchical networks.

In Figure 3, we plot the expected number of hops in a lookup as a function of the availability of the peers in a Chord system, for different values of N . With smaller values of p , although peers in the ring are not totally reliable, we are still able to advance quite quickly on the ring. Indeed, while the best finger for a target peer is unavailable with probability p , the probability of the second best choice to be also down is p^2 , which is far smaller than p .

Despite the good scalability of the Chord lookup algorithm in a flat configuration, the hierarchical architecture can yet significantly decrease the lookup delay. Figure 4 gives the expected number of hops for the flat and hierarchical schemes, for different values of P , I , and p_r ($p_s = 0$). We suppose in all cases groups of $\frac{P}{I}$ peers. Since the number of steps is directly related to the lookup delay, we can conclude that the average lookup delay is severely improved in the hierarchical DHT.

5 Conclusion

Hierarchical organizations in general improve overall system scalability. In this paper, we have proposed a generic framework for the hierarchical organization of peer-to-peer overlay network, and we have demonstrated the various advantages it offers over a flat organization. A hierarchical design offers higher stability by using more “reliable” peers (superpeers) at the top levels. It can use various inter- and intra-group lookup algorithms simultaneously, and treats join/leave events and key migration as local events that affect a single group. By gathering peers into groups based on topological proximity, a hierarchical organization also generates less messages in the wide area and can significantly improve the lookup performance. Finally, our architecture is ideally suited for caching popular content in local groups. We have presented an instantiation of our hierarchical peer organization using Chord at the top level. The Chord lookup algorithm required only minor adaptations to deal with groups instead of individual peers. When all peers are available, a hierarchical organization reduces the length of the lookup path by a factor of $\frac{\log P}{\log I}$, for I groups and P peers. A hierarchical organization reduces the length of the lookup path dramatically when superpeers are far more stable than regular peers.

References

1. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM*, 2001.
2. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of SIGCOMM 2001*, Aug. 2001.
3. A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, (Heidelberg, Germany), pp. 329–350, November 2001.
4. B. Y. Zhao, J. Kubiawicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr 2001.
5. K. Aberer, "P-grid: A self-organizing access structure for P2P information systems," in *Proceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, (Trento, Italy), 2001.
6. "Gnutella." <http://gnutella.wego.com>.
7. "Kazaa." <http://www.kazaa.com>.
8. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *Proceedings of Infocom'02*, (New York City, NY), 2002.
9. B. Krishnamurthy, J. Wang, and Y. Xie, "Early measurements of a cluster-based architecture for P2P systems," in *ACM SIGCOMM Internet Measurement Workshop*, (San Francisco, CA), November 2001.
10. B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiawicz, "Brocade: Landmark routing on overlay networks," in *In Proceedings of IPTPS'02*, (Cambridge, MA), March 2002.
11. M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks," Tech. Rep. MSR-TR-2002-82, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 2002.
12. K. W. Ross, "Hash-routing for collections of shared web caches," *IEEE Network Magazine*, vol. 11, 7, pp. 37–44, Nov-Dec 1997.
13. D. Karger, A. Sherman, A. Berkhemier, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," in *Eighth International World Wide Web Conference*, May 1999.
14. J. F. Kurose and K. W. Ross, *Computer Networks: A Top-Down Approach Featuring the Internet*, 2nd edition. Addison Wesley, 2002.
15. E. Adar and B. A. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, Oct. 2000.

Enabling Peer-to-Peer Interactions for Scientific Applications on the Grid^{*}

Vincent Matossian and Manish Parashar

The Applied Software Systems Laboratory
Department of Electrical and Computer Engineering
Rutgers University, Piscataway NJ 08855, USA
{vincentm,parashar}@caip.rutgers.edu

Abstract. P2P and Grid communities are actively working on deploying and standardizing infrastructure, protocols, and mechanisms, to support decentralized interactions across distributed resources. Such an infrastructure will enable new classes of applications based on continuous, seamless and secure interactions, where the application components, Grid services, resources and data interact as peers. This paper presents the design, implementation and evaluation of a peer-to-peer messaging framework that builds on the JXTA protocols to support the interaction and associated messaging semantics required by these applications.

1 Introduction

The emergence of Grid applications coincides with that of Peer-to-Peer (P2P) applications such as Napster or SETI@HOME[1]. This parallel is manifested in the similarities in infrastructural requirements of Grid and P2P systems, such as the underlying decentralized (overlay) network architecture, the dynamic discovery of resources, the aggregation of distributed resources, or the need for system integrity and security guarantees. A key requirement for enabling the application-level interactions on the Grid is a P2P messaging layer that supports the interactions and their associated messaging semantics. The overall objective of this work is to prototype such a messaging framework to support Grid applications in a P2P environment. A number of messaging solutions for P2P/Grid systems have been proposed in recent years. These include Message Oriented Middleware (MOM) systems such as JMS [2], LeSubscribe [3], and IBM MQSeries[4], as well as Grid oriented systems such as GridRPC [5], NaradaBrokering [6], and ICENI[7]. This paper presents Pawn, a publisher/subscriber messaging substrate that offers interaction services for distributed object management, monitoring and steering, group formation, and collaboration through guaranteed, flexible, and stateful messaging. Pawn combines properties from messaging and P2P messaging on the Grid to provide advanced messaging semantics such as guaranteed message delivery, push, pull, transactions, and request/response interactions. It also supports synchronous and asynchronous communications, coordination through message ordering, and remote procedure calls. Unlike

^{*} Support for this work was provided by the NSF via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

other publisher/subscriber systems, Pawn focuses on interaction services to support application monitoring and steering, collaboration, and application execution on the Grid. This paper makes three contributions: (1) the definition of messaging requirements for scientific investigation in a P2P Grid environment, (2) the identification and implementation of corresponding services and mechanisms in Pawn, (3) the deployment of the Pawn messaging substrate and its evaluation using a “real-world” Grid application.

The rest of this paper is organized as follows. Section 2 presents the motivating application process and describes the underlying interactions. Section 3 presents the design and implementation of Pawn. Section 4 describes the use of Pawn to enable the interactions required by the target application. Section 5 presents an experimental evaluation of Pawn. Section 6 presents some conclusions.

2 Motivating Application: Enabling Autonomic Oil Reservoir Optimization

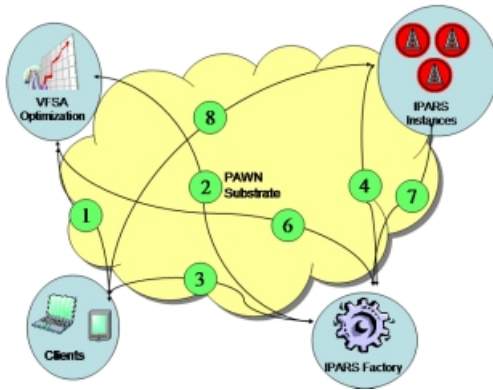


Fig. 1. Autonomous optimization in IPARS using VFSA

The research presented in this paper and the Pawn messaging substrate is motivated by the autonomic oil reservoir optimization process on the Grid. The goal of the process is to dynamically optimize the placement and configuration of oil wells to maximize revenue. The overall application scenario is illustrated in Figure 1. The peer components involved include: **Integrated Parallel Accurate Reservoir Simulator (IPARS)** [8], providing sophisticated simulation components that encapsulate complex mathematical models of the physical interaction in

the subsurface. **IPARS Factory**, responsible for configuring IPARS simulations, executing them on resources on the Grid and managing their execution. **Very Fast Simulated Annealing (VFSA)**, an optimization service based on statistical physics. **Economic Modeling Service** that uses IPARS simulation outputs and current market parameters (oil prices, costs, etc.) to compute estimated revenues for a particular reservoir configuration. **DISCOVER Middleware** that integrates Globus [9] Grid services (GSI, MDS, GRAM, and GASS), via the CorbaCoG [10], and DISCOVER remote monitoring, interactive steering, and collaboration services. **DISCOVER Collaborative Portals** providing experts (scientists, engineers) with collaborative access to the other peers components. These entities need to dynamically discover and interact with one another as peers to achieve the overall application objectives. Figure 1 shows the different interactions involved numbered 1 through 8. (1) The experts use the portals to interact with the DISCOVER middleware and the Globus Grid services to discover and allocate appropriate resource, and to deploy the IPARS Factory, VFSA and Economic model peers.

(2) The IPARS Factory discovers and interacts with the VFSA service peer to configure and initialize it. (3) The experts interact with the IPARS Factory and VFSA to define application configuration parameters. (4) The IPARS Factory then interacts with the DISCOVER middleware to discover and allocate resources and to configure and execute IPARS simulations. (5) The IPARS simulation now interacts with the Economic model to determine current revenues, and discovers and interacts with the VFSA service when it needs optimization. (6) VFSA provides IPARS Factory with optimized well information, which then (7) launches new IPARS simulations. (8) Experts can at anytime discover, collaboratively monitor and interactively steer IPARS simulations, configure the other services and drive the scientific discovery process. Once the optimal well parameters are determined, the IPARS Factory configures and deploys a production IPARS run.

3 Design and Implementation of Pawn

A conceptual overview of the Pawn framework is presented in Figure 2 and is composed of peers (computing, storage, or user peers), network and interaction services, and mechanisms. These components are layered to represent the requirements stack enabling interactions in a Grid environment. The figure can be read from bottom to top as “Peers compose messages handled by services through specific interaction modalities”.

Pawn builds on Project JXTA[11], a general-purpose peer-to-peer framework. JXTA concepts include peer, peergroup, advertisement, module, pipe, rendezvous, and security. JXTA defines protocols for : discovering peers (Peer Discovery Protocol, PDP), binding virtual end-to-end communication channels between peers (Pipe Binding Protocol, PBP), resolving queries (Peer Resolver Protocol, PRP), obtaining information on a particular peer, such as its available memory or CPU load (Peer Information Protocol,

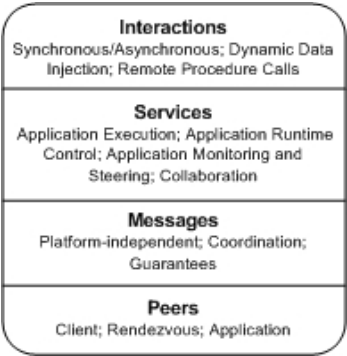


Fig. 2. Pawn requirements stack.

PIP), propagating messages in a peergroup (Rendezvous protocol, RVP), determining and routing from a source to a destination using available transmission protocols (Endpoint Routing Protocol, ERP). Protocols in JXTA define the format of the messages exchanged as well as the behavior adopted on receiving a message. In Pawn, peers can implement one or more services (behaviors); the combination of services implemented by a peer defines its role. Typical roles for a peer are client, application or rendezvous. A client peer deploys applications on resources and accesses them for interactive monitoring and/or steering. It also collaborates with other peers in the peergroup. An application peer exports its application interfaces and controls to a peergroup. These interfaces are used by other peers to interact with the application. The rendezvous peer distributes and relays messages, and filters them en route to their destination.

3.1 Pawn Services

A network service is a functionality that can be implemented by a peer and made available to a peergroup. File-sharing or printing are typical examples of network services. In Pawn, network services are application-centric and provide the mechanisms to query, respond, subscribe, or publish information to a peergroup. Pawn offers four key services to enable dynamic collaborations and autonomic interactions in scientific computing environments. These services are *Application Runtime Control*, *Application Monitoring and Steering*, *Application Execution*, and *Group Communication*.

Application Execution service [AEX]: The Application Execution service enables a peer to remotely start, stop, get the status of, or restart an application peer. This service requires a messaging mechanism supporting synchronous and guaranteed remote calls for resource allocation and application deployment (transaction oriented interaction).

Application Monitoring and Steering service [AMS]: The Application Monitoring and Steering service handles Request/Response interactions, application querying (i.e. PULL), and dynamic steering (i.e. PUSH) of application parameters. It requires support for synchronous and asynchronous communications, guaranteed message delivery, and dynamic data injection (e.g. push information to an application at runtime).

Application Runtime Control service [ARC]: The application runtime control service announces the existence of an application to the peergroup, sends application responses, publishes application update messages, and notifies the peergroup of an application termination. This service requires a mechanism to push information to the peergroup and respond to queries (i.e. PUSH and Request/Response interactions).

Collaboration Service [Group communication, Presence]: The collaboration service supports collaborative tools and group communications. Collaborating peers need to establish direct end-to-end communications through synchronous/asynchronous channels (e.g. for file transfer or text communication), and be able to publish information to the peergroup (Transaction and PULL interactions).

3.2 Implementation of Pawn Services

Pawn builds on the current Java implementation of the JXTA protocols. JXTA defines unicast pipes that provide a communication channel between two endpoints, and propagate pipes that can propagate a message to a peergroup. It also defines the Resolver Service that sends and receives messages in an asynchronous manner. Recipients of the message can be a specific peer or an entire peergroup. The pipe and resolver service use the available underlying transport protocol (TCP, HTTP, TLS). To realize the four services identified above, Pawn extends the pipe and resolver services to provide stateful and guaranteed messaging. This messaging is then used to enable the key application-level interactions such as synchronous/asynchronous communication, dynamic data injection, and remote procedure calls.

Stateful Messages: In Pawn, every message contains a source and destination identifier, a message type, a message identifier, a payload, and a handler tag. The handler tag uniquely identifies the service that will process the message. State is maintained through the payload that contains system/application parameters. These messages are defined in XML to provide platform-independence.

Message guarantees: Pawn implements application-level communication guarantees by combining stateful messages and a per-message acknowledgment table maintained at every peer. Guarantees are provided by using FIFO message queues to handle incoming and outgoing messages. Every outgoing message that expects a response is flagged in the table as awaiting acknowledgment. This flag is removed once the message is acknowledged. Messages contain a default timeout value representing an upper limit on the estimated response time. If an acknowledgement is not received and/or the timeout value expires, the message is resent, blocking or not blocking the current process depending on the communication type (synchronous or asynchronous). The message identifier is composed from the destination and sender's unique peer identifiers. It is incremented for every transaction during a session (interval between a peer joining and leaving a peer group) to enable the application-level message ordering guarantees.

Synchronous/Asynchronous communication: Communication in JXTA can be synchronous when using blocking pipes, or asynchronous when using non-blocking pipes or the resolver service. In order to provide reliable messaging, Pawn combines these communication modalities with stateful messaging and guarantee mechanism.

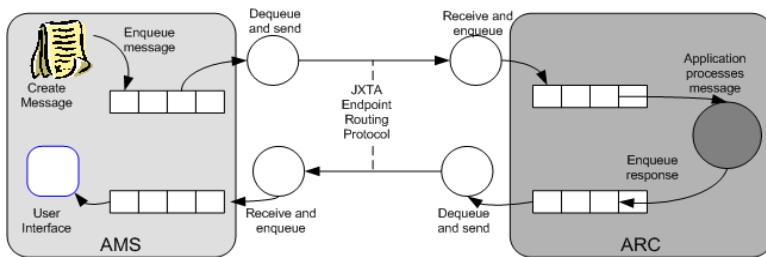


Fig. 3. AMS to ARC interaction

Dynamic Data Injection: In Pawn, every peer advertisement contains a pipe advertisement, which uniquely identifies a communication channel to it. This pipe is used by other peers to create an end-to-end channel to send and receive messages dynamically. Every interacting peer implements a message handler that listens for incoming messages on the peer's input pipe channel. The application-sensitive data contained in the message payload can be dynamically passed to the application/service identified by the handler tag field.

Remote Method Calls (PawnRPC): The PawnRPC mechanism provides the low-level constructs for building applications interactions across distributed peers. Using PawnRPC, a peer can invoke a method on a remote peer dynamically by passing its request as an XML message through a pipe. The interfaces of the methods that can be remotely invoked are published as part of the peer advertisement during peer discovery. The XML message is a composition of the destination address, the remote method name, the arguments of the method, and the arguments associated types. Upon receiving an RPC message, a peer locally checks the credentials of the sender, and if the sender is authorized, the peer invokes the appropriate method and returns a response to the requesting

peer. The process may be done in a synchronous or asynchronous manner. PawnRPC uses the messaging guarantees to assure delivery ordering, and stateful messages to tolerate failure.

4 Enabling Autonomic Reservoir Optimization Using Pawn

In this section we describe how the interaction services provided by Pawn are used to support the autonomic oil reservoir process outlined in Figure 1 (Section 2). Every interacting component described in Section 2 is a peer that implements Pawn services. The IPARS Factory, VFSA, and the DISCOVER middleware are Application peers and implement ARC and AEX services. The DISCOVER portals are Client peers and implement AMS and Group communication services. Key operations using Pawn are described below.

Peer Deployment: The Application Execution service in Pawn uses the CorbaCoG kit (and the Globus Grid services it provides) in conjunction with the DISCOVER middleware to provide client peers access to Grid resources and services, and to deploy application peers.

Autonomic Optimization: Autonomic optimization involves interactions between the IPARS simulation, VFSA service and the IPARS Factory. VFSA provides the IPARS Factory with an initial guess based on the initialization parameters entered by the client. The IPARS factory then configures and spawns an IPARS instance. The simulation output is used by the Economic Model to generate the estimated revenue. This revenue is normalized and then fed back to VFSA, which generates a subsequent guess. The process continues until the terminating condition is reached (e.g. revenue stabilizes). These interactions build on the Application Runtime Control service (ARC) and the PawnRPC mechanism.

Collaboration and Interactive Monitoring and Steering: Users can collaboratively connect to, monitor, and interactively steer the IPARS Factory, VFSA, Economic model and an IPARS simulation instance. The Collaboration Services enable communication with other users, as well as the interactions between the Application Monitoring and Steering service (AMS) and the Application Runtime Control service (ARC), such an interaction is presented on Figure 3.

5 Experimental Evaluation of Pawn

Pawn was deployed on 20 hosts (1.5GHz Pentium IV processors with 512 MB of RAM, running Linux RedHat 7.2) located at Rutgers University. The Wide Area measurements were performed using two PCs (750 MHz Pentium III with 256 MB RAM and a 350 MHz Pentium II with 256 MB RAM) deployed on a private ISP. The evaluation consisted of the following experiments:

Round Trip Time communication (RTT) over LAN: This experiment evaluates the round trip time of messages sent from a peer running the *Application Monitoring and Steering service* to peers running the *Application Runtime Control service* over a LAN. The message size varies from 10 bytes to 1 Megabyte. The Pawn services (AMS and ARC) build the query and response messages that are then sent using the JXTA Endpoint

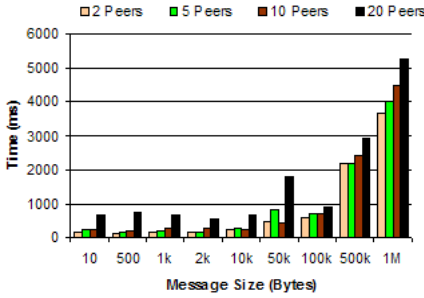


Fig. 4. RTT measurement in a LAN

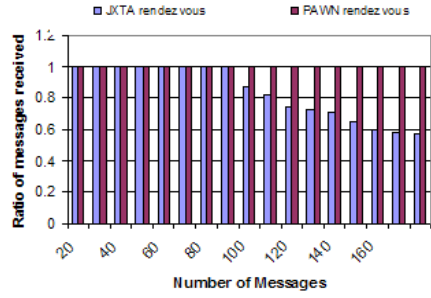


Fig. 5. Effectiveness of message queuing

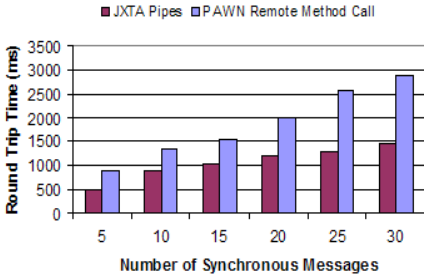


Fig. 6. PawnRPC over JXTA pipes

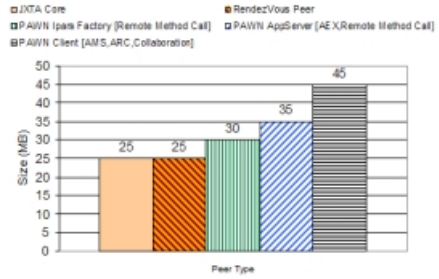


Fig. 7. Memory cost of JXTA and Pawn

Routing Protocol. The overall message transfer time is tightly bound to the performance of the JXTA platform, and is likely to improve with the next generation JXTA platform. The results are plotted in Figure 4. Note that the difference in RTT between 2 peers and 20 peers decreases as the message size increases.

Effectiveness of message queuing: This experiment compares the behavior of a Pawn rendezvous peer implementing the application-level message queuing to the behavior of a core JXTA rendezvous peer. The number of messages published on the rendezvous peer range from 10 to 500. The ratio of messages received is plotted in Figure 5. It can be seen that the message queuing functionality guarantees that no application-level messages are dropped even under heavy load.

Memory costs of JXTA and Pawn: In order to bootstrap a peer in Pawn, every peer has to load a core application implementing the JXTA and Pawn protocols. Figure 7 plots the memory used for loading the Java Virtual Machine, and the core JXTA protocols for every type of peer in Pawn. Pawn services add, on an average, an overhead of 20% to the JXTA core. Note that the client peer loads a Graphical User Interface in addition to the AMS and Collaboration services.

Overhead of PawnRPC on JXTA pipes: Figure 6 shows a comparison between PawnRPC and JXTA pipes. Using PawnRPC, a message describing the remote call is marshaled and sent to the remote peer. The remote peer unmarshals the request, processes it before marshaling and sending back a response to the requesting peer. The marshal-

ing, unmarshaling, and invocation add an overhead on the plain pipe transaction. This overhead remains however less than 50% on average.

6 Summary and Conclusions

This paper presented the design, implementation, and evaluation of Pawn, a peer-to-peer messaging substrate that builds on project JXTA to support peer-to-peer interactions for scientific applications on the Grid. Pawn provides stateful and guaranteed messaging to enable key application-level interactions such as synchronous/asynchronous communication, dynamic data injection, and remote procedure calls. It exports these interaction modalities through services at every step of the scientific investigation process, from application deployment to interactive monitoring and steering and group collaboration. The use of Pawn to enable peer-to-peer interactions for an oil reservoir optimization application on the Grid was presented. Pawn is motivated by our conviction that the next generation of scientific and engineering Grid applications will be based on continuous, seamless and secure interactions, where the application components, Grid services, resources (systems, CPUs, instruments, storage) and data (archives, sensors) interact as peers.

