

Lecture Notes in Computer Science

2763

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

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Victor Malyshev (Ed.)

Parallel Computing Technologies

7th International Conference, PaCT 2003
Nizhni Novgorod, Russia, September 15-19, 2003
Proceedings



Springer

Series Editors

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

Volume Editor

Victor Malyshkin
Russian Academy of Sciences
Institute of Computational Mathematics and Mathematical Geophysics
pr. Lavrentiev 6, Novosibirsk
630090 Russia
E-mail: malysh@ssd.sssc.ru

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 Nationalbibliografie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): D, F.1-2, C, I.6

ISSN 0302-9743

ISBN 3-540-40673-5 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: 10930922 06/3142 5 4 3 2 1 0

Preface

The PaCT-2003 (Parallel Computing Technologies) conference was a four-day conference held in Nizhni Novgorod on September 15–19, 2003. This was the 7th international conference of the PaCT series, organized in Russia every odd year.

The first conference, PaCT-91, was held in Novosibirsk (Academgorodok), September 7–11, 1991. The next PaCT conferences were held in: Obninsk (near Moscow), 30 August–4 September, 1993; St. Petersburg, September 12–15, 1995; Yaroslavl, September 9–12, 1997; Pushkin (near St. Petersburg) September 6–10, 1999; and Akademgorodok (Novosibirsk), September 3–7, 2001. The PaCT proceedings are published by Springer-Verlag in the LNCS series.

PaCT-2003 was jointly organized by the Institute of Computational Mathematics and Mathematical Geophysics of the Russian Academy of Sciences (Novosibirsk) and the State University of Nizhni Novgorod.

The purpose of the conference was to bring together scientists working with theory, architectures, software, hardware and solutions of large-scale problems in order to provide integrated discussions on Parallel Computing Technologies. The conference attracted about 100 participants from around the world. Authors from 23 countries submitted 78 papers. Of those submitted, 38 papers were selected for the conference as regular ones; there were also 4 invited papers. In addition, a number of posters were presented. All the papers were internationally reviewed by at least three referees. As usual a demo session was organized for the participants.

Many thanks to our sponsors: the Russian Academy of Sciences, the Russian Fund for Basic Research, the Russian State Committee of Higher Education, IBM and Intel (Intel laboratory in Nizhni Novgorod) for their financial support. The organizers highly appreciate the help of the Association Antenne-Provence (France).

June 2003

Victor Malyskin
Novosibirsk, Academgorodok

Organization

PaCT-2003 was organized by the Supercomputer Software Department, Institute of Computational Mathematics and Mathematical Geophysics, Siberian Branch, Russian Academy of Sciences (SB RAS) in cooperation with the State University of Nizhni Novgorod.

Program Committee

V. Malyskin	Chairman (Russian Academy of Sciences)
F. Arbab	(Centre for MCS, The Netherlands)
O. Bandman	(Russian Academy of Sciences)
T. Casavant	(University of Iowa, USA)
A. Chambarel	(University of Avignon, France)
P. Degano	(State University of Pisa, Italy)
J. Dongarra	(University of Tennessee, USA)
A. Doroshenko	(Academy of Sciences, Ukraine)
V. Gergel	(State University of Nizhni Novgorod, Russia)
B. Goossens	(University Paris 7 Denis Diderot, France)
S. Gorlatch	(Technical University of Berlin, Germany)
A. Hurson	(Pennsylvania State University, USA)
V. Ivannikov	(Russian Academy of Sciences)
Yu. Karpov	(State Technical University, St. Petersburg, Russia)
B. Lecussan	(State University of Toulouse, France)
J. Li	(University of Tsukuba, Japan)
T. Ludwig	(University of Heidelberg, Germany)
G. Mauri	(Università degli Studi di Milano-Bicocca, Italy)
M. Raynal	(IRISA, Rennes, France)
B. Roux	(CNRS-Universités d'Aix-Marseille, France)
G. Silberman	(IBM T.J. Watson Research Center, USA)
P. Sloot	(University of Amsterdam, The Netherlands)
V. Sokolov	(Yaroslavl State University, Russia)
R. Strongin	(State University of Nizhni Novgorod, Russia)
V. Vshivkov	(State Technical University of Novosibirsk, Russia)

Organizing Committee

V. Malyskin	Co-chairman (Novosibirsk)
R. Strongin	Co-chairman (Nizhni Novgorod)
V. Gergel	Vice-chairman (Nizhni Novgorod)
V. Shvetsov	Vice-chairman (Nizhni Novgorod)
B. Chetverushkin	Member (Moscow)
L. Nesterenko	Member (Nizhni Novgorod)
Yu. Evtushenko	Member (Moscow)
S. Pudov	Secretary (Novosibirsk)
T. Borets	Vice-secretary (Novosibirsk)
O. Bandman	Publication Chair (Novosibirsk)
N. Kuchin	Member (Novosibirsk)
Yu. Medvedev	Member (Novosibirsk)
I. Safronov	Member (Sarov)
V. Voevodin	Member (Moscow)

Referees

D. van Albada	B. Goossens	M. Montangero
M. Alt	S. Gorlatch	M. Ostapkevich
F. Arbab	V. Grishagin	S. Pelagatti
O. Bandman	J. Guillen-Scholten	C. Pierik
H. Bischof	K. Hahn	S. Piskunov
R. Bisseling	A. Hurson	M. Raynal
C. Bodei	V. Ivannikov	L. Ricci
M. Bonuccelli	E. Jeannot	W. Ro
T. Casavant	T. Jensen	A. Romanenko
A. Chambarel	Yu. Karpov	B. Roux
V. Debelov	J.-C. de Kergommeaux	E. Schenfeld
P. Degano	V. Korneev	G. Silberman
J. Dongarra	M. Kraeva	M. Sirjani
A. Doroshenko	B. Lecussan	P. Sloot
D. Etiemble	J. Li	V. Sokolov
K. Everaars	A. Lichnewsky	P. Spinnato
P. Ferragina	R. Lottiaux	C. Timsit
J. Fischer	F. Luccio	L. van der Torre
S. Gaissaryan	T. Ludwig	V. Vshivkov
J. Gaudiot	V. Markova	P. Zoetewij
V. Gergel	G. Mauri	
C. Germain-Renaud	R. Merks	

Table of Contents

Theory

Mapping Affine Loop Nests: Solving of the Alignment and Scheduling Problems	1
<i>Evgeniya V. Adutskevich, Nikolai A. Likhoded</i>	
Situated Cellular Agents in Non-uniform Spaces	10
<i>Stefania Bandini, Sara Manzoni, Carla Simone</i>	
Accuracy and Stability of Spatial Dynamics Simulation by Cellular Automata Evolution.....	20
<i>Olga Bandman</i>	
Resource Similarities in Petri Net Models of Distributed Systems	35
<i>Vladimir A. Bashkin, Irina A. Lomazova</i>	
Authentication Primitives for Protocol Specifications	49
<i>Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Corrado Priami</i>	
An Extensible Coloured Petri Net Model of a Transport Protocol for Packet Switched Networks	66
<i>Dmitry J. Chaly, Valery A. Sokolov</i>	
Parallel Computing for Globally Optimal Decision Making	76
<i>V.P. Gergel, R.G. Strongin</i>	
Parallelization of Alternating Direction Implicit Methods for Three-Dimensional Domains	89
<i>V.P. Il'in, S.A. Litvinenko, V.M. Sveshnikov</i>	
Interval Approach to Parallel Timed Systems Verification	100
<i>Yuri G. Karpov, Dmitry Sotnikov</i>	
An Approach to Assessment of Heterogeneous Parallel Algorithms	117
<i>Alexey Lastovetsky, Ravi Reddy</i>	
A Hierarchy of Conditions for Asynchronous Interactive Consistency	130
<i>Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, Matthieu Roy</i>	
Associative Parallel Algorithms for Dynamic Edge Update of Minimum Spanning Trees	141
<i>Anna S. Nepomniaschaya</i>	

The Renaming Problem as an Introduction to Structures for Wait-Free Computing	151
<i>Michel Raynal</i>	
Graph Partitioning in Scientific Simulations: Multilevel Schemes versus Space-Filling Curves	165
<i>Stefan Schamberger, Jens-Michael Wierum</i>	
Process Algebraic Model of Superscalar Processor Programs for Instruction Level Timing Analysis	180
<i>Hee-Jun Yoo, Jin-Young Choi</i>	
Software	
Optimization of the Communications between Processors in a General Parallel Computing Approach Using the Selected Data Technique	185
<i>Hervé Bolvin, André Chambarel, Dominique Fougere, Petr Gladkikh</i>	
Load Imbalance in Parallel Programs	197
<i>Maria Calzarossa, Luisa Massari, Daniele Tessera</i>	
Software Carry-Save: A Case Study for Instruction-Level Parallelism	207
<i>David Defour, Florent de Dinechin</i>	
A Polymorphic Type System for Bulk Synchronous Parallel ML	215
<i>Frédéric Gava, Frédéric Loulergue</i>	
Towards an Efficient Functional Implementation of the NAS Benchmark FT	230
<i>Clemens Grelck, Sven-Bodo Scholz</i>	
Asynchronous Parallel Programming Language Based on the Microsoft .NET Platform	236
<i>Vadim Guzev, Yury Serdyuk</i>	
A Fast Pipelined Parallel Ray Casting Algorithm Using Advanced Space Leaping Method	244
<i>Hyung-Jun Kim, Yong-Je Woo, Yong-Won Kwon, So-Hyun Ryu, Chang-Sung Jeong</i>	
Formal Modeling for a Real-Time Scheduler and Schedulability Analysis	253
<i>Sung-Jae Kim, Jin-Young Choi</i>	
Disk I/O Performance Forecast Using Basic Prediction Techniques for Grid Computing	259
<i>DongWoo Lee, R.S. Ramakrishna</i>	
Glosim: Global System Image for Cluster Computing	270
<i>Hai Jin, Guo Li, Zongfen Han</i>	

Exploiting Locality in Program Graphs	276
<i>Joford T. Lim, Ali R. Hurson, Larry D. Pritchett</i>	
Asynchronous Timed Multimedia Environments Based on the Coordination Paradigm	291
<i>George A. Papadopoulos</i>	
Component-Based Development of Dynamic Workflow Systems Using the Coordination Paradigm	304
<i>George A. Papadopoulos, George Fakas</i>	
A Multi-threaded Asynchronous Language	316
<i>Hervé Paulino, Pedro Marques, Luís Lopes, Vasco Vasconcelos, Fernando Silva</i>	
An Efficient Marshaling Framework for Distributed Systems	324
<i>Konstantin Popov, Vladimir Vlassov, Per Brand, Seif Haridi</i>	
Deciding Optimal Information Dispersal for Parallel Computing with Failures	332
<i>Sung-Keun Song, Hee-Yong Youn, Jong-Koo Park</i>	
Parallel Unsupervised k -Windows: An Efficient Parallel Clustering Algorithm	336
<i>Dimitris K. Tasoulis, Panagiotis D. Alevizos, Basilis Boutsinas, Michael N. Vrahatis</i>	
Applications	
Analysis of Architecture and Design of Linear Algebra Kernels for Superscalar Processors	345
<i>Oleg Bessonov, Dominique Fougère, Bernard Roux</i>	
Numerical Simulation of Self-Organisation in Gravitationally Unstable Media on Supercomputers	354
<i>Elvira A. Kuksheva, Viktor E. Malyshkin, Serguei A. Nikitin, Alexei V. Snytnikov, Valery N. Snytnikov, Vitalii A. Vshivkov</i>	
Communication-Efficient Parallel Gaussian Elimination	369
<i>Alexander Tiskin</i>	
Alternative Parallelization Strategies in EST Clustering	384
<i>Nishank Trivedi, Kevin T. Pedretti, Terry A. Braun, Todd E. Scheetz, Thomas L. Casavant</i>	
Protective Laminar Composites Design Optimisation Using Genetic Algorithm and Parallel Processing	394
<i>Mikhail Alexandrovich Vishnevsky, Vladimir Dmitrievich Koshur, Alexander Ivanovich Legalov, Eugenij Moiseevich Mirkes</i>	

Tools

A Prototype Grid System Using Java and RMI	401
<i>Martin Alt, Sergei Gorlatch</i>	
Design and Implementation of a Cost-Optimal Parallel Tridiagonal System Solver Using Skeletons	415
<i>Holger Bischof, Sergei Gorlatch, Emanuel Kitzelmann</i>	
An Extended ANSI C for Multimedia Processing	429
<i>Patricio Bulić, Veselko Guštin, Ljubo Pipan</i>	
The Parallel Debugging Architecture in the Intel® Debugger	444
<i>Chih-Ping Chen</i>	
Retargetable and Tuneable Code Generation for High Performance DSP .	452
<i>Anatoliy Doroshenko, Dmitry Ragozin</i>	
The Instruction Register File	467
<i>Bernard Goossens</i>	
A High Performance and Low Cost Cluster-Based E-mail System	482
<i>Woo-Chul Jeun, Yang-Suk Kee, Jin-Soo Kim, Soonhoi Ha</i>	
The Presentation of Information in mpC Workshop Parallel Debugger ...	497
<i>A. Kalinov, K. Karganov, V. Khatzkeovich, K. Khorenko, I. Ledovskikh, D. Morozov, S. Savchenko</i>	
Grid-Based Parallel and Distributed Simulation Environment	503
<i>Chang-Hoon Kim, Tae-Dong Lee, Sun-Chul Hwang, Chang-Sung Jeong</i>	
Distributed Object-Oriented Web-Based Simulation	509
<i>Tae-Dong Lee, Sun-Chul Hwang, Jin-Lip Jeong, Chang-Sung Jeong</i>	
GEPARD – General Parallel Debugger for MVS-1000/M	519
<i>V.E. Malyshkin, A.A. Romanenko</i>	
Development of Distributed Simulation System	524
<i>Victor Okol'nishnikov, Sergey Rudometov</i>	
CMDE: A Channel Memory Based Dynamic Environment for Fault-Tolerant Message Passing Based on MPICH-V Architecture ...	528
<i>Anton Selikhov, Cécile Germain</i>	
DAxML : A Program for Distributed Computation of Phylogenetic Trees Based on Load Managed CORBA	538
<i>Alexandros P. Stamatakis, Markus Lindermeier, Michael Ott, Thomas Ludwig, Harald Meier</i>	

D-SAB: A Sparse Matrix Benchmark Suite	549
<i>Pyrrhos Stathis, Stamatis Vassiliadis, Sorin Cotofana</i>	
DOVE-G: Design and Implementation of Distributed Object-Oriented Virtual Environment on Grid	555
<i>Young-Je Woo, Chang-Sung Jeong</i>	
Author Index	569

Mapping Affine Loop Nests: Solving of the Alignment and Scheduling Problems

Evgeniya V. Adutskevich and Nickolai A. Likhoded

National Academy of Sciences of Belarus,
Institute of Mathematics,
Surganov str., 11 , Minsk 220072 BELARUS
{zhanya, likhoded}@im.bas-net.by

Abstract. The paper is devoted to the problem of mapping affine loop nests onto distributed memory parallel computers. An algorithm to find an efficient scheduling and distribution of data and operations to virtual processors is presented. It reduces the scheduling and the alignment problems to the solving of linear algebraic equations. The algorithm finds the maximal degree of pipelined parallelism and tries to minimize the number of nonlocal communications.

1 Introduction

A wide class of algorithms may be represented as affine loop nests (loops whose loop bounds and array accesses are affine functions of loop indices). The implementation of such algorithms in parallel computers is undoubtedly important.

While mapping affine loop nests onto distributed memory parallel computers it is necessary to distribute data and computations to processors and to determine the execution order of operations. A number of problems appear: scheduling [1,2,3], alignment [3,4,5,6], space-time mapping [6,7,8,9,10], blocking [7,9,11,12]. Scheduling is a high-level technique for parallelization of loop nests. Scheduling of a loop nest for parallel execution consists in transforming this nest into an equivalent one for which a number of loops can be executed in parallel. The alignment problem consists in mapping data and computations to processors with the aim of minimizing the communications. The problem of space-time mapping is to assign operations to processors and to express the execution order. Blocking is a technique to increase the granularity of computations, the locality of data references, and the computation-to-communication ratio. An essential stage of these techniques is to find linear or affine functions (scheduling functions, statement and array allocation functions) satisfying certain constraints.

One of the preferable parallelization schemes is to use several scheduling functions to achieve pipelined parallelism [8,9,11]. Such a scheme has a number of advantages: regular code, point-to-point synchronization, amenable to blocking. At the same time the alignment problem should still be solved.

In this paper, an efficient algorithm to implement pipelined parallelism and to solve the scheduling problem and the alignment problem is proposed. The simultaneous solving these problems allows us to choose scheduling functions and allocation functions which complement each other in the best way.

2 Main Definitions

Let an algorithm be represented by affine loop nest. Briefly, affine loop nest is the set of sequential programs consisting of arbitrary nestings and sequences of loops whose array indices and bounds of the loops are affine functions of outer loop indices or loop-invariant variables.

Let a loop nest contain K statements S_β and use L arrays a_l . By V_β denote the index domain of statement S_β , by W_l denote the index domain of array a_l . Let n_β be the number of surrounding loops of statement S_β , ν_l be the dimension of array a_l ; then $V_\beta \in \mathbb{Z}^{n_\beta}$, $W_l \in \mathbb{Z}^{\nu_l}$. By $\bar{F}_{l\beta q}(J)$ denote an affine expression that maps an iteration J to the array index computed by the q th access function in instruction S_β to array a_l : $\bar{F}_{l\beta q}(J) = F_{l\beta q}J + f^{(l,\beta,q)}$, $J \in V_\beta \subset \mathbb{Z}^{n_\beta}$, $F_{l\beta q} \in \mathbb{Z}^{\nu_l \times n_\beta}$, $f^{(l,\beta,q)} \in \mathbb{Z}^{\nu_l}$.

Given a statement S_β , a computation instance of S_β is called the operation and is denoted by $S_\beta(J)$, where J is the iteration vector (the vector whose components are the values of the surrounding loop indices).

There is a dependence between operations $S_\alpha(I)$ and $S_\beta(J)$ ($S_\alpha(I) \rightarrow S_\beta(J)$) if: 1) $S_\alpha(I)$ is executed before $S_\beta(J)$; 2) $S_\alpha(I)$ and $S_\beta(J)$ refer to a memory location M , and at least one of these references is write; 3) the memory location M is not written between iteration I and iteration J .

Let $P = \{(\alpha, \beta) \mid \exists I \in V_\alpha, J \in V_\beta, S_\alpha(I) \rightarrow S_\beta(J)\}$, $V_{\alpha,\beta} = \{J \in V_\beta \mid \exists I \in V_\alpha, S_\alpha(I) \rightarrow S_\beta(J)\}$. The set P determines the pairs of dependent operations. Let $\bar{\Phi}_{\alpha,\beta}: V_{\alpha,\beta} \rightarrow V_\alpha$ be a dependence functions: if $S_\alpha(I) \rightarrow S_\beta(J)$, $I \in V_\alpha$, $J \in V_{\alpha,\beta} \subset V_\beta$, then $I = \bar{\Phi}_{\alpha,\beta}(J)$. Suppose $\bar{\Phi}_{\alpha,\beta}$ are affine functions: $\bar{\Phi}_{\alpha,\beta}(J) = \Phi_{\alpha,\beta}J - \varphi^{(\alpha,\beta)}$, $J \in V_{\alpha,\beta}$, $(\alpha, \beta) \in P$, $\Phi_{\alpha,\beta} \in \mathbb{Z}^{n_\alpha \times n_\beta}$, $\varphi^{(\alpha,\beta)} \in \mathbb{Z}^{n_\alpha}$.

Let function $t^{(\beta)}: V_\beta \rightarrow \mathbb{Z}$, $1 \leq \beta \leq K$, assign an integer $t^{(\beta)}(J)$ to each operation $S_\beta(J)$. Let $t^{(\beta)}$ be a generalized scheduling function (g-function). This means that

$$t^{(\beta)}(J) \geq t^{(\alpha)}(\Phi_{\alpha,\beta}J - \varphi^{(\alpha,\beta)}), \quad J \in V_{\alpha,\beta}, (\alpha, \beta) \in P. \quad (1)$$

In other words, if $S_\alpha(I) \rightarrow S_\beta(J)$, $I = \Phi_{\alpha,\beta}J - \varphi^{(\alpha,\beta)}$, then $S_\beta(J)$ is executed in the same iteration as $S_\alpha(I)$ or $S_\beta(J)$ is executed in an iteration that comes after the iteration that executes $S_\alpha(I)$. Suppose $t^{(\beta)}$ are affine functions: $t^{(\beta)}(J) = \tau^{(\beta)}J + a_\beta$, $1 \leq \beta \leq K$, $J \in V_\beta$, $\tau^{(\beta)} \in \mathbb{Z}^{n_\beta}$, $a_\beta \in \mathbb{Z}$.

3 Statement of the Problem

We shall exploit a pipelined parallelism. Pipelining has many benefits over wavefronting: barriers are reduced to point-to-point synchronizations, processors need not work on the same wavefront at the same time, the SPMD code to implement pipelining is simpler, the processors tend to have better data locality [8,9].

If there are n independent sets of g-functions $t^{(1)}, \dots, t^{(K)}$, then there is a way to implement pipelined parallelism. The way is to use any $n - 1$ of the sets as components of an $(n - 1)$ -dimensional spatial mapping, and to use the remaining set to serialize the computations assigned to each processor; blocking

can be used to reduce the frequency and volume of the computations. Thus we consider g-functions $t^{(1)}, \dots, t^{(K)}$ both scheduling and allocation functions.

This parallelization scheme solves the problem of space-time mapping loop nests onto virtual processors. The purpose of this paper is to propose an algorithm that exploits a pipelined parallelism and solves both the problem of space-time mapping and the problem of aligning data and computations.

Let functions $d^{(l)}: W_l \rightarrow \mathbb{Z}$, $1 \leq l \leq L$, determine which processor each array element is allocated to. Suppose $d^{(l)}$ are affine functions: $d^{(l)}(F) = \eta^{(l)}F + y_l$, $1 \leq l \leq L$, $F \in W_l$, $\eta^{(l)} \in \mathbb{Z}^{\nu_l}$, $y_l \in \mathbb{Z}$.

Functions $t^{(\beta)}$ and $d^{(l)}$ are to satisfy some constraints.

It follows from (1) that $\tau^{(\beta)}J + a_\beta \geq \tau^{(\alpha)}(\Phi_{\alpha,\beta}J - \varphi^{(\alpha,\beta)}) + a_\alpha$, that is

$$(\tau^{(\beta)} - \tau^{(\alpha)}\Phi_{\alpha,\beta})J + \tau^{(\alpha)}\varphi^{(\alpha,\beta)} + a_\alpha - a_\beta \geq 0, \quad J \in V_{\alpha,\beta}, \quad (\alpha, \beta) \in P, \quad (2)$$

for all n sets of g-functions.

Let $t^{(1)}, \dots, t^{(K)}$ be one of the $n - 1$ sets of allocation functions. Operation $S_\beta(J)$ is assigned to execute at virtual processor $t^{(\beta)}(J)$. Array element $a_l(\overline{F}_{l\beta q}(J))$ is stored in the local memory of processor $d^{(l)}(F_{l\beta q}(J))$. Consider the expressions $\delta_{l\beta q}(J) = t^{(\beta)}(J) - d^{(l)}(\overline{F}_{l\beta q}(J))$. The communication of length $\delta_{l\beta q}(J)$ is equal to the distance between $S_\beta(J)$ and $a_l(\overline{F}_{l\beta q}(J))$. Since $\delta_{l\beta q}(J) = \tau^{(\beta)}J + a_\beta - (\eta^{(l)}\overline{F}_{l\beta q}(J) + y_l) = \tau^{(\beta)}J + a_\beta - \eta^{(l)}(F_{l\beta q}J + f^{(l,\beta,q)}) - y_l = (\tau^{(\beta)} - \eta^{(l)}F_{l\beta q})J + a_\beta - \eta^{(l)}f^{(l,\beta,q)} - y_l$, we obtain the conditions for only fixed-size (independent of J) communications:

$$\tau^{(\beta)} - \eta^{(l)}F_{l\beta q} = 0. \quad (3)$$

The aim of further research is to obtain n independent sets of functions $t^{(\beta)}$ and $n - 1$ sets of functions $d^{(l)}$ such that 1) for all n sets of $t^{(\beta)}$ conditions (2) are valid; 2) n is as large as possible; 3) for $n - 1$ sets of $t^{(\beta)}$ and $d^{(l)}$ conditions (3) are valid for as many l, β, q as possible.

4 Main Results

First let us introduce some notation:

$$\sigma_0 = 0, \quad \sigma_j = \sum_{i=1}^j n_i, \quad 1 \leq j \leq K, \quad \sigma_{K+j} = \sigma_K + \sum_{i=1}^j \nu_i, \quad 1 \leq j \leq L;$$

$x = (\tau^{(1)}, \dots, \tau^{(K)}, \eta^{(1)}, \dots, \eta^{(L)}, a_1, \dots, a_K)$ is a vector of order $\sigma_{K+L} + K$ whose entries are parameters of functions $t^{(\beta)}$ and $d^{(l)}$;

$0^{i \times j}$ is a null $i \times j$ matrix;

$E^{(i)}$ is an identity $i \times i$ matrix;

$0^{(i)}$ is the zero column vector of size i ;

$e_j^{(i)}$ is a column vector of order i whose entries are all zeros except that the j th entry is equal to unity;

$$\tilde{\Phi}_{\alpha,\beta} = \begin{pmatrix} 0^{\sigma_{\beta-1} \times n_\beta} \\ E^{(n_\beta)} \\ 0^{(\sigma_{K+L} - \sigma_\beta + K) \times n_\beta} \end{pmatrix} - \begin{pmatrix} 0^{\sigma_{\alpha-1} \times n_\beta} \\ \Phi_{\alpha,\beta} \\ 0^{(\sigma_{K+L} - \sigma_\alpha + K) \times n_\beta} \end{pmatrix};$$

$$\tilde{\varphi}^{(\alpha, \beta)} = \begin{pmatrix} 0^{(\sigma_{\alpha-1})} \\ \varphi^{(\alpha, \beta)} \\ 0^{(\sigma_{K+L}-\sigma_{\alpha}+K)} \end{pmatrix} + e_{\sigma_{K+L}+\beta}^{(\sigma_{K+L}+K)} - e_{\sigma_{K+L}+\alpha}^{(\sigma_{K+L}+K)};$$

$$\Delta_{l\beta q} = \begin{pmatrix} 0^{\sigma_{\beta-1} \times n_{\beta}} \\ E^{(n_{\beta})} \\ 0^{(\sigma_{K+L}-\sigma_{\beta}+K) \times n_{\beta}} \end{pmatrix} - \begin{pmatrix} 0^{\sigma_{K+l-1} \times n_{\beta}} \\ F_{l\beta q} \\ 0^{(\sigma_{K+L}-\sigma_{K+l}+K) \times n_{\beta}} \end{pmatrix}.$$

With the notation, conditions (2) and (3) can be written in the form

$$x\tilde{\Phi}_{\alpha, \beta} J + x\tilde{\varphi}^{(\alpha, \beta)} \geq 0, \quad J \in V_{\alpha, \beta}, \quad (4)$$

$$x\Delta_{l\beta q} = 0. \quad (5)$$

Now we state sufficient conditions ensuring the fulfillment of constraints (4) for some practical important cases.

Lemma. Let $(\alpha, \beta) \in P$, $p^{(\alpha, \beta)}$ be a vector such that $p^{(\alpha, \beta)} \leq J$ for all $J \in V_{\alpha, \beta}$. Constraints (4) are valid for any values of outer loop indices if

$$x(\tilde{\Phi}_{\alpha, \beta} p^{(\alpha, \beta)} + \tilde{\varphi}^{(\alpha, \beta)}) \geq 0. \quad (6)$$

and one of the following sets of conditions is valid:

$$1. \quad x\tilde{\Phi}_{\alpha, \beta} \geq 0; \quad (7)$$

$$2. \quad J_{k_1} \leq J_{k_2} + q^{(\alpha, \beta)} \text{ for all } J = (J_1, \dots, J_{n_{\beta}}) \in V_{\alpha, \beta}, \quad p_{k_1}^{(\alpha, \beta)} = p_{k_2}^{(\alpha, \beta)} + q^{(\alpha, \beta)}, \\ q^{(\alpha, \beta)} \in \mathbb{Z},$$

$$x(\tilde{\Phi}_{\alpha, \beta})_k \geq 0, \quad k \neq k_1, \quad x((\tilde{\Phi}_{\alpha, \beta})_{k_1} + (\tilde{\Phi}_{\alpha, \beta})_{k_2}) \geq 0, \quad (8)$$

where $(\tilde{\Phi}_{\alpha, \beta})_k$ denotes the k th column of matrix $\tilde{\Phi}_{\alpha, \beta}$;

$$3. \quad J_{k_1} \leq J_{k_2} + q_1^{(\alpha, \beta)}, \quad J_{k_1} \leq J_{k_3} + q_2^{(\alpha, \beta)} \text{ for all } J = (J_1, \dots, J_{n_{\beta}}) \in V_{\alpha, \beta}, \\ p_{k_1}^{(\alpha, \beta)} = p_{k_2}^{(\alpha, \beta)} + q_1^{(\alpha, \beta)} = p_{k_3}^{(\alpha, \beta)} + q_2^{(\alpha, \beta)}, \quad q_1^{(\alpha, \beta)}, \quad q_2^{(\alpha, \beta)} \in \mathbb{Z},$$

$$x(\tilde{\Phi}_{\alpha, \beta})_k \geq 0, \quad k \neq k_1, \quad x((\tilde{\Phi}_{\alpha, \beta})_{k_1} + (\tilde{\Phi}_{\alpha, \beta})_{k_2} + (\tilde{\Phi}_{\alpha, \beta})_{k_3}) \geq 0. \quad (9)$$

Proof. Write condition (4) in the form

$$x\tilde{\Phi}_{\alpha, \beta}(J - p^{(\alpha, \beta)}) + x(\tilde{\Phi}_{\alpha, \beta} p^{(\alpha, \beta)} + \tilde{\varphi}^{(\alpha, \beta)}) \geq 0, \quad J \in V_{\alpha, \beta}. \quad (10)$$

If conditions (6), (7) are valid, then conditions (10) are valid; hence (4) are valid.

Denote $S_{k_1, k_2} = x(\tilde{\Phi}_{\alpha, \beta})_{k_1}(J_{k_1} - p_{k_1}^{(\alpha, \beta)}) + x(\tilde{\Phi}_{\alpha, \beta})_{k_2}(J_{k_2} - p_{k_2}^{(\alpha, \beta)})$, then

$$x\tilde{\Phi}_{\alpha, \beta}(J - p^{(\alpha, \beta)}) = \sum_{k \neq k_1, k \neq k_2} x(\tilde{\Phi}_{\alpha, \beta})_k(J_k - p_k^{(\alpha, \beta)}) + S_{k_1, k_2}. \quad (11)$$

Write S_{k_1, k_2} in the form $S_{k_1, k_2} = x(\tilde{\Phi}_{\alpha, \beta})_{k_1}(J_{k_1} - p_{k_1}^{(\alpha, \beta)}) + x(\tilde{\Phi}_{\alpha, \beta})_{k_2}((J_{k_1} - p_{k_1}^{(\alpha, \beta)}) + (J_{k_2} - J_{k_1} + q_1^{(\alpha, \beta)}) + (p_{k_1}^{(\alpha, \beta)} - p_{k_2}^{(\alpha, \beta)} - q_1^{(\alpha, \beta)})) = (J_{k_1} - p_{k_1}^{(\alpha, \beta)})(x(\tilde{\Phi}_{\alpha, \beta})_{k_1} +$

$x(\tilde{\Phi}_{\alpha,\beta})_{k_2}) + x(\tilde{\Phi}_{\alpha,\beta})_{k_2}(J_{k_2} - J_{k_1} + q_1^{(\alpha,\beta)})$. If (8) are valid, then the right part of (11) is nonnegative, i.e. $x\tilde{\Phi}_{\alpha,\beta} \geq 0$. If (6) are also valid, then (10) are valid; hence (4) are valid.

The sufficiency of conditions (6), (9) can be proved analogously. $\square$

Let us remark that sufficient conditions formulated in Lemma are necessary if $p^{(\alpha,\beta)} \in V_{\alpha,\beta}$, functions $\tilde{\Phi}_{\alpha,\beta}$, $t^{(\alpha)}$, $t^{(\beta)}$ are independent of outer loop indices and domain $V_{\alpha,\beta}$ is large enough.

Let us introduce in the consideration the following matrices.

D_1 is a matrix whose columns are nonzero and not identical vectors $\tilde{\Phi}_{\alpha,\beta}p^{(\alpha,\beta)} + \tilde{\varphi}^{(\alpha,\beta)}$ and columns of the matrices $\tilde{\Phi}_{\alpha,\beta}$. Let the matrix D_1 have μ_1 columns: $D_1 \in \mathbb{Z}^{(\sigma_{K+L}+K) \times \mu_1}$.

D_2 is a matrix whose columns are not identical columns of the matrices $\Delta_{l\beta q}$. Let the matrix D_2 have μ_2 columns: $D_2 \in \mathbb{Z}^{(\sigma_{K+L}+K) \times \mu_2}$.

$D = (D_1|D_2)$, $D \in \mathbb{Z}^{(\sigma_{K+L}+K) \times (\mu_1+\mu_2)}$.

B is a matrix obtained by elementary row transformations of D . It is valid $B = PD$, where matrix $P \in \mathbb{Z}^{(\sigma_{K+L}+K) \times (\sigma_{K+L}+K)}$ can be constructed by applying the same row transformations to the identity matrix.

Theorem. *Suppose leading μ_1 elements of a certain row of B are nonnegative and the next μ_2 elements are zeros; then the corresponding row of P determines the vector x whose entries are parameters of functions $t^{(\beta)}$ and $d^{(l)}$ such that $t^{(\beta)}$ are g -functions (i.e. conditions (2) are valid) and $t^{(\beta)}$, $d^{(l)}$ determine a one-dimensional spatial mapping onto virtual processors with only fixed-size communications (i.e. conditions (3) are valid). If not all the μ_2 elements are zero, then the number of zeros characterizes the number of nonlocal (depending of J) communications.*

Proof. Write conditions (6), (7), (5) in the vector-matrix form $\begin{cases} xD_1 \geq 0, \\ xD_2 = 0. \end{cases}$ The solution of the system is the solution of system $\begin{cases} xD_1 = (z_1, \dots, z_{\mu_1}), \\ xD_2 = (z_{\mu_1+1}, \dots, z_{\mu_1+\mu_2}) \end{cases}$ or

$$(x|z) \begin{pmatrix} D \\ -E^{(\mu_1+\mu_2)} \end{pmatrix} = 0, \quad z = (z_1, \dots, z_{\mu_1+\mu_2}), \quad (12)$$

provided that $z_1, \dots, z_{\mu_1}$ are nonnegative and $z_{\mu_1+1}, \dots, z_{\mu_1+\mu_2}$ are zeros. By assumption, the row of B provides these requirements; let the row be the i th row of B , $(B)_i$. Besides, $(B)_i$ satisfies system (12) because of any row of $(B|P)$ satisfies:

$$(P|B) \begin{pmatrix} D \\ -E^{(\mu_1+\mu_2)} \end{pmatrix} = (P|PD) \begin{pmatrix} D \\ -E^{(\mu_1+\mu_2)} \end{pmatrix} = PD - PD = 0. \text{ Thus the first}$$

statement of the theorem is proved. To prove the second statement suppose that not all μ_2 elements of the row $(B)_i$ are equal to zero. If any element of $(B)_i$ is not zero, then $(P)_i \Delta_{l\beta q} \neq 0$ for some l, β, q . This implies that there is a nonlocal (depending on J) communication. $\square$

Composing matrix D we keep in mind sufficient conditions (6), (7). If we use conditions (6), (8), then the sum of the k_1 th and the k_2 th columns of $\tilde{\Phi}_{\alpha,\beta}$ is included in D_1 instead of the k_1 th column. If we use conditions (6), (9), then the sum of the k_1 th, the k_2 th, the k_3 th columns of $\tilde{\Phi}_{\alpha,\beta}$ is included in D_1 instead of the k_1 th column.

The following algorithm is based on the proved Theorem.