References

1. SETI@Home. Internet: <http://setiathome.ssl.berkeley.edu> (1998)
2. Monson-Haefel, R., Chappell, D.: Java Message Service. O'Reilly & Associates, Sebastopol, CA, USA (2000)
3. Fabret, F., Jacobsen, A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. *ACM SIGMOD Record* **30** (2001) 115–126
4. IBM MQSeries. Internet: <http://www-3.ibm.com/software/ts/mqseries/> (2002)
5. Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Parashar, M., ed.: *Proceedings of the Third International Workshop on Grid Computing (GRID 2002)*, Baltimore, MD, USA, Springer (2002) 274–278
6. Fox, G., Pallickara, S., Rao, X.: A Scaleable Event Infrastructure for Peer to Peer Grids. In: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, Seattle, Washington, USA, ACM Press (2002) 66–75
7. Furmento, N., Lee, W., Mayer, A., Newhouse, S., Darlington, J.: ICENI: An Open Grid Service Architecture Implemented with Jini. In: *SuperComputing 2002 (SC2002)*, Baltimore, MD, USA (2002) 10 pages in CDROM.
8. IPARS: Integrated Parallel Reservoir Simulator. Internet: <http://www.ticam.utexas.edu/CSM> (2000) Center for Subsurface Modeling, University of Texas at Austin.
9. Foster, I., Kesselman, C.: The Globus Project: A Status Report. In: *IPPS/SPDP'98 Heterogeneous Computing Workshop*, Orlando, Florida, USA (1998) 4–18
10. Parashar, M., Laszewski, G.V., Verma, S., Gawor, J., Keahey, K., Rehn, H.N.: A CORBA Commodity Grid Kit. *Special Issue on Grid Computing Environments, Concurrency and Computation: Practice and Experience* **14** (2002) 1057–1074
11. Project JXTA. Internet: <http://www.jxta.org> (2001)

A Spontaneous Overlay Search Tree

Hung-Chang Hsiao, Chuan-Mao Lin, and Chung-Ta King*

Department of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan 300
hsiao@pads1.cs.nthu.edu.tw

Abstract. It is often necessary to maintain a data structure, for example a search tree, for a set of computing nodes, which are interconnected dynamically and spontaneously. Instead of relying on a centralized “official” server to maintain the data structure, a grass-root and spontaneous approach is to distribute data structure to the participating nodes by taking advantages of their distributed resources. This study shows the feasibility of such an approach by designing a distributed search tree structure, called *Pyramid*, to operate atop end systems without relying on a centralized server. Pyramid allows the participating nodes to manipulate the data structure transparently through high-level operations such as search, insert, delete, update and query. Pyramid is designed based on the peer-to-peer model. Its self-configuration and -healing features enable the manipulation of information in an unreliable and unstable environment. A wireless application based on a prototype of Pyramid is presented.

1 Introduction

Imagining in a country fair in the not distant future, a talent contest is being held. While the official judges are busy with the scoring, the audiences also like to take an active role and make their choices. They use their PDA, cellular phone, smart watch, etc., to enter their ranking. They can also query for the current standings of the performers. A straightforward way of implementing this system is to call for the fair organizer to provide a centralized, official web site to collect, maintain, and disseminate the audiences’ scores. However, for one reason or another, we may want to resort to a more grass-root approach, in which no “official” web site is needed and the voting community is formed spontaneously using only the devices carries by the audiences. How this can be done?

Apparently, the devices need to be connected through one or more networks, perhaps through an ad hoc network. Beyond the basic connectivity, we need middleware services to maintain the scores and answer queries. One possibility is to overlay a data structure, say a search tree, over the spontaneous network formed by the devices. We then have to address the following issues.

* This work was supported in part by National Science Council, R.O.C., under Grant NSC 90-2213-E-007-076 and by Ministry of Education, R.O.C., under Grant MOE 89-E-FA04-1-4.

- We cannot rely on any particular device to host the search tree, because the device may not be reliable. The device may introduce a single point of failure and a performance bottleneck.
- A distributed search tree co-managed by the participating devices is thus desirable. However, the devices may join and leave at any time, and the network connectivity may be unreliable and unstable. The system thus needs to be adaptive, self-organized, and robust.
- The number of participating devices may be large and change constantly. The system therefore has to be scalable and incur little overhead to manage.

This study designs and implements a distributed search tree structure, called *Pyramid*, which is robust and scalable for storing, deleting, updating and querying data items scattered in a dynamic network. Pyramid is designed based on the *peer-to-peer* (P2P) model and aimed at aggregating the distributed resources in the network to aid the manipulation of the deployed search tree. After the search tree is created, any authorized end system should be able to transparently manipulate the tree via high-level operations. To accommodate the dynamics of the networked environment, Pyramid has to manage dynamic node configuration, utilize untrusting nodes, and tolerate faults, all without centralized administration. If nodes and communication links, overlaid by the search tree, fail, then the search tree performs self-healing. Pyramid is implemented on top of Tornado [4], a capability-aware P2P data naming, routing and storing middleware.

The proposed Pyramid search tree has the following features: (1) it can utilize distributed resources and does not depend on a centralized sever. (2) It is self-organized and can be operated in an unreliable environment. (3) It is shown highly available by probability analysis and simulation. (4) It can be maintained with nearly zero-memory overhead. (5) It is very scalable by following an almost stateless design.

A Pyramid tree based on such a wireless network is deployed. An application, the *virtual music scoring board*, is implemented to demonstrate the use of Pyramid.

2 Related Works

Distributed data structures have been investigated by [3][5][6]. These studies assumed that each node in a system had global knowledge of all participating nodes, including their network addresses. This knowledge was used to deliver messages to others. In [6], a distributed linear hash structure was proposed, and the issues of distributing hash data structures in a cluster computer environment were investigated in [3]. Distributed tree structures were studied in [5], which similarly assumed that each node had global knowledge of the network address of every node within a single administrative domain. The targeted system was reliable but was centrally controlled. Pyramid considers a computing environment which is quite different from the above. The environment consists of dynamic nodes assembled in an *ad hoc* fashion—centralized and artificial control is infeasible.

Information retrieval can be roughly categorized as (1) a search for an exact keyword, such as in [4][7][8][9][10]; (2) a search for partial keywords matching, such as in [1][2]. These studies either are based on flooding or rely on a virtual data

structure to organize related data items and/or machines. Pyramid is different from the above by not concentrating on a single type of search, but rather directly implementing an overlay search tree in an untethered environment. Search trees provided by Pyramid are capable of supporting *structured queries* such as SQL that cannot be implemented by methods mentioned above.

3 Pyramid

The ubiquitous wired and wireless Internet allows scattered resources in devices such as sensors, PDAs, desktop PCs, workstations and servers to be aggregated to provide distributed data and computation services. Pyramid is a generic middleware on top of Tornado that provides distributed tree data structures in spontaneous network environments. Each node participating in Pyramid contributes some of its storage and computational power to help assemble and access the tree structures in the system. A participating node can act as a client to request tree operations from other nodes in Pyramid, while simultaneously act as a server to supply data items stored in the trees to other nodes.

3.1 Overview

Pyramid maintains *search tree data structures* over spontaneous networks. Each participating node can construct a tree structure that will be maintained by multiple nodes in the system. Nodes can access a tree using high-level operations, described below. The integrity of constructed data structures can be ensured by the RSA algorithm, for example. Nodes that intend to access a particular tree must first obtain the corresponding public keys, and only those nodes with private keys can modify the contents of such a data structure.

Pyramid is implemented on top of Tornado and is therefore built according to the hash-based addressing scheme. This allows Pyramid to support nomadic data structures by decoupling the location of a data structure from its address. The hash-based scheme also supports anonymity. In other words, nodes that intend to access tree structures do not necessarily know where they should locate or how to organize the data structure. The scheme thus leverages the reliability of the underlying system and the durability of the data structures, while reducing malicious attacks on the nodes.

Pyramid provides the following interfaces to access the data of a search tree.

- *create* (s, t): Construct a search tree s with a lease time t in Pyramid.
- *insert* (s, v): Insert a key value v into the named tree s .
- *delete* (s, v): Remove v from s .
- *update* (s, v_1, v_2): Update a key value v_1 in s to v_2 .
- *search* (s, v): Search for exact v in s .
- *query* (s, v_1, v_2): Collect a set of values from v_1 to v_2 in s .

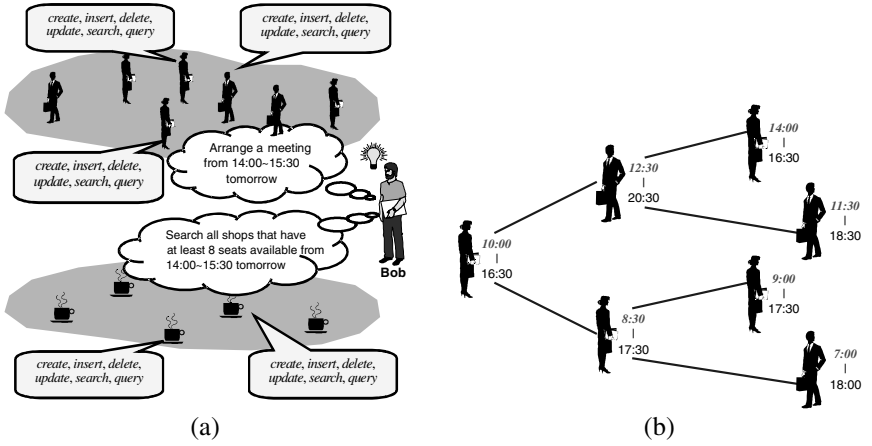


Fig. 1. A usage example of Pyramid. (a) There are two ad hoc Pyramid spaces, one organizes the schedules of colleagues and the other maintains the reservation information of each participating coffee shop. (b) A tree structure represents the Pyramid space for the schedules of colleagues by referring to the time values in Italics.

Figure 1(a) illustrates an example usage of Pyramid. Note that with the aid of Tornado, Pyramid can function over a global area. The example here, which shows ad hoc Pyramid spaces in a localized setting, demonstrates the diversity of Pyramid. In this example, two Pyramid spaces are presented: the schedules of colleagues, and the reservation information of coffee shops. Suppose Bob would like to call for a meeting with his colleagues currently in the office. He issues a *request query* to the Pyramid space “the schedules of colleagues”. The issued request specifies the date and time slots for meeting. Meanwhile, Bob searches all nearby coffee shops to reserve a place for the meeting.

Figure 1(b) shows a Pyramid tree that represents the schedules of colleagues. Each colleague may insert, delete and update his own schedule in the Pyramid tree. Note that a colleague may enter or leave the office at any time, and thus his schedule may join or depart from the Pyramid tree. Similarly, the Pyramid space, “reservation information of coffee shops”, tracks the seat availability of each participating coffee shop.

3.2 Design

Pyramid maintains an m -way search tree over a spontaneous network. A tree node r hosts m key values and pointers, and has the following properties.

$$Key_{1r} < Key_{2r} < Key_{3r} < \dots < Key_{mr} \quad (1)$$

and

$$Key_{vr} < Key_{v+1r} < Key_{(v+1)r}, \quad (2)$$

where Key_{kr} denotes the keys hosted by node r , $1 \leq k \leq m$ and $Key_{1r} = -\infty$. Key_{vr}'' represents a key in the v -th sub-tree *Sub-tree_v* associated with node r , where

$1 \leq v \leq m-1$. No more than m pointers point to m m -way search sub-trees in each tree node. For example, in Figure 2, Key_1'' , Key_2'' , Key_3'' and Key_m'' in the sub-trees $Sub-tree_1$, $Sub-tree_2$, $Sub-tree_3$ and $Sub-tree_m$, respectively, satisfy $Key_1 < Key_1'' < Key_2$, $Key_2 < Key_2'' < Key_3$, $Key_3 < Key_3'' < Key_4$ and $Key_m < Key_m''$.

3.2.1 Naming

Each tree node in Pyramid is addressed using the hash-based naming scheme supplied Tornado. Thus, given its hashing ID, a node can be dynamically mapped to a virtual home in Tornado.

To initialize a search tree in Pyramid, a request is first sent to the root of the tree. The root controls access to data items in the tree. The hashing value of the root of a tree named C (for example, schedules of colleagues), is calculated as,

$$H(C), \quad (3)$$

where H is the hashing function applied by Tornado. The hashing value C_i^v of each $Sub-tree_v$'s root of a node with the hashing value i in C is defined as if $1 \leq v \leq m-1$, then,

$$C_i^v = H\left(H(C) \oplus H(Key_{vi}) \oplus H(Key_{(v+1)i})\right). \quad (4)$$

Otherwise, if $v=m$, then,

$$C_i^v = H\left(H(C) \oplus H(Key_{vi})\right), \quad (5)$$

where $\oplus$ stands for concatenation. Notably, (1) each tree node also maintains a pointer that points to the hashing address of the node's parent. (2) Given the naming scheme for each sub-tree, Pyramid do not need to maintain pointers that point to a set of sub-trees, providing state-less tree construction and maintenance.

3.2.2 Creating a Tree

First, the hashing value of the namespace that represents a given tree structure is calculated. Then, the tree root node is stored in the corresponding virtual home with the hashing key closest to the hashing value of the tree root node. Notably, Tornado provides the *Publish* operation, where $Publish(x, y)$ stores a data item to a virtual home with the hashing key closest to x in the Tornado space called y .

Since the virtual home that hosts a tree root node in Pyramid may fail and the communication links to the home may be broken, Pyramid relies on the fault-resilient route supported by Tornado. The hash-based naming delay the binding of the actual network address (for example, the IP address) of a tree root node to the name of the overlaying virtual home.

The k -replication mechanism of Tornado can further increase the reliability of Pyramid. The mechanism sends replicas of k identical copies to k Tornado virtual homes whose IDs are closest to that of the tree root node. Each Pyramid tree root node is thereby implicitly replicated to k spaces deployed over distinct Tornado virtual homes. When the home that hosts a Pyramid root node fails, the contents stored in the failing node in all instance can be recovered from the replica homes. The

tree data structure need not be reconstructed since the replica home already has the most up-to-date key values maintained by the failing home.

3.2.3 Exact Search

To find a particular key in a Pyramid tree, a search request is firstly sent to the tree root node. This is done by issuing a routing request to a virtual home in Tornado whose ID is closest to the hashing value of the designated namespace. After the virtual home has been consulted, it fetches the keys associated with the designated Pyramid tree. If the consulted home cannot yield the requested key, the request is routed to a virtual home whose ID is closest to that of the sub-tree's pointer, where such a sub-tree may store the requested key (Line 6.c). If the search is successful, the hashing address of the tree node that hosts the requested key is returned. Otherwise, the hash ID of a tree node that can host the requested key is returned.

3.2.4 Key Insertion

To insert a key into a Pyramid tree, the search algorithm is first performed. This algorithm routes the insertion request to a tree node r that is responsible for accommodating the inserted key. If the inserted key cannot be found in tree nodes along the route toward node r , then node r stores the inserted key while examining whether the number of valid keys it has maintained exceeds. If the number of keys exceeds m , then node r informs its parent to segment the keys hosted by node r . The parent accomplishes the segmentation by issuing *Publish* operations to introduce two tree nodes as the node's sub-trees. One tree node replicates the first to the $(\lceil m/2 \rceil - 1)$ -th keys from node r and the other copies the $(m - \lceil m/2 \rceil)$ -th to the m -th keys. Then, node r 's parent performs similar operations after the $\lceil m/2 \rceil$ -th key hosted by node r is stored in node r 's parent. Note that after node r 's parent inserts node r 's $\lceil m/2 \rceil$ -th key into its local storage, the pointer that points to node r is no longer maintained and the keys stored in node r are gradually removed when their associated time contracts have expired.

To maintain the reliability of a tree, first, tree nodes introduced by segmentation are spontaneously replicated to k virtual homes in Tornado. The root node mentioned in Section 3.2.2 is also replicated. After a virtual home hosting a particular tree node fails or its communication link is broken, requests to such a failing home are routed to another virtual home that includes identical copies which can be provided by the failure home.

Second, producers of data should proactively and periodically refresh published keys using operations similar to those performed by the insertion algorithm. If a key to be refreshed does not appear in the designated tree, then it is added thereto.

Third, the tree constructor periodically issues the re-publishing messages to the tree roots that it constructed. When a tree root receives such a message, it repeatedly broadcasts the re-publishing messages to each of its sub-trees according to the valid keys that it maintains. Conceptually, the re-publishing messages are epidemically delivered to all tree nodes. If the virtual home whose hash address is closest to the tree node's ID i of the named tree C that receives the re-publish message, is a newly joining home and could be responsible for hosting the key values in the sub-tree C_i^v ,

then this virtual home allocates storage space to the tree node that represents key values from Key_{vi} to $Key_{v(i+1)}$ if $1 \leq v \leq m-1$ or the values above Key_{vi} otherwise. This allocation ensures that a Pyramid tree is deployed over Tornado virtual homes whose IDs are closest to those of tree nodes.

3.2.5 Deleting and Updating Keys

In Pyramid, only a data producer can erase the keys generated by itself. To delete a key x , a data producer can explicitly nullify the time contract of key x or simply not re-publish such a key. After key x 's time contract has expired or is nullified, it is removed from its associated tree. Then, the tree node with the hashing address i , hosting key x , re-publishes a new sub-tree to host key values from $Key_{(v-1)i}$ to $Key_{(v+1)i}$ if $Key_{(v-1)i} < key\ x < Key_{(v+1)i}$ if $1 < v \leq m-1$, or key values above $Key_{(m-1)i}$ if key x is Key_{mi} .

Note that key u originally appears in the sub-tree $Sub-tree_{v-1}$ or $Sub-tree_v$. It will not be available if the sub-tree responsible for the keys from $Key_{(v-1)i}$ to $Key_{(v+1)i}$ is renewed following the deletion of a key. However, a data producer will re-publish the keys it generated and then key u will be made available by its re-insertion into the tree (Section 3.2.4).

To update a key from x_1 to x_2 in Pyramid, x_1 can firstly be simply deleted from the designated tree and then x_2 inserted.

After a key is deleted or updated, the probability of failing to access a key due to changes in the mapping from tree nodes to Tornado nodes previously stored in a complete m -way search tree with height h can be similarly estimated as $\nabla(m, h)$, as shown in Equation 9, where $\nabla(m, h) \rightarrow 0$ when $h \rightarrow \infty$.

3.2.6 Sub-Range Query

To collect keys with values from v_1 to v_2 in a Pyramid tree, a corresponding request is firstly issued to the tree root node of the designated tree. The tree root then invokes several sub-range queries if multiple keys have values between v_1 and v_2 . Tree nodes that receive the sub-range query requests perform similar operations.

4 Implementation and an Application

Pyramid has been implemented on top of a light Tornado, called Trailblazer, over a wireless network. Each wireless device in the Trailblazer network is a WINCE-based HP568 PDA. The PDA is equipped with a 206 MHz Intel SA-1100 microprocessor, 64 Mbytes RAM and an 802.11b Wireless CF interface. The wireless devices are randomly named using a uniformly random hash function. Trailblazer guarantees that a message with a designated hash address can be efficiently routed to a wireless device whose hash key is closest to the specified one. Each wireless device provides

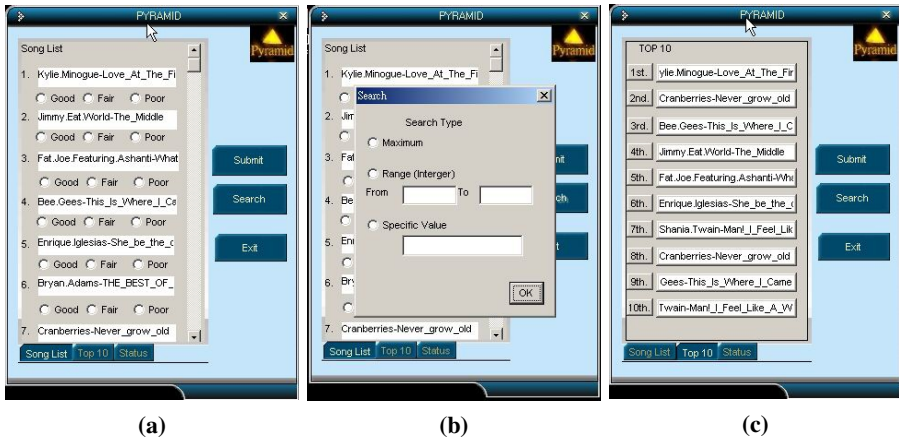


Fig. 2. The user interfaces of the virtual music scoring board, where (a) shows the scoring interface, (b) shows the query interface and (c) shows the results for the top-10 favorite records.

some storage space and every device in Trailblazer can access the aggregated storage space.

As a proof-of-concept, an application called the *virtual music scoring board* is implemented. Users can dynamically participate in the virtual music scoring board and access related information, e.g. the top-ten most popular music records. Users can also enter their scores into the virtual music scoring board. The trailblazer network is dynamically organized on top of the participating wireless devices. One Pyramid tree, based on the Trailblazer network, is deployed and used to handle all the music records that the participating users are interested. The Pyramid tree is based on the lexical order of the named musical records. Each user has a local profile that represents a set of favorite musical records with associated grades.

Figure 2(a) shows that each user can mark a particular record as “good”, “fair” or “poor”. A user can submit his or her scores to the shared Pyramid tree. Each user can then search for the most popular music record or submit a range query for finding the records that are from $x\%$ to $y\%$ most popular, where x is smaller than y (Figure 2(b)). A user can also query the score of a particular album submitted by all participating users. Figure 2(c) presents the top-ten popular music records.

5 Conclusions

This study presents a novel design for search tree data structures, which is based on the peer-to-peer model. Pyramid can utilize resources of widely distributed end systems without centralized administration. A set of rich operations, including search, insert, delete, update and query, is provided to manipulate data items in the tree. The search tree can be adapted to the changing environment while providing robust operations and ensuring data availability. As a proof-of-concept, a wireless

application based on a prototype of Pyramid is presented. We believe that Pyramid is generic and it can be applied to other peer-to-peer infrastructures.

References

- [1] E. Cohen, A. Fiat, and H. Kaplan. "A Case for Associative Peer to Peer Overlays," In *ACM Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.
- [2] Crespo and H. Garcia-Molina. "Routing Indices for Peer-to-Peer Systems," In *International Conference on Distributed Computing Systems*, pages 19–28, July 2002.
- [3] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, Distributed Data Structures for Internet Service Construction," In *USENIX Symposium on Operating Systems Design and Implementation*, October 2000.
- [4] H.-C. Hsiao and C.-T. King. "Tornado: Capability-Aware Peer-to-Peer Storage Networks," In *IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.
- [5] Kroll and P. Widmayer. "Distributing a Search Tree among a Growing Number of Processors," In *Proceedings of ACM SIGMOD Conference*, pages 265–276, June 1994.
- [6] W. Litwin, M.-A. Neimat, and D. A. Schneider. "LH*—A Scalable, Distributed Data Structure," *ACM Transactions on Database Systems*, 21(4), pages 480–525, December 1996.
- [7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A Scalable Content-Addressable Network," In *ACM SIGCOMM*, pages 161–172, August 2001.
- [8] Rowstron and P. Druschel. "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.
- [9] Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," In *ACM SIGCOMM*, pages 149–160, August 2001.
- [10] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," *Technical Report UCB/CSD-01-1141*, April 2000.

Fault Tolerant Peer-to-Peer Dissemination Network

Konstantinos G. Zerfiris and Helen D. Karatza

Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece
{zerf,karatza}@csd.auth.gr

Abstract. The widespread use of broadband networks gave new ground for Peer-to-Peer systems. The evolution of these systems made P2P file sharing networks one of the most popular ways of sharing content. Due to their distributed nature, such networks tend to be a reliable source for highly anticipated files. However, along with the benefits of P2P networks, certain patterns became apparent. Uneven flow of data and intersperse congestion points could result on increased mean response time or even network failure. In this paper the structure of Peercast, an agent based dissemination network, is presented. Emphasis is given to the self organizing nature of the dissemination tree, and simulation results depict its behavior under strenuous conditions...

1 Introduction

While today the servers are able to acquire more bandwidth, they can not keep up with the rapidly increasing requests of the users. The demand for faster service increased as broadband connections became available. But if a file of considerable size has to be disseminated to a considerable amount of receivers, the network could be saturated quickly, clogging the host computer. Such is the case for example when any highly anticipated software is released and several people are trying to download it at the same time. This became known as the midnight madness problem [1].

As today's needs for data transfer steadily increase, traditional ways of making data available to the masses become obsolete. Conventional FTP servers can no longer serve as a way of distributing large amounts of data. Mirroring the required content on several dispersed servers, cannot always compensate for the rapid traffic increase.

The main architecture used for casting data through the Internet is IP multicast, which mainly targets real time non-reliable applications. It extends the IP architecture so that packets travel only once on the same parts of a network to reach multiple receivers. A transmitted packet is replicated only if it needs to, on network routers along the way to the receivers. Although it has been considered as the foundation for Internet distribution and it is available in most routers and on most operating systems, IP multicast has not so far lived up to early expectations. Its fundamental problem is that it requires that all recipients

receive the content at the same time. The most popular solution to this problem was to multicast the content multiple times until all of the recipients obtain it. IP multicast might be considered ideal for applications that require relatively high and constant throughput but not much delay. However it is not suitable for applications that may tolerate significant delays but no losses. This is the case with file distribution.

These days, a new way of disseminating files emerged. File sharing networks [2] are perhaps the most commonly used Peer-to-Peer applications. Such systems have been used for diverse applications: combining the computational power of thousands of computers, forming collaborative communities, instant messaging, etc. P2P file sharing networks' main purpose is to create a common pool of files where everybody can search and retrieve any shared files. Depending on the algorithm used, these sharing networks can be divided in two groups. Networks that maintain a single database of peers and their content references are known as centralized. Such file sharing networks [3] have several advantages, such as easy control and maintenance, and some disadvantages as, for example, server overload. On the other hand, dynamically reorganizing networks such as Gnutella [4], have a rather more elaborate service discovery mechanism, avoiding this way the use of a centralized server. Those kinds of networks are known as decentralized, and their main advantage is the absence of a single point of failure.

File sharing networks had never been designed for file dissemination. Nevertheless people turn to them to find highly anticipated software or even video files, when the official server stops responding due to high demand. Extensive research has been done about how existing P2P networks operate over time and how they can be optimized [4,5]. However, the dissemination process of highly anticipated files on P2P networks over unreliable network connections remains unexplored. Peercast, a P2P network first presented in [6], is designed to assist the dissemination of a file in a heterogeneous network of clients. The purpose of this paper is to show Peercast's performance in the long term under different network conditions. Simulation results depict how network failures can affect this process.

The structure of this paper is as follows. In section 2 the Peercast's structure is shown, along with its latest extensions. Section 3 elaborates on the simulation model of the system and the simulated network failures. The results and drawn conclusions are summarized in section 4 and finally, section 5 presents suggestions for further research.

2 The Network

When a file needs to be downloaded by more clients than the server can handle, alternative algorithms have to be utilized. The naive way of avoiding retransmissions is to pipeline the file through all the clients. But this is not a viable solution because clients might have to indefinitely wait to be served.

The proposed algorithm uses a self-organizing tree of clients. The server can upload the file to a certain number of clients simultaneously. When the server

successfully uploads a file to a client, it keeps a reference of this client to a short list (up to 100 entries). The server has a queue, but most of the clients are expected to find this queue full. This is the case especially at the beginning of the dissemination process, as clients arrive faster than the server can handle. In this case, the server sends to the client the list of clients that already downloaded the file. This way, the new client can download the file from a peer that was already served, removing the congestion from the server.

When a client finishes the download, it acts as a server for other clients. Similarly to the server, the clients have a short queue. If a client A requests the file from a client B that has it, and that client B can not serve client A immediately, A is queued. If the queue is full, client B sends its own list of clients that it served to client A, so that it can continue searching. If a client is not able to be served or queued, it retries after a certain period of time to contact the server.

The peers are not expected to stay on-line for ever. But when a peer leaves the network, the dissemination tree is left in an inconsistent state. That's because the clients who were served from that peer are no longer accessible from the tree's root. In order to take advantage of all the peers that are willing to help in the dissemination process, the clients that are not occupied by serving other clients, periodically check with their parent peer. If the parent is not on-line or it is not accessible due to network failure, the client contacts the server and assigns itself as the server's child. If this fails, it requests from the server its list of served clients and tries to assign itself to one of those clients. If this fails also, the client waits for a certain amount of time and retries.

As it was mentioned earlier, in order to avoid server explosion, the list of children that the server has, is limited. If this list is full and a new child has to be added, the server removes from the list its oldest child, accepts the new one, and forces the new peer to adopt the removed client to its own list. This way, the server always has a list of serving peers that recently became available and therefore are more likely to have empty queues and stay on-line longer.

In order to utilize all the available upload bandwidth, a single peer can serve several clients concurrently. Additionally, each client can initiate multiple concurrent download connections in order to utilize all the available download bandwidth. At the end of the transfer, the downloading client chooses randomly one of the assisting peers and requests to be listed on that one. The way to derive the optimal number of simultaneous upload connections and queue size is discussed in [6]. In brief, Peercast dynamically increases the number of connection slots used when the rate of arrivals balances with the rate of clients being served as shown in figure 1a. As a mean of keeping track of the number of served clients, the server instructs a small percentage (5%) of the arriving clients, to send back a message when they finish downloading the file. This way the server can estimate when the balance will occur. At that point it propagates a message to all the peers in its list and to every new client in order to increase the number of concurrent connections and decrease the queue size.

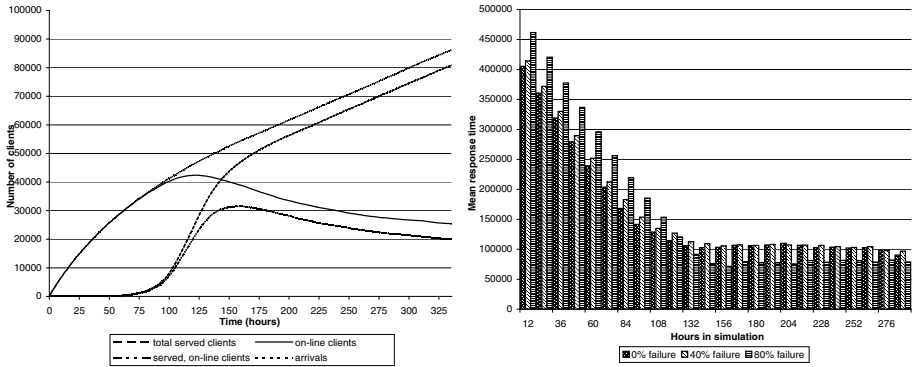


Fig. 1. a) Network's state over time, b) Mean response time in 12-hour intervals according to each client's arrival

Several issues arise about the performance of this algorithm under different network conditions in a heterogeneous network of clients. For example, how is the mean response time affected by several local congestion points or network failures? Can such problems affect dramatically the dissemination process? How does the system respond if a considerable number of peers refuse to assist?

3 Simulation Model

In this section details are presented about the simulation model for the proposed network, and show how different strategies might affect the dissemination process. The system was populated with clients arriving according to the exponential distribution. The simulation period was set to be 2 weeks (1209600 seconds). During the first week the mean interarrival time was incremented linearly from 5 to 20 sec in order to simulate demand on a highly anticipated file. For the second week the exponential distribution was used with 20 sec mean interarrival time. The file size was set to be 650MB (the size of a full CD).

All the clients that populated the system were set to have broadband connections to the Internet, resembling cable modems and DSL. This is done in order to use a realistic model. As in many cases, such connections have different download and upload speeds. Four different categories of users were used. The first category (10% of the clients) had download and upload speed of 256 Kbps, the second (40%) had 384 Kbps and 128 Kbps, the third (20%) had 384 Kbps (download and upload), and the fourth (30%) had 1.5 Mbps and 384 Kbps respectively. This configuration is a theoretical model, and is used to compare how the same network performs under different conditions.

These kinds of clients are always on-line. However, they are not expected to share that file for ever. Therefore they were set to leave the dissemination network with exponential distribution and mean time of four days. The server (a client from category 4) was set to never go off line. An additional difference

between the server and the clients is that the server keeps a limited list of up to 100 clients that it served, whereas the clients have an unlimited list. That's because clients are not expected to stay on the network for ever, and therefore they do not run the risk of collapsing because of an overwhelming list.

The actual connection speed between two clients is calculated at the beginning of each session, taking into consideration the theoretical maximum speed they could achieve and an exponentially distributed surcharge, in order to simulate additional network traffic and sparse bottlenecks. If a new client cannot be served or queued immediately, it waits for 600 seconds and retries.

At the beginning of the dissemination, two download and two upload connections were used in order to speed up the creation of a critical mass of served clients. The critical mass is the point where the rate of served clients in the system starts to decline (figure 1a). That happens when the rate of arriving and the rate of departing clients balance out. When the server estimates that the critical mass has been reached, it propagates a message through the peers in its list, notifying the clients to change the upload/download slots from 2 to 4. Additionally, the waiting queue on each client drops from 8 to 4 entries. The number of concurrent uploading and downloading streams at any time and the queue size are derived from simulation results showed in [6]. This switch is done in order to reduce mean response time, as newly arriving clients should not be queued on long queues. When the critical mass has been reached, they are more likely to find service from clients found deeper in the dissemination tree, optimizing this way all the available network's resources.

As it was mentioned earlier, the behavior of this network can change significantly under certain conditions. The system's performance is investigated at the beginning (2 weeks) of the dissemination, under different conditions. Our focus is on how the system behaves under network failure. More specifically, the simulations tested the system's performance when a certain percentage of connections failed. At any given time, the connection between any two clients (including the server) was set to have a predefined chance of failure. The system was tested using 0, 40 and 80 percent chance that any given client cannot contact another peer in order to be served or queued on it. This is done in order to simulate local network bottlenecks and network or system failures. If a client cannot contact a peer, it tries to find another. If no clients are found, it contacts the server to request an updated list of clients. In the case that even the server is not accessible, it retries after 600 seconds.

Additional simulations show the system's behavior when a percentage of the clients refuse assistance to other peers. In this case it is assumed that those clients go off-line immediately after they finish the download, and do not rejoin the network later on. This is expected to decrease dramatically the performance of the dissemination process. Nevertheless it is a behavior that can be expected. We test the system's performance when 0, 10 and 40 percent of the clients depart from the network immediately after they have been served. All of these simulations are done using a reliable network (0% connection failures). It should be noted that all network failure simulations were done using 10% of such clients.

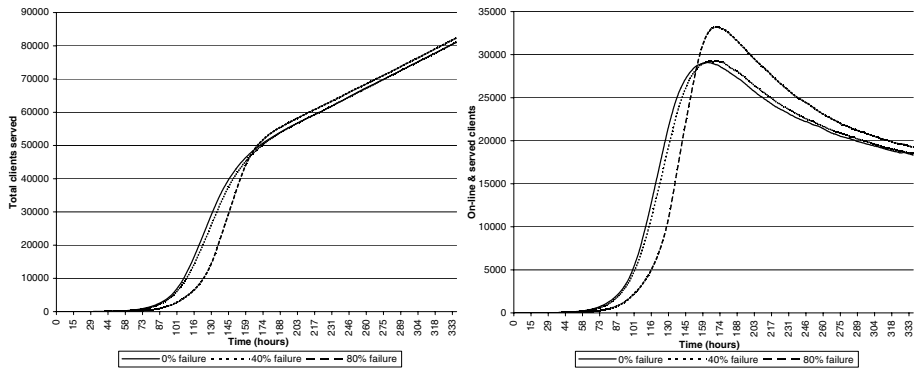


Fig. 2. Network failure tests over time: a) total clients served, b) served clients that are still on-line

The theoretical case that all the clients are willing to cooperate in the process for a certain period of time is examined in order to compare the degradation of performance that occurs to the rest of the cases.

4 Simulation Results and Conclusion

Figure 2a reveals that local network or system failures have an effect on the dissemination process, but not significant. More specifically, at the beginning of the dissemination there is a drop at the number of served clients. This is more obvious in the 80% network failure simulation. Nevertheless, as shown in table 1, the mean response time increased only by an average of 3.5% for the 40% case and 8.5% for the 80% case. This disproportional drop of performance is justifiable, as clients that are temporarily inaccessible by one peer, can be accessible by another. Figure 2a depicts that the network's resources are volatile at the beginning of the process, but as more peers join the network, their resource utilization increases steadily.

In figure 2b, the number of currently on-line peers is shown to be increased in the case of 80% network failure. That's because the server is not able to determine accurately when the critical mass has been built. This is also responsible for the higher mean response time shown in table 1. The increased mean response time in all cases can be explained as the clients that arrive early on the dissemination process have to wait for a long period of time to be served. When the rate of arrivals balances with the rate of clients being served, the mean response time stabilizes to lower levels. Therefore, clients arriving later in the system benefit from a faster service. This is depicted in figure 1b where mean response time is shown in 12 hour intervals according to each client's arrival in the system.

An additional test that was executed at the end of the simulations showed the integrity of the dissemination tree under these conditions. More specifically, in the case of the 40% network failure, by iterating through the tree, 0.5% of the

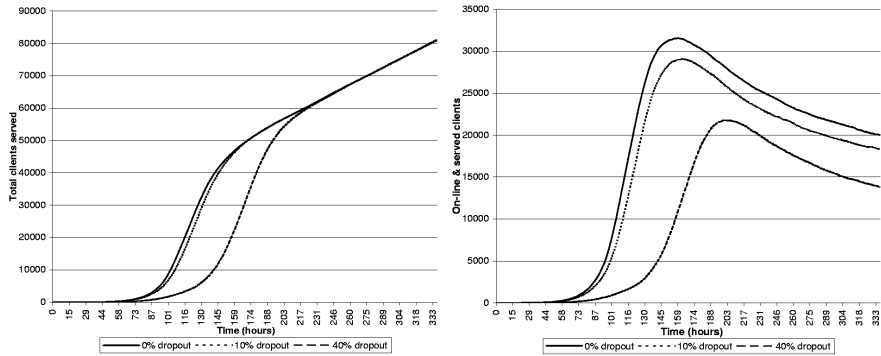


Fig. 3. Percentage of clients departs from the network immediately after completing the download a) total clients served over time, b) served clients that are still on-line

currently on-line serving-peers were found to be unreachable. This percentage increased to 10% for the 80% case. Those clients were in a timeout loop in order to try to reassign themselves in the tree. This shows that the nature of the self-organizing tree is relatively reliable to harsh conditions.

In figure 3a, the degradation of performance is shown to be proportional to the number of clients that depart from the system without assisting in the dissemination process. This is also shown in table 2 where the mean response time increased by an average of 4% when 10% of the clients leave the system as soon as they are served, and by 45% in the 40% case. Additionally, figure 3b shows that the critical mass is reached much earlier in the first case. That's because it is essential for the performance of the dissemination process that each client within this system acts as a server for other peers when it is served.

Further simulation results, not presented here due to space limitations, show that if a file of smaller size is used the impact of the clients that refuse to serve other peers is getting smaller. Overall, the system's behavior in strenuous conditions can be considered satisfactory. Its performance decrease is minimal in network failures, and the decentralized self-organizing tree of served clients proved to be reliable and scalable. The utilization of peer-to-peer technology for this task revealed a flexible way of reliably disseminating static data in a high number of clients, as long as the interested parties act collectively.

5 Future Work

Additional simulation experiments are under way, using distributions varying with time for more realistic long run simulations, as depicted in [7]. Peercast is an evolving platform. For the current P2P network implementation we used a monolithic approach: all the data has to be sent to a client, before this client starts sending it to another peer. A new version that replicates groups of 256KB packets, to adjacent peers as they arrive, is under way. This is expected to

Table 1. Mean response time over different amount of network failures (10% dropouts)

Net failures	256/256	384/128	384/384	1.5/384
0%	210377	199498	199672	182107
40%	216783	207291	207037	188381
80%	227610	216455	217156	197314

Table 2. Mean response time over different percentage of clients leaving immediately

Dropouts	256/256	384/128	384/384	1.5/384
0%	201812	192129	192077	174603
10%	210377	199498	199672	182107
40%	289458	278221	278129	257649

alleviate the problems that are caused from peers that go off-line immediately or soon after they finish downloading the requested file. The synchronization between the peers is done in predetermine time intervals, called epochs [8]. The peers are segmented in virtual groups according to their bandwidth and the epoch size depends on an estimation of the minimum bandwidth between the peers that form each dissemination group. Simulation results from this network are expected to show alleviation of several issues raised in this paper.

References

1. Schooler, E., Gemmell, J.: Using Multicast FEC to solve the Midnight Madness Problem. Technical Report, Microsoft research (1997)
2. Parameswaran, M., Susarla, A., Whinston, A.B.: P2P Networking: An Information Sharing Alternative. *IEEE Computer Journal*, Vol. 34. (2001) 31-38.
3. Shirky C.: Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology / Listening to Napster. I.A. Oram (ed.), O'Reilly & Associates.
4. Ripeanu, M., Foster, I., Iamnitchi, A.: Mapping the Gnutella Network: Properties of large scale peer to peer systems and implications for system design. *Internet Computing Journal*, IEEE Computer Society (2002) 50-57
5. Markatos, E.P.: Tracing a large-scale Peer to Peer System: an hour in the life of Gnutella. *Proceedings of the CCGrid 2002, Second IEEE/ACM International Symposium on Cluster Computing and the Grid* (2002) 65-74
6. Zerfiridis K.G., Karatza, H.D.: Large Scale Dissemination using a Peer-to-Peer Network. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* 2003, Tokyo (2003) 421-427
7. Karatza, H.D.: Task Scheduling Performance in Distributed Systems with Time Varying Workload. *Neural, Parallel & Scientific Computations*, Vol. 10. Dynamic Publishers, Atlanta, (2002) 325-338
8. Karatza, H.D., Hilzer R.C.: Epoch Load Sharing in a Network of Workstations. *Proceedings of the 34th Annual Simulation Symposium*, IEEE Computer Society Press, SCS, Seattle, Washington (2001) 36-42

Exploring the Catalactic Coordination Approach for Peer-to-Peer Systems*

Oscar Ardaiz¹, Pau Artigas¹, Torsten Eymann², Felix Freitag¹, Roc Messeguer¹,
Leandro Navarro¹, and Michael Reinicke²

¹ Computer Architecture Department, Polytechnic University of Catalonia, Spain
{oardaiz,partigas,felix,meseguer,leandro}@ac.upc.es

² Institute for Computer Science and Social Studies
Albert-Ludwigs-University Freiburg, Germany
{reinicke,eymann}@iig.uni-freiburg.de

Abstract. Efficient discovery and resource allocation is one of the challenges of current Peer-to-Peer systems. In centralized approaches, the user requests can be matched to the fastest, cheapest or most available resource. This approach, however, shows scalability limits. In this paper, we explore the catalactic coordination as a decentralized economic approach for resource allocation in peer-to-peer networks. The economic model of the catalaxy is based on the self-interested maximization of utility and the negotiation of prices between agents. We evaluate the feasibility of our approach by means of simulations and compare the proposed system with a centralized baseline approach. Our results indicate that while in the catalactic approach the number of control messages exchanged between the peers grows due to the negotiation process, its service provision rate is fairly constant in different dynamic environments.

1 Introduction

Peer-to-peer (P2P) systems are a class of distributed systems or applications to achieve a certain functionality in a decentralized manner. In this model, peers give resources and receive resources in turn. These resources can consist of computing power, storage of data or content, network bandwidth or presence. Application domains are distributed computing, data and content sharing, and collaborative applications.

A number of successful applications like Napster [9], Gnutella [5], and Freenet [4] for file and content sharing, and SETI@home [11] for distributed computing have demonstrated the feasibility of this approach.

Current P2P systems, however, have mainly focused on the exchange of objects like files and music clips, which are “small”. In future systems, however, the content

* This work was supported in part by the Ministry of Science and Technology of Spain under Contract TIC2002-04258-C03-01 and TIC2001-5193-E, and the European Union under Contract IST-2001-34030 CATNET.

will be of any form, including audio, video, and large data sets. Peer-to-peer systems may be used to set up multi-cast services for large-scale global audiences, provide services for storing ultra large data sets, and to allow the execution of parallel applications requiring teraflops of processing power. To achieve an efficient performance of such systems, more intelligent decisions than we have in today's systems are required, concerning particularly from where the content should be retrieved and on which path it should travel.

In this paper, we propose the usage of the economic paradigm of the catallaxis [3] for the decentralized resource allocation in P2P networks. In the catallactic coordination model the decisions of the peers are based on economic principles, being aware that resources like bandwidth, processing power and storage are limited. Peers are in negotiation with each other to optimize their own benefits, and the decision of service provision takes into account their cost and benefits involved.

Recent research in Grid computing has also recognized the value of price generation and negotiation, and in general the use of economic models for trading resources and services and for the regulation of supply and demand of resources in increasingly large-scale and complex Grid environment. Examples are the Nimrod/G Resource Broker and the GridBus project [1; 6].

In order to study the catallactic coordination in P2P networks, we use the CATNET application layer simulator [2], which builds on top of a TCP/IP network simulator the agents to form a peer-to-peer network. We evaluate the proposed system with several experiments and compare the achieved service provision with a baseline system.

In the following section 2 we describe the motivation of our approach. In section 3 we explain the used simulator and the experimental framework. Section 4 contains the evaluation of the proposed system and the discussion of the results. In section 5 we conclude the paper.

2 Decentralized Economic Coordination with the Catallaxy Paradigm

Application layer networks such as peer-to-peer networks are software architectures which allow the provision of services by connecting a large number of individual computers for information search, content download, parallel processing or data storage.

In order to keep such a network operational, service control and resource allocation mechanisms are required. Performing such task with a centralized coordinator, however, has several difficulties: 1) A continuously updating mechanism would be needed to reflect the changes in the service demands and node connectivity; and 2) Long latencies for obtaining updates about the nodes at the edge of the network as the diameter of the network grows.

These drawbacks motivate the evaluation of a decentralized coordination concept, which is able to allocate services and resources without having a dedicated and centralized coordinator instance.

The catallaxy coordination approach [3; 7] is a coordination mechanism for information systems consisting of autonomous network elements, which is based on constant negotiation and price signaling. Autonomous agents are able to adapt their heuristic strategies using machine learning mechanisms. This constant revision of prices leads to an evolution of the agent strategies, a stabilization of prices throughout the system and self-regulating coordination patterns [3]. The resulting patterns are comparable to those witnessed in human market negotiation experiments [10].