Algorithm (search of pipelined parallelism and minimization of the number of nonlocal communications)

1. Compose the matrix $D \in \mathbb{Z}^{(\sigma_{K+L+K}) \times (\mu_1 + \mu_2)}$.
2. Obtain the matrix $(P|H)$ by elementary row transformations of the matrix $(E^{(\sigma_{K+L+K})}|D)$, where H is the normal Hermite form of the matrix D (up to a permutation of rows and columns)
3. Obtain the matrix $(P|B)$ by addition of rows of the matrix $(P|H)$ with a view to derive as many nonnegative leading μ_1 elements of the rows of B as possible and to derive as many zeros next μ_2 elements of the rows of B as possible.
4. Choose n rows of $(P|B)$ such that the rows of P are nondegenerate, leading μ_1 elements of the rows of B are nonnegative and $n-1$ rows of B have as many zeros among the next μ_2 elements as possible. Use the elements of $n-1$ row of P as the components of an $(n-1)$ -dimensional spatial mapping (defined by $t^{(\beta)}$ and $d^{(l)}$) of operations and data. Use the elements of the remaining row as the components of scheduling functions $t^{(\beta)}$.

It should be noted that any solution of (2) can be found as a linear combination of rows of the matrix P . Thus the algorithm can find the maximal number of independent sets of functions $t^{(\beta)}$ determining the pipelined parallelism.

5 Example

Let $A = (a_{ij})$, $1 \leq i, j \leq N$, be a lower triangular matrix, $a_{ij} = 0$, $i < j$, $a_{ii} = 1$, $1 \leq i \leq N$. Consider a solution algorithm for a system of linear algebraic equations $Ax = B$:

```

S1 : x[1] = b[1]
for (i = 2 to N) do
  S2 : x[i] = b[i];
  for (j = 1 to i-1) do
    S3 : x[i] = x[i] - a[i,j]x[j];

```

The loop nest has three statements S_1 , S_2 , S_3 and elements of three arrays a , b , x ; $n_1 = 0$, $n_2 = 1$, $n_3 = 2$, $\nu_1 = 2$, $\nu_2 = \nu_3 = 1$, $V_1 = \{(1)\}$, $V_2 = \{(i) \in \mathbb{Z} \mid 2 \leq i \leq N\}$, $V_3 = \{(i, j) \in \mathbb{Z}^2 \mid 2 \leq i \leq N, 1 \leq j \leq i-1\}$, $W_1 = V_3$, $W_2 = W_3 = \{(i) \in \mathbb{Z} \mid 1 \leq i \leq N\}$; $\bar{F}_{131}(i, j) = E^{(2)}(i, j)^T$, $\bar{F}_{211}(1) = E^{(1)}(1)$, $\bar{F}_{221}(i) = E^{(1)}(i)$, $\bar{F}_{311}(1) = E^{(1)}(1)$, $\bar{F}_{321}(i) = E^{(1)}(i)$, $\bar{F}_{331}(i, j) = \bar{F}_{332}(i, j) = (1 \ 0)(i, j)^T$, $\bar{F}_{333}(i, j) = (0 \ 1)(i, j)^T$; $\bar{\Phi}_{1,3}(i, 1) = (0 \ 0)(i \ 1)^T + 1$, $(i, 1) \in V_{1,3} = \{(i, 1) \in \mathbb{Z}^2 \mid 2 \leq i \leq N\}$, $\bar{\Phi}_{2,3}(i, 1) = (1 \ 0)(i \ 1)^T$, $(i, 1) \in V_{2,3} = V_{1,3}$, $\bar{\Phi}_{3,3}^{(1)}(i, j) = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} (i, j)^T - (0 \ 1)^T$, $(i, j) \in V_{3,3} =$

$$\{(i, j) \in \mathbb{Z}^2 \mid 3 \leq i \leq N, 2 \leq j \leq i-1\}, \quad \bar{\Phi}_{3,3}^{(2)}(i, j) = E^{(2)}(i, j)^T - (0 \ 1)^T, \\ (i, j) \in V_{3,3}^{(2)} = V_{3,3}^{(1)}.$$

We have $x = (\tau_1^{(2)}, \tau_1^{(3)}, \tau_2^{(3)}, \eta_1^{(1)}, \eta_2^{(1)}, \eta_1^{(2)}, \eta_1^{(3)}, a_1, a_2, a_3)$ (the vector $\tau^{(1)}$ is 0-dimensional and it does not enter into x); $\sigma_0 = 0, \sigma_1 = 0, \sigma_2 = 1, \sigma_3 = 3, \sigma_4 = 5, \sigma_5 = 6, \sigma_6 = 7$;

$$\begin{aligned} \tilde{\Phi}_{1,3} &= \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T, \quad \tilde{\varphi}^{(1,3)} = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ -1 \ 0 \ 1)^T, \\ \tilde{\Phi}_{2,3} &= \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T, \quad \tilde{\varphi}^{(2,3)} = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ -1 \ 1)^T, \\ \tilde{\Phi}_{3,3}^{(1)} &= \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T, \quad \tilde{\varphi}^{(3,3)(1)} = (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)^T, \\ \tilde{\Phi}_{3,3}^{(2)} &= 0, \quad \tilde{\varphi}^{(3,3)(2)} = \tilde{\varphi}^{(3,3)(1)}; \\ p^{(1,3)} &= p^{(2,3)} = (2, 1), \quad p^{(3,3)(1)} = p^{(3,3)(2)} = (3, 2); \\ J_2 &\leq J_1 - 1 \text{ for all } J \in V_{3,3}^{(1)} = V_{3,3}^{(2)}, \quad p_2^{(3,3)(1)} = p_1^{(3,3)(1)} - 1, \quad p_2^{(3,3)(2)} = p_1^{(3,3)(2)} - 1; \\ \Delta_{131} &= \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T, \quad \Delta_{221} = (1 \ 0 \ 0 \ 0 \ 0 \ -1 \ 0 \ 0 \ 0 \ 0)^T, \\ \Delta_{321} &= (1 \ 0 \ 0 \ 0 \ 0 \ 0 \ -1 \ 0 \ 0 \ 0)^T, \quad \Delta_{333} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix}^T, \\ \Delta_{331} &= \Delta_{332} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T. \end{aligned}$$

According to the algorithm we compose the matrix

$$D = \begin{pmatrix} 0 & -2 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

By $R_i + aR_j$ denote the following elementary row transformation: to add the row j multiplied by a to the row i . By $-R_i$ denote the sign reversal of the elements of the row i . We make the following elementary row transformations of $(E^{(10)}|D)$: $R_2 + 2R_8, R_3 + R_8, R_{10} + R_8, R_1 - 2R_9, R_2 + 2R_9, R_3 + R_9, R_{10} + R_9, -R_8, -R_9, R_2 + R_4, R_3 + R_5, R_1 + R_6, -R_4, -R_5, -R_6, -R_7, -R_1, R_2 - R_1, R_1 + R_7, R_2 - R_7$ and obtain the matrix $(P'|H)$. The matrix $(P'|H)$ is also the matrix $(P|B)$.

$$(P|B) = \left(\begin{array}{cccccccccc|cccccccccccc} -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right).$$

Then we choose the second and the third rows of $(P|B)$. It follows from the Theorem that the second row of P determines the components of a one-dimensional spatial mapping that results in one nonlocal communication (there are unities in the 13th and the 14th columns of B). We use the elements of the third row of P as the components of scheduling functions. Thus, we have $t^{(1)} = 2$, $t^{(2)}(i) = i$, $t^{(3)}(i, j) = i$ for mapping the operations, and $\eta^{(1)}(i, j) = i$, $\eta^{(2)}(i) = i$, $\eta^{(3)}(i) = i$ for mapping the data, and $t^{(1)} = 1$, $t^{(2)}(i) = 1$, $t^{(3)}(i, j) = j$ for scheduling the operations.

According to the functions obtained we write the SPMD code for the algorithm. The processor's ID is denoted by p ; the i th wait(q) executed by processor p stalls execution until processor q executes the i th signal(p).

```

if (1 < p < N+1) then
  for (t = 1 to p-1) do
    if (p > 2 and t = 1) then wait(p-1);
    if (p = 2) then
      S1: x[1] = b[1];
    if (t = 1) then
      S2: x[p] = b[p];
    S3: x[p] = x[p] - a[p,t]x[t];
    if (p < N and t = 1) then signal (p+1);

```

6 Conclusion

Thus we present a new method for the mapping of affine loop nests onto distributed memory parallel computers. The aim is to obtain pipelined parallelism and to minimize the number of nonlocal communications in the target virtual architecture.

The main theoretical and technical contributions of the paper are: the reduction of scheduling and alignment problems to the solving system of linear algebraic equations; the statement and proof of conditions such that the solution of the system is the solution of the problems; the algorithm realizing parallelization scheme based on pipelining and taking into account alignment problem. The algorithm can be used for automatic parallelization.

Further work could be oriented towards a generalization of the presented method: to consider scheduling and allocation functions depending on outer parameters; to take into account not only nonlocal but also local communications, communication-free partitions; to consider the constraints for data reuse.

References

1. Darte, A., Robert, Y.: Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *J. of Parallel and Distrib. Computing* **29** (1) (1995) 43–59
2. Feautrier, P.: Some efficient solutions to the affine scheduling problem. *Int. J. of Parallel Programming* **21** (5,6) (1992) 313–348,389–420
3. Voevodin, V.V., Voevodin, V.I.: *Parallel computing* (St. Petersburg, BHV-Petersburg, 2002) (in Russian)
4. Dion, M., Robert, Y.: Mapping affine loop nests. *Parallel Computing* **22** (1996) 1373–1397
5. Frolov, A.V.: Optimization of arrays allocation in FORTRAN programs for multiprocessor computing systems. *Programming and Computer Software* **24** (1998) 144–154
6. Lee, H.-J., Fortes, J. A. B.: Automatic generation of modular time-space mappings and data alignments. *J. of VLSI Signal Processing* **19** (1998) 195–208
7. Darte, A., Robert, Y.: Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing* **20** (1994) 679–710
8. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing* **24** (3,4) (1998) 445–475
9. Lim, A.W., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. *Proceedings of the 1-sth ACM SIGARCH International Conference on Supercomputing* (1999)
10. Bakhanovich, S.V., Likhoded, N.A.: A method for parallelizing algorithms by vector scheduling functions. *Programming and Computer Software* **27** (4) (2001) 194–200
11. Frolov, A.V.: Finding and using directed cuts of real graphs of algorithms. *Programming and Computer Software* **23** (4) (1997) 230–239
12. Lim, A.W., Liao, S.-W., Lam, M.S.: Blocking and array contraction across arbitrary nested loops using affine partitioning. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Programming Languages* (2001)

Situated Cellular Agents in Non-uniform Spaces^{*}

Stefania Bandini, Sara Manzoni, and Carla Simone

Department of Informatics, Systems and Communication

University of Milano-Bicocca

Via Bicocca degli Arcimboldi 8 20126 Milan - Italy

{bandini, manzoni, simone}@disco.unimib.it

Abstract. This paper presents Situated Cellular Agents (SCA), a special class of Multilayered Multi Agent Situated Systems (MMASS). Situated Cellular Agents are systems of reactive agents that are heterogeneous (i.e. characterized by different behavior and perceptive capabilities), and populate a single layered structured environment. The structure of this environment is defined as a non-uniform network of sites in which the agents are situated. The behavior of Situated Cellular Agents (i.e. change of state and position) is influenced by states and types of agents that are situated in adjacent and at-a-distance sites. In the paper it will be outlined an ongoing project whose aim is to develop a set of tools to support the development and execution of SCA application. In particular it will be described the algorithm designed and implemented to manage field diffusion throughout structurally non-uniform environments.

1 Introduction

The paper presents Situated Cellular Agents (SCA) that is, systems of reactive agents situated in environments characterized by a non-uniform structure. The behavior of Situated Cellular Agents is influenced by spatially adjacent as well as by at-a-distance agents. In the latter case this happens according to a field emission-propagation-perception mechanism. Situated Cellular Agents constitute a special class of Multilayered Multi Agent Situated Systems (MMASS [1]). The MMASS has been designed for applications to Multi Agent Based Simulation (MABS) in complex domains that are intrinsically distributed and, thus, require distributed approaches to modelling and computation. The Multi Agent Systems (MAS [2]) approach can be used to simulate many types of artificial worlds as well as natural phenomena [3,4,5]. MABS is based on the idea that it is possible to represent a phenomenon as the result of the interactions of an assembly of simple agents with their own operational autonomy [2].

A SCA is an heterogeneous MAS, where agents with different features, abilities and perceptive capabilities coexist and interact in a structured environment

^{*} The work presented in this paper has been partially funded by the Italian Ministry of University and Research within the project ‘Cofinanziamento Programmi di Ricerca di Interesse Nazionale’

(i.e. space). Each situated agent is associated with a site of this space and agents' behavior is strongly influenced by its position. Spatial relationships among situated agents are derived by spatial relationships among the sites they are situated in. This means, for instance, that adjacent agents correspond to agents situated in spatially adjacent sites. Agent interactions are spatial dependent: agent behavior is influenced by other agents (i.e. by their presence or by the signals they emit), and both type of interactions are strongly dependent of the spatial structure of the agent environment. Agent presence is perceived only in the agent neighborhood (i.e. adjacent sites) while signals propagate according to the environment structure. Both agent state and position can be changed by the agent itself according to a perception–deliberation–action mechanism. Each agent, after the perception of signals emitted by other agents, selects the action to be undertaken (according to its state, position and type) and executes it. Agents are heterogeneous that is, they are characterized by a type that determines their abilities and perceptive capabilities (e.g. sensitivity to external stimuli). A language to specify agent behavior according to an action model based on reaction–diffusion metaphor has been described in [6].

Basic mechanisms that are shared by SCA applications (e.g. field diffusion throughout a non–uniform structured environment, conflict resolution on sites within the set of mobile agents) are going to be tackled within a project whose aim is to provide developers with tools to facilitate and support the development and execution of applications based on the SCA model. In this paper, after a description of Situated Cellular Agents (Section 2), this project will be briefly described and some details on an algorithm designed and implemented to manage field diffusion in a structurally non–uniform network of sites will be given (Section 3). Finally two application contexts of the SCA model will be described in Section 4 even if the description of these applications are out of the scope of this paper.

2 Situated Cellular Agents

A system of Situated Cellular Agents can be denoted by:

$$< Space, F, A >$$

where *Space* is the single layered structured environment where the set *A* of agents is situated, acts autonomously and interacts via the propagation of the set *F* of fields. The *Space* is defined as made up of a set *P* of sites arranged in a network (i.e. an undirect graph of sites). Each *site* $p \in P$ can contain at most one agent and is defined by $< a_p, F_p, P_p >$. $a_p \in A \cup \{\perp\}$ is the agent situated in *p* ($a_p = \perp$ when no agent is situated in *p*, that is *p* is empty); $F_p \subset F$ is the set of fields active in *p* ($F_p = \emptyset$ when no field is active in *p*); and $P_p \subset P$ is the set of sites adjacent to *p*.

An agent $a \in A$ is defined by $< s, p, \tau >$, where: $s \in \Sigma_\tau$ denotes the *agent state* and can assume one of the values specified by its type; $p \in P$ is the site of the *Space* where the agent is situated; and τ is the *agent type* describing the

set of states the agent can assume, a function to express agent sensitivity to fields emitted by other agents and propagating throughout the space (see field definition in the following), and the set of actions that the agent can perform.

Agent heterogeneity allows to define different abilities and perceptive capabilities to agents according to their type. The action set that is specified by their type defines agents ability to emit fields in order to communicate their state, to move along the space edges and to change their state. Moreover, agent type defines the set of states that agents can assume and their capability to perceive fields emitted by other agents. Thus, an agent type τ is defined by

$$\langle \Sigma_\tau, Perception_\tau, Action_\tau \rangle$$

where: Σ_τ defines the set of states that agents of type τ can assume. $Perception_\tau : \Sigma_\tau \rightarrow [\mathbb{N} \times W_{f_1}] \dots [\mathbb{N} \times W_{f_{|F|}}]$ is a function associating to each agent state the vector of pairs

$$\left(c_\tau^1(s), t_\tau^1(s) \right), \left(c_\tau^2(s), t_\tau^2(s) \right), \dots, \left(c_\tau^{|F|}(s), t_\tau^{|F|}(s) \right)$$

where for each i ($i = 1 \dots |F|$), $c_\tau^i(s)$ and $t_\tau^i(s)$ express respectively a coefficient to be applied to the field value f_i and the agent sensibility threshold to f_i in the given state s . In this way, agents situated at the same distance from the agent that emits a field can have different perceptive capabilities of it.

$Actions_\tau$ denotes the set of actions that agents of type τ can perform. $Actions_\tau$ specifies whether and how agents change their state and/or position, how they interact with other agents, and how neighboring and at-a-distance agents can influence them. Specifically, *trigger* defines how the perception of a field causes a change of state in the receiving agent, while *transport* defines how the perception of a field causes a change of position in the receiving agent.

The behavior of Situated Cellular Agents is influenced by at-a-distance agents through a field emission–diffusion–perception mechanism. Agents can communicate their state and thus influence non-adjacent agent by the emission of fields. Field diffusion along the space allows other agents to perceive it. $Perception_\tau$ function, characterizing each agent type, defines the possible reception of broadcast messages conveyed through a field, if the sensitivity of the agent to the field is such that it can perceive it. This means that a field can be neglected by an agent of type τ if its value at the site where the agent is situated is less than the sensitivity threshold computed by the second component of the $Perception_\tau$ function. This means that an agent of type τ in state $s \in \Sigma_\tau$ can perceive a field f_i only when it is verified

$$Compare_{f_i}(c_\tau^i(s) \cdot w_{f_i}, t_\tau^i(s))$$

that is, when the first component of the i -th pair of the perception function (i.e. $c_\tau^i(s)$) multiplied for the received field value w_{f_i} is greater than the second component of the pair (i.e. $t_\tau^i(s)$). This is the very essence of the broadcast interaction pattern, in which messages are not addressed to specific receivers but potentially to all agents populating the space.

The set of values that a field emitted by agents of type τ can assume are denoted by the pair $\langle w_\tau, n \rangle$, where the first component represent the emission

value and can assume one of the states allowed for that agent type (i.e. $w_\tau \in \Sigma_\tau$) and $n \in \mathbb{N}$ indicates the field intensity. This component of field values allows the modulation of the emission value during the field propagation throughout the space according to its spatial structure. Field diffusion occurs according to the function that characterizes the field as well. Finally, field comparison and field composition functions are defined in order to allow field manipulation. Thus, a field $f_\tau \in F$ that can be emitted by agents of type τ is denoted by

$$< W_\tau, Diffusion_\tau, Compare_\tau, Compose_\tau >$$

where:

- $W_\tau = \Sigma_\tau \times \mathbb{N}$ denotes the set of values that the field can assume;
- $Diffusion_\tau : P \times W_\tau \times P \rightarrow (W_\tau)^+$ is the diffusion function of the field computing the value of a field on a given site taking into account in which site and with which value it has been emitted. Since the structure of a *Space* is generally not regular and paths of different lengths can connect each pair of sites, $Diffusion_\tau$ returns a number of values depending on the number of paths connecting the source site with each other site. Hence, each site can receive different values of the same field along different paths.
- $Compare_\tau : W_\tau \times W_\tau \rightarrow \{True, False\}$ is the function that compares field values. For instance, in order to verify whether an agent can perceive a field value.
- $Compose_\tau : (W_\tau)^+ \rightarrow W_\tau$ expresses how field values have to be combined (for instance, in order to obtain the unique value of the field at a site).

Moreover, Situated Cellular Agents are influenced by agents situated on adjacent positions. Adjacent agents, according to their type and state, synchronously change their states undertaking a two-steps process (named *reaction*). First of all, the execution of a specific protocol allows to synchronization of the set of adjacent computationally autonomous agents. When an agent wants to react with the set of its adjacent agents since their types satisfy some required condition, it starts an *agreement* process whose output is the subset of its adjacent agents that have agreed to react. An agent agreement occurs when the agent is not involved in other actions or reactions and when its state is such that this specific reaction could take place. The agreement process is followed by the synchronous reaction of the set of agents that have agreed to it. Let us consider an agent $a = < s, p, \tau >$, reaction can be specified as an agent action, according to MMAS notation [1], by:

action : $reaction(s, a_{p_1}, a_{p_2}, \dots, a_{p_n}, s')$
condit : $state(s), agreed(a_{p_1}, a_{p_2}, \dots, a_{p_n})$
effect : $state(s')$

where $state(s)$ and $agreed(a_{p_1}, a_{p_2}, \dots, a_{p_n})$ are verified when the agent state is s and agents situated in sites $\{p_1, p_2, \dots, p_n\} \subset P_p$ have previously agreed to undertake a synchronous reaction. The effect of a reaction is the synchronous change in state of the involved agents; in particular, agent a changes its state to s' .

3 Supporting the Application of Situated Cellular Agents

In order to facilitate and support the design, development and execution of the applications of Situated Cellular Agents, a dedicated platform is under development. The aim of this platform is to facilitate and support application developers in their activity avoiding them to manage aspects that characterize the SCA modelling approach and that are shared by all SCA applications. These aspects are, for instance, the diffusion of fields throughout the environment structure and agent synchronization to perform reaction. Thus, developers can exploit the tools provided by the platform and can better focus on aspects that are more directly related to their target applications.

In particular the platform will provide tools to describe system entities (i.e. sites, spaces, agents and fields) and tools to manage:

- agents' autonomous behavior based on the perception–deliberation–action mechanism;
- agents' awareness to the local and dynamic environment they are situated in (e.g. adjacent agents, free adjacent sites);
- field diffusion throughout the structured environment;
- conflicts potentially arising among a set of mobile agents that share an environment with limited resources;
- synchronization of a set of autonomous agents when they need to perform a reaction.

This work is part of an ongoing project. The platform architecture has been designed in a way that allows to incrementally integrate new designed and developed tools that provide new management functionalities. It can also be extended to include new tools providing the same functionalities according to other management strategies in order to better tackle the requirements of the target application. The platform has been designed according to the Object Oriented paradigm and developed in the Java programming language and platform. The currently developed tools allows to satisfy all the listed management functionalities according to one of the possible strategies. For instance, an algorithm has been designed and implemented in order to manage field diffusion over a generally irregular spatial structures [7]. An analysis has been performed to compare different possible solutions. However, we claim that there is not a generally optimal algorithm, but each SCA application presents specific features that must be taken into account in the choice (or design) of a strategy for field diffusion.

The proposed algorithm provides the generation of infrastructures to guide field diffusion and a specification of how sites should perform it, according to the diffusion function related to the specific field type. It was designed under the assumption of an irregular space (i.e. a non-directed, non-weighted graph), with a high agents–sites ratio and very frequent field emissions. Fields propagate instantly throughout the space, according to the modulation specified by the field diffusion function; in general fields could diffuse throughout all sites in the structured environment. Moreover, the model is meant to be general and thus makes no assumption on the synchronicity of the system. Under these assumptions we considered the possibility of storing a spatial structure representation for each

site, and namely a Minimum Spanning Tree (MST) connecting it to all other sites, since the use of these structures is frequent and the overhead for their construction for every diffusion operation would be relevant.

There are several algorithms for MST building, but previously explained design choices led to the analysis of approaches that could be easily adapted to work in a distributed and concurrent environment. The breadth first search (BSF) algorithm starts exploring the graph from a node that will be the root of the MST, and incrementally expands knowledge on the structure by visiting at phase k nodes distant k hops from the root. This process can be performed by nodes themselves (sites, in this case), that could offer a basic service of local graph inspection that could even be useful in case of dynamism in its structure. The root site could inspect its neighborhood and require adjacent sites to do the same, iterating this process with newly known sites until there is no more addition to the visited graph. An important side effect of this approach is that this MST preserves the distance between sites and the root: in other words the path from a site to the root has a number of hops equal to its distance from the root. Fields propagate through the edges of the MST and thus the computation of the diffusion function is facilitated.

The complexity of the MST construction using this approach is the order of $O(n + e)$ where n is the number of sites and e is the number of edges in the graph. Such an operation should be performed by every site, but with a suitable design of the underlying protocol they could proceed in parallel. Field diffusion requires at most $O(\log_b n)$, where b is the branching factor of the MST centered in the source site and the field propagation between adjacent sites is performed in constant time. The issue with this approach is the memory occupation of all those structures, that is $O(n^2)$ (in fact it is made up of n MSTs, each of those provides $n - 1$ arcs); moreover if the agents–sites ratio is not high or field emission is not very frequent to keep stored the MST for every site could be pointless, as many of those structures could remain unused.

4 Applications of Situated Cellular Agents

Situated Cellular Agents have been defined in order to provide a MAS based modelling approach that require spatial features to be taken into account and distributed approaches both from the modelling and the computational point of views. Two application domains of the Situated Cellular Agents will be briefly described in this section: the immune system modelling [8] and the guides placement in museum.

4.1 Immune System Modelling

The Immune System (IS) of vertebrates constitutes the defence mechanism of higher level organisms (fishes, reptiles, birds and mammals) to molecular and micro organismic invaders. It is made up of specific organs (e.g. thymus, spleen, lymph nodes) and of a very large number of cells of different kind that have or acquire distinct functions. The response of the IS to the introduction of a

foreign substance that might be harmful (i.e. antigen) involves thus a collective and coordinated response of many autonomous entities [9].

Other approaches to represent components and processes of the IS and simulate its behavior have been presented in the literature. A relevant and successful one is based on Cellular Automata (CA [10,11]). In this case, as according to our approach, the entities that constitutes the immune system and their behavior are described by specific rules defined by immunologists. In both approaches there is a clear correspondence between domain entities and model concepts, thus it is easy for the immunologist to interact with it using her language. A serious issue with approaches based on CA is that rules of interaction between cells and other IS entities must be globally defined. Therefore each entity that constitutes the CA-based model (i.e. cell) must be designed to handle all possible interactions between different types of entities. This problem is particularly serious as research in the area of immunology is very active and the understanding of the mechanisms of the IS is still far from complete. New researches and innovative results in the immunology area may require a complete new design of the IS model.

The goal of the application of the Situated Cellular Agents approach to IS modelling was to provide a modelling tool that is more flexible and at the same time allows a more detailed and complete representation of IS behavior. In fact, SCA allows to modify, extend and detail in an incremental way the representation of IS entities, their behaviors and interactions. Moreover, SCA allows a more detailed representation of the IS (e.g. more than just a probabilistic representation of interactions between entities is possible) and an expressive and natural way to describe all the fundamental mechanisms that characterize the IS (e.g. at-a-distance interaction through virus and antibody diffusion).

4.2 Guides Placement in Museums

The Situated Cellular Agents has also been proposed to support the decision making process about the choice of the best position for a set of museum guides into a building halls. This problem requires a dynamical and adaptive placement of guides: guides must be located in order to respond to all requests in a timely fashion and, thus, effectively serve visitors that require assistance and information.

A suitable solution to this problem must consider that guides and visitors dynamically change their position within the museum building and that visitor requests can vary according to their position and state. The SCA approach has been applied to this problem and it has allowed to effectively represent dynamic and adaptable behaviors that characterize guide and visitor agents, and to obtain the localization of objects as an emergent result of agents interactions. Moreover the Situated Cellular Agents approach has allowed to explicitly represent the environment structure where agents are situated and to provide agent behavior and interaction mechanisms that are dependant to this spatial structure. These aspects are of particular relevance in problems, like guide placement, in which the representation of spatial features is unavoidable.



Fig. 1. Some screenshots of the simulation performed to study guide placement in museums.

This problem has been implemented exploiting a system for three-dimensional representation of virtual worlds populated by virtual agents. Figure 1 shows some screenshots of simulation performed within the virtual representation of the Frankfurt Museum fur Kunsthandwerk¹ in which a guide placement problem has been studied.

5 Concluding Remarks and Future Works

In this paper Situated Cellular Agents have been presented. Situated Cellular Agents are systems of reactive agents whose behavior is influenced by adjacent as well as by at-a-distance situated agents. Situated Cellular Agents are heterogeneous (i.e. different abilities and perceptive capabilities can be associated to different agent types) and populates environment whose structure is generally not uniform. Moreover, the paper has briefly described two application examples that require suitable abstractions for the representation of spatial structures and relationships, and the representation of local interaction between autonomous agents (i.e. the immune system modelling and the guides placement

¹ The museum graphic model has been obtained adding color shaping, textures, and objects to a graphic model downloaded from Lava web site (lava.ds.arch.tue.nl/lava)

in museum). Finally, mechanisms to support the development and execution of applications of the proposed approach have been considered. The latter is the main topic of an ongoing project that aims developing a platform to facilitate and support developers in their activities for SCA applications. In particular a mechanism to support field diffusion throughout the non-uniform structure of the environment has been presented. This mechanism has already been implemented in the preliminary version of the platform and it can be exploited by developers of SCA applications.

The advantages in the Situated Cellular Agents approach and in particular the possibility to represent agent situated in environments with non-uniform structure have been evaluated and will be applied in the near future to the urban simulation domain. In particular, within a collaboration with the Austrian Research Center Seibersdorf (ARCS), a microeconomic simulation model is under design in order to model the fundamental socio-economic processes in residential and industrial development responsible for generating commuter traffic in urban regions. A second application in the same domain will concern a collaboration with the Department of Architectural Design of the Polytechnic of Turin. The main aim of this ongoing project is to design and develop a virtual laboratory for interactively designing and planning at urban and regional scales (i.e. UrbanLab [12]). Within this project the Situated Cellular Agents approach will be applied to urban and regional dynamics at the building scale. Currently, both projects are in the problem modelling phase and further investigations will be done in collaboration with domain experts in order to better define the details to apply the proposed approach to this domain.

References

1. Bandini, S., Manzoni, S., Simone, C.: Enhancing cellular spaces by multilayered multi agent situated systems. In Bandini, S., Chopard, B., Tomassini, M., eds.: Cellular Automata, Proceeding of 5th International Conference on Cellular Automata for Research and Industry (ACRI 2002), Geneva (Switzerland), October 9–11, 2002. Volume 2493 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (2002) 155–166
2. Ferber, J.: Multi-Agent Systems. Addison-Wesley, Harlow (UK) (1999)
3. Sichman, J.S., Conte, R., Gilbert, N., eds.: Multi-Agent Systems and Agent-Based Simulation, Proceedings of the 1st International Workshop (MABS-98), Paris, France, July 4–6 1998. Volume 1534 of Lecture Notes in Computer Science., Springer (1998)
4. Moss, S., Davidsson, P., eds.: Multi Agent Based Simulation, 2nd International Workshop, MABS 2000, Boston, MA, USA, July, 2000, Revised and Additional Papers. Volume 1979 of Lecture Notes in Computer Science. Springer (2001)
5. Sichman, J.S., Bousquet, F., Davidsson, P., eds.: Multi Agent Based Simulation, 3rd International Workshop, MABS 2002, Bologna, Italy, July, 2002, Revised Papers. Lecture Notes in Computer Science. Springer (2002)
6. Bandini, S., Manzoni, S., Pavesi, G., Simone, C.: L*MASS: A language for situated multi-agent systems. In Esposito, F., ed.: AI*IA 2001: Advances in Artificial Intelligence, Proceedings of the 7th Congress of the Italian Association for Artificial Intelligence, Bari, Italy, September 25–28, 2001. Volume 2175 of Lecture Notes in Artificial Intelligence., Berlin, Springer-Verlag (2001) 249–254

7. Bandini, S., Mauri, G., Vizzari, G.: Supporting action-at-a-distance in situated cellular agents. Submitted to *Fundamenta Informaticae* (2003)
8. Bandini, S., Manzoni, S., Vizzari, G.: (Situated cellular agents and immune system modelling) Submitted to WOA 2003 – Dagli oggetti agli agenti, 10–11 Sep. 2003, Villasimius (CA), Italy.
9. Kleinstein, S.H., Seiden, P.E.: Simulating the immune system. *IEEE Computing in Science and Engineering* **2** (2000)
10. Celada, F., Seiden, P.: A computer model of cellular interactions in the immune system. *immunology Today* **13** (1992) 56–62
11. Bandini, S.: Hyper-cellular automata for the simulation of complex biological systems: a model for the immune system. Special Issue on Advance in Mathematical Modeling of Biological Processes **3** (1996)
12. Caneparo, L., Robiglio, M.: Urbanlab: Agent-based simulation of urban and regional dynamics. In: *Digital Design: Research and Practice*, Kluwer Academic Publisher (2003)

Accuracy and Stability of Spatial Dynamics Simulation by Cellular Automata Evolution

Olga Bandman

Supercomputer Software Department
ICMMG, Siberian Branch
Russian Academy of Science
Pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
`bandman@ssd.sssc.ru`

Abstract. Accuracy and stability properties of fine-grained parallel computations, based on modeling spatial dynamics by cellular automata (CA) evolution, are studied. The problem arises when phenomena under simulation are represented as a composition of a CA and a function given in real numbers, and the whole computation process is transferred into a Boolean domain. To approach the problem accuracy of real spatial functions approximation by Boolean arrays, as well as of some operations on cellular arrays with different data types are determined and approximation errors are assessed. Some methods of providing admissible accuracy are proposed. Stability is shown to depend only of the nonlinear terms in hybrid methods, the use of CA-diffusion instead of Laplace operator having no effect on it. Some experimental results supporting the theoretical conclusions are presented.

1 Introduction

Fine-grained parallelism is the concept which attracts a great interest due to its compatibility both with the growing demands of natural phenomena simulation tools, and of the modern tendency towards multiprocessor architecture development. Among a scope of fine-grained parallel models for spatial dynamics simulation the discrete ones are the most extensively studied. Almost all of them descend from the classical cellular automaton (CA) [1], and are either its modification or its extension. Some of them are well studied and proved to be an alternative to the corresponding continuous models. Such are CA-diffusion models [2,3] and Gas-Lattice [4,5] models. There are also such ones which have no continuous alternatives [6]. The attractiveness of CA-models is founded upon their natural parallelism admitting any kind of parallel realization, simplicity of programming, as well as the computation stability and absence of round off errors. Nevertheless, up to now there is not so many good CA-models of spatial dynamics. The reason is in that there is no systematic methods to construct automata transition rules from any kind of spatial dynamic description. This fact has favored the appearance of a hybrid approach which combines CA-evolution with computations in reals [7]. This approach may be used in all those cases,

when the phenomenon under simulation comprises a component for which CA model is known. The bright manifestation of hybrid approach applicability is the wide range of reaction-diffusion processes [8]. Due to its novelty, the hybrid approach is not yet well studied. Particularly, the computation parameters, such as accuracy and stability, have not yet been investigated. Although they are main computation parameters, which may be compared with the similar ones characterizing PDE solution. Such comparison seems to be the most practical way to assess computation properties of CA models in physics. The comparison is further performed with the explicit numerical methods for concentrating the whole study in the domain of fine-grained parallelism.

Since CAs are accurate by their nature, the study of this property is focused on CA interaction with real functions. So, the accuracy assessment is concerned with two types of errors: the approximation errors when transferring from real spatial function to the equivalent Boolean array, and the deflections from the true values when performing the inverse transfer.

As distinct from the accuracy, CA are not always stable. From the point of view of stability CAs are divided into four classes in [9]. CAs from the first class have trivial attractors (all cell states are equal to 1, or all are equal to 0). CAs from the second class have attractors in the form of stable patterns. The third and the forth class comprise CAs having no stable attractors (or have so called "strange attractors"), exhibiting *complex behavior*, the notion meaning that there is no other way to describe global states than to indicate the state of each cell. Excluding from the consideration chaotic phenomena described by 3d and 4th classes, the attention is further focused to those CAs, whose evolution tends to a stable state, i.e. to the CAs from two first classes. The most known CAs of this type are CA-diffusion [2], Gas-lattice models [4,5], percolation [10], phase transition, pattern formation [9]. Such CAs by themselves are absolutely stable, and no care is required to provide stability of their evolution, though the instability may be caused by nonlinear functions in hybrid methods.

Apart from Introduction and Conclusion the paper contains three sections. The second presents a brief presentation of CA and hybrid models. The third is destined to the accuracy problem. In the fourth the stability is considered.

2 Cellular Automata in Spatial Dynamics Simulation

2.1 Representation of Cellular Arrays

Simulating spatial dynamics is computing a function $u(\mathbf{x}, t)$, where u is a scalar, representing a certain physical value, which may be pressure, density, velocity, concentration, temperature, etc. A vector $\mathbf{x}$ represents a point in a continuous space, t stands for time. In the case of D -dimensional Cartesian space the vector components are spatial coordinates. For example, in 2D case $\mathbf{x}=(x_1, x_2)$.