3 Experimental Framework

3.1 The Simulator for P2P Networks

In order to evaluate the behavior of a P2P system with the catallactic coordination mechanism, we have used the CATNET network simulator [2]. CATNET is a simulator for an application layer network, which allows creating different types of agents to form a network. This simulator is implemented on top of the JavaSim network simulator [8]. JavaSim simulates a general TCP/IP network and provides substantial support for simulating real network topologies and application layer services, i.e. data and control messages among application network instances.

We have implemented two main control mechanisms for the network coordination: the baseline and the catallactic control mechanism. The baseline mechanism computes the service/resource allocation decision in a centralized instance. In the catallactic mechanism, autonomous agents take their decisions in a decentralized way, having only local information about the environment. Each agent disposes of a strategy to take decisions, which targets to increase the agent's own benefit.

In the simulations, we consider a service as the functionality, which is exchanged among the peers in the network. The concept of service and the functions, or "personalities", a peer can assume in the CATNET simulator, are the following:

- **Service:** a service encapsulates a general function performed in the P2P network. A service is the provision of a resource such as computing power, data storage, content, or bandwidth. The service provision includes the search for a resource and its reservation for availability.
- **Client:** A peer may act as a client or consumer of a service. As such it needs to access the service, use it for a defined time period, and then continues with its own program sequence.
- **Resource:** A peer, which is the owner of a required functionality. This functionality, for instance, may represent content, storage or processing power. The functionality, which is required by the clients or consuming peers, is encapsulated in a service.

- **Service copy:** A peer acting as a service copy offers a service as an intermediary, however it is not the owner of the components to provide the service. It must cooperate with the resource to be able to provide the service. Service copies offer the service to requesting clients.

3.2 Money and Message Flow

In the simulator, the network activity is characterized by a continuous exchange of control messages and service provision. Different control messages in the two coordination mechanisms are used to accomplish the negotiation between peers.

In Figure 1 we show the money and message flow used in the catallactic coordinated system. The requests from clients are broadcasted and forwarded to the service copies. Compared with the flooded requests model used in Gnutella [5], however, the numbers of hops a request can be forwarded in the simulator is limited. Service copies initiate negotiations with the resources they know to provide the service. Upon successful negotiations the service copies offer the service to the client. If the client accepts, then the service copy provides the service by means of a resource.

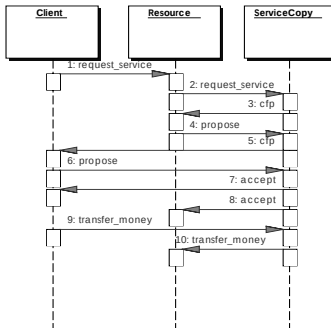


Fig. 1. Money and message flows. Catallactic coordinated system.

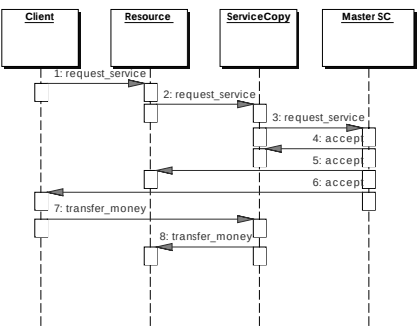


Fig. 2. Money and message flows: Baseline approach.

In the centralized baseline system (Figure 2), the master service copy (MSC) receives the client requests with additional information through the resource/service copy pairs. Taking into account the distance and availability, it selects a resource/service copy pair and sends back an accept/reject message to the client. The resource allocates the required resource units and the service copy provides the service to the client.

4 Experimental Evaluation

4.1 Experimental Setup

With experiments we wish to measure if a P2P network coordinated by the catallaxy paradigm is able to successfully provide service to requesting clients. A second goal is to compare qualitatively the obtained results with the centrally coordinated baseline system.

In our experiments we explore as design space of the system the node density and node dynamics of the network (Figure 3). First, we simulate the P2P network with different densities of the service and resource providing agents in a high dynamics environment (Figure 1 experiments 1A-C). Then, we simulate the high node density network in environments with different values of the dynamics (Figure 3 experiments 2A-C).

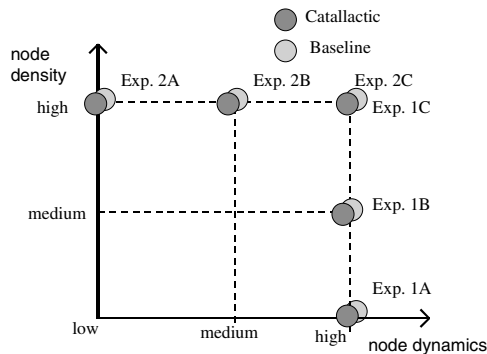


Fig. 3. Design space of the system evaluated experimentally. Experiments 1A-C: Node density. Experiments 2A-C: Node dynamics.

In the simulations the input is a trace of client demands with requests for service. The service request specifies the amount of service, a price, and its duration. In all experiments the same demand trace is used.

The physical network topology used in the experiments is organized in three levels of pentagons with leaves on the outer level, such as shown in Figure 4. Although other specific or random topologies of the nodes could be used as well, we applied this topology since it facilitates controlled experiments.

On the top of the physical network an application layer network is build. On each node of the network, peers can be instantiated. Peers are instantiated having one of the previously described types of personalities, which can be a client, service copy or resource agent. Depending on the particular experiment, a node may contain several agents or no agent at all. In the second case, the node acts as a router.

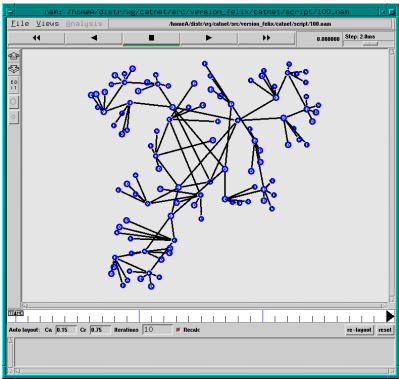


Fig. 4. Example of the network topology with approx. 100 nodes used in the experiments.

The relation of the experimental configuration to real world P2P systems is the following: A high value for the dynamics is interpreted to reflect the high level of connection and disconnection in P2P networks. A high level of node density represents the large number of machines with limited capability as found in P2P networks. In the simulations with high node density, we reduce the capacity of the resource agents, in order to represent small machines at the edge of the network. In the low node density scenario, on the other hand, the capacity of the service copy is increased such that the total amount of service available by the network is equal over all experiments.

In Table 1, the configuration of the experiments is detailed.

Table 1. Experiment description.

Input trace	- 2000 service requests generated randomly by 75 clients over a time interval of 100 s. - each request is for 2 service units. - each service has a duration of 5 s.
Node topol.	- 106 physical nodes
Node density	- 75 clients on the leaves of the physical network - different density of resource and service copy agents. Each Resource has one service copy associated. Exp 1A: low node density: 5 resources with capacity 60. Exp 1B: medium node density: 25 resources with capacity 12. Exp 1C: high node density: 75 resources with capacity 4.
Node dynamics	Dynamic behavior: On average 70% of the service copies are connected. Exp 2A: Service copies do not change its state (static network) Exp 2B: Each 200 ms every service copy can change its state (connected/disconnected) with a probability of 0.2. Exp 2C: Each 200 ms every service copy can change its state (connected/disconnected) with a probability of 0.4.

The main parameters we are interested to measure are the number of client requests the network is able to provide a service for in the different scenarios. The sce-

nario we are particularly interested in is the one with high node dynamics and high node density, as this configuration can be related to the conditions found in P2P networks.

4.2 Experimental Results

In Figure 5 the service provision rate of a network with *different node density* in a highly dynamic environment is shown (experiments 1A – C). It can be observed that the network using catallactic coordination achieves a higher service provision rates than the baseline system with a smooth decrease for increasing node density. In Figure 6 the service provision rate of a network with high node density in *different dynamic environment* is shown (experiments 2A – C). It can be observed that the service provision rate of the catallactic system is rather independent to the dynamics. The baseline system, on the other hand, decreases with increasing dynamics.

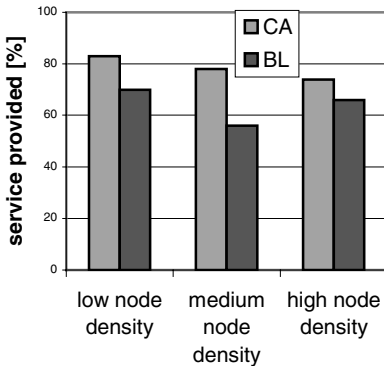


Fig. 5. Service provision in % in a highly dynamic network environment with different node density. CA = catallactic coordinated system. BL = baseline system.

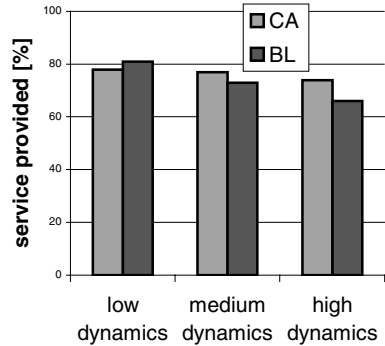


Fig. 6. Service provision in % in a high node density environment with different dynamics. CA = catallactic coordinated system. BL = baseline system.

Considering the achieved service provision rate, our experimental results indicate that service provision in networks with many small nodes in a highly dynamics environment could be coordinated successfully by the catallaxy paradigm.

Exploring additional parameters of the system has the potential to provide more insight in the behavior of such a complex system. Currently, we examine the influence of other parameters on the performance tendencies discovered so far. One of the drawbacks we found of the catallactic approach is the time needed to establish a service provision, which is high due to the negotiation protocol carried out by agents. Other parameters we study is how scale affects the performance of the system.

5 Conclusions

We have first indicated the need for an intelligent decision mechanism for service provision in future P2P networks, which shall cost-consciously decide from where the content should be retrieved and on which path it should travel. Such mechanism should not only achieve the functionality, but also reduce the overall cost to provide this functionality.

We have proposed the catallactic coordination as a decentralized economic approach for resource allocation in P2P networks. In this approach, the decisions of the peers are based on economic principles, being aware that resources like bandwidth, processing power and storage are limited. Autonomous peers negotiate with each other for service provision in order to optimize their own benefits.

With simulations we have investigated if service provision in P2P networks can be achieved by the catallactic coordination approach. We compared the obtained results with a centralized baseline approach. We observed in the experiments that the service provision in the catallactic coordination is rather independent of the dynamics of the network. The service provision capability of the baseline approach appears to be sensitive to the dynamics, reducing its performance in highly dynamic environments.

References

1. Buyya, R., D. Abramson, and J. Giddy, "A Case for Economy Grid Architecture for Service-Oriented Grid Computing". Proc. 10th IEEE International Heterogeneous Computing Workshop (HCW 2001). San Francisco, 2001.
2. CatNet project. "CATNET". <http://research.ac.upc.es/catnet>
3. Eymann, T. and B. Padovan, "The Catallaxy as a new Paradigm for the Design of Information Systems". Proceedings of The World Computer Congress 2000 of the International Federation for Information Processing. 2000.
4. Freenet. 2003. The Freenet home page. <http://www.freenetproject.org>
5. Gnutella. 2003. The Gnutella home page. <http://www.gnutella.com>
6. Grid Computing and Distributed Systems (GRIDS) Laboratory. "GRIDBUS Project". The University of Melbourne, Australia. <http://www.gridbus.org/>, 2002-11-28.
7. Hayek, F.A., W.W. Bartley, P.G. Klein, and B. Caldwell. The collected works of F.A. Hayek. University of Chicago Press, Chicago, 1989.
8. JavaSim Project. "JavaSim". Ohio State University EEng Dept. <http://www.javasim.org/>, 2 A.D.-11-29.
9. Napster. 2003. The Napster home page. <http://opennap.sourceforge.net/>
10. Pruitt, D.G. Negotiation Behavior. Academic Press, New York, 1981.
11. SETI@HOME. <http://setiathome.ssl.berkeley.edu/>

Incentives for Combatting Freeriding on P2P Networks

Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina

Stanford University

Abstract. We address the freerider problem on P2P networks. We first propose a specific participation metric, which we call a peer's EigenTrust score. We show that EigenTrust scores accurately capture several different participation criteria. We then propose an incentive scheme that may be used in conjunction with any numerical participation metric. We show that, when these incentives are used in conjunction with EigenTrust scores, they reward participatory peers but don't exclude less active peers.

1 Introduction

A notable problem with many of today's P2P file-sharing networks is the abundance of *freeriders* on the network – peers who take advantage of the network without contributing to it. Up to 70% of Gnutella clients do not share any files, and nearly 50% of all responses are returned by 1% of the peers [1]. This abundance of freeriders, and the load imbalance it creates, punishes those peers who do actively contribute to the network by forcing them to overuse their resources (e.g. bandwidth).

We address this problem by providing incentives for peers to make active contributions to the network. For example, active participants may get preference when they are competing for another peer's resources, such as bandwidth. Our approach is simple: each peer gets a certain participation score, and it receives rewards based on its participation score. The challenges here lie in how to structure incentives that reward active participants without completely excluding peers that are less active.

Previous work in this area has focused primarily on currency-based systems wherein peers gain currency for uploading files, and use currency when downloading files [2]. We take a different approach, rewarding peers with high participation scores with advanced services, such as faster download times or an increased view of the network. Our approach may be used in conjunction with currency-based approaches.

In this work, we describe a scoring system that accurately quantifies participation, even in the presence of malicious peers trying to subvert the system, and we propose some incentives, and show empirically that these incentives benefit participatory peers in a fair manner.

2 EigenTrust

In the incentives that we propose in this paper, we assume the existence of some scoring system that measures the relative participation levels of peers in the system. One useful metric is a peer's EigenTrust score [4]. EigenTrust was developed as a reputation metric

for P2P systems. In this work, we show that EigenTrust is also a good measure of a peer’s relative participation level.

To test how well a peer’s EigenTrust score reflects its participation level, we simulate a P2P network in the manner described in [3]. The simulator described in [3] is an event-driven simulator that proceeds by query cycles. At each query cycle, peers submit and respond to queries according to certain distributions over peers’ interests and the files peers share, and download files from peers who respond to their queries. Freeriders and malicious peers sharing inauthentic files are modeled as well as active participatory peers.

In Figure 1 we plot each peer in the network on a graph where the x-axis represents the EigenTrust score of the peer, and the y-axis represents the number of authentic uploads that the peer provides in a given timespan (15 query cycles). Notice that the EigenTrust score is correlated with the number of authentic uploads; those peers that provide many authentic uploads also have high EigenTrust score. The correlation coefficient is 0.97 indicating a close relationship between the number of authentic uploads and the EigenTrust score.

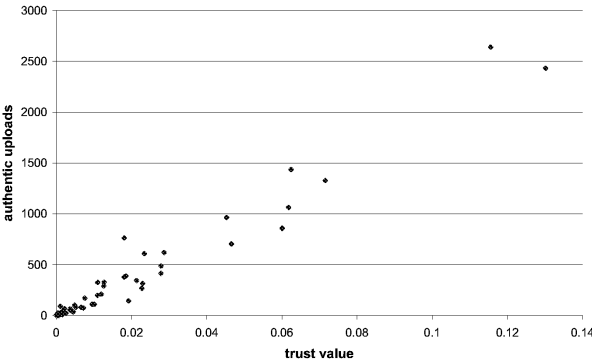


Fig. 1. Correlation between EigenTrust scores (y-axis) and the number of authentic uploads (x-axis). Each point represents a peer.

In order to give the reader further intuition on EigenTrust scores, we examine the following simple example users in our P2P simulation. These users, and their characteristics, are described below and summarized in Table 1.

Angela is an active participator and shares many popular files across many content categories. Bob is an average user who shares a moderate number of files, many of them popular, some of them unpopular, from a couple of content categories. Corcoran is an occasional user who shares a few files, most of them popular. David is an eccentric user, sharing many obscure files that few people want. Ebeniezer is a freerider, who doesn’t share any files. And Forster is a malicious user, who shares many corrupt files.

Based on these descriptions, Angela is the most active participator, followed by Bob, Corcoran, David, Ebeniezer, and Forster, in that order. Table 1 shows that EigenTrust

captures this ranking. Notice that the EigenTrust scores of Ebeniezer the freerider and Forster the malicious peer are both 0. One may wish for a scoring system in which the malicious peer gets a lower score than a freerider. However, it should be noted that, due to the ease of entry in P2P networks, giving a malicious peer a score below that of a freerider is not very effective. A malicious user with a poor score may simply create a new peer and enter the network as a new freerider.

Table 1. EigenTrust vs. other rankings for each of our seven sample users. Each of these users has the same uptime (20%), and the same number of connections (10).

	# Content Cat.	# Files	Popularity	ET Score	ET Rank
ANGELA	10	1,000	Typical	.02875	1
BOB	3	100	Typical	.00462	2
CORCORAN	3	50	Typical	.00188	3
DAVID	3	200	Unpopular	.00115	4
EBENIEZER	0	0	-	0	5 (tie)
FORSTER	10	2,000	Malicious	0	5 (tie)

We also examine the following pairs of peers, where the first member of each pair is more participatory in a different way. Again, we use these examples to give the reader intuition on the behavior of EigenTrust scores.

1. The first scenario involves Tim and Jim. Tim and Jim share the exact same files, except Tim is always online while Jim turns his computer off at night.
2. The second scenario involves Stan and Jan. Stan and Jan share files in the same content categories. However, Stan shares twice as many files as Jan.
3. The third scenario involves Stephanie and Bethany. Stephanie and Bethany share five files each from the same content category. However, Stephanie shares five very popular files, and Bethany shares five unpopular files.
4. The final scenario involves Mario and Luigi. Mario and Luigi share the same number of files, but Mario has a diverse collection, sharing files in many content categories, while Luigi has narrow interests, and shares files from only one content category.

Table 2 compares the EigenTrust scores of each of these pairs of peers, and shows that each of these characteristics we tested (uptime, number of shared files, popularity of shared files, and diversity of shared files) are reflected in the EigenTrust score of a peer. For example, Tim (who shares the same files as Jim, but has a greater uptime) has a greater number of authentic uploads, and a greater EigenTrust score, than Jim.

3 Incentives

Our goal is to provide incentives that reward participatory peers and punish freeriders and malicious peers. However, we do not want to punish freeriders so much that they are not able to download files and share them to become participators if they so choose.

Two ways to reward participatory peers is to award them faster download times, and grant them a wider view of the network. In this section, we propose two score-based

Table 2. Comparing Pairs of Peers

	# Cont.	Cat.	# Files	Popularity	# Links	Uptime	# Auth.	Uploads	ET Score
TIM	3		1000	Typical	10	92%	965		0.045
JIM	3		1000	Typical	10	35%	415		0.028
STAN	3		500	Typical	10	33%	325		0.011
JAN	3		1000	Typical	10	33%	379		0.018
STEPHANIE	1		5	5 Most Pop.	10	35%	15		0.00038
BETHANY	1		5	5 Least Pop.	10	35%	0		0
MARIO	5		5000	Typical	10	25%	267		0.023
LUIGI	1		5000	Typical	10	25%	81		0.0052

incentive schemes: a bandwidth incentive scheme and a TTL incentive scheme. Notice that these score-based schemes that we propose are completely general, and may be used with any scoring scheme that gives scores for which peers should be rewarded.

Bandwidth. The first incentive that we propose is to give active participators preference when there is competition for bandwidth. More specifically, we propose that, if peer i and peer j are simultaneously downloading from another peer k , then the bandwidth of peer k is divided between peer i and peer j according to their participation scores. So if Tim and Jim are simultaneously downloading from some peer k , then Tim will get $\frac{S_T}{S_T+S_J} * 100\%$ of peer k 's bandwidth (where S_T represents Tim's participation score, and S_J is Jim's participation score), and Jim will get $\frac{S_J}{S_T+S_J} * 100\%$ of peer k 's bandwidth.

This also works when more than 2 peers are simultaneously downloading from the same peer. The protocol for this incentive is given in Algorithm 1.

Peer i with available bandwidth b , assigns bandwidth to each peer j downloading from it as follows:

foreach peer j downloading from peer i **do**
 $bandwidth(\text{peer } j) = \frac{\text{score}(\text{peer } j)}{\sum_j \text{score}(\text{peer } j)} b;$
end

Algorithm 1: Bandwidth incentive

Notice that if a peer has a participation score of 0, it will get none of peer k 's bandwidth if it is competing against other peers for peer k 's bandwidth. However, this doesn't exclude that peer from the network, since it is able to download from peers that are not servicing other peers. We show in Section 4 that freeriders are not excluded from the network when this incentive is implemented.

TTL. Currently, Gnutella assigns each peer a time-to-live of 7 for each query. Our second incentive is to assign each peer a TTL based on its participation score, giving active participators a wider view of the network. There are many ways to do this, but one simple way would be to give each peer who has an above-average participation score a

high TTL (for example, 10), and to give each peer who has a below-average participation score a low TTL (for example, 5). The protocol for this incentive is given in Algorithm 2.

```

if  $score(peer\ j) > mean\_score$  then
     $TTL(peer\ j) = high\_ttl;$ 
else
     $TTL(peer\ j) = low\_ttl;$ 
end

```

Algorithm 2: TTL incentive

One problem here is that each peer must know the average participation score in the network, and explicitly computing this can be prohibitively costly in terms of message complexity, since it would require each peer to know the participation scores of every other peer in the network. However, if EigenTrust scores are used, this isn't a problem. Since the EigenTrust scores in the network sum to 1, the average EigenTrust score for any network is $1/n$, where n is the number of peers in the network. (We assume that a peer either knows or can approximate the number of peers in the network. If this is impractical for the given network, a peer can simply substitute its own EigenTrust score for $1/n$.) Therefore, each peer can compare its own EigenTrust score to $1/n$, and if its EigenTrust score is greater, the peer may issue a query with a TTL of 5. Otherwise, the peer may issue a query with a TTL of 3. A peer never needs to explicitly compute the average participation score in the network.

4 Experiments

We have shown in Section 2 that EigenTrust scores are a good measure of participation. Our task in this section is to show that the proposed incentives achieve their stated goals: to reward participatory peers with faster download times and a wider view of the network without completely excluding less active peers from the network. Again, we use our sample peers to give the reader intuition on how these incentives reward peers in the network.

Bandwidth. The bandwidth incentive aims to reward peers with faster downloads. To test this, we again examine the sample peers Angela et. al., and measure their average download speeds in our simulations with and without the bandwidth incentive implemented.

In Table 3, we show the average download speed for each of our sample users when the bandwidth incentive is implemented in conjunction with the EigenTrust scoring scheme, compared to the average download speed for each of our sample users when no incentive is implemented. To simulate a congested network, we employ the following congestion model: for each download that peer i begins to download from peer j , he competes for peer j 's bandwidth resources against somewhere between 0 to 4 other peers (chosen from a discrete uniform random distribution) who are also downloading files from peer j . Notice that, when the bandwidth incentive is implemented in this

simulation in conjunction with EigenTrust scores, active participators are compensated for their participation, but not at too great an expense to the less active peers.