When numerical methods of PDE solution are used for simulation spatial dynamics, space is converted into a discrete grid, which is further referred to as a *cellular space* according to cellular automata terminology. For the same reason the function $u(\mathbf{x})$ is represented in the form of a *cellular array*.

$$U(\mathbf{R}, M) = \{(u, m) : u \in \mathbf{R}, m \in M\} \quad (1)$$

which is the set of cells, each cell being a pair (u, m) , where u is a state variable with the domain usually taken as the real interval $(0, 1)$, $m \in M$ is the name of a cell in a discrete cellular space M , which is called a *naming set*. To indicate the state value of a cell named m a notation $u(m)$ is used. In practice, the names are given by coordinates of the cells in the cellular space. For example, in case of the cellular space represented by a 2D Cartesian lattice, the set of names is $M = \{(i, j) : i, j = 0, 1, 2, \dots\}$, where $i = x_1/h_1, j = x_2/h_2$, h_1 and h_2 being space discretization steps. For simplicity we take $h_1 = h_2 = h$. In theory, it is more convenient to deal with a generalized notion of the naming set considering $m \in M$ as a discrete spatial variable. A cell named m is called *empty*, if its state is zero. A cellular array with all cells being empty is called an *empty array* further denoted as $\Omega = \{(0, m) : \forall m \in M\}$.

When CA models are to be used for spatial dynamics simulation, the discretization should be performed not only on time and space, but also on the function values transforming a "real" cellular array into a Boolean one

$$V(B, M) = \{(v, m) : v \in B, m \in M\}, \quad B = \{0, 1\}. \quad (2)$$

In order to define this type of discretization, some additional notions should be introduced. A set of cells

$$Av(m) = \{(v, \phi_k(m)) : v \in B, k = 0, 1, \dots, q\} \quad (3)$$

is called the *averaging area* of a cell named m , $q = |Av(m)|$ being its size. Functions $\phi_k(m)$, $k = 0, \dots, q$ are referred to as *naming functions*, indicating the names of cells in the averaging area, and forming an *averaging template*

$$T(m) = \{\phi_k(m) : k = 0, 1, \dots, q\}. \quad (4)$$

In the naming set $M = \{(i, j)\}$ the naming functions are usually given in the form of shifts, $\phi_k(i, j) = (i + a, j + b)$, a, b being integers not exceeding a fixed r , called a *radius* of averaging.

The *averaged state* of a cell is

$$z(m) = \frac{1}{q} \sum_{k=0}^q v(\phi_k(m)). \quad (5)$$

Computing averaged states for all $m \in M$ according to (5) yields a cellular array $Av(V) = Z(Q, M)$ called the *averaged form* of $V(B, M)$. From (5) it follows, that $Q = \{0, 1/q, 2/q, \dots, 1\}$ is a finite set of real numbers forming a discrete alphabet.

It follows herefrom, that a Boolean array represents a spatial function through the distribution of "ones" over the discrete space. Averaging is the procedure of computing the density of this distribution, which transfers a Boolean array into a cellular array with real state values from a discrete alphabet. The

inverse procedure of obtaining a Boolean array representation of a given cellular array with real state values is more important and more complicated.

A Boolean array $V(B, M)$ such that its averaged form $Z(Q, M) = Av(V)$ approximates a given cellular array $U(\mathbf{R}, M)$ is called its *Boolean discretization* $Disc(U)$. Obtaining $Disc(U)$ is based on the fact, that for any $m \in M$ the probability of the event that $v(m) = 1$ is equal to $u(m)$, i.e.

$$P_{v(m)=1} = u(m). \quad (6)$$

2.2 Computation on Cellular Arrays

As it was already said, CA-models are used for spatial-dynamics simulation in two ways. The first is possible when there exists a "pure" (classical) CA, which is the model of the phenomenon under simulation. In this case Boolean discretization and averaging are performed once at the start and at the end of the simulation, respectively, which causes no accuracy and stability problems. The second way is possible when there exist CA-models of phenomena which are components of that to be simulated, the other components being given in the real domain. In this case the *hybrid* approach is used, which transfers the whole computation process into the Boolean domain by means of approximate operations on cellular arrays at each iterative step, generating approximation errors, and, hence, the need to take care for providing accuracy and stability. A bright manifestation of a hybrid method application are reaction-diffusion processes, where the diffusion part is modeled by a CA, and the reaction is represented as a nonlinear function in the real domain. The details of the hybrid method for this type of processes are given in [7]. In general case, spatial dynamics is represented as a composition of cellular array transformations, which may have different state domains. Specifically, two types of operations on cellular arrays are to be defined: transformations and compositions.

Transformations of Boolean arrays are as follows.

- 1) Application of a CA-transition rules $\Phi(V)$, resulting in a Boolean array.
- 2) Computation of a function $F(Av(V))$ whose argument is in real arrays domain, but the result $Disc(F(Av(V)))$ should be a Boolean array.
- 3) All kind of *superpositions* of the above transformation are allowed, the mostly used are the following:

- $\Phi(Disc(U))$ – application of CA-rules to a Boolean discretization of U ,
- $Disc(F(Av(\Phi(V))))$ – discretization of a real array obtained by averaging a CA-rules application result.

Composition operations are addition (subtraction) and multiplication. They are determined in the domain of the set of cellular arrays, belonging to one and the same group $K(M, T)$, characterized by a naming set M , and an averaging template $T = \{(\phi_k(m)) : k = 0, \dots, q\}$.

- 1) Boolean cellular arrays addition (subtraction). A Boolean array $V(B, M)$ is called a sum of two Boolean arrays $V_1(B, M)$ and $V_2(B, M)$,

$$V(B, M) = V_1(B, M) \oplus V_2(B, M), \quad (7)$$

if its averaged form $Z(Q, M) = Av(V)$ is a matrix-like sum of $Z_1(Q, M) = Av(V_1)$ and $Z_2(Q, M) = Av(V_2)$. This means, that for any $m \in M$: $z(m) = z_1(m) + z_2(m)$, where $(z, m), z_1(m), z_2(m)$, are cell states in $Z(Q, M), Z_1(Q, M), Z_2(Q, M)$ respectively. Using (5) and (6) the resulting array may be obtained by allocating the "ones" in the cells of an empty array with the probability

$$P_{0 \rightarrow 1} = \frac{1}{q} \left(\sum_{k=0}^q v_1(\phi_k(m)) + \sum_{k=0}^q v_2(\phi_k(m)) \right). \quad (8)$$

When Boolean array addition is used as an intermediate operation it is more convenient to obtain the resulting array by means of updating one of the operands so, that it equals the resulting Boolean array. It may be done as follows. Let $V_1(B, M)$ should be changed into $V_1(B, M) \oplus V_2(B, M)$. Then some cells $(v_1, m) \in V_1(B, M)$ with $v_1(m) = 0$ have to invert their states. The probability of such an inversion is the relation of the value to be added to the amount of "zeros" in the averaging area $Av(m) \in V_1(B, M)$, i.e.

$$P_{0 \rightarrow 1} = \frac{z_2(m)}{(1 - z_1(m))}. \quad (9)$$

Subtraction also may be performed in two ways. The first is similar to (8), the resulting difference

$$V(B, M) = V_1(B, M) \ominus V_2(B, M)$$

being obtained by allocating the "ones" in the cells of an empty array with the probability

$$P_{0 \rightarrow 1} = z_1(m) - z_2(m);. \quad (10)$$

The second is similar to (9) taking allowance for the inversion be done in the cells with states $v_1(m) = 1$, the probability of the inversion being the relation of the amount of "ones" to be subtracted to the total amount of "ones" in the averaging area, i.e.

$$P_{1 \rightarrow 0} = \frac{z_2}{z_1}. \quad (11)$$

2) Boolean and real cellular arrays addition (subtraction), which is also referred to as a *hybrid* operation, differs from the given above only in that one of the operand is initially given in normalized real form.

3) Multiplication of two Boolean arrays. A Boolean array $V(B, M)$ is called a product of $V_1(B, M)$ and $V_2(B, M)$, which is written as $V(B, M) = V_1(B, M) \otimes V_2(B, M)$, if its averaged form $Z(Q, M) = Av(V)$ has cell states, which are products of corresponding cell states from $Z_1(Q, M) = Av(V_1)$ and $Z_2(Q, M) = Av(V_2)$. It means, that for all $m \in M$

$$\frac{1}{q} \sum_{k=0}^q v(\phi_k(m)) = \frac{1}{q} \sum_{k=0}^q v_1(\phi_k(m)) \times \frac{1}{q} \sum_{k=0}^q v_2(\phi_k(m)) \quad (12)$$

The resulting array may be obtained by allocating the "ones" in the cells of an empty array with the probability

$$P_{0 \rightarrow 1} = \frac{1}{q} \sum_{k=0}^q v_1(\phi_k(m)) \times \frac{1}{q} \sum_{k=0}^q v_2(\phi_k(m)). \quad (13)$$

4) Multiplication of a Boolean array by a real cellular array (hybrid multiplication). A Boolean array $V(B, M)$ is a product of a Boolean array $V_1(B, M)$ and $Z_2(Q, M)$, which is written as $V(B, M) = V_1(B, M) \otimes Z_2(Q, M)$, if its averaged form $Z(Q, M) = Av(V)$ has cell states, which are products of corresponding cell states from $Z_1(Q, M) = Av(V_1)$ and $Z_2(Q, M)$. The resulting array is obtained by allocating the "ones" in the cells of an empty array with the probability

$$P_{0 \rightarrow 1} = \frac{z_2(m)}{q} \sum_{k=0}^q v_1(\phi_k(m)), \quad (14)$$

Clearly, multiplication of a Boolean array $V_1(B, M)$ by a constant $a \in Q$, $Q = \{0, 1/q, \dots, 1\}$ is the same that multiplication $V(B, M)$ by $Z_2(a, M)$ with all cells having equal states $z_2(m) = a$.

2.3 Construction of a Composed Cellular Automaton

Usually, natural phenomena to be simulated are represented as a composition of a number of simple well studied processes, which are further referred to as *component processes*. Among those the most known are diffusion, convection, phase separation, pattern formation, reaction functions, etc., which may have quite different form of representation. For example, reaction functions can be given by a continuous real nonlinear functions, the phase separation process – by a CA, and pattern formation process – by a semi-discrete cellular-neural network [11]. Obviously, if the process under simulation is the sum of components with different representation types, then the usual real summation of cell states does not work. Hence, we are forced to use cellular array composition operations.

The procedure of constructing a composed phenomenon simulation algorithm is as follows. Let the initial state of the process under simulation be a cellular array given as functions of time in two forms: $V(0), Y(0) = Av(V(0))$. Without loss of generality let's assume the phenomenon be a reaction-diffusion process which is composed of two components: the diffusion represented by a CA with transition rules $\Phi(V) = \{(\Phi(v), m) : m \in M\}$, and the reaction represented by a nonlinear function $F(Y) = \{(F(y), m) : M \in M\}$. A CA of the composition

$$\Psi(V) = \Phi(V) \oplus F(Y)$$

should have the transition function, such that the CA-evolution $V^* = \{V(0), V(1), \dots, V(t), V(t+1), \dots, V(T)\}$ simulates the composed process.

Let the t -th iteration result be a pair of cellular array $V(t)$ and $Y(t)$. Then the transition to their next states comprises the following steps.

1. Computation of $\Phi(V(t))$ by applying $\Phi(v, m)$ to all cells $(v, m) \in V(t)$.
2. Computation of $F(V(t))$ by calculating $F(y)$ for all cells $(y, m) \in Y(t)$.
3. Computation of the result of cellular array addition $V(t+1) = \Phi(V(t)) \oplus F(Y(t))$ by applying (9) or (11) (depending on the sign of $F(y, m)$) to all cells $(v, m) \in V(t)$.
4. Computation of $Y(t+1) = Av(V(t+1))$ by applying (5) to all cells of $V(t+1)$.

It's worth noting that computations in pp.1 and 2 may be done in parallel, being adjunct to the fine-grained parallelism by cells in the whole procedure.

The following example illustrates the use of above procedure when simulating composed spatial dynamics.

Example 1. There is a well known CA [6], simulating phase separation in a 2D space. It works as follows. Each cell changes its states according to the following rule $\Phi_1(V)$:

$$v(t+1) = \begin{cases} 0, & \text{if } S < 4 \text{ or } v = 5, \\ 1, & \text{if } S > 5 \text{ or } v = 4. \end{cases} \quad (15)$$

where $S = \sum_{k=0}^8 v_k$, v_k being the state of the k -th ($k = 0, 1, \dots, 9$) neighbor (including the cell itself) of the cell $(v, m) \in V$.

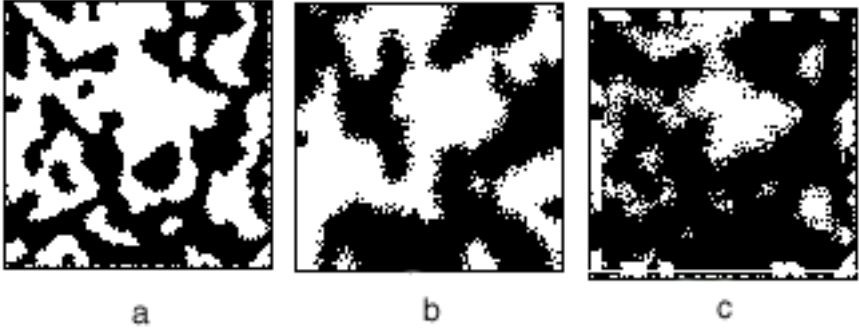


Fig. 1. Simulation of three separation phase processes. The snapshots at $T=20$ are shown, the initial cellular array having randomly distributed "ones" with the density $d=0.5$: a) a process, given by a CA (15), b) a process composed of two CAs: CA (15) and a CA-diffusion, c) a process composed of three components: CA (15), the CA-diffusion and a nonlinear reaction $F(u)=0.5u(1-u)$.

This CA separates "zeros" (white cells) from "ones" (black cells) forming a stable pattern. In Fig.1a the Boolean array $V_1(T)$ at $T = 20$ obtained according to (15) is shown, the evolution having started at $V_1(0)$ being a random distribution of "ones" with the density 0.5. If in combination with the separation process a diffusion $\Phi_2(V)$ also takes place, cellular arrays addition $V_1(t) \oplus V_2(t)$ should be done according to (9) on each iterative step. So, the composed process

is $\Phi(V) = \Phi_1(V) \oplus \Phi_2(V)$. In the experiment in Fig.1b CA-diffusion $\Phi_2(V)$ with Margolus neighborhood (in [3] this model is called a Block-Rotation diffusion) is used. Fig.1c shows the snapshot ($T = 20$) of the process $\Psi(V) = \Phi(V) \oplus F(Y)$, obtained by one more cellular addition of a chemical reaction, given by a non-linear function $F(u) = 0.5u(1 - u)$.

Since our main objective is to analyze accuracy and stability, a notice about these properties in the above example is appropriate. Clearly, in case of phase separation according to (15) no problems arise both in accuracy and in stability, due to the absence of approximation procedures. In the second and third cases cellular addition with averaging procedure (9) contributes accuracy errors. As for stability there is no problems at all, because the both CAs and the $F(u)$ are intrinsically stable.

3 Accuracy of Cellular Computations

3.1 Boolean Discretization Accuracy

The transitions between real and discrete representations of cellular arrays, which take place on each iteration in composed processes simulation, incorporate approximation errors. The first type of approximation is replacing the continuous alphabet $(0, 1)$ by a discrete one $Q = \{0, 1/q, \dots, 1\}$, the error being

$$e_1 \leq 1/q, \quad (16)$$

The second type of errors are those brought up by Boolean discretization of a real array with subsequent averaging. Let $V(B, M) = \text{Disc}(Y)$ be obtained according to the probabilistic rule (6), its averaged form being $Z(Q, M)$. Then expected value $\mu(y(m))$ for any $m \in M$ is equal to the mean state value $y'(m)$ of $y(m)$ over the averaging area $Av(m)$, which in its turn is equal to $z(m)$, i.e.

$$\mu(y(m)) = \frac{1}{q} \sum_{k=0}^q v(\phi_k(m)) P_{v(\phi_k(m)=1)} = \frac{1}{q} \sum_{k=0}^q y(\phi_k(m)) = y'(m) = z(m). \quad (17)$$

From (17) it follows, that the discretization error vanishes in those cells where

$$y(m) = y'(m) = \frac{1}{q} \sum_{k=0}^q y(\phi_k(m)). \quad (18)$$

The set of such cases includes, for example, all linear functions and parabolas of odd degree, considered on the averaging area relative to a coordinate system with the origin in the cell named m . When (18) is not satisfied, the error of Boolean discretization

$$e_2(m) = z(m) - y(m) \neq 0 \quad (19)$$

is the largest at the cells where $y(m)$ has extremes.

Generalized accuracy parameters which is intended further to be used in experimental practice is the *mean discretization error*

$$E = \frac{1}{M} \sum_{m \in M} \frac{|y(m) - z(m)|}{y(m)}. \quad (20)$$

which should satisfy the accuracy requirements

$$E < \epsilon, \quad (21)$$

ϵ being the admissible approximation error.

3.2 Methods for Providing Accuracy

From (16) and (19) it follows, that discretization errors depend on the cardinality $q = |Av(m)|$ and on the behavior of $y(m)$ on $Av(m)$. Both these parameters are conditioned by the spatial discretization step h , which should be taken small, allowing q to be chosen large enough to smooth function extremes. It may be done by two ways: 1) to divide the physical space S into small cells of size $h = S/|M|$, i.e. taking a naming set of large cardinality, and 2) to increase the dimension of the Boolean space making it a multilayer one. Since no analytical method exists for evaluation the accuracy, the only way to get the insight to the problem is to perform the computer experiments.