Table 3. The average download speed (and percent bandwidth) for each peer using the bandwidth incentive in conjunction with the EigenTrust Score (left), and with no incentive (right)

	ET Score	Bandwidth Incentive	No Incentive
		% Bandwidth Download Speed	% Bandwidth Download Speed
ANGELA	.02875	.6549	.4566
BOB	.00462	.5307	.4566
CORCORAN	.00188	.5978	.4566
DAVID	.00115	.4737	.4566
EBENIEZER	0	.2	.4566
FORSTER	0	.2	.4566

TTL. Again, the important issue to investigate is whether this incentive compensates participators enough to be a useful incentive while giving nonparticipators enough resources so that they have the option of becoming participators if they so choose.

To do this, we simulate 15 query cycles with the TTL incentive scheme activated as in Algorithm 2. Since this is a small network (100 peers), we define the default TTL to be 4 (peers can reach 75 other peers in this network on average with a TTL of 4). In this case, we define *high-ttl* to be 5, and *low-ttl* to be 3.

For reporting our results, we split the peers into two groups: premium users (those peers with EigenTrust scores larger than the average EigenTrust score $\frac{1}{N}$), and moderate users (those peers with EigenTrust scores less than the average EigenTrust score $\frac{1}{N}$). Table 4 shows the number of premium users and moderate users and the average number of peers within their respective TTL range. Note that activating the TTL incentive scheme with these settings decreases the query load on the network while beefing up the service levels for premium users, which is a very desirable result: With the TTL scheme switched on, $77 * 27 + 23 * 98 = 4367$ query messages will be generated throughout the network when all peers issue a query. With the TTL scheme switched off, $100 * 75 = 7500$ messages will be generated in the same process. Also, notice that even the freeriders will not be excluded from the network, as they receive a TTL of 3 for their queries.

Table 4. The TTL, average number of peers reached (PR), and average number of responses per query (RPQ) for each peer using the TTL incentive in conjunction with the EigenTrust Score (left) and with no incentive (right).

	ET Score	TTL Incentive		No Incentive	
		#Peers	#Peers in TTL Range	#Peers	#Peers in TTL Range
MODERATE USERS	$< \frac{1}{N}$	77	27	74	75
PREMIUM USERS	$> \frac{1}{N}$	23	98	26	75

5 Conclusion

Two main results are presented in this paper: First, we show that a peer's EigenTrust score is a good participation metric according to three natural participation criteria. Second, we present two incentive protocols based on these scores, and show that they reward participatory peers without completely excluding non-participatory peers.

References

1. E. Adar and B. Huberman. Free riding on gnutella. *First Monday*, 5(10), October 2000.
2. P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge. Incentives for sharing in peer-to-peer networks. In *WELCOM01*, 2001.
3. S. Kamvar and M. Schlosser. Simulating a File-Sharing P2P Network. In *SemPGRID03*, 2003.
4. S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *WWW 2003*, 2003.

Topic 18

Demonstrations of Parallel and Distributed Computing

Ron Perrott, Henk Sips, Jarek Nabrzyski, and Michael Kropfberger
Topic Chairs

For many years Euro-Par has provided a forum for researchers from all over the world with the opportunity to present their research ideas and results to other colleagues; colleagues with similar interests as well as colleagues with related interests. This year for the first time Euro-par has enhanced this forum of international activities with a special session based solely on ‘Demonstrations’.

The rationale is based on the premise that it is important not only to research new ideas but also to show their usage. Demonstrations are a means of showing the advantages of the underlying technologies within real-use environments and based on proof-of-concept applications. This new session is therefore designed as an incentive and catalyst for researchers to show the feasibility and necessity of their work.

The focus of the reviewers was therefore not only on scientific contributions, but also on demonstrations which can illustrate proof of concepts and e.g. show the ease of use of certain new tools, or performance gains in comparison to commonly used algorithms or concepts.

The number of contributions submitted to the “Demo Session” was 12, consisting of demonstrations from a wide range of areas relevant to Euro-Par. The reviewers of the “Demo Session” selected 6 of the submitted papers which hopefully will add to the portfolio of Euro-Par activities and will encourage more submissions in future years.

Since this is the first time for this topic, the organisers would welcome feedback on the topic and suggestions for its promotion at future conferences. The reviewers are convinced of its necessity in the field and would like to thank all who submitted contributions.

Demonstration of P-GRADE Job-Mode for the Grid¹

P. Kacsuk¹, R. Lovas¹, J. Kovács¹, F. Szalai¹, G. Gombás¹, N. Podhorszki¹,
Á. Horváth², A. Horányi², I. Szeberényi³, T. Delaitre⁴, G. Terstyánszky⁴, and
A. Gourgoulis⁴

¹Computer and Automation Research Institute (MTA SZTAKI)

Hungarian Academy of Sciences, 1518 Budapest, P.O. Box 63, Hungary
{kacsuk, rlovas, smith, szalai, gombasg, pnorbert}@sztaki.hu

²Hungarian Meteorological Service, H-1525 Budapest, P. O. Box 38, Hungary
{horvath.a, horanyi.a}@met.hu

³Department of Control Engineering and Information Technology, Budapest University of
Technology and Economics, H-1117 Budapest, XI., Pázmány Péter sétány 1/D, Hungary
szebi@iit.bme.hu

⁴Centre for Parallel Computing (CPC), Cavendish School of Computer Science, University of
Westminster, 115 New Cavendish Street, London W1W 6UW, UK
{delaitt, terstyg, agourg}@cpc.wmin.ac.uk

Abstract. The P-GRADE job execution mode will be demonstrated on a small Grid containing 3 clusters from Budapest and London. The first demonstration illustrates the Grid execution of a parallel meteorology application. The parallel program will be on-line monitored remotely in the Grid and locally visualized on the submitting machine. The second demonstration will use a parallel traffic simulation program developed in P-GRADE to show the usage of the P-GRADE job mode for Grid execution. The parallel program will be checkpointed and migrated to another cluster of the Grid. On-line job and execution monitoring will be demonstrated.

1 Introduction

MTA SZTAKI has developed a parallel program development environment (P-GRADE) [1] for hiding the low-level PVM and MPI APIs from the users and providing a much higher-level graphical programming model with full support for parallel program debugging, monitoring, performance visualization, mapping, load-balancing, etc. P-GRADE is now extended towards the Grid and it stands for Parallel Grid Run-time and Application Development Environment. This environment enables the Grid application programmer to develop a parallel program that can be executed as a Grid job on any parallel site (like a supercomputer, or a cluster) of a Grid in a transparent way.

¹ This work was partially supported by the following grants: EU DataGrid IST-2000-25182, EU GridLab IST-2001-32133, Hungarian Scientific Research Fund (OTKA) no. T032226, Hungarian OMFB-02307/2000, and Hungarian IKTA-00075/2001.

2 Current Version of P-GRADE

P-GRADE supports the interactive development of a parallel program as well as their job execution mode. The interactive execution can be on a single processor system, on a supercomputer or on a cluster. The recommendation is that the editing, compiling and debugging activities should be on a single processor system while mapping, and performance monitoring should take place on parallel systems like supercomputers or clusters. If the program is correct and delivers the expected performance on parallel systems, the user can switch to the job mode. Here the user should specify the resource requirements, input files, output files and error file of the job. Then P-GRADE automatically generates the appropriate job from the parallel program developed in the interactive working mode. Currently two job types can be generated:

- Condor job
- PERL-GRID job

3 Condor Job Mode of P-GRADE

By integrating P-GRADE and Condor our goal was to provide a Grid-enabled runtime system for P-GRADE. In Condor mode, P-GRADE automatically constructs the necessary job description file containing the resource requirements of the parallel job. The mapping function of P-GRADE was changed according to the Condor needs. In Condor the user can define machine classes from which Condor can reserve as many machines as it is defined by the user in the job description file. When the user generates Condor job under P-GRADE this is the only Condor-related task. P-GRADE supports this activity by offering a default set of machine classes for the user.

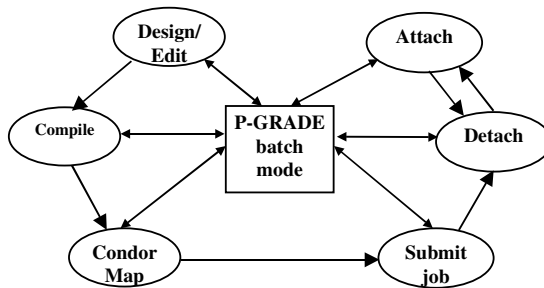


Fig. 1. P-GRADE services in job mode

Finally, after submitting the Condor job the user can detach P-GRADE from the job. It means that the job does not need the supervision of P-GRADE when it is executed in the Grid (for that purpose we use Condor). Meanwhile the P-GRADE generated

Condor job is running in the Grid, P-GRADE can be turned off or it can be used for developing other parallel applications. However, at any time the user can attach again P-GRADE to the job to watch the current status and results of the job. The program development and execution mechanism of P-GRADE for the Grid is shown in Fig. 1.

4 PERL-GRID Job Mode of P-GRADE

The Condor job mode can be used only if the submit machine is part of a Condor pool. However, such a restriction is too strong in the Grid where there are many Condor pools as potential resources and these should be accessed from the submit machine. This is the case, for example, in the Hungarian ClusterGrid where the Grid itself is a collection of Condor pools (clusters) but the user's submit machine is typically not part of the ClusterGrid.

In such situation some new functionalities should be provided for the user. In order to provide these missing functionalities we have created a thin layer, called PERL-GRID, between P-GRADE and Condor with the following tasks:

- The selection of the high-performance computing site (Condor pool) is the task of a Grid resource broker. Currently a random selection algorithm is applied by PERL-GRID. (This will be replaced by a Grid resource broker in the future.)
- PERL-GRID takes care of contacting the selected site and executing the mutual authentication based on the SSH technology.
- PERL-GRID also takes care of file staging by transferring the input files and the code file into a temporary directory at the reserved Grid site. Finally, it passes the job to the local job manager. Currently it is Condor but soon others, like SGE will be supported.
- If the selected Grid site becomes overloaded, the whole parallel application is check-pointed and migrated to another Grid site by PERL-GRID.

An automatic parallel check-point mechanism has been elaborated for parallel programs developed under P-GRADE. By this check-point mechanism, all those features of Condor that are provided for sequential jobs in the Standard Universe (job migration, fault-tolerance, guaranteed execution) are supported even for parallel jobs under P-GRADE.

Running a parallel job in the Grid requires the on-line monitoring and steering possibility of that job. Using the Grid resource and job monitoring techniques developed in the GridLab project we can on-line monitor and visualize the status of P-GRADE jobs. More than that, the processes of a parallel application can be monitored by integrating the GRM monitor of P-GRADE either with the GridLab Grid monitor or with the R-GMA monitoring infrastructure of the DataGrid project. In either case using PROVE of P-GRADE enables the on-line visualization of the process interactions of a parallel application running anywhere in the Grid.

5 Description of the Demonstration

All the features of the PERL-GRID job mode of P-GRADE described in the previous section will be demonstrated by two scenarios. During the demonstration we will use a demonstration Grid consisting of 3 clusters: SZTAKI cluster in Budapest, cluster of the Budapest University of Technology and Economics, cluster of the University of Westminster in London.

5.1 Scenario 1

Scenario 1 will demonstrate the Grid execution of the MEANDER nowcast program package of the Hungarian Meteorology Service. The goal of the MEANDER package is to provide ultra-short range (up to 1 hour) weather forecast of dangerous situations like storm and fog on a high resolution regular mesh (10km -> 1km). To achieve this goal members of OMSZ and SZTAKI have jointly parallelised the six most time consuming calculation module of MEANDER by P-GRADE [2].

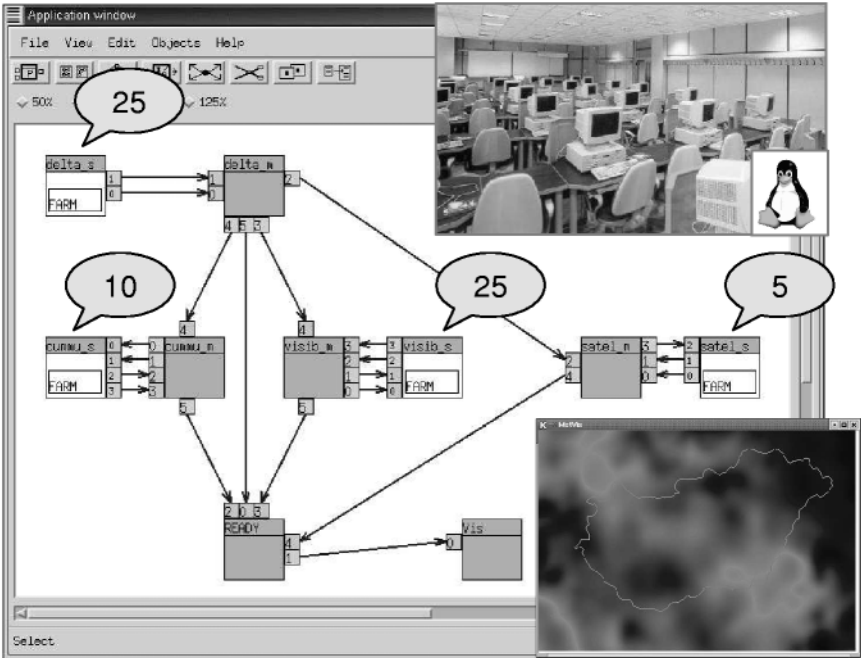


Fig. 2. The P-GRADE demonstration version of the MEANDER program

We will use a simplified version of MEANDER containing only 4 algorithms for the demonstration. The P-GRADE view of this MEANDER version is shown in Fig. 2. It is clear on Fig. 2 that the four algorithms are realized by the processor farm concept. The numbers in the clouds represent the number of processors to be used in the dem-

onstration. It can also be seen that these algorithms are connected like a workflow. First the delta algorithm should be executed, then in the second phase, the other three algorithms can be executed in parallel. At that time 40 processors will be used in parallel. Finally, a visualization process is applied to draw the weather map shown in the right bottom of Fig. 2.

The parallel job will be generated by P-GRADE and passed to PERL-GRID which will transfer the executable code to the 58-processor Linux cluster of SZTAKI (shown in the right top of the picture) to execute the job. Then PERL-GRID collects the necessary meteorology database input file from the Hungarian Meteorology Service and passes the job with all the files to Condor at the SZTAKI cluster. Condor takes care of the parallel execution of the job at SZTAKI. When monitoring is requested, PERL-GRID delivers the local monitor code, too and the collected trace file is sent back and visualized by PROVE on the submit machine at Klagenfurt whenever we request a trace collection by PROVE. The final trace file visualization picture is shown in Fig. 3.

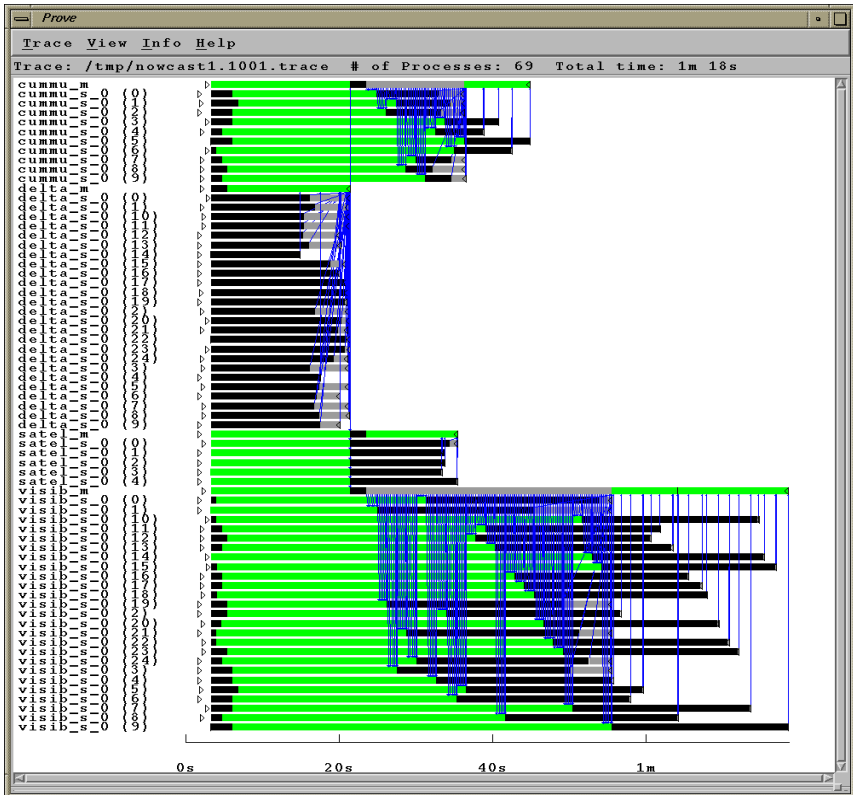


Fig. 3. Process space-time diagram of the MEANDER program

The picture clearly shows that first the delta algorithm is running on 25 processors and when it is finished, it triggers the execution of the other three algorithms that are executed simultaneously. When the whole job is finished, PERL-GRID takes care of

transferring the result file back to Klagenfurt and removing the temporary directory it created for the job at the SZTAKI cluster. Finally, the weather-forecast map of Hungary will be displayed at Klagenfurt.

5.2 Scenario 2

Scenario 2 will demonstrate the Grid execution of an urban traffic simulation program developed at the University of Westminster [3]. The program will be started as a PERL-GRID job under P-GRADE. PERL-GRID will select one of the three clusters of the demonstration Grid and transfers the necessary files to the selected cluster where it passes the job to Condor. When the program starts we will increase the load of the selected cluster by starting other higher priority jobs there. The application will be check-pointed and PERL-GRID will select another cluster and migrates the job there and passes it to the local Condor job manager. Using the GridLab Grid monitor system we will on-line monitor and visualize the job status as well as its processes and their interactions by PROVE.

6 Conclusions

P-GRADE provides a high-level graphical environment to develop parallel applications both for parallel systems and the Grid. One of the main advantages of P-GRADE is that the user has not to learn the different APIs for parallel systems and the Grid, simply by using the same environment will result in a parallel application transparently applicable either for supercomputers, clusters or the Grid. The current version of P-GRADE supports the interactive execution of parallel programs as well as the creation of a Condor or PERL-GRID job to execute the parallel program in the Grid. Remote monitoring and performance visualization of Grid applications are also supported by P-GRADE.

P-GRADE will be used in the Hungarian ClusterGrid that connects the Condor pools of the Hungarian higher educational institutions into a high-performance, high-throughput Grid system. Though the Grid execution mode of P-GRADE is currently strongly connected to Condor, the use of PERL-GRID enables the easy connection to other local job managers like SGE and PBS. This work is planned as part of the Hungarian Grid activities.

References

1. P. Kacsuk: Visual Parallel Programming on SGI Machines, Invited paper, Proc. of the SGI Users' Conference, Krakow, Poland, pp. 37–56, 2000
2. R. Lovas, et al: Application of P-GRADE Development Environment in Meteorology, Proc. of DAPSYS'2002, Linz, pp. 30–37, 2002
3. Gourgoulis, et al: Using Clusters for Traffic Simulation, Proc. of Mipro'2003, Opatija, 2003

Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster

J r mie Allard, Bruno Raffin, and Florence Zara

Laboratoire Informatique et Distribution
Projet APACHE ID-IMAG
CNRS – INPG – INRIA - UJF
38330 Montbonnot, France

Abstract. Recent developments make it possible for PC clusters to drive multi-display visualization environments. This paper shows how we coupled parallel codes with distributed 3D graphics rendering, to enable interactions with complex simulations in multi-display environments.

1 Introduction

Visualization appears as an efficient way to analyze results of complex simulations. Immersive environments, like CAVEs [1], enhance the visualization experience. They provide a high resolution and large surface display created by assembling multiple video projectors. Interactivity and stereoscopic visualization further improve the possibility of these workspaces. These environments are classically powered by dedicated graphics supercomputers, like SGI Onyx machines.

Today, the anatomy of supercomputing is quickly and deeply changing. Clusters of commodity components are becoming the leading choice architecture. They are scalable and modular with a high performance/price ratio. Clusters have proved efficient for classical (non interactive) intensive computations. Recently the availability of low cost high performance graphics cards have foster researches to use these architectures to drive immersive environments [2,3]. The first goal was to harness the power of multiple graphics cards distributed on different PCs. But the scalability and performance of PC clusters allow to go beyond only distributed graphics rendering. While some cluster nodes have graphics cards to power the immersive environment, complex simulations can take advantage of extra nodes to decrease their execution time and reach an update rate suitable for interactivity.

The demo we propose is based on two interactive applications running on a PC cluster driving a multi-display environment: a cloth simulation [4] and a fluid simulation [5]. Both applications involve a parallel simulation coupled with a distributed graphics rendering.

2 Softwares and Environments

The applications were developed on the following open source softwares:

- **CLIC:** Clic [6] is a Linux distribution dedicated to PC clusters. It includes several pre-configured free software tools to ease cluster installation, administration and monitoring.
- **Net Juggler:** Net Juggler [3] enables an application to use the power of multiple graphics cards distributed on different PCs. It parallelizes graphics rendering computations for multi-display environments by replicating the application on each node and using MPI to ensure copies are coherent and displayed images are synchronized. Net Juggler is based on VR Juggler [7], a platform for virtual reality applications.
- **Athapascan:** Athapascan [8,9] is a parallel programming language designed to develop portable applications and to enable efficient executions on PC clusters. An Athapascan program consists of a description of parallel tasks communicating through a shared memory. At runtime, Athapascan builds a data flow graph from the data dependencies between tasks. Based on this graph and on a cluster description, it controls task scheduling and data distribution.

3 Coupling Parallel Simulation with Parallel Rendering

We present two applications, a fluid and a cloth simulation. Both consist of a parallel simulation coupled with a distributed 3D graphics rendering. They use two different paradigms to parallelize the simulation part: the fluid simulation uses MPI while the cloth simulation uses Athapascan.

3.1 First Case Study: Fluid Simulation

The fluid simulation is based on the implementation of the Navier-Stokes equation solver proposed by Stam [10]. The space is discretized on a grid of cells, where the fluid can flow. Each cell holds a fluid velocity vector and a scalar fluid density characterizing the fluid present in a given cell. At each time step, the solver updates these data. These computations require simple matrix computations, a conjugate gradient and a Poisson solver.

The solver is implemented with PETSC [11], a mathematical library distributing matrix-based operations on the cluster using MPI. As Net Juggler already sets up an MPI environment, it is easy to integrate PETSC in Net Juggler. More details can be found in [5].

The simulation involves 2 fluids flowing on a 2D grid (Fig. 1). The user can interactively disturb the fluid flow with a pointer that applies a force on the fluids. He can also add or remove obstacles. The simulation and the visualization are executed synchronously.

With the solver executed on one node, the obtained frame rate is about 8 frames per second (fps). When the solver is parallelized on four nodes, the frame rate is close to 20 fps.

3.2 Second Case Study: Cloth Simulation

This application uses two different parallelization paradigms. The simulation is parallelized with the parallel programming environment Athapascan [9]. Rendering computations are distributed with Net Juggler [3] on graphics nodes.