Let's begin with the second method by constructing a Boolean discretization $V(B, M \times L) = Disc(Y)$ with a naming set having a L -layered structure of the form

$$M \times L = \bigcup_{l=1}^L M_l, \quad M_l = \{m_1^{(l)}, \dots, m_N^{(l)}\}.$$

The cell state values $v(m_i^{(l)})$ of $V(B, M \times L)$ are obtained in all layers in the one and the same way according to the rule (6). Averaging of $V(B, M \times L)$ is done over the multilayer averaging area with a size $q \times L$. The result of averaging $Z(Q, M)$ is again an one-layer array, where cell states are as follows.

$$z(m) = \frac{1}{q \times L} \sum_{l=1}^L \sum_{k=0}^q v(\phi_k(m^{(l)})) \quad \forall m_i^{(l)} \in M. \quad (22)$$

Example 2. Boolean discretization of a one-dimensional half-wave $u = \sin x$ with $0 < x < \pi$ is chosen for performing an experimental assessment of Boolean discretization accuracy. The objective is to obtain the mean error dependence of the number of layers. The experiment has been stated as follows. The cellular array representation $Y(Q, M)$ of the given continuous function is found as follows.

$$y(m) = \sin\left(\frac{\pi}{|M|}m\right), \quad m = 0, 1, \dots, |M|, \quad (23)$$

For the real cellular array $Y(Q, M)$ a number of Boolean discretizations $\{V_l(B, M \times l) : l = 1, \dots, 20, |M| = 360\}$ with $|Av_l| = q \times l$, have been obtained by applying (6) to all layers cells, and $E(l)$ have been computed for all $l = 1, 2, \dots, 20$.

The dependence $E(l)$ (Fig.4) shows that the mean error decrease is essential for a few numbers of layers, remaining then unchanged. Moreover, similar experiment on a 2D function $u(x, y) = \sin(\sqrt{x^2 + y^2})$ showed no significant decrease of mean errors, the price for it being fairly high, since $q = (2r + 1)^2$, where r is the radius of the averaging area. The most efficient method for providing accuracy

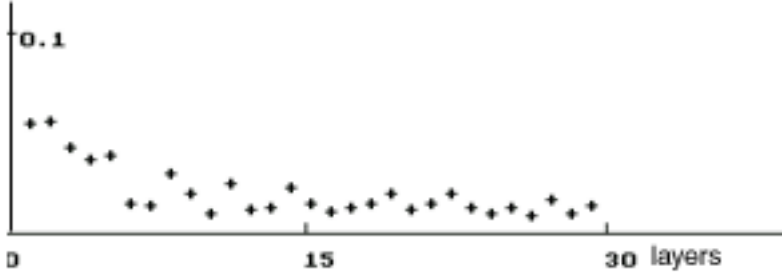


Fig. 2. Mean discretization error dependence of the number of layers in the Boolean cellular array for the function (23), the spatial step $h = 0.5^\circ$.

is the one, mentioned as first in the beginning of the section, which is to take large naming set cardinality in each layer (if there are many).

Example 3. Considering $u(x) = \sin x$ ($0 < x < \pi$) be a representative example for a wide range of nonlinear phenomena, this function is chosen again for experimental assessment of Boolean discretization error accuracy via $|M|$ and $|Av|$. For obtaining the dependence $E(|M|)$, a number of Boolean discretizations of (23) $\{V_k(B, M_k) : k = 1, \dots, 30\}$ have been constructed with such that $|M_k| = c \times k$, c being a constant, $c = 60$, the argument domain being $60 < |M_k| < 1800$ which corresponds to $2^\circ > h > 0.1^\circ$. For each $\{V_k(B, M_k)$ its averaged form $Z_k(Q, M_k) = Av_k(V_k)$ has been constructed with $|Av_k| = 0.2|M_k|$, and the mean errors E_k have been computed according to (20). The dependence $E(|M|)$ (Fig.2) shows that the mean error value follows the decrease of a spatial step and does not exceed 1% with $h < 0.5^\circ$.

To obtain the discretization error dependence of the averaging area, a number of Boolean discretizations of (23) $\{V_j(M \times L) : j = 1, \dots, 30\}$ ($L = 20$) have been obtained with fixed $|M| = 360$ but different $|Av_j| = 5 \times j \times L$. The dependence $E(q)$ (Fig.3) shows, that the best averaging area is about 36° .

Remark Of course, it is allowed to use cellular arrays with different spatial steps and different averaging areas over the cellular space, as well as to change them dynamically during the simulation process.

When a spatial function has sharp extremes or breaks, discretization error elimination may be achieved by using *extreme compensation method*. The cells, where the function has the above peculiarities, are further referred to as *ex-*

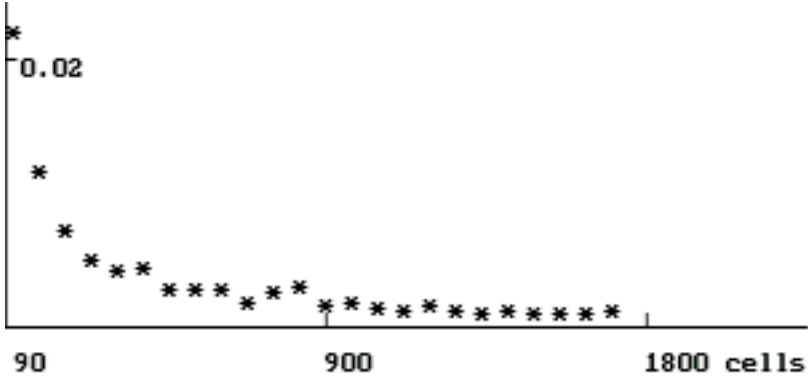


Fig. 3. Mean discretization error dependence of the naming set cardinality $|M|$ with $|Av| = 0.2|M|$ for the function (23)

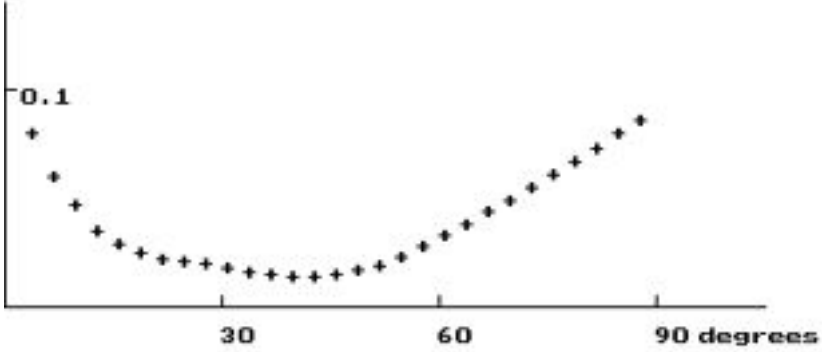


Fig. 4. Mean discretization error dependence of the averaging area for the function (23), the spatial step $h = 0.5^\circ$.

treme cells their names being denoted as m^* (Fig.5). The method provides for replacing the initial cellular array $Y(Q, M)$ by a "virtual" one $Y^*(Q, M)$, which is obtained by substituting the subarrays $Av(m^*)$ in $Y(Q, M)$ for the "virtual" ones $Av^*(m^*)$.

For determining new states $y^*(\phi_k(m^*))$ in the cells of $Av^*(m^*)$, error correcting values $\tilde{y}(\phi_k(m^*))$

$$\tilde{y}(\phi_k(m^*)) = 2y(m^*) - y(\phi_k(m^*)) \quad (24)$$

with $\phi_0(m) = \phi_0(m^*) = m^*$, which compensate averaging errors are found, and cell states in virtual averaging areas are computed as follows.

$$y^*(\phi_k(m^*)) = \frac{1}{2} \left(y(\phi_k(m^*)) + \tilde{y}(\phi_k(m^*)) \right) = y(m^*). \quad (25)$$

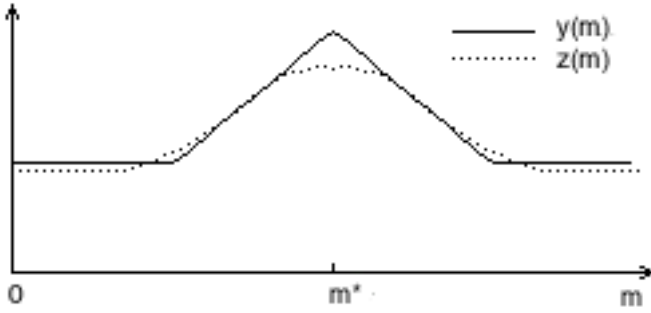


Fig. 5. A spatial function $y(m)$ with sharp extremes and its averaged Boolean discretization $z(m)$

From (25) it is easily seen, that when the function under Boolean discretization is piece-wise linear, all cell states in $Av^*(m^*)$ are equal to $y(m^*)$, i.e.

$$Av^*(m^*) = \{(y(m^*), \phi_k(m^*)) : k = 0, \dots, q\}, \quad (26)$$

So, in many cases it makes sense to obtain the piece-wise linear approximation of $Y(Q, M)$, and then perform Boolean discretization with the use of the extreme compensation method. Of course, the spatial discretization step should be chosen in such a way that the distance between two nearest extremes be larger than $2r$, r being the radius of the averaging area. The efficiency of the method is illustrated in Fig.6 by the results of Boolean discretization of a piece-wise linear function $Y(m)$, shown in Fig.5. The averaged Boolean discretization $Z(Q, M) = Av(Disc(Y^*))$ coincides with the given initial one.

3.3 Stability of Cellular Computations

When a CA used to simulate spatial dynamics is intrinsically stable, there is no need to take care for providing stability of computation. It is a good property of CA-models, which nevertheless cannot be assessed quantitatively for those cases, where no other models exist (for example, snow-flakes formation, percolation, crystallization). The comparison may be made for those CA-models, which have the counterparts as PDEs, where stability requirements impose an essential constraint on the time step. The latter should be small enough to satisfy the Courant's constraint, which is $c < 1/2$, $c < 1/4$ and $c < 1/6$ for 1D, 2D and 3D cases, respectively. The parameter $c = \tau d/h^2$ (τ - the time step, d -diffusion coefficient, h - spatial step), is a coefficient with Laplace operator. Sometimes, for example, when Poisson's equation is solved, this constraint is essential. Meanwhile, CA-model simulates the same process with $c = 1$ for 1D case, $c=1,5$ for 2D case, and $c = 23/18$ for 3D case [3], these parameters being inherent to the model, having no relation to the stability. So, in the 2D case the convergence rate of the computation is 6 times larger when CA-model is used, if there is no

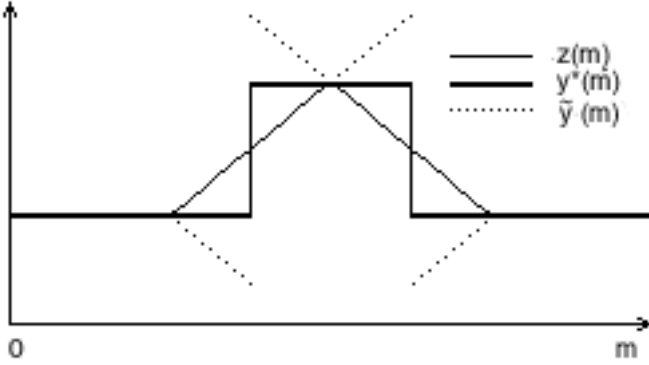


Fig. 6. Virtual cellular array $Y^*(Q, M)$ (thick lines) construction, the initial array being $Y(Q, M)$ (thin lines) from Fig 5. The compensating values are shown in dots. $Z(Q, M) = Av(Disc(Y^*))T$ coincides with $Y(Q, M)$

other restricting conditions. The comparative experiments of CA-diffusion are given in detail in [2]. Though they are rather roughly performed, the difference in iterative steps numbers is evident. Unfortunately, there is no such investigation comparing Gas-Lattice fluid flow simulation with Navier-Stokes equations solution, which would allow to make similar conclusions.

When CA-diffusion is used in reaction-diffusion simulation, that is the reaction part of the process which may cause instability, any known method being allowed for this. The following example shows how the use of CA-diffusion in reaction-diffusion simulation improves the computational stability.

Example 4. 1D Burger's equation solution. Burger's equation describes a wave propagation with a growing front steepness. The right hand side of the equation has two parts: a Laplace operator and a nonlinear shifting.

$$u_t = \lambda u u_x + \nu u_{xx}, \quad (27)$$

where subscripts mean the derivatives, λ and ν are constants. After time and space discretization it looks like this.

$$u_i(t+1) = u_i(t) + \frac{\tau\lambda u_i(t)}{2h}(u_{i-1}(t) - u_{i+1}(t)) + \frac{\tau\nu}{h^2}(u_{i-1}(t) + u_{i+1}(t) - 2u_i(t)), \quad (28)$$

where $i = x/h$, $i \in M$ is a point in a discrete space or a cell name in CA notation, h and τ being space and time discretization steps. Taking a for $\tau\lambda/2$ and b for $\tau\nu/h^2$ and $V(B, M)$ as a Boolean discretization of $U(i)$, (27) is represented as a cellular form.

$$V(t+1) = a\Phi(V(t)) \oplus bF(Z(t)), \quad (29)$$

where $\Phi(V(t))$ is a result of one iteration of CA-diffusion applied to $V(t)$, $F(Z(t))$ is the cellular array with states

$$f_i(z) = z_i(z_{i-1} + z_{i+1}), \quad z_i = \frac{1}{q} \sum_{k=0}^q v_k(\phi_k(i)). \quad (30)$$

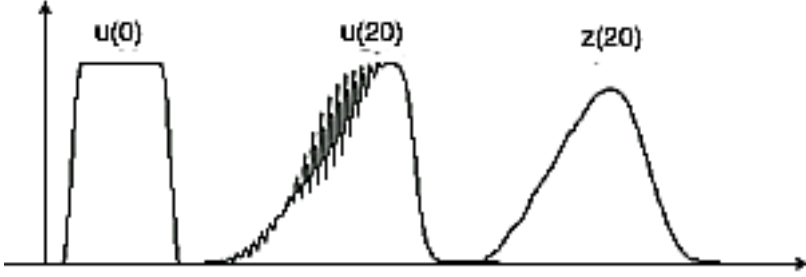


Fig. 7. 1D Burgers equation solution: the initial cellular state $u(i)$ at $t = 0$, a snapshot of numerical PDE solution $u(20)$ at $t = 20$ and a snapshot of hybrid solution at $t = 20$;

The equation (25) was solved with $a = 0.05, b = 0.505, i = 0, \dots, 200$ by using two methods: a numerical iterative method with explicit discretization according to (26), and a hybrid method with a 1D CA-diffusion algorithm [2] with probabilistic updating according to (9). The initial state is a flash of high concentration between $15 < i < 45$ ($u(0)$ in Fig.7). Border conditions are of Neumann type: $z_i = z_r$ for $i = 0, \dots, r$, and $z_i = N - r - 1$ for $i = N - r - 1, \dots, N - 1$. In Fig.7 a snapshot at $t = 20$ ($u(20)$) obtained by a numerical method (26) is shown. The unstable behavior, generated by diffusion instability ($b > 0.5$) is clearly seen. The snapshot obtained on the same time with the same parameters but by using hybrid method has no signs of instability. Moreover, the hybrid evolution remains absolutely stable up to $t = 100$, when the instability of the nonlinear function $F(z)$ starts to be seen.

4 Conclusion

From the above results of accuracy and stability investigation it follows, that the use of CA models in spatial dynamics simulation improves the computational properties, relative to the explicit methods of PDE solution. Of course, these results are preliminary ones. The complete assessment may be made on the base of a great experience on simulation of large-scale phenomena using multiprocessor computers.

References

1. von Neumann, J.: Theory of self reproducing automata. Uni. of Illinois, Urbana (1966)
2. Bandman O.: Comparative Study of Cellular-Automata Diffusion Models. In: Malyshkin V.(ed.):Lecture Notes in Computer Science, 1662. Springer-Verlag, Berlin (1999), 395–409.
3. Malinetski G.G., Stepanov M.E.: Modeling Diffusive Processes by Cellular Automata with Margolus Neighborhood. Zhurnal Vychislitel'noy Matematiki i Matematicheskoy fiziki, Vol. 36, N 6. (1998), 1017–1021 (in Russian)
4. Wolfram S.: Cellular automata fluids 1: Basic Theory. Journ. Stat. Phys., Vol. 45 (1986), 471–526
5. F.Rothman D.H.,Zaleski,S.: Lattice-Gas Cellular Automata. Simple models of complex hydrodynamics. Cambridge, University Press (1997)
6. Vichniac G.: Simulating Physics by Cellular Cellular Automata. Physica, Vol. 10 D, (1984), 86–115
7. Bandman O.: Simulating Spatial Dynamics by Probabilistic Cellular Automata. Lecture Notes in Computer Science, Vol. 2493. Springer, Berlin Heidelberg New York (2002), 10–19
8. Bandman O.: A Hybrid Approach to Reaction-Diffusion Processes Simulation. Lecture Notes in Computer Science, Vol. 2127. Springer, Berlin Heidelberg New York (2001), 1–16
9. Wolfram S.: A new kind of Science. Wolfram media Inc., Champaign, Il. USA (2002)
10. Bandini S., Mauri G., Pavesi G, Simone C.: A Parallel Model Based on Cellular Automata for Simulation of Pesticide Percolation in the Soil. Lecture Notes in Computer Science, Vol. 1662, Springer, Berlin (1999)
11. Chua L.: A Paradigm for Complexity. World Scientific, Singapore, (1999)

Resource Similarities in Petri Net Models of Distributed Systems^{*}

Vladimir A. Bashkin¹ and Irina A. Lomazova²

¹ Yaroslavl State University
Yaroslavl, 150000, Russia
`bas@uniyar.ac.ru`

² Moscow State Social University
Moscow, 107150, Russia
`irina@lomazova.pereslavl.ru`

Abstract. *Resources* are defined as submultisets of Petri net markings. Two resources are called *similar* if replacing of one by another doesn't change the net's behavior. Two resources are called *similar under a certain condition* if one of them can be replaced by another without changing an observable behavior provided that a comprehending marking contains also some additional resources. The paper studies conditional similarity of Petri net resources, for which the (unconditional) similarity is a special case. It is proved that the resource similarity is a semilinear relation and can be represented as a finite union of linear combinations over a finite set of base conditional resource similarities. The algorithm for computing a finite approximation for conditional resource similarity relation is also presented.

1 Introduction

Nowadays one of the most popular formalisms for modelling and analysis of complex systems is a formalism of Petri nets. Petri nets are widely used in different application areas: from the development of parallel and distributed information systems to the modelling of business processes.

Models based on Petri nets are simple and illustrative. However they are enough powerful: ordinary Petri nets have infinite number of states and reside strictly between finite automata and Turing machines.