The cloth is modeled as a triangular mesh of particles linked up by springs. The object is block partitioned using Athapascan tasks [4]. At each time step, a particle position is computed based on two steps of a Newton's equation integration. Data mapping, task scheduling and communications are managed at runtime by Athapascan.

At the end of each time step, computed positions are sent in a socket to Net Juggler for graphics rendering (Fig. 2).

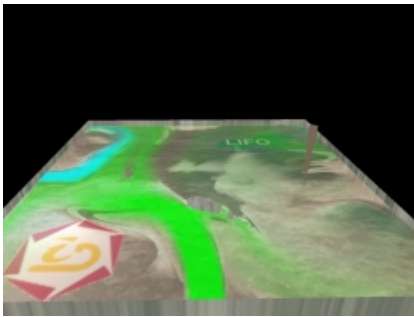


Fig. 1. Interactive simulation of two 2D fluids.

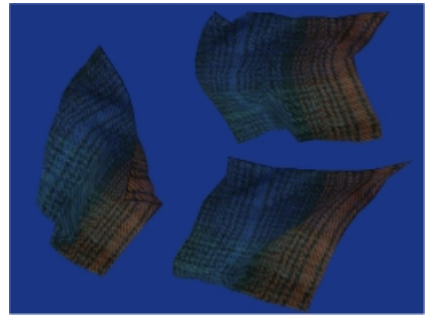


Fig. 2. Piece of cloth of 100 particles with two or one corner(s) fastened.

4 Demo Setup

The demo setup consists of a 6 nodes cluster using a fast Ethernet network and driving 3 video-projectors. We will alternatively show the fluid simulation demo and the cloth simulation demo. The attendees will be able to evaluate by themselves the quality and performance of commodity graphics cards, the synchronization level provided by Net Juggler to create a seamless 3 projector display, the interactivity level reached on complex simulations when parallelized. These demo will also give attendees the opportunity to have some insights about the CLIC distribution.

Short videos of the demos are available at

<http://netjuggler.sourceforge.net/Gallery.php>.

5 Conclusion

Interactive simulation execution coupled with advanced visualization environments can greatly help to analyze and understand complex data. We showed

in this paper how PC clusters can be used to execute such applications. The fluid and cloth simulations take advantage of two levels of parallelism to enable the interactive execution of complex simulations in a multi-display environment. The first one distributes graphics rendering using Net Juggler, and the second one parallelizes the core of the simulation using MPI or Athapascan.

Future works will focus on developing solutions for automatic coupling, targeting executions on large clusters and grid computing infrastructures. Interactive processing and visualization of large data sets will also be considered.

References

1. Cruz-Neira, C., Sandin, D.J., DeFanti, T.A., Kenyon, R.V., Hart, J.C.: The Cave Audio Visual Experience Automatic Virtual Environment. *Communication of the ACM* **35** (1992) 64–72
2. Samanta, R., Funkhouser, T., Li, K., Singh, J.P.: Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In: *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. (2000)
3. Allard, J., Gouranton, V., Lecointre, L., Melin, E., Raffin, B.: Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In: *IEEE VR*, Orlando, USA (2002) 275–276
4. Zara, F., Faure, F., Vincent, J.M.: Physical cloth simulation on a PC cluster. In D. Bartz, X.P., Reinhard, E., eds.: *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany (2002)
5. Allard, J., Gouranton, V., Melin, E., Raffin, B.: Parallelizing pre-rendering computations on a net juggler PC cluster. In: *Immersive Projection Technology Symposium*, Orlando, USA (2002)
6. MandrakeSoft, ID, L., Bull: (The clic linux cluster distribution) <http://clic.mandrakesoft.com>.
7. Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In: *IEEE VR 2001*, Yokohama, Japan (2001)
8. Roch, J.L., *et al.*: Athapascan: Api for asynchronous parallel programming. Technical report, INRIA Rhône-Alpes, projet APACHE (2003)
9. Galilée, F., Roch, J.L., Cavalheiro, G., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In *IEEE*, ed.: *Pact'98*, Paris, France (1998)
10. Stam, J.: Stable Fluids. In: *SIGGRAPH 99 Conference Proceedings*. (1999) 121–128
11. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: PETSc 2.0 Users Manual. Technical Report ANL-95/11 - Revision 2.0.29, Argonne National Laboratory (2000)

Kerrighed: A Single System Image Cluster Operating System for High Performance Computing

Christine Morin¹, Renaud Lottiaux¹, Geoffroy Vallée², Pascal Gallard¹,
Gaël Utard¹, R. Badrinath^{1,3}, and Louis Rilling⁴

¹ IRISA/INRIA – PARIS project-team

² EDF

³ IIT Kharagpur

⁴ ENS-Cachan, antenne de Bretagne

Abstract. Kerrighed is a single system image operating system for clusters. Kerrighed aims at combining high performance, high availability and ease of use and programming. Kerrighed implements a set of global resource management services that aim at making resource distribution transparent to the applications, at managing resource sharing in and between applications and at taking benefit of the whole cluster resources for demanding applications. Kerrighed is implemented as a set of modules extending the Linux kernel. Legacy multi-threaded applications and message-passing based applications developed for an SMP PC running Linux can be executed without re-compilation on a Kerrighed cluster. The proposed demonstration presents a prototype of Kerrighed running on a cluster of four portable PCs. It shows the main features of Kerrighed in global memory, process and stream management by running multi-threaded and MPI applications on top of Kerrighed.

1 Topics

We propose to demonstrate a prototype of Kerrighed¹, a single system image operating system for clusters. The research work presented in this prototype has been carried out in the PARIS project-team at IRISA/INRIA and relates to several topics:

Topic 01: Support Tools and Environments. A single system image operating system like Kerrighed can be considered as a tool or an environment to conveniently and efficiently execute parallel applications on clusters.

Topic 03: Scheduling and Load Balancing. Kerrighed's prototype implements a configurable global scheduler to balance the load on cluster nodes. Kerrighed's global scheduler is based on novel efficient process management mechanisms.

¹ Kerrighed (previously named Gobelins) has been filed as a community trademark.

Topic 09: Distributed Algorithms. The prototype of Kerrighed implements the container concept for efficient global memory management. Based on containers, Kerrighed provides shared virtual memory segments to threads executing on different cluster nodes, a cooperative file cache and remote memory paging.

Topic 14: Routing and Communication in Interconnection Networks. Kerrighed's prototype implements a portable high performance reliable communication system providing a kernel level interface for the implementation of Kerrighed's distributed system services. The standard communication interface used by communicating Linux processes (pipe, sockets, etc. . .) is also available in Kerrighed on top of this communication system. In Kerrighed, communicating processes can be transparently migrated and can still efficiently communicate with other processes after migration.

2 Originality of the Demonstrated Prototype

The main originality of the prototype we propose to present is that it is a single system image operating system for clusters. Kerrighed provides the same interface as the standard operating system running on each cluster node. The current prototype has been implemented as a set of Linux modules and a small patch to the kernel (less than 200 lines of code, mainly for exporting kernel functions). Hence, existing applications that have been developed for an SMP PC running Linux can be executed on a cluster without even being recompiled. Unix applications can be easily ported on Kerrighed by recompiling them. So, sequential processes requiring huge processing and/or memory resources, multi-threaded applications and parallel applications based on message passing can be easily and efficiently executed on a cluster running Kerrighed. Unlike other systems Kerrighed supports efficiently both the message-passing and the shared memory programming models on clusters.

There are very few other research projects working on the design and implementation of a single system image operating system for clusters. Mosix [3] and Genesis [1] are examples of operating systems targeting the single system image properties. Mosix offers processor load balancing on top of a cluster. However, it does not support memory sharing between threads or processes executing on different cluster nodes unlike Kerrighed. A process which has migrated in Mosix cannot communicate efficiently with other processes after migration as messages are forwarded to it by the node on which it was created. Moreover, processes in Mosix cannot take benefit of the local file cache of their current execution node after migration, leading to poor performance for file accesses. Kerrighed has none of these drawbacks as it preserves direct communications between processes even after migration and as it implements a cooperative file cache.

Genesis is a single system image operating system for clusters which, in contrast to Kerrighed, has been developed from scratch and is based on the micro-kernel technology. As Kerrighed it supports both the shared memory and the message-passing programming paradigms. However, it implements a distributed

shared memory system which does not provide a standard interface. Thus, legacy shared memory parallel applications (such as Posix multi-threaded applications) cannot be executed on Genesis without a substantial porting effort. To our knowledge, Genesis only supports PVM for message-passing parallel applications.

3 Mechanisms for Demonstrating the Prototype

We will show a cluster of five portable PCs interconnected by a Fast Ethernet network. Four nodes of the cluster will run Linux and the Kerrighed modules. On this cluster, we will show the execution of instances of two parallel applications, one being a multi-threaded application, Volrend, the other one being an MPI application. Volrend is a multi-threaded application which implements ray-tracing to display 3D images. The fifth node executes a standard Linux system and is used to display the applications' results and to show some performance values (processor load, memory occupation and network throughput for instance) as well as the location of all applications' threads and processes on the cluster's nodes (graphical interface with different colors for the different applications).

Kerrighed's features that are demonstrated are the container concept for memory sharing between threads (Volrend), the process migration mechanism which allows to migrate any process or thread even if it communicates with other threads or processes by shared memory (Volrend) or by message-passing (Mandelbrot fractal), the configurable global scheduler in which the scheduling policy can be changed without stopping Kerrighed and the applications currently running on top of it.

Note that a demonstration of an initial prototype of Kerrighed (not including process management and global scheduling features) has been successfully demonstrated at Supercomputing in Baltimore (USA) in November 2002 and at Linux Expo in Paris in February 2003 on the same cluster of portable PCs. The demonstration is stand-alone and does not require an Internet access.

4 Scientific Content

Our goal in designing Kerrighed is to combine high performance, high availability and ease of use [5]. Kerrighed performs global resource management. With a set of global resource management services, Kerrighed aims to make resource distribution transparent to the applications, to manage resource sharing in and between applications and to take benefit of the whole cluster resources for demanding applications. These services are distributed and each of them is in charge of the global management of a particular resource (memory, processor, disk).

Global memory management in Kerrighed is based on the container concept [4]. In modern operating systems, block devices and memory are managed on page granularity. A container is a software object allowing to store and share pages cluster wide. Containers are managed in a software layer in between high level services of the node's standard operating system (virtual memory, virtual

file system) and its device managers. Thus, the standard user interface is kept while Kerrighed offers virtual shared memory segments, a cooperative file cache and remote paging, all based on containers.

Kerrighed's global process management service consists in a global scheduler based on efficient process state management mechanisms. As characteristics of the workloads executed on clusters may differ from one environment to another, Kerrighed's global scheduler has been designed to allow the specialization of the scheduling policy [6]. Changing the global scheduling policy can be done dynamically without rebooting the cluster nodes. The description of the scheduler to be used in a particular environment need only be described using XML configuration files. Kerrighed implements a basic primitive to extract the state of a process from the kernel. This primitive is exploited to perform process duplication, migration and check-pointing [7,2]. Processes exchanging messages can also be efficiently migrated as Kerrighed implements migrable sockets that ensure a direct communication between two communicating processes even after migration of one of them. These sockets are implemented on top of a dynamic stream service that allows processes to attach or detach from a stream.

Kerrighed is an open source software distributed under the GNU GPL license. Scientific publications and a first (demonstration) version can be downloaded at <http://www.kerrighed.org>.

References

1. M.J. Hobbs A.M. Goscinski and J. Silock. Genesis : The operating system managing parallelism and providing single system image on cluster. Technical Report TR C00/03, School of Computing and Mathematics, Deakin University, February 2000.
2. Ramamurthy Badrinath and Christine Morin. Common mechanisms for supporting fault tolerance in DSM and message passing systems. Rapport de recherche 4613, INRIA, November 2002.
3. Amnon Barak, Shai Geday, and Richard G. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.
4. Renaud Lottiaux and Christine Morin. Containers: A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 66–73, Brisbane, Australia, May 2001.
5. Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. In *ICA3PP*, 2002.
6. Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, and Louis Rilling. A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In *International Parallel and Distributed Processing Symposium*, April 2003.
7. Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, Ivan Dutka Malen, and Renaud Lottiaux. Process migration based on Gobelins distributed shared memory. In *Proc. of the workshop on Distributed Shared Memory (DSM'02) in CCGRID 2002*, pages 325–330, Berlin, Allemagne, May 2002. IEEE Computer Society.

ASSIST Demo: A High Level, High Performance, Portable, Structured Parallel Programming Environment at Work^{*}

M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo

Dept. of Computer Science – University of Pisa – Viale Buonarroti 2, 56127 Pisa

Abstract. This work summarizes the possibilities offered by parallel programming environment ASSIST by outlining some of the features that will be demonstrated at the conference *demo session*. We'll substantially show how this environment can be deployed on a Linux workstation network/cluster, how applications can be compiled and run using ASSIST and eventually, we'll discuss some ASSIST scalability and performance features. We'll also outline how the ASSIST environment can be used to target GRID architectures.

Keywords. Structured parallel programming, skeletons, coordination languages.

1 Demo Background

ASSIST (*A Software development System based on Integrated Skeleton Technology*) is a parallel programming environment based on skeleton and coordination language technology [8,9,3,2]. ASSIST provides the user/programmers with a structured parallel programming language (ASSISTcl), an integrated set of compiling tools (**astCC**) and a portable run time (the actual runtime *CLAM*, and the loader/runner **assistrun**). ASSIST is based on both skeleton and coordination languages technology, and comes after some other different experiences of our group related to skeleton based parallel programming [5,4]. It builds on the experience gained in these projects.

The main goals in the design of ASSIST have been: high level programmability, rapid prototyping and suitability for complex multidisciplinary applications; functional and performance portability across a range of different target architectures; software reuse and interoperability. These goals have been achieved by taking a number of design choices and using several different implementation techniques.

^{*} This work has been partially supported by the National Research Council Coordinated Project CNRC0014B3 “Development environment for multiplatform and multilanguage high-performance applications, based upon the objects model and structured parallel programming (Agenzia2000)” and ASI-PQE2000 “Earth observation application development with High Performance Computing Tools” projects

Programmability: the coordination language allows programmers to express both standard skeleton/coordination patterns as well as more complex parallelism exploitation patterns. This because a new¹, general purpose, highly configurable parallelism exploitation pattern has been included in the language (the **parmod** one) and because completely general graphs of **parmods** can be used to describe the parallel structure of applications. Furthermore, **ASSISTcl** allows a controlled usage of *external objects* (e.g. existing, possibly parallel, libraries) after programmer requests

Performance: **ASSIST** environment design is highly layered. The source code is first translated into an intermediate “task code”. The task code, in turn, is compiled to an abstract machine built on top of **ACE** (*Adaptive Coordination Environment* [1]) and **AssistLib**. **ACE** provides communication and process framework. **AssistLib** is a C++ library implementing specific mechanisms needed to implement task code on top of **ACE**. The whole compiler design is based on OO design pattern technology, thus allowing easily replacement of compiler parts as well as introduction of new features without affecting the **ASSIST** support overall design. In addition, the whole compile process and both the task code and the **AssistLib** level have been carefully optimized in order to avoid any kind of bottlenecks as well as of overhead sources.

Interoperability and software reuse: all the sequential portions of code needed to instantiate **parmod** parallelism exploitation patterns within an **ASSISTcl** program can be written in any of the C, C++ or F77 languages, and the details a programmer has usually to deal with when using such a programming language mix are handled by the **ASSISTcl** compiling tools (and further languages, such as Java, are being taken into account). Furthermore, sequential portions of code can invoke external CORBA object services, and a whole **ASSISTcl** application could be automatically wrapped into a CORBA object (IDL code generation is automatic), in such a way that its “parallel services” could be invoked from outside the **ASSIST** world. Existing, highly optimized, parallel scientific libraries can be integrated into **ASSISTcl** applications in such a way that they look like “normal” **ASSISTcl parmods** [7]

Our group is currently working to **ASSIST** enhancement and evolution within several National research projects. Within a couple of National Research Council Strategy projects the **ASSIST** environment is currently being migrated on GRIDs [6]. Further **ASSIST** environment enhancements are planned within another, three year, large, National Research Council Project (*GRID.it*). The interested reader can refer to different papers describing **ASSIST**: [8,9,2,7,6,3].

2 ASSIST Framework Setup

ASSIST currently runs on cluster or networks of workstations. In order to install **ASSIST** the user needs to install some separate, public domain software packages. In particular, **ACE**, the *Adaptive Communication Environment* is needed, as well

¹ with respect to previously developed skeleton based programming environments

as DVSA, the *Distributed Virtual Shared Areas library*. The former providing the basic functions of ASSIST abstract machine, the latter providing the support for data sharing across network connected processing elements. Both libraries assume a POSIX TCP/IP framework. ACE runs on both Linux and Windows or Mac (OS/X) boxes. DVSA originally runs on top of Linux, and is currently begin tested on the other environments by our research group.

Once these libraries have been configured, the installation of ASSIST is a matter of a couple of **make** commands. The correct installation of the ASSIST package provides user with the **astCC** and **assistrun** commands, i.e. the compiler and run commands, respectively.

Compiling ASSIST programs. Once an ASSIST program has been produced, using any available editor, it can be compiled using the **astCC** command. The compiler basically produces a set of C++ “object code” files, a kind of configuration file storing an XML representation of the resources needed to run the program, the **makefiles** needed to compile actual object code out of these sources, and eventually executes a sort of **make all** completing the generation of actual object code. Several parameters of the **astCC** command allow, for instance, the source files to be kept after the production of object code, in such a way the programmer may intervene directly at the “task code” or **AssistLib** level, i.e. at the level of the intermediate abstract machine, if needed.

Running ASSIST programs. In order to run an ASSIST executable, the user must basically perform two steps: first, a *CLAM* (*Coordination Language Abstract Machine*, the one built out of ACE and **AssistLib**, basically) instance must be run on the processing elements of the target architecture. This can be done running by hand the *CLAM* process onto every node of the target architecture. ASSIST provides scripts that make this process automatic, once the (IP) names of the nodes are known. This step can be performed once and for all, as *CLAM* is an execution server and it can be invoked multiple times, with different object codes. Every time, *CLAM* loads object code, configuration info and provides to actually run the ASSIST object code. Second, we must issue an **assistrun** command. The command accepts as a parameter the XML configuration file produced by the compiler² and consequently properly configures *CLAM* and actually starts computation.

3 Programmability

The time needed to develop running, scalable ASSISTcl programs is significantly smaller than the time needed to develop equivalent (both in the functional and in the performance sense) applications with different, more classical parallel programming tools, such as MPI, for instance. The programmer has handy ways to express simple as well as complex parallelism exploitation patterns. All the

² and possibly manipulated by the programmer/user

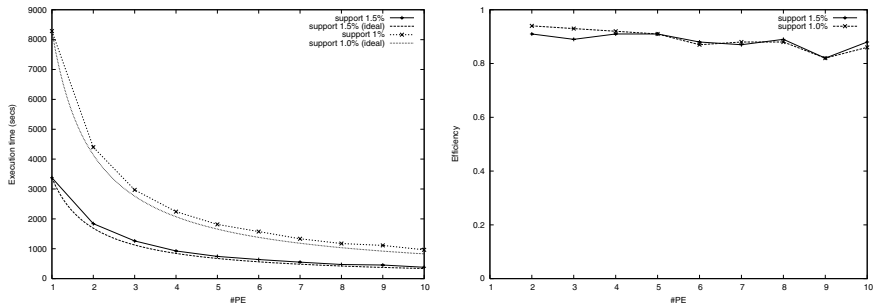


Fig. 1. Scalability and efficiency of Apriori data mining application

details needed to implement parallelism exploitation are handled by the compiling tools. Therefore, on the one hand the programmers may write the parallel structure of the application very quickly, while on the other hand performance exploitation is in charge of the tools, and again this consistently shortens the application development time.

4 Performance Results

We experimented different synthetic benchmarks as well as complete applications written in ASSISTcl. Typical performance numbers got out of Intel/Linux cluster are depicted in Figure 1 (left). Provided that computational grain is medium to coarse grain, ASSIST demonstrates good speedup and efficiency. The Figure plots values achieved running a data mining application exploiting an “a priori” algorithm.³ In the Apriori code execution, efficiency is constantly more than 80%, as shown in the Figure 1 (right).

5 Interoperability

ASSIST programs can invoke external services/code using several mechanism including typical CORBA ones. In particular, any portion of code included in an ASSIST program may call external CORBA objects methods and a whole ASSISTcl program can be wrapped into a CORBA object whose services/methods⁴ can be called from elsewhere. To demonstrate this feature, we prepared a Nbody program whose computational intensive part is performed in an ASSISTcl program but visualizing results accessing an X display via CORBA. The graphical output of the program is in Figure 2. The Nbody program is actually a code implementing the naive, n^2 algorithm, as the goal there was only to demonstrate

³ The parameters of these runs can be summarized as follows: #DBtransactions = 1236000, average transaction len = 30, #items = 1000. Large item-sets: #patterns = 2000, average pattern len = 10, correlation between consecutive patterns = 0.5, average confidence in a rule = 0.75, variation in the confidence = 0.1

⁴ computation of the parallel program onto a given input data set, actually

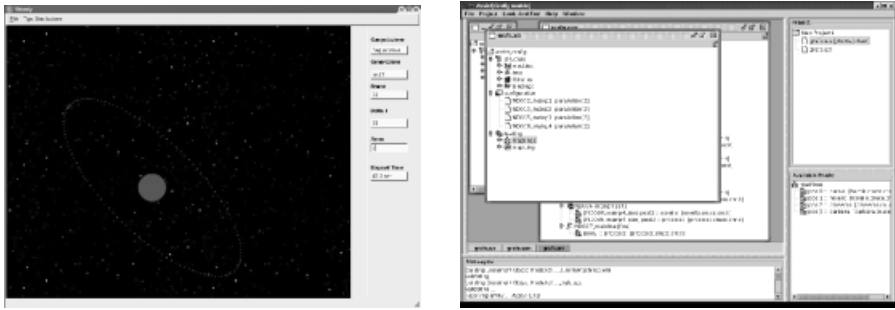


Fig. 2. N-body ASSISTcl program interacting with graphic display via CORBA (left) and assistConf configuration tools (right)

interoperability via CORBA, even in case of small granularity ops (i.e. graphic display operations).

6 Heterogeneous Target Architecture & GRID

We are currently adapting the ASSISTcl compiling tools to produce object code for heterogeneous architectures (networks). Due to the structured layered implementation of the compiling tools of the ASSIST framework, in order to address heterogeneous architectures we simply have to perform two steps: first, we need to activate the possibility of delivering external data representation messages between hosts. ACE has a full support for such “processor neutral” data representation but at the moment the marshaling and unmarshaling routines do not use such feature. Second, we must arrange the `makefile` production in such a way that different DLLs (object code) are produced for the different machines in the target architectures. When all the needed different object codes are available, the `assistrun` command may exploit them accordingly to the contents of the XML configuration file.