In this paper we consider the behavioral aspects of Petri net models. The bisimulation equivalence [7] captures the main features of an observable behavior of a system. As a rule, the bisimulation equivalence is a relation on sets of states. Two states are bisimilar, if they are undistinguishable modulo systems behavior. For ordinary Petri nets the state (marking) bisimulation is undecidable [5].

In [1] for ordinary Petri nets a more weak place bisimulation was introduced and proved to be decidable. The place bisimulation is a relation on sets of places.

^{*} This research was partly supported by the Presidium of the Russian Academy of Science, program "Intellectual computer systems", project 2.3 – "Instrumental software for dynamic intellectual systems" and INTAS-RFBR (Grant 01-01-04003).

Roughly speaking, two places are bisimilar, if replacing a token in one place by a token in another one in all markings doesn't change the system behavior. Place bisimulation can be used for reducing the size of a Petri net, since bisimilar places can be merged without changing the net's behavior.

In [3] we presented the notion of the resource similarity. A resource in a Petri net is a part of a marking. Two resources are similar for a given Petri net if replacing one of them by another in any marking doesn't change the net's behavior. It was proved, that the resource similarity can be generated by a finite basis. However, the resource similarity turned to be undecidable. So, a more strict equivalence relation — the resource bisimulation was defined, for which the place bisimulation of C. Autant and Ph. Schnoebelen is a special case. For a given Petri net and a natural number n the largest resource bisimulation relation on resources of a size not greater than n can be effectively computed.

In this paper we present the notion of the *conditional resource similarity*. Two resources are conditionally similar if one of them can be replaced by another in any marking in the presence of some additional resources. For many applications the notion of the conditional resource similarity is even more natural than unconditional one. For instance, one can replace an excessive memory subsystem by a smaller one with the required maximal capacity provided.

It is shown, that the conditional resource similarity has some nice properties. It is a congruence closed under addition and subtraction of resources. We prove that for each Petri net the maximal plain (unconditional) similarity can be represented as a semilinear closure over some finite basis of conditionally similar pairs of resources.

The conditional resource similarity is undecidable. However, the approximation algorithm from [3] can be modified for computing approximations for both kinds of similarities.

The paper is organized as follows. In section 2 we recall basic definitions and notations on multisets, congruences, Petri nets and bisimulations. In section 3 the conditional resource similarity and its correlation with the resource similarity is studied. In section 4 some basic properties of the resource bisimulation are considered and the algorithm for computing approximations of the unconditional and conditional resource similarities is presented. Section 5 contains some conclusions.

2 Preliminaries

Let S be a finite set. A *multiset* m over a set S is a mapping $m : S \rightarrow \text{Nat}$, where Nat is the set of natural numbers (including zero), i.e. a multiset may contain several copies of the same element.

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S : m(s) \leq m'(s)$ (the inclusion relation). The sum and the union of two multisets m and m' are defined as usual: $\forall s \in S : m + m'(s) = m(s) + m'(s)$, $m \cup m'(s) = \max(m(s), m'(s))$. By $\mathcal{M}(S)$ we denote the set of all finite multisets over S .

Non-negative integer vectors are often used to encode multisets. Actually, the set of all multisets over finite S is a homomorphic image of $Nat^{|S|}$. A binary relation $R \subseteq Nat^k \times Nat^k$ is a congruence if it is an equivalence relation and whenever $(v, w) \in R$ then $(v + u, w + u) \in R$ (here ‘+’ denotes coordinate-wise addition). It was proved by L. Redei [6] that every congruence on Nat^k is generated by a finite set of pairs. Later P. Jančar [5] and J. Hirshfeld [4] presented a shorter proof and also showed that every congruence on Nat^k is a semilinear relation, i.e. it is a finite union of linear sets.

Recall, that a *quasi-ordering* (a qo) is any reflexive and transitive relation $\leq$ over S . A *well-quasi-ordering* (a wqo) is any quasi-ordering $\leq$ such that, for any infinite sequence $x_0, x_1, x_2, \dots$ in S , there exist indexes $i < j$ with $x_i \leq x_j$. If $\leq$ is a wqo, then any infinite sequence contains an infinite increasing subsequence and any infinite sequence contains a finite number of minimal elements.

Let P and T be disjoint sets of *places* and *transitions* and let $F : (P \times T) \cup (T \times P) \rightarrow Nat$. Then $N = (P, T, F)$ is a *Petri net*. A *marking* in a Petri net is a function $M : P \rightarrow Nat$, mapping each place to some natural number (possibly zero). Thus a marking may be considered as a multiset over the set of places. Pictorially, P -elements are represented by circles, T -elements by boxes, and the flow relation F by directed arcs. Places may carry tokens represented by filled circles. A current marking M is designated by putting $M(p)$ tokens into each place $p \in P$. Tokens residing in a place are often interpreted as resources of some type consumed or produced by a transition firing. A simple example, where tokens represent molecules of hydrogen, oxygen and water respectively is shown in Fig. 1.

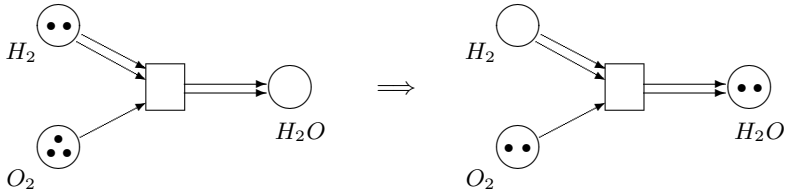


Fig. 1. A chemical reaction.

For a transition $t \in T$ an arc (x, t) is called an *input arc*, and an arc (t, x) — an *output arc*; the *preset* $\bullet t$ and the *postset* $t \bullet$ are defined as the multisets over P such that $\bullet t(p) = F(p, t)$ and $t \bullet(p) = F(t, p)$ for each $p \in P$. A transition $t \in T$ is *enabled* in a marking M iff $\forall p \in P \ M(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking $M' =_{\text{def}} M - \bullet t + t \bullet$, i.e. $M'(p) = M(p) - F(p, t) + F(t, p)$ for each $p \in P$ (denoted $M \xrightarrow{t} M'$).

To observe a net behavior transitions are marked by special labels representing observable actions or events. Let Act be a set of action names. A *labelled Petri net* is a tuple $N = (P, T, F, l)$, where (P, T, F) is a Petri net and $l : T \rightarrow Act$ is a labelling function.

Let $N = (P, T, F, l)$ be a labelled Petri net. We say that a relation $R \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ conforms the *transfer property* iff for all $(M_1, M_2) \in R$ and for every step $t \in T$, s.t. $M_1 \xrightarrow{t} M'_1$, there exists an imitating step $u \in T$, s.t. $l(t) = l(u)$, $M_2 \xrightarrow{u} M'_2$ and $(M'_1, M'_2) \in R$. The transfer property can be represented by the following diagram:

$$\begin{array}{ccc} M_1 & \sim & M_2 \\ \downarrow t & & \downarrow (\exists)u, l(u) = l(t) \\ M'_1 & \sim & M'_2 \end{array}$$

A relation R is called a *marking bisimulation*, if both R and R^{-1} conform the transfer property.

For every labelled Petri net there exists the largest marking bisimulation (denoted by $\sim$) and this bisimulation is an equivalence. It was proved by P. Jančar [5], that the marking bisimulation is undecidable for Petri nets.

3 Resource Similarities

From a formal point of view the definition of a resource doesn't differ from the definition of a marking. Thus, every marking can be considered as a resource and every resource can be considered as a marking. We differentiate these notions because of their different substantial interpretation. Resources are constituents of markings which may or may not provide this or that kind of net behavior, e.g. in Fig. 1 two molecules of hydrogen and one molecule of oxygen form a resource — enough to produce two molecules of water. We could use the term 'submarkings', but we prefer 'resources', since we consider a resource not in the context of 'all submarkings of a given marking', but as a common part of all markings containing it.

Definition 1. Let $N = (P, T, F, l)$ be a labelled Petri net. A resource $R \in \mathcal{M}(P)$ in a Petri net $N = (P, T, F, l)$ is a multiset over the set of places P .

Resources $r, s \in \mathcal{M}(P)$ are called similar (denoted by $r \approx s$) iff for every resource $m \in \mathcal{M}(P)$ we have $m + r \sim m + s$.

Thus if two resources are similar, then in every marking each of these resources can be replaced by another without changing the observable system's behavior. Some examples of similar resources are shown in Fig. 2.

The following proposition states that the resource similarity is a congruence w.r.t. addition of resources.

Proposition 1. Let $m, m', r, s \in \mathcal{M}(P)$. Then

1. $m \approx m' \ \& \ r \approx s \ \& \ r \subseteq m \Rightarrow m - r + s \approx m'$;
2. $m \approx m' \ \& \ r \approx s \Rightarrow m + r \approx m' + s$;
3. $m \approx r \ \& \ r \approx s \Rightarrow m \approx s$.

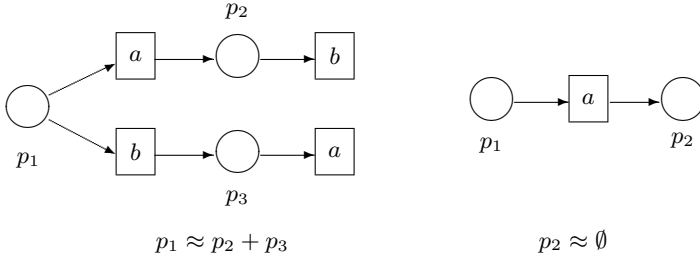


Fig. 2. Examples of similar resources.

Proof. 1) From the definition.

2) From the first claim.

3) Since the largest marking bisimulation $\sim$ is closed under the transitivity. $\square$

Now we define the *conditional similarity*.

Definition 2. Let $r, s, b \in \mathcal{M}(P)$. Resources r and s are called similar under a condition b (denoted $r \approx_{|b} s$) iff for every resource $m \in \mathcal{M}(P)$ s.t. $b \subseteq m$ we have $m + r \sim m + s$.

Resources r and s are called conditionally similar (denoted $r \approx_{|} s$) iff there exists $b' \in \mathcal{M}(P)$ s.t. $r \approx_{|b'} s$.

The conditional similarity has a natural interpretation. Consider, for example, a net in Fig. 3(a). The resources p_1 and p_2 are not similar since in the marking p_1 no transitions are enabled while in the marking p_2 the transition a may fire. However, they are similar *under the condition q* , i.e. in the presence of the resource q resources p_1 and p_2 can replace each other. Another example is given in Fig. 3(b). It is clear, that for this net any number of tokens in the place p can be replaced by any other nonzero number of tokens, i.e. *under the condition* that at least one token resides in this place.

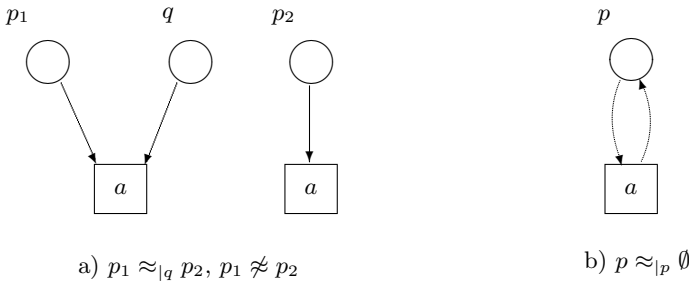


Fig. 3. Examples of conditionally similar resources.

The next proposition states some important properties of the conditional similarity.

Proposition 2. *Let $r, s, b, b', m, m' \in \mathcal{M}(P)$.*

1. $m + r \approx m + s \Leftrightarrow r \approx_{|m} s$.
2. $m \approx_{|m'} m', r \approx_{|s} s \Rightarrow m + r \approx_{|m'} m' + s$.
3. $r \approx_{|b} s, b \subseteq b' \Rightarrow r \approx_{|b'} s$.
4. $m + r \approx_{|b} m + s \Leftrightarrow r \approx_{|b+m} s$.
5. $m + r \approx_{|m} m + s \Leftrightarrow r \approx_{|s} s$.
6. $m \approx m', m + r \approx m' + s \Rightarrow r \approx_{|s} s$.
7. $m \approx_{|b} m', m + r \approx_{|b'} m' + s \Rightarrow r \approx_{|s} s$.

Proof. 1) Immediately from the definitions.

2) Let $m \approx_{|b} m'$ and $r \approx_{|b'} s$. Then from the claim 1 we have

$$m + b \approx m' + b \text{ and } r + b' \approx s + b'.$$

From the second claim of proposition 1

$$m + r + b + b' \approx m' + s + b + b'.$$

Applying the claim 1 once again we get $m + r \approx_{|b+b'} m' + s$.

3) From the definitions.

4) From the definitions.

5) An immediate corollary of the claim 4.

6) Due to the congruence property from $m \approx m'$ and $m + r \approx m' + s$ we get $m + r \approx m + s$, i.e. $r \approx_{|m} s$.

7) From the claim 1 we have

$$m + b \approx m' + b \text{ and } m + r + b' \approx m' + s + b'.$$

Since the similarity is closed under the addition, we get

$$m + b + b' \approx m' + b + b' \text{ and } m + r + b' + b \approx m' + s + b' + b.$$

Thus, from the claim 6 we get $r \approx_{|s} s$. □

In words the statements of Proposition 2 can be formulated as follows: The conditional resource similarity is closed under the addition. It is invariant modulo the condition enlargement. Claims 4 and 5 state that the common part can be removed from both similar resources. Claims 6 and 7 state that the difference of similar, as well as conditionally similar, resources is also conditionally similar.

So, unlike the plain similarity, the conditional similarity is closed under the subtraction. This property can be used as a foundation for constructing an additive base for the conditional similarity relation.

Definition 3. *Let $r, s, r', s', r'', s'' \in \mathcal{M}(P)$. A pair $r \approx_{|s} s$ of conditionally similar resources is called minimal if it can't be decomposed into a sum of two other non-empty conditionally similar pairs, i.e. for every non-empty pair $r' \approx_{|s'} s'$ of conditionally similar resources $r = r' + r''$ and $s = s' + s''$ implies $r = r'$ and $s = s'$.*

From the proposition 2.7 one can easily obtain

Corollary 1. *Every pair of conditionally similar resources can be decomposed into a sum of minimal pairs of conditionally similar resources.*

Proposition 3. *For every Petri net the set of minimal pairs of conditionally similar resources is finite.*

Proof. Multisets over a finite set of places can be encoded as non-negative integer vectors. Then minimal pairs of conditionally similar resources are represented by minimal (w.r.t. coordinate-wise comparison) non-negative integer vectors of double length.

For non-negative integer vectors the coordinate-wise partial order $\leq$ is a well-quasi-ordering, hence there can be only finitely many minimal elements. $\square$

Theorem 1. *The set of all pairs of conditionally similar resources is an additive closure of the finite set of all minimal pairs of conditionally similar resources.*

Immediately from the previous propositions.

Definition 4. *A pair $r \approx s$ of similar resources is called minimal if it can't be represented as a sum of a pair of similar resources and a pair of conditionally similar resources, i.e. for every non-empty pair $r' \approx s'$ of similar resources $r = r' + r''$ and $s = s' + s''$ implies $r = r'$ and $s = s'$.*

From the proposition 2.6 and the theorem 1 we have

Corollary 2. *Every pair of similar resources can be decomposed into the sum of one minimal pair of similar resources and several minimal pairs of conditionally similar resources.*

The next proposition states the interconnection between the plain and the conditional similarities.

Proposition 4. *Let $r, s, m, m' \in \mathcal{M}(P)$, $m \approx m'$. Then $m + r \approx m' + s$ iff $r \approx_{|m} s$.*

Proof. $(\Rightarrow)$ Let $m + r \approx m' + s$. Since $m \approx m'$, by the congruence property we get $m + r \approx m + s$. Then from the proposition 2.1 $r \approx_{|m} s$.

$(\Leftarrow)$ Let $r \approx_{|m} s$. From the proposition 2.1 we have $m + r \approx m + s$. Then, since $m \approx m'$, by the congruence property we get $m + r \approx m' + s$. $\square$

Proposition 5. *For every pair $r \approx_{|} s$ of conditionally similar resources the set of all its minimal conditions (w.r.t. the coordinate-wise comparison) is finite.*

Proof. Since the coordinate-wise ordering $\leq$ is a well-quasi-ordering. $\square$

The conditional similarity is closed under the addition of resources. The exact formulation of this property is given in the following

Proposition 6. *Let $r, r', m, m', b_1, b_2 \in \mathcal{M}(P)$. If $m \approx_{|b_1} m'$ and $r \approx_{|b_2} r'$ then $m + r \approx_{|b_1 \cup b_2} m' + r'$.*

Proof. Since $m + b_1 \approx m' + b_1$ and $r + b_2 \approx r' + b_2$ by the congruence property we get $m + r + b_1 \cup b_2 \approx m' + r' + b_1 \cup b_2$. $\square$

Obviously, this proposition can be generalized to any number of pairs.

Definition 5. Let $R \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ be some set of pairs of conditionally similar resources ($r \approx_{\mid} s$ for every $(r, s) \in R$). Let

$$B = \{ (u, v) \in \mathcal{M}(P) \times \mathcal{M}(P) \mid u \approx v \quad \wedge \quad \forall (r, s) \in R \quad u + r \approx v + s \}$$

be a set of all common conditions for R . By $\text{Cond}(R)$ we denote the set of all minimal elements of B (w.r.t. $\leq$, considering B as a set of vectors of length $2|P|$).

Note, that due to the proposition 4 for $(u, v) \in \text{Cond}(R)$ both u and v are conditions for every $(r, s) \in R$.

Proposition 7. For every R the set $\text{Cond}(R)$ is finite.

Proof. Since the coordinate-wise ordering $\leq$ is a well-quasi-ordering. $\square$

Definition 6. Let $u, v \in \mathcal{M}(P)$ and $u \approx v$. By $S(u, v)$ we denote the set of all potential (w.r.t. the similarity) additives to the pair (u, v) :

$$S(u, v) = \{ (r, r') \in \mathcal{M}(P) \times \mathcal{M}(P) \mid u + r \approx v + r' \}.$$

By $S_{\min}(u, v)$ we denote the set of all minimal elements of $S(u, v)$ (considering B as a set of vectors of length $2|P|$).

Proposition 8. Let $u, v, u', v' \in \mathcal{M}(P)$ and $u \approx v$.

- 1) $S(u, v)$ is a congruence;
- 2) $u \approx v, u' \approx v', (u, v) \leq (u', v') \Rightarrow S(u, v) \subseteq S(u', v')$;
- 3) $S_{\min}(u, v)$ is finite.

Proof. 1) It is clear that $S(u, v)$ is an equivalence relation. Let us show that whenever $(r, s) \in S(u, v)$ then $(r + m, s + m) \in S(u, v)$.

By definition $(r, s) \in S(u, v)$ implies $r + u \approx s + v$. Since the resource similarity is a congruence, one can add the resource m to the both sides of this pair. Hence $r + u + m \approx s + v + m$ and we get $(r + m, s + m) \in S(u, v)$.

2) Denote $(u', v') = (u, v) + (w, w')$. Let $u + r \approx v + r'$ for some pair (r, r') . We immediately have $u + w + r \approx v + w + r' \approx u + w + r' \approx v + w' + r'$, i.e. $u' + r \approx v' + r'$.

- 3) Since the coordinate-wise ordering is a well-quasi-ordering. $\square$

Definition 7. Let N be a Petri net. By $A(N)$ we denote the set of all sets of potential additives in N :

$$A(N) = \{ H \mid \exists (u, v) : u \approx v \wedge H = S(u, v) \}.$$

Proposition 9. *The set $A(N)$ is finite for any Petri net N .*

Proof. Assume this is not true. Then there exist infinitely many different sets of potential additives. Consider the corresponding pairs of similar resources.

There exist infinitely many such pairs, hence there exists an infinite increasing sequence (u_i, v_i) of similar pairs with $S(u_i, v_i) \neq S(u_j, v_j)$ for every $i \neq j$. Since $(u_i, v_i) < (u_{i+1}, v_{i+1})$ for every i , from the second claim of the proposition 8 we have $S(u_i, v_i) \subset S(u_{i+1}, v_{i+1})$.

Recall that each $S(u_i, v_i)$ is a congruence and hence it is finitely generated by the set of its minimal pairs. But the infinite chain of inclusions leads to the infinite growth of the basis and thus contradicts to this property. $\square$

Let $R \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$. By $lc(R)$ we denote the set of all *linear combinations* over R :

$$lc(R) = \{(r, s) \mid (r, s) = (r_1, s_1) + \dots + (r_k, s_k) : (r_i, s_i) \in R \ \forall i = 1, \dots, k\}.$$

Let also $S \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$. By $R + S$ we denote the set of all sums of pairs from R and S :

$$R + S = \{(u, v) \mid (u, v) = (r + r', s + s') : (r, s) \in R, (r', s') \in S\}.$$

Theorem 2. *Let N be a Petri net, $(\approx)$ — the set of all pairs of similar resources for N , $(\approx|)$ — the set of all pairs of conditionally similar resources for N . The set $(\approx)$ is semilinear. Specifically, there exists a finite set $R \subseteq (\approx|)$ s.t.*

$$(\approx) = \bigcup_{\mathcal{R} \in 2^R} [Cond(\mathcal{R}) + lc(\mathcal{R})],$$

where 2^R is the set of all subsets of R .

Proof. $(\supseteq)$ It is clear that for all $\mathcal{R} \subseteq (\approx|)$ we have $[Cond(\mathcal{R}) + lc(\mathcal{R})] \subseteq (\approx)$.

$(\subseteq)$ Consider some pair $u \approx v$. Let (u', v') be the minimal pair of resources such that

- $(u', v') \leq (u, v)$;
- $u' \approx v'$;
- $S(u', v') = S(u, v)$.

Let us prove that $(u, v) \in (u', v') + lc(S_{\min}(u', v'))$. Consider $(u_1, v_1) =_{def} (u - u', v - v')$. Then $u_1 \approx| v_1$ and there exists a pair $(w_1, w'_1) \in S_{\min}(u', v')$ such that $(w_1, w'_1) \leq (u_1, v_1)$. If $(w_1, w'_1) = (u_1, v_1)$, we get the desired decomposition.

Suppose $(w_1, w'_1) < (u_1, v_1)$. Then we have $(u', v') < (u' + w_1, v' + w'_1) < (u' + u_1, v' + u'_1) = (u, v)$. From $S(u', v') = S(u, v)$ we obtain $S(u' + w_1, v' + w'_1) = S(u, v)$.

Consider $(u_2, v_2) =_{def} (u_1 - w_1, v_1 - w'_1)$. Reasoning as above, we can show that $u_2 \approx| v_2$ and hence there exists a pair $(w_2, w'_2) \in S_{\min}(u', v')$ such that $(w_2, w'_2) \leq (u_2, v_2)$. If $(w_2, w'_2) = (u_2, v_2)$, we get the desired decomposition. If

$(w_2, w'_2) < (u_2, v_2)$, then we repeat the reasoning and obtain pairs (u_3, v_3) and (w_3, w'_3) and so on.

Since $(u_1, v_1) > (u_2, v_2) > (u_3, v_3) > \dots$, for some step we get $(w_j, w'_j) = (u_j, v_j)$ and hence

$$(u, v) = (u', v') + (w_1, w'_1) + \dots + (w_j, w'_j) \in (u', v') + lc(S_{\min}(u', v')).$$

Let us show now that the set R is finite. It is sufficient to show that there are only finitely many candidates to be (u', v') in the previous reasoning for all possible similar pairs.

Recall that there are only finitely many different sets $S(u, v)$ (proposition 9). Since the natural order $\leq$ (coordinate-wise comparison) is a well-quasi-ordering, there are also finitely many minimal pairs $(u', v') \in (\approx)$ with $S(u', v') = S(u, v)$. $\square$

This theorem shows the correlation between the plain resource similarity and the conditional resource similarity. There could be a question, if it is possible to use just the *minimal* conditionally similar resources in this decomposition. Indeed, it would be fine to produce the complete plain resource similarity from only minimal conditionally similar pairs, rather than from 'some' finite subset. Unfortunately, it is not possible.

Consider a small example in figure 4.



Fig. 4. A cycle with double arcs.

It is easy to see, that the minimal conditionally similar pair of resources for this Petri net is $0 \approx_{|_2} 1$. One token is similar to any number of tokens if there are at least 2 another tokens in the only place of the net. However, there exists another (not minimal) conditionally similar pair $1 \approx_{|_1} 2$ with a smaller minimal condition 1.

In Fig. 5 we give also an example, showing that a sum of conditionally similar pairs can have a smaller minimal condition than its components. Indeed, pairs $m_1 \approx_{|_{b_1}} m'_1$ and $m_2 \approx_{|_{b_2}} m'_2$ are minimal pairs of conditionally similar resources, but the pair $m_1 + m_2 \approx m'_1 + m'_2$ has the empty condition.

So in the additive decompositions of unconditionally similar resources we are to take into account not just the minimal conditionally similar pairs, but also some other pairs, depending on decomposed resources.

4 Resource Bisimulation

In practical applications a question of interest is whether two given resources in a Petri net are similar or not. So, one would like to construct an appropriate

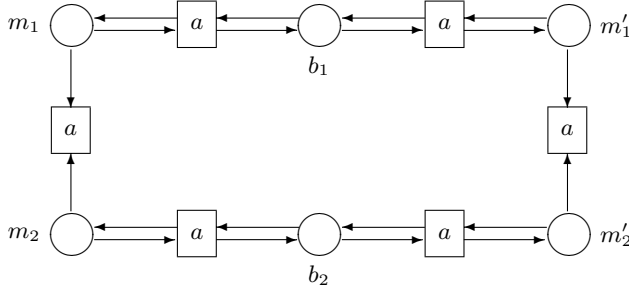


Fig. 5. A bigger example.

algorithm, answering this question or computing the largest resource similarity. Unfortunately, it is not possible in general:

Theorem 3. [3] *The resource similarity is undecidable for Petri nets.*

Hence from the proposition 2.1 we immediately get

Corollary 3. *The conditional resource similarity is undecidable for Petri nets, i.e. it is impossible to construct an algorithm, answering whether a given pair of resources is similar under a given condition.*

However, it is possible to construct a special structured version of the resource similarity — the resource bisimulation. The main advantage of the resource bisimulation is that there exists an algorithm, computing the parameterized approximation of the largest resource bisimulation for a given Petri net.

Definition 8. *An equivalence $B \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ is called a resource bisimulation if B^{AT} is a marking bisimulation (where B^{AT} denotes the closure under the transitivity and the addition of resources of the relation B).*

The relation of the resource bisimulation is a subrelation of the resource similarity:

Proposition 10. [3] *Let N be a labelled Petri net. If B is a resource bisimulation for N and $(r, s) \in B$, then $r \approx s$.*

The relation B^{AT} is a congruence, so it can be generated by a finite number of minimal pairs [6,4]. Moreover, in [3] it was proved, that a finite bases of B^{AT} can be described as follows.

Define a partial order $\sqsubseteq$ on the set $B \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ of pairs of resources: for “loop” pairs let

$$(r_1, r_1) \sqsubseteq (r_2, r_2) \stackrel{def}{\iff} r_1 \subseteq r_2;$$

for “non-loop” pairs “loop” and nonintersecting addend components are compared separately

$$(r_1 + o_1, r_1 + o'_1) \sqsubseteq (r_2 + o_2, r_2 + o'_2) \stackrel{def}{\iff}$$

$$\stackrel{def}{\Leftrightarrow} o_1 \cap o'_1 = \emptyset \ \& \ o_2 \cap o'_2 = \emptyset \ \& \ r_1 \subseteq r_2 \ \& \ o_1 \subseteq o_2 \ \& \ o'_1 \subseteq o'_2.$$

Note that by this definition reflexive and non-reflexive pairs are incomparable. Let B_s denote the set of all minimal (w.r.t. $\sqsubseteq$) elements of B^{AT} . We call B_s the *ground basis* of B .

Theorem 4. [3] *Let $B \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ be a symmetric and reflexive relation. Then $(B_s)^{AT} = B^{AT}$ and B_s is finite.*

So, it is sufficient to deal with the ground basis — a finite resource bisimulation, generating the maximal resource bisimulation.

Definition 9. *A relation $B \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ conforms the weak transfer property if for all $(r, s) \in B$, for all $t \in T$, s.t. $\bullet t \cap r \neq \emptyset$, there exists an imitating step $u \in T$, s.t. $l(t) = l(u)$ and, writing M_1 for $\bullet t \cup r$ and M_2 for $\bullet t - r + s$, we have $M_1 \xrightarrow{t} M'_1$ and $M_2 \xrightarrow{u} M'_2$ with $(M'_1, M'_2) \in B^{AT}$.*

The weak transfer property can be represented by the following diagram:

$$\begin{array}{ccc} r & \approx_B & s \\ \bullet t \cup r & & \bullet t - r + s \\ \downarrow t & & \downarrow (\exists)u, \ l(u) = l(t) \\ M'_1 & \sim_{B^{AT}} & M'_2 \end{array}$$

Theorem 5. [3] *A relation $B \subseteq \mathcal{M}(P) \times \mathcal{M}(P)$ is a resource bisimulation iff B is an equivalence and it conforms the weak transfer property.*

Due to this theorem to check whether a given finite relation B is a resource bisimulation, one needs to verify the weak transfer property for only a finite number of pairs of resources. We can use this fact for computing a finite approximation of the conditional resource similarity.

Actually, we use the weak transfer property to compute the largest plain resource bisimulation for resources with a bounded number of tokens and then produce the corresponding conditional similarity.

Let $N = (P, T, F, l)$ be a labelled Petri net and $\mathcal{M}_q(P)$ denote the set of all its resources, containing not more than q tokens (residing in all places). The largest resource bisimulation on $\mathcal{M}_q(P)$ is defined as the union of all resource bisimulations on $\mathcal{M}_q(P)$. We denote it by $B(N, q)$. By $C(N, q)$ we denote the subset of the conditional resource similarity $B(N, q)$ of the net N , obtained from $B(N, q)$ as follows:

$$\begin{aligned} C(N, q) = \{ & r \approx_b s \mid (r + b, s + b) \in B(N, q) \ \& \ r \cap s = \emptyset \ \& \\ & \wedge \ \nexists b' < b : (r + b', s + b') \in B(N, q) \} \end{aligned}$$

$C(N, q)$ is just a set of elements of $B(N, q)$ with a distinguished “loop” part (the condition). The set $C(N, q)$ of pairs of conditionally similar resources completely describes the relation $B(N, q)$ (cf. proposition 2). The set $B(N, q)$ is finite, and hence $C(N, q)$ can be effectively constructed.

Computing $B(N, q)$ is based on the finiteness of the set $\mathcal{M}_q(P)$ and uses the weak transfer property of the resource bisimulation.

Algorithm.

input: a labelled Petri net $N = (P, T, F, l)$, a positive integer q .

output: the relation $C(N, q)$

step 1: Let $NB = \{(\emptyset, \emptyset)\}$ be an empty set of pairs (further the set of non-bisimilar pairs of resources).

step 2: Let $B = (\mathcal{M}_q(P) \times \mathcal{M}_q(P)) \setminus NB$.

step 3: Compute a ground basis B_s .

step 4: Check if B_s conforms the weak transfer property:

- If the weak transfer property is valid, then B is $B(N, q)$.
- Otherwise, there is a pair $(r, s) \in B_s$ and a transition $t \in T$ with $\bullet t \cap r \neq \emptyset$ s.t. $\bullet t \cup r \xrightarrow{t} M_1'$ cannot be imitated from $\bullet t - r + s$. Then add pairs (r, s) , (s, r) to NB and return to step 2.

step 5: Compute $C(N, q)$ from $B(N, q)$ by subtracting the reflexive parts and determining the minimal conditions.

The relation $B(N, q)$ can be considered as an approximation of the largest resource bisimulation $B(N)$. It is clear that for $q \leq q'$, $B(N, q) \subseteq B(N, q')$ and $B(N) = \bigcup_q B(N, q)$. By increasing q , we produce a closer approximations of $B(N)$. Since $B(N)$ has a finite ground basis, there exists q_0 s.t. $B(N) = B(N, q_0)$. The problem is to evaluate q_0 .

The question, whether the largest resource bisimulation can be effectively computed, is still open. We suppose, that the problem of evaluating q_0 is undecidable, since we believe (but cannot prove), that the largest resource bisimulation of a Petri net coincides with its resource similarity, and the resource similarity is undecidable.

For practical applications an upper bound for q_0 can be evaluated either by experts in the application domain, or by analysis of a concrete net. Then the algorithm computing $B(N, q)$ and $C(N, q)$ can be used for searching similar resources.

5 Conclusion

In this paper we presented the plain and conditional resource similarity relations on Petri net markings, which allow to replace a submarking by a similar one without changing an observable net's behavior.

These relations can be used for analysis of dependencies between resources in a modelled system. Resource similarities can be used also as simplifying patterns for reduction of a net model [2].

It is shown in the paper, that the resource similarity relations have some nice properties and, being infinite, can be represented by a finite basis. An algorithm computing the parameterized approximation of the largest resource similarity for a given Petri net is also presented.

The definitions and results presented here for ordinary Petri nets can be naturally generalized for other Petri net models, e.g. high-level Petri nets and nested Petri nets, as it was done for the resource bisimulation in [3].

References

1. C. Autant and Ph. Schnoebelen: Place Bisimulations in Petri Nets. In: Proc. 13th Int. Conf. Application and Theory of Petri Nets, Lecture Notes in Computer Science, Vol. 616. Springer, Berlin Heidelberg New York (1992), 45–61
2. V. A. Bashkin and I. A. Lomazova: Reduction of Coloured Petri Nets based on Resource Bisimulation. Joint Bulletin of NCC & IIS, Series: Computer Science, Vol. 13. Novosibirsk, Russia (2000), 12–17
3. V. A. Bashkin and I. A. Lomazova: Resource Bisimulations in Nested Petri Nets. In: Proc. of CS&P'2002, Vol.1, Informatik-Bericht Nr.161, Humboldt-Universitat zu Berlin, Berlin (2002),39–52
4. Hirshfeld Y: Congruences in Commutative Semigroups . Research Report ECS-LFCS-94-291, Department of Computer Science, University of Edinburgh (1994)
5. P. Jančar: Decidability Questions for Bisimilarity of Petri Nets and Some Related Problems. In: Proc. STACS'94, Lecture Notes in Computer Science, Vol. 775. Springer-Verlag, Berlin Heidelberg New York (1993), 581–592
6. Redei L.: The Theory of Finitely Generated Commutative Semigroups. Oxford University Press, New York (1965)
7. R. Milner: A Calculus of Communicating Systems. Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, Berlin Heidelberg New York (1980)
8. Ph. Shnoebelen and N. Sidorova: Bisimulation and the Reduction of Petri Nets. In: Proc. 21st Int. Conf. Application and Theory of Petri Nets, Lecture Notes in Computer Science, Vol. 1825. Springer-Verlag, Berlin Heidelberg New York (2000), 409–423
9. N. Sidorova: Petri Nets transformations. PhD theses, Yaroslavl State University, Yaroslavl, Russia, (1998). In Russian

Authentication Primitives for Protocol Specifications*

Chiara Bodei¹, Pierpaolo Degano¹, Riccardo Focardi², and Corrado Priami³

¹ Dipartimento di Informatica, Università di Pisa
Via Filippo Buonarroti, 2, I-56127 Pisa, Italy {chiara,degano}@di.unipi.it

² Dipartimento di Informatica, Università Ca' Foscari di Venezia,
Via Torino 155, I-30173 Venezia, Italy
focardi@dsi.unive.it

³ Dipartimento di Informatica e Telecomunicazioni, Università di Trento
Via Sommarive, 1438050 Povo (TN), Italy priami@science.unitn.it

Abstract. We advocate here the use of two authentication primitives we recently propose in a calculus for distributed systems, as a further instrument for programmers interested in authentication. These primitives offer a way of abstracting from various specifications of authentication and obtaining idealized protocols “secure by construction”. We can consequently prove that a cryptographic protocol is the correct implementation of the corresponding abstract protocol; when the proof fails, reasoning on the abstract specification may drive to the correct implementation.

1 Introduction

Security in the times of Internet is something people cannot do without. Security has to do with confidentiality, integrity and availability, but also with non-repudiation, authenticity and even more, depending on the application one has in mind. The technology of distributed and parallel systems and networks as well influences security, introducing new problems and scenarios, and updating some of the old ones.

A big babel of different properties and measures have been defined to guarantee that a system is secure. All the above calls for formal methods and flexible tools to catch the elusive nature of security.

Mostly, problems arise because it is necessary to face up to the heterogeneity of administration