Both these tasks do not present any technical difficulty. They have been postponed only due to lack of human resources (programmers) in the project and we plan to have a working version of ASSISTcl compiler targeting heterogeneous architecture by the end of year 2003.⁵

In the meanwhile, we are moving the whole ASSIST framework to the GRID. As a first step, a tool has been built [6] that allows to manipulate (within a nice graphical interface) the XML configuration file according to information taken from GRID information services, in such a way that ASSISTcl programs can be eventually run on GRIDs. Figure 2 shows a snapshot of the tool. Actually, most of the work needed to run ASSISTcl programs on GRID is to be performed by hand by the programmer. In particular, most of the dynamic features of GRID⁶

⁵ development, experiments and debugging is performed on a mixed Linux/Pentium and MacOS/X/PowerPC machines

⁶ e.g. resource lookup and reservation

are completely in charge of the programmer interacting with the configuration tool.

7 Conclusion

We discussed some features of the ASSIST structured, parallel programming environment, that will be demonstrated during this conference in the new, for Europar, “demo session”. More precise information concerning ASSIST can be found in the other papers of our group.

Acknowledgements. We wish to thank people that contributed in different, essential ways, to the development of ASSIST: R. Baraglia, D. Laforenza, M. Lettere, D. Guerri, S. Magini, S. Orlando, A. Paternesi, R. Perego, A. Petroccelli, E. Pistoletti, L. Potiti, N. Tonellotto, L. Vaglini, P. Vitale.

References

1. The Adaptive Communication Environment home page.
<http://www.cs.wustl.edu/~schmidt/ACE-papers.html>, 2003.
2. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. A framework for experimenting with structured parallel programming environment design. In *Proceedings of PARCO'03*, 2003. to appear.
3. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an Environment for Parallel and Distributed Programmind. In *Proceedings of Europar'03*, 2003. to appear.
4. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P³L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.
5. B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec. 1999.
6. R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 193–200. IEEE, February 2003. ISBN 0-7695-1875-3.
7. P. D’Ambra, M. Danelutto, D. di Serafino, and M. Lapeгна. Integrating MPI-Based Numerical Software into an Advanced Parallel Computing Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 283–291. IEEE, February 2003. ISBN 0-7695-1875-3.
8. M. Vanneschi. ASSIST: an environment for parallel and distributed portable applications. Technical Report TR 02/07, Dept. Comp. Sc., Univ. of Pisa, May 2002.
9. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs

Bernd Mohr¹ and Felix Wolf²

¹ Forschungszentrum Jülich,
Zentralinstitut für Angewandte Mathematik,
52425 Jülich, Germany
b.mohr@fz-juelich.de

² Innovative Computing Laboratory,
Computer Science Department,
University of Tennessee,
Knoxville, TN 37996
fwolf@cs.utk.edu

Abstract. Today's parallel computers with SMP nodes provide both multithreading and message passing as their modes of parallel execution. As a consequence, performance analysis and optimization becomes more difficult and creates a need for advanced performance tools that are custom made for this class of computing environments. Current state-of-the-art tools provide valuable assistance in analyzing the performance of MPI and OpenMP programs by visualizing the run-time behavior and calculating statistics over the performance data. However, the developer of parallel programs is still required to filter out relevant parts from a huge amount of low-level information shown in numerous displays and map that information onto program abstractions without tool support.

The KOJAK project (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks) is aiming at the development of a generic automatic performance analysis environment for parallel programs. Performance problems are specified in terms of execution patterns that represent situations of inefficient behavior. These patterns are input for an analysis process that recognizes and quantifies the inefficient behavior in event traces. Mechanisms that hide the complex relationships within event pattern specifications allow a simple description of complex inefficient behavior on a high level of abstraction.

The analysis process transforms the event traces into a three-dimensional representation of performance behavior. The first dimension is the kind of behavior. The second dimension describes the behavior's source-code location and the execution phase during which it occurs. Finally, the third dimension gives information on the distribution of performance losses across different processes or threads. The hierarchical organization of each dimension enables the investigation of performance behavior on varying levels of granularity. Each point of the representation is uniformly mapped onto the corresponding fraction of execution time, allowing the convenient correlation of different behavior using only a single view. In addition, the set of predefined performance problems can be extended to meet individual (e.g., application-specific) needs.

1 Short Description of the KOJAK Tool Set

Figure 1 gives an overview about the architecture of the current prototype and its components. The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. The first subprocess is called semi-automatic because it requires the user to slightly modify the makefile and execute the application manually.

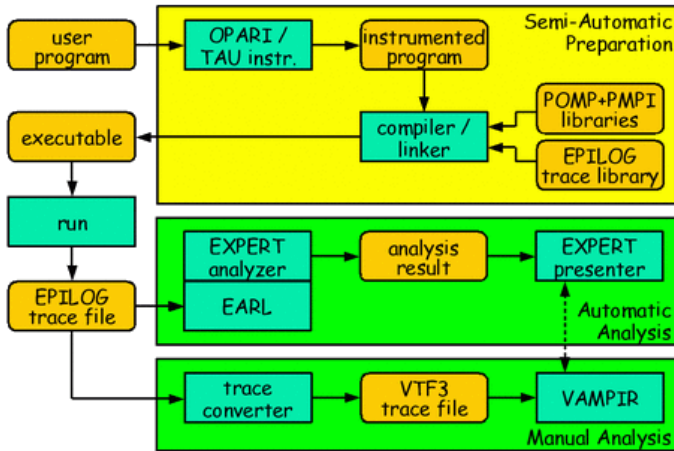


Fig. 1. KOJAK tool architecture

To begin the process, the user supplies the application's source code, written in either C, C++, or Fortran, to OPARI (OpenMP Pragma And Region Instrumentor), which performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level based on the POMP API [6]. Instrumentation of user functions is done either on the source-code level using TAU or using a compiler-supplied profiling interface. Instrumentation for MPI events is accomplished using a PMPI wrapper library, which generates MPI-specific events by intercepting calls to MPI functions. All MPI, OpenMP, and user-function instrumentation call the EPILOG (Event Processing, Investigating and LOGging) run-time library, which provides mechanisms for buffering and trace-file creation. At the end of the instrumentation process the user has a fully instrumented executable.

Running this executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT (Extensible Performance Tool) analyzer. The analyzer uses EARL (Event Analysis and Recognition Language) to provide a high-level view of the raw trace file. We call this view the *enhanced event model*, and it is where the actual analysis takes place. The analyzer generates an analysis report, which serves as input for the EXPERT presenter. Figure 2 shows a screendump of the EXPERT presenter.

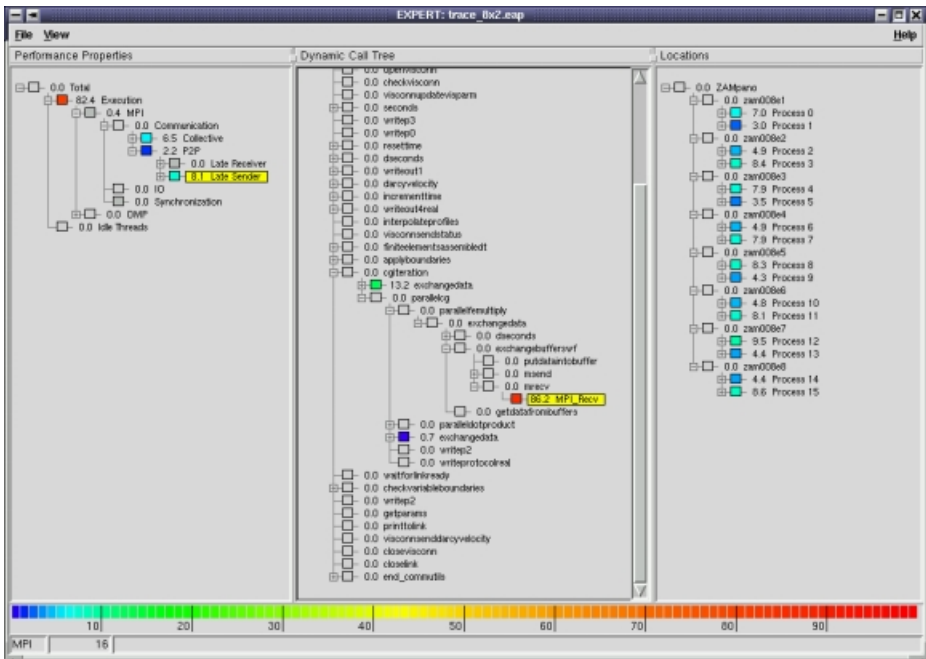


Fig. 2. EXPERT Presenter Example Screenshot. Using the color scale shown on the bottom, the severity of performance problems found (left pane) and their distribution over the program's call tree (middle pane) and machine locations (right pane) is displayed. By expanding or collapsing nodes in each of the three trees, the analysis can be performed on different levels of granularity.

In addition, it is possible to convert EPILOG traces into VTF3 format and analyze them manually with the VAMPIR event trace analysis tool [8]. We are currently working on integrating both tools so that the instance with the highest severity of each performance problem found can be displayed by VAMPIR on request. This would provide the ability to analyze the history of inefficient behavior in a time-line diagram or to do further statistical analysis using VAMPIR's powerful features.

Currently, the measurement components are available for the following platforms:

- Linux IA-32 cluster
- IBM POWER3 AND POWER4 cluster
- SGI MIPS cluster (Onyx, Challenge, Origin 2000, Origin 3000)
- SUN SUN FIRE cluster
- CRAY T3E
- HITACHI SR8000-FI

On Linux clusters and HITACHI SR8000-FI systems the instrumentation of user functions is done automatically using the unpublished profiling interface of the PGI compiler or of the HITACHI compiler, respectively. For IBM systems, EPILOG provides an automatic binary instrumentor, which has been implemented on top of DPCL. On all other systems,

user function instrumentation must be carried out on the source-code level either manually or automatically utilising the TAU instrumentation facilities [4]. The analysis components are based on Python and run on any workstation or laptop.

2 Additional Information

- The KOJAK tool suite including the source code can be downloaded from the KOJAK website. The website also provides a variety of technical papers, presentations, and screen dumps showing the analysis of example applications.
→ <http://www.fz-juelich.de/zam/kojak/>
- For more information on EXPERT's analysis and presentation features, see Felix Wolf's Ph.D. thesis [2]. The theoretical aspects can also be found in [3]. A more detailed overview (than this short description) about KOJAK can be found in [1].
- Details on instrumentation of OpenMP applications based on the POMP interface are described in [6] and [7].
- More information on the source-code instrumentation of user functions can be found on the homepages of the TAU [4] and PDT [5] projects.
→ <http://www.cs.uoregon.edu/research/paracomp/tau/>
→ <http://www.cs.uoregon.edu/research/paracomp/pdtoolkit/>
- The KOJAK project is part of the European IST working group APART.
→ <http://www.fz-juelich.de/apart/>

References

1. F. Wolf, B. Mohr. *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*. 11th Euromicro Conference on Parallel, Distributed and Network Based Processing, 2003.
2. F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Dissertation, NIC Series, Vol. 17, Forschungszentrum Jülich, 2002.
3. F. Wolf and B. Mohr. Specifying Performance Properties of Parallel Applications Using Compound Events. *Parallel and Distributed Computing Practices (Special Issue on Monitoring Systems and Tool Interoperability)*, Vol. 4, No. 3.
4. S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145. ACM, August 1998.
5. K. A. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proc. of Supercomputing 2000*, Dallas, TX, 2000.
<http://www.sc2000.org/techpaper/papers/pap.pap167.pdf>
6. B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.
7. B. Mohr, A. Malony, H.-Ch. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, S. Shah. A Performance Monitoring Interface for OpenMP. In *Proc. of 4th European Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, 2002.
<http://www.caspar.it/ewomp2002/prog.html>
8. Pallas GmbH. Visualization and Analysis of MPI Programs
<http://www.pallas.de/e/products/vampir/>

Visual System for Developing of Parallel Programs

O.G. Monakhov

Institute of Computational Mathematics and Mathematical Geophysics SB RAS,
Novosibirsk, 630090, Russia
`monakhov@rav.sccc.ru`

Abstract. A program system for manipulating and translating of multimedia program skeletons ("films") into parallel programs is considered. The main goal of this system is to make easier to create parallel programs for various parallel computing systems.

In this paper a programming environment for visual developing of parallel programs is presented. The environment allows writing parallel programs in abstract mode without consideration of particular computing system. Also the environment uses multimedia tools (graphics, text, animation, sound) for presentation of parallel algorithms to make developing of parallel programs more clear and understandable. The environment has a flexible GUI (Graphical User Interface) written in Java.

The system is based on idea of program skeletons. A program skeleton is a multimedia, hierarchical, tree-like representation of parallel program in "film" format [1],[3]. The film describes in animation mode (by sequence of multimedia stills) the dynamic activity of the algorithm in time-space coordinates. The film includes a parameterized template code of parallel computational algorithms. On the basis of the template code it is possible to create parallel program code after definition of parameters, variables and formulas in time-space points of algorithm activity. The film database includes the parallel algorithmic skeletons, proposed in [4], and can be extended by any user-defined program skeletons.

The programming environment for visual developing of parallel programs consists of the following main parts:

- Graph and film editors for creating, editing, manipulating, composition and viewing of program multimedia skeletons.
- Icon language based on an open set of intuitively understandable small images for representing computational schemes, operations and formulas in points of algorithmic activity.
- Modules for attachment of parameters, variables and formulas to skeletons and two translators for generating of code for sequential C-program and for parallel ANSI C-program with using MPI library for execution on many parallel and distributed systems and clusters.

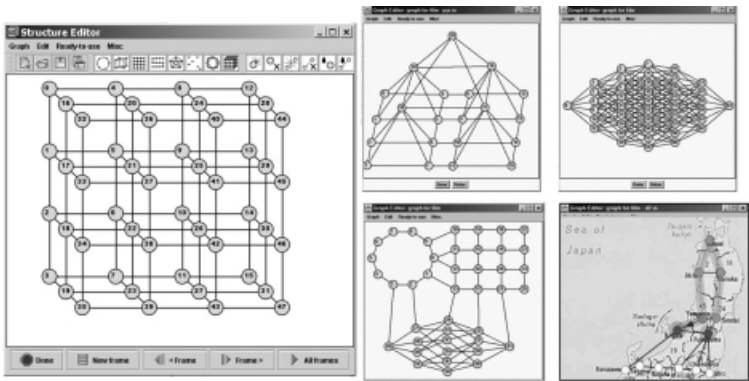


Fig. 1. Structures examples for films

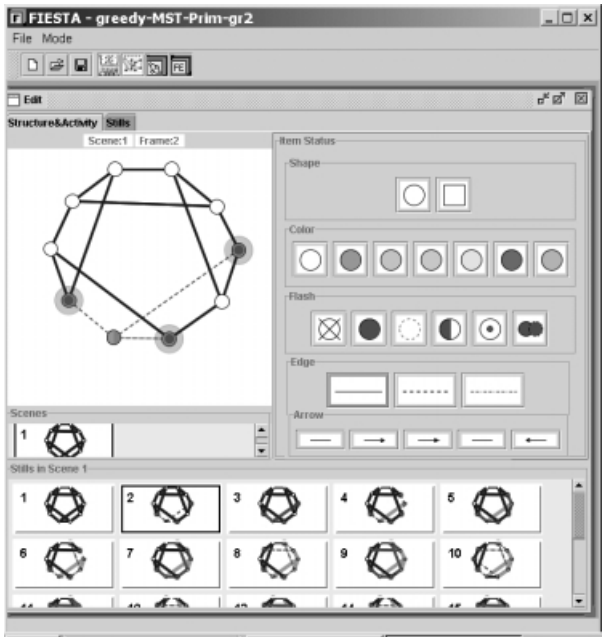


Fig. 2. Still editing interface of the film

The programming environment for visual developing of parallel programs is inspired by Active Knowledge Studio (AKS)[2], a framework supporting the development, the acquisition and the use of self-explanatory components related to methods of computational mathematics, physics and other natural science.

The system provides seamless workflow from visual film format to automatic executable code generation and visualization of results. It includes film compo-

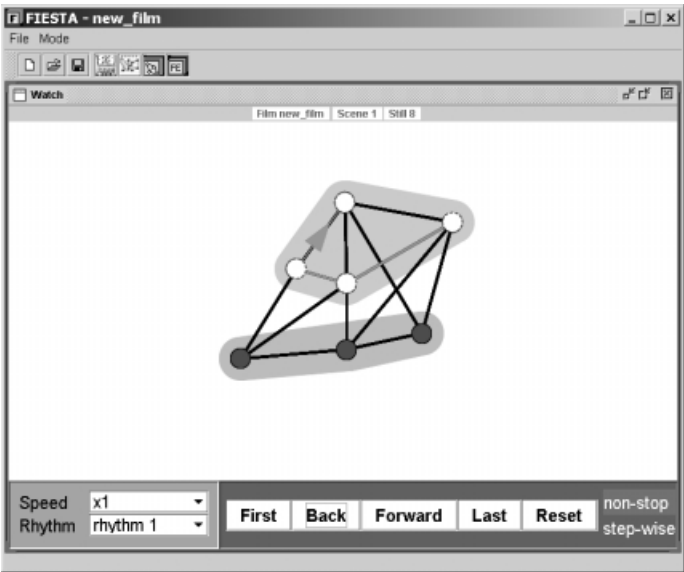


Fig. 3. Watching interface of the system.

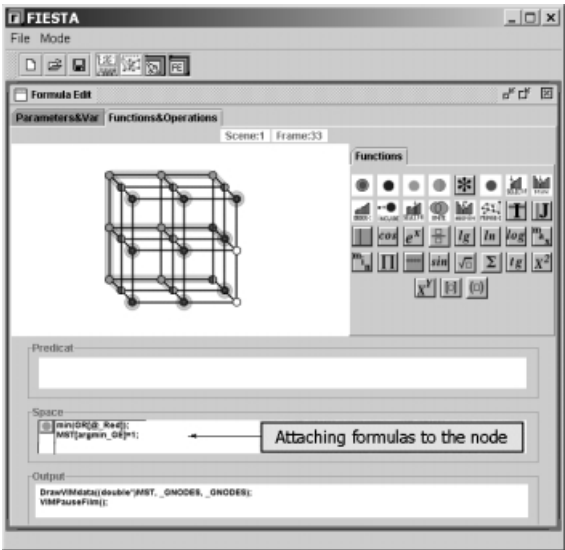


Fig. 4. A panel of the node specifying mode: input/edit formulas

sition and manipulation part, attaching formula/sound and program synthesis part, program execution and result visualisation part.

The film composition and manipulation part includes "Structure Editor" (depicted by Fig.1) and "2D Rendering Engine" modules for film creation and ma-

nipulations with self-explanatory components. This part provides a user interface for watching (Fig.2), editing (Fig.3), and composing modes for film management system.

The attaching formulas and program synthesis part includes "Formula Editor" (Fig.4) and "Program Synthesis/Compiling" modules for formulas attachment and executable code generation. This part provides a library of template programs and corresponding template control files representing basic algorithmic skeletons. This part also includes executable component codes being created.

The program execution and result visualisation part provides execution of generated program and user interface for visualisation of input/output data for the program.

References

1. N. Mirenkov,: VIM Language Paradigm, in: Parallel Processing: CONPAR'94-VAPP'IV, Lecture Notes in Computer Science, Vol.854, B.Buchberger, J.Volkert (Eds.), Springer-Verlag, pp. 569–580, 1994.
2. N. Mirenkov, A. Vazhenin, R. Yoshioka, T. Ebihara, T. Hirotomi, and T. Mirenkova,: Self-explanatory components: a new programming paradigm, International Journal of Software Engineering and Knowledge Engineering, Vol.11, No. 1, World Scientific, 2001, 5–36.
3. N. Mirenkov, O. Monakhov, R. Yoshioka,: Self-explanatory components: visualization of graph algorithms, in: Proc. of the 8th Internat. Conference on Distributed Multimedia Systems (DMS'02), (Workshop on Visual Computing), 2002, San Francisco, CA, USA, IEEE Press, p. 562–567.
4. M. Cole,: Algorithmic Skeletons: Structured Management of Parallel Computation, The MIT Press, 1989.

Peer-to-Peer Communication through the Design and Implementation of Xiangqi

Abdulmotaleb El Saddik and Andre Dufour

Multimedia Communications Research Lab (MCRLab)
University of Ottawa, Ottawa, Canada, K1N 6N5
abed@mcrlab.uottawa.ca

Abstract. In this work, we conducted a case study by implementing the Xiangqi game using a P2P networking technology. In so doing, we gained a deeper understanding of P2P communication in general and a more in-depth knowledge of the specific technology used. We encountered a number of issues while implementing our game using the JXTA P2P framework, but we were nevertheless able to create a working prototype that functioned satisfactorily.

1 Introduction

Peer-to-peer communication is emerging as one of the most potentially disruptive technologies in the networking sector. If the interest in such technologies as Napster, Morpheus and Gnutella is any indication, peer-to-peer networks will be a major component in the future of computer communications and the Internet. Indeed, the Morpheus P2P network had, on average, 300,000 simultaneous users in 2001 and now regularly exceeds 3,000,000 users [1]. These numbers are too large to be ignored and the rapid growth of P2P is a testimonial to its usefulness. When we add the host of users using instant messaging clients or SETI@Home, to name just those two, it becomes clear that P2P networking technologies have proven their value and have made their way into mainstream use.

In this work, we propose to investigate the suitability of P2P networking technologies for multimedia gaming applications through a case study. There were two objectives to the experiment: we wanted to gain a deeper understanding of peer-to-peer communication in general and of one technology in particular, and we wished to evaluate the suitability of P2P networks for gaming applications of a particular type.

2 Application Characteristics

Given that Xiangqi is a turn-based game somewhat similar to western chess, it has very low bandwidth requirements. Other than initial players discovery and board setup, messages are only exchanged when a player makes a move, in order to notify the opponent, and their content is very compact: they simply encapsulate a starting point and an ending point. Since time between moves can be of the order of several tens of seconds or more, latency and jitter are not major concerns. Indeed, the time for

a message to reach a player should be of the order of milliseconds or seconds at most, which is negligible when compared to the time that a user may legitimately expect to wait for the other player’s move. Likewise, even relatively high jitter (with respect to message transit time) would have negligible impact on the gaming experience because players expect certain variability in the time between moves. We do require, however, that messages be sent reliably, which is to say that they arrive in sequence and intact, otherwise the game will enter an undefined state.

Overall, we have created a fairly undemanding application. This was done intentionally because the gaming application is little more than an interesting test harness in this connection. Since we were mostly concerned with learning about P2P networks it seemed appropriate to select the characteristics of the test application such that they could be easily supported by the network architecture. That way, we improved our chances of achieving a successful implementation.

3 Software Architecture

Our Xiangqi implementation is based on the Java binding of the JXTA protocol suite [5]. Given that the low-level communication functions were handled using Java, it seemed natural to use Java/Swing for the graphical user interface. We strove to design our application such that the GUI and game logic were decoupled from the communications functionality. This increased code modularity and clarity and avoided the propagation of JXTA’s idiosyncrasies into all parts of the code. The following figure gives a summary of the software architecture for the Xiangqi application.

Fig. 1. Xiangqi Application Architecture

Graphical User Interface and Game Logic
Communications Infrastructure
JXTA Communications Interface
JXTA Services and Core (taken from the JXTA project)

As we can see, the GUI and application logic are not separated. This is because they are closely related. For example, the game board, a graphical component, is responsible for determining if moves requested by the user are valid or not. It seemed most logical to proceed in this way because the information required to make this determination is contained within the board.

The communications infrastructure mediates between the GUI and the JXTA-specific primitives in the lower layers of the architecture, providing simple send and receive primitives that do not don’t reveal the complexity of interacting with JXTA.

4 Game Implementation Issues

In order to gauge the suitability of P2P networks, and in particular JXTA, to our gaming application, it is necessary to review the difficulties encountered in implementing the solution with the given technology.

4.1 Reliability

One of the most salient issues we had to contend with during the implementation of Xiangqi was the fact that JXTA does not provide reliable messaging. Indeed, a significant amount of effort was required to devise an acknowledgement scheme using sequence numbers in order to confirm that messages arrived at their destination. It was also necessary to include a CRC in the messages to ensure the integrity of their content. While we didn't fully implement a scheme to resend missing or erroneous messages, it would have been useful to do so. Also, we noted that JXTA's internal messages, such as service and pipe advertisements, did not include any protection against lost or corrupted messages. This caused us significant problems because the pipe advertisements we embedded in the Xiangqi service advertisements were frequently corrupted in transit. The following is part of a JXTA pipe advertisement:

```
uuid-2CD0915F41244B19B341AB9F49204F4  
3174A4A3EF51B4A9785C296532AFCC01404
```

As we can see, there are 66 hexadecimal numbers in this UUID. If only one of these 264 bits is corrupted, the peer on the receiving end will attempt to bind to an inexistent pipe. This situation occurred fairly frequently. It would have been necessary to implement a checksum or CRC algorithm in a class derived from JXTA's advertisement classes in order to ensure the reliability of these advertisements.

We can see that a significant amount of developer effort is required in order to ensure true reliability when using JXTA. Unfortunately, any application layer reliability scheme is redundant in the case where JXTA is running over a protocol such as TCP, which already provides for reliable transport. This is a significant failing in JXTA.

4.2 Network Reliability

The issue with the messaging was not the only difficulty we encountered with respect to reliability in JXTA. We also found the JXTA P2P network to be somewhat unreliable.

JXTA uses so-called rendezvous peers to propagate discovery requests throughout the network. Sun also defines rendezvous peers as "well-known peers that have agreed to cache a large number of advertisements in support of a peer group". Any peer can be configured to be a rendezvous peer, but in reality, relatively few are. If no rendezvous peers are present on the network or if they are configured such that they don't form an uninterrupted connection between two new peers wishing to play Xiangqi, it will not be possible for the game to take place: the players will never be able to find each other. In this situation, the network is effectively "down". We observed this several times during our experimentation. This is similar to the failure of a central server and quite an unexpected situation for a P2P network! However, it is important to note that the JXTA network is still in its infancy and that if the technology is generally accepted and becomes popular, it should be very rare to see the network fail for lack of rendezvous peers. Nevertheless, this did occasionally present a problem for our prototype application.

4.3 Ability to Serialize Advertisements

One further problem we encountered was the fact that JXTA advertisements are not serializable. This makes it impossible to directly convert the objects to a form suitable for transmission, such as an octet stream. The issue first arose when we wanted to embed a pipe advertisement in `GameRequestMessage` objects and found that we couldn't. Creating a subclass of the advertisement and making it serializable would not have helped because every class in the hierarchy must be serializable for the operation to succeed. Ultimately, we were forced to construct a serializable wrapper class that, given an advertisement, extracted the values of the instance variables and stored them before transmission.

Once the wrapper object was reconstructed on the receiving side we could create a new advertisement using the stored values and work with that object as though it had itself been transmitted. This strategy was quite effective, but it seems a strange *gaucherie* on the part of JXTA's designers to not have implemented the `Serializable` interface in the first place.

5 Conclusion and Future Work

JXTA's flaws notwithstanding, we were able to successfully implement the Xiangqi application using P2P technology. The implementation did require a significant amount of effort in order to ensure some reliability and more would doubtless be required to guarantee complete dependability of the inter-peer messaging, but we are confident that the objective is attainable. We feel that the idea of using peer-to-peer communication for gaming is sound and that the advantages far outweigh the drawbacks and difficulties associated with our particular implementation. Indeed, the fact that our application functions satisfactorily proves the concept.

While successful, our implementation is still only a prototype and as such has a number of limitations and a number of areas could be improved upon. In particular, it would be desirable to complete the work required in order to make the P2P communication fully reliable. This might even go so far as to introduce a "keepalive" mechanism to periodically inspect the integrity of the pipe connection. From a feature standpoint, it would be attractive to users if they could see what players are online and select the opponent they wish to play with. A chat facility would also make the game play more enjoyable. There is much work to do and while this work describes a particular frozen version of the project, we intend to make the source code available on the Internet so that others may add to it and keep the project very much alive.

References

- [1] <http://www.openp2p.com/pub/a/p2p/2001/07/11/numbers.html>, "Gnutella Blown Away? Not exactly. [July 10, 2001]", visited Dec. 4, 2002
- [2] Sun's Java Technology and Web services Documentation: http://www.uo.com/newplayer/newplay_0.html, "ORIGIN – Ultima Online – Main", visited Dec. 4, 2002
- [3] http://www.tiaranetworks.com/news/010919pr_mfr.html, "Tasman News", visited Dec. 4, 2002

- [4] <http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html?page=2>, "What is P2P... and What Isn't [Nov. 24, 2000]", visited Dec. 4, 2002
- [5] GONG, Li, Project JXTA: A Technology Overview, Sun Microsystems inc., Palo Alto, CA, USA, 2001.
- [6] <http://www.openp2p.com/pub/a/p2p/2001/05/30/hailstorm.html?page=2>, OpenP2P.com:HailStorm : Open Web Services Controlled by Microsoft [May 30, 2001]", visited Dec. 5, 2002
- [7] <http://www.microsoft.com/net/services/passport/license.asp>, "Licensing .NET Passport",visited Dec. 5, 2002
- [8] <http://www.sun.com/software/jini/faqs/index.html#1q15>, "Jini Network Technology", visited Dec. 5, 2002

Author Index

- Agarwal, Amit 230
Agarwal, Saurabh 160
Agarwal, Tarun 230
Aguilar-Saborit, Josep 328
Akutagawa, Tomoyoshi 1082
Aldinucci, Marco 712, 1295
Aliagas, Carles 616
Allard, Jérémie 1287
Almási, George 543
Aloisio, Giovanni 421
Alt, Martin 427, 742
Alves, Albano 969
Amorim, Claudio L. de 1207
An, Sunshin 1153
Anshus, Otto J. 47
Anvik, John 81
Ardaiz, Oscar 1265
Artigas, Pau 1265
- Badía-Contelles, José M. 310
Badrinath, R. 1291
Bailey, Mark 261
Bal, Henri E. 4
Balaton, Zoltán 404
Barchet-Estefanel, Luiz Angelo 632
Barli, Niko Demus 603
Batista, Rodolfo 517
Baydal, Elvira 958
Beckett, Duan H. 533
Beer, Wolfgang 1064
Beletsky, Volodymyr 297
Bell, Robert 17
Bellofatto, Ralph 543
Bellotti, Francesco 1129
Benthin, Carsten 499
Bergen, Benjamin 840
Berger, Michael 1109
Berta, Riccardo 1129
Berthold, Jost 732
Biersack, E.W. 1230
Bigonha, Mariza Andrade da Silva 1074
Bigonha, Roberto da Silva 1074
Bischof, Holger 682
Bister, Michel 624
Bjørndalen, John Markus 47
- Blanco, Vicente 704
Blochinger, Wolfgang 722
Bochmann, Gregor v. 342, 1046
Boissier, Olivier 1091
Bongo, Lars Ailo 47
Bosschere, Koen De 556
Boukerche, Azzedine 1099
Boyle, Eamonn 640
Brand, Per 470, 675
Brasileiro, Francisco Vilar 169
Brüning, Ulrich 481
Bruhn, Andrés 481
Brunheroto, José 543
Brunheroto, José R. 109
Brunie, Lionel 374
Bunimov, Viktor 923
Byun, Ilsoo 1170
- Cafaro, Massimo 421
Campa, Sonia 712, 1295
Carabelea, Cosmin 1091
Caşcaval, Călin 543
Castaños, José G. 543
Cesar, Eduardo 141
Ceze, Luis 543
Chan, C.Y. 870
Chan, Tim K.T. 1017
Chatzigianakis, I. 1003
Chen, G. 271
Cheung, Wai-leung 537
Choi, Gyu Sang 160
Choi, Jin-Seok 661
Choi, Jong-Mu 661
Chopra, Sumit 230
Choudhary, A. 271, 279
Christian, Volker 1064
Chu, Kenneth M.K. 1017
Chung, Sang-Hwa 995
Chunlin, Li 980
Cirne, Walfredo 169
Ciullo, Pierpaolo 712, 1295
Cláudio, Ana Paula 57
Clint, Maurice 640
Coppola, Massimo 712, 1295
Cordasco, Gennaro 911

- Cores, Fernando 859
 Coulon, Cédric 318
 Cox, Simon J. 357, 412, 525, 533, 1148
 Crumley, Paul 543
 Cunha, João Duarte 57

 Danelutto, M. 1295
 Daoudi, El Mostafa 844
 Das, Chita R. 160
 Decker, Keith 384
 Delaitre, T. 1281
 DeRose, Luiz 7, 109
 Dessloch, Stefan 3
 Devillers, Sylvain 1216
 Dickens, Phillip M. 938
 Dietrich, Andreas 499
 Dimitriou, T. 1003
 Dong, Ligang 236
 Dongarra, Jack 394
 Downton, Andy 750
 Duato, J. 1190
 Dufour, Andre 1309
 Dutra, Inês 509

 El-Khatib, K. 1046
 El Saddik, Abdulmotaleb 1309
 Engelen, Robert van 261, 421
 Ercan, Muhammet F. 537
 Eres, Hakki 525, 1148
 Erway, C. Christopher 543
 Exposto, José 969
 Eymann, Torsten 1265

 Faber, Peter 303
 Fàbrega, Josep 989
 Fahringer, Thomas 27
 Fairman, Matthew J. 357
 Felber, P.A. 1230
 Feldmann, Anja 230
 Fernández, Juan C. 491
 Fernández, M. 1190
 Ferreto, Tiago C. 7
 Ferretti, Edmondo 1129
 Ferscha, Alois 1064
 Finney, Sean 1160
 Fischer, Markus 481
 Flammini, Michele 1056
 Fleury, Martin 750
 Flich, José 947
 Fonseca, Tiago 517

 Fortes, José A.B. 98
 Franchetti, Franz 251
 Freitag, Felix 1265
 Furlinger, Karl 127
 Fujie, Tetsuya 451
 Fujita, Satoshi 201
 Fung, Yu-fai 537

 Gabarró, Joaquim 640
 Gagliano, Joseph 543
 Gallard, Pascal 930, 1291
 Gambosi, Giorgio 1056
 Ganchev, Kuzman 1160
 Garcés-Erice, L. 1230
 Garcia, Montse 616
 Garcia-Molina, Hector 1273
 Gasparini, Alessandro 1056
 Gaudiot, Jean-Luc 576
 Gautama, Hasyim 88
 Gemund, Arjan J.C. van 88
 Gerndt, Michael 127
 Gil-García, Reynaldo 310
 Gijzen, Martin van 820
 Giné, Francesc 212
 Gloria, Alessandro De 1129
 Goldschmidt, Balázs 1199
 Gombás, Gábor 404, 1281
 Gonzalez, Antonio 616
 González, Jesús A. 704
 Gorlatch, Sergei 427, 682, 742
 Gourgoulis, A. 1281
 Goussevskaia, Olga 1118
 Granat, Robert 800
 Griebel, Martin 303
 Guilloud, Cyril 38
 Guirado, Fernando 218
 Guo, H-F. 694
 Gupta, G. 694

 Hadibi, N. 1046
 Hämäläinen, Timo 1141
 Hännikäinen, Marko 1141
 Hagersten, Erik 760
 Hagihara, Kenichi 135
 Haridi, Seif 470, 675
 Hasegawa, Atsushi 1082
 He, Ligang 195
 Heinemann, Andreas 1038
 Hernández, Porfidio 212, 859
 Hernández, Vicente 491

- Herrero, José R. 461
- Hiett, Ben 533
- Hiraki, Kei 609
- Ho, Tin-kin 537
- Hoare, C.A.R. 1
- Holmgren, Fredrik 470, 675
- Horányi, A. 1281
- Horspool, Nigel 242
- Horváth, Á. 1281
- Housni, Ahmed 669
- Hsiao, Hung-Chang 1248
- Hu, Zhenjiang 789
- Huedo, Eduardo 366
- Hülsemann, Frank 840
- Hung, Luong Dinh 603
- Huszák, Árpád 1137
- Ibañez, Pablo 586
- Ibe, Akihiro 1082
- Imre, Sándor 1137
- Ino, Fumihiko 135
- Ishar, Syarraiени 624
- Iwama, Chitaka 603
- Jang, Si Woong 880
- Jarvis, Stephen A. 195
- Jiang, Hong 224
- Jiao, Zhuoan 412
- Johnston, David 750
- Jonsson, Isak 810
- Kacsuk, Peter 119, 1281
- Kadayif, I. 279
- Kågström, Bo 800, 810
- Kalogeropoulos, Spiros 597
- Kammenhuber, Nils 230
- Kamvar, Sepandar D. 1273
- Kandemir, M. 271, 279
- Kangasharju, Jussi 1038
- Karakoy, M. 279
- Karasti, Olavi 1141
- Karatza, Helen D. 1257
- Keane, Andy 525
- Keen, Aaron W. 770
- Kergommeaux, Jacques Chassin de 38
- Kerhervé, Brigitte 342
- Kim, Han-gyoo 1224
- Kim, Jai-Hoon 661
- Kim, Jin-Ha 160
- King, Chung-Ta 1248
- Kitzelmann, Emanuel 682
- Klusik, Ulrike 732
- Ko, Young-Bae 661
- Kohn, Emil-Dan 1180
- Kolcu, I. 271
- Korch, Matthias 830
- Kounoike, Yuusuke 451
- Kovács, J. 1281
- Kowarschik, Markus 441
- Kral, Stefan 251
- Kranzlmüller, Dieter 74
- Kreahling, William 261
- Krysta, Piotr 230
- Küchlin, Wolfgang 722
- Kwon, Hyuk-Chul 995
- Kwon, Jin B. 851
- Lacroix, Michel 669
- Laforest, Christian 903
- Lakhouaja, Abdelhak 844
- Larriba-Pey, Josep-L. 328
- László, Zoltán 417, 1199
- Lawley, Richard 384
- Layuan, Li 980
- Lee, Ben 995
- Lee, Jack Y.B. 870
- Lee, Kee Cheol 1224
- Lee, Seong-Won 576
- Lehner, Wolfgang 348
- Lengauer, Christian 303
- León, Coromoto 704
- Leonardi, Letizia 1027
- Leung, Karl R.P.H. 1017
- Lezzi, Daniele 421
- Li, Chun Hung 1017
- Li, Xiaolin 181
- Lieber, Derek 543
- Lim, Kyungsoo 1153
- Lin, Chuan-Mao 1248
- Liu, Hua 66
- Llabería, Jose Maria 586
- Llorente, Ignacio M. 366
- Lobosco, Marcelo 1207
- Löf, Henrik 760
- Logie, Hans 556
- Loogen, Rita 732
- López, Pedro 947, 958
- Loques, Orlando 1207
- Lorenz, Juergen 251
- Lottiaux, Renaud 1291

- Loulergue, Frédéric 781
 Loureiro, Antonio A.F. 1118
 Lovas, R. 1281
 Luck, Michael 384
 Luque, Emilio 141, 212, 218, 566, 859
 Lyardet, Fernando 1038

 Magini, Silvia 712
 Malony, Allen D. 17
 Mamei, Marco 1027
 Margarone, Massimiliano 1129
 Martín, María J. 287
 Marín, Mauricio 338
 Martins, Thelmo 517
 Martorell, Xavier 543
 Masukawa, Masayuki 201
 Mateus, Geraldo Robson 1118
 Matossian, Vincent 1240
 Matsuzaki, Kiminori 789
 Mavronicolas, M. 1003
 Mehofer, Eduard 242
 Mehrmann, Lars 1064
 Melo, Alba Cristina Magalhaes A. 517
 Melo, Renata Cristina F. 517
 Mesa, J.G. 141
 Messeguer, Roc 1265
 Michael, Maged M. 651
 Miguel-Alonso, José 98
 Miller, Jim 2
 Mindlin, Pedro 109
 Miura, Hideyuki 603
 Mizutani, Yasuharu 135
 Mohr, Bernd 1301
 Molina, Carlos 616
 Molinari, Marc 412
 Monakhov, O.G. 1305
 Montero, Rubén S. 366
 Moreau, Luc 384
 Moreira, José E. 109, 543
 Morillo, Pedro 1190
 Morin, Christine 930, 1291
 Moure, Juan C. 566
 Mühlhäuser, Max 1038
 Muñoz, Xavier 989
 Muntés-Mulero, Victor 328

 Nakajima, Tatsuo 1082
 Nardelli, Marcelo 517
 Navarra, Alfredo 1056
 Navarro, Gonzalo 338

 Navarro, Juan J. 461
 Navarro, Leandro 1265
 Negro, Alberto 911
 Newhall, Tia 1160
 Ng, Joseph Kee-Yin 1017
 Ngoh, Lek Heng 236
 Nikolettseas, S. 1003
 Nudd, Graham R. 195

 Özsu, M. Tamer 318
 Olsson, Ronald A. 770
 Orduña, Juan M. 1190
 Outada, Halima 844

 Pacitti, Esther 318
 Page, David 509
 Parashar, Manish 66, 181, 1240
 Pardines, I. 206
 Park, Taesoon 1170
 Park, Woojin 1153
 Park, Yong Woon 880
 Payne, Terry 384
 Peñalver, Lourdes 491
 Pereira, Fernando Magno 1074
 Pesciullesi, Paolo 712, 1295
 Pierson, Jean-Marc 374
 Pina, António 969
 Podhorszki, Norbert 119, 1281
 Pohl, Thomas 441
 Pons-Porrata, Aurora 310
 Pontelli, E. 694
 Popov, Konstantin 470, 675
 Poromaa, Peter 800
 Potiti, Laura 712
 Pound, Graeme E. 357, 525
 Priebe, Steffen 732

 Qazzaz, Bahjat 859
 Qin, Xiao 224

 Radović, Zoran 760
 Rafea, Mahmoud 470, 675
 Raffin, Bruno 1287
 Ramparany, Fano 1091
 Rantanen, Tapio 1141
 Rauber, Thomas 830
 Ravazzolo, Roberto 712, 1295
 Reinicke, Michael 1265
 Renard, Hélène 148
 Rexachs, Dolores I. 566

- Rilling, Louis 1291
Ripoll, Ana 218, 859
Rivera, Francisco F. 206, 287
Robert, Yves 148
Robles, Antonio 947
Rodríguez, Casiano 704
Rodríguez, Germán 704
Roig, Concepció 218
Rose, César A.F. De 7
Rosenberg, Arnold L. 911
Ross, K.W. 1230
Rüde, Ulrich 441, 840
Rufino, José 969

Sakai, Shuichi 603
Sakellariou, Rizos 189
Sanomiya, Alda 543
Santos Costa, Vitor 509
Scarano, Vittorio 911
Scarpa, Michael 74
Schaeffer, Jonathan 81
Schimmler, Manfred 923
Schlesinger, Lutz 348
Schlosser, Mario T. 1273
Schmollinger, Martin 885
Scholz, Bernhard 242
Schulz, Martin 433
Schuster, Assaf 1180
Seitz, Christian 1109
Seitz, Ludwig 374
Shavlik, Jude 509
Shende, Sameer 17
Shinano, Yuji 451
Sibeyn, Jop F. 894
Siedlecki, Krzysztof 297
Sikiö, Janne 1141
Silva, Anderson 1207
Silva, Daniel Paranhos da 169
Simon, Vilmos 1137
Singh, David E. 287
Slogsnat, David 481
Slusallek, Philipp 499
Solsona, Francesc 212
Somogyi, Csongor 417
Song, Ha Yoon 1224
Song, Wenbin 525
Sorribes, Joan 141
Spiegel, Michael 1160
Spirakis, P. 1003
Spoonner, Daniel P. 195

Stein, B. de Oliveira 38
Stewart, Alan 640
Strauss, Karin 543
Subramanyan, Rajesh 98
Suppi, Remo 859
Swanson, David R. 224
Szabó, Sándor 1137
Szafron, Duane 81
Szalai, F. 1281
Szeberényi, I.
Szeberényi, Imre 417, 1281

Tagashira, Shigeaki 201
Takagi, Masamichi 609
Takeichi, Masato 789
Takesue, Masaru 917
Tan, Joo Geok 236
Tan, Kai 81
Tanaka, Hidehiko 603
Tashiro, Daisuke 603
Terstyánszky, G. 1281
Thomas, Ken S. 533
Torquati, Massimo 712, 1295
Torres, Enrique F. 586
Trehel, Michel 669
Trinitis, Carsten 433
Truong, Hong-Linh 27
Tubella, Jordi 616

Ueberhuber, Christoph W. 251
Uh, Gang-Ryung 261
Urvoy-Keller, G. 1230
Utard, Gaël 1291

Vadhiyar, Sathish 394
Valente, Marco Tulio 1074
Vallée, Geoffroy 1291
Vallejo, Isabel 640
Vandierendonck, Hans 556
Vanhatupa, Timo 1141
Vanneschi, Marco 712, 1295
Villaverde, K. 694
Viñals, Victor 586
Vivien, Frédéric 148
Vlassov, Vladimir 470
Vöcking, Berthold 230
Volkert, Jens 74

Waddell, Michael 509
Wald, Ingo 499

- Walter, Maria Emilia Telles 517
Wason, Jasmin 412
Weickert, Joachim 481
Weskamp, Nils 732
Whalley, David 261
Wheelhouse, Ed 433
Wilke, Jens 441
Wolf, Felix 1301
Woo, Sinam 1153

Xu, Fenglian 1148
Xue, Gang 357

Yamamoto, Kouji 1082
Yang, Xiaoyuan 859

YarKhan, Asim 394
Ye, Haiwei 342
Yeom, Heon Y. 851
Yoo, Anydy B. 160
Yoon, In-Su 995
Yuan, Xiao 218
Yuan, Xin 261

Zambonelli, Franco 1027
Zara, Florence 1287
Zerfridis, Konstantinos G. 1257
Zhao, Henan 189
Zhu, Yifeng 224
Zoccolo, Corrado 712, 1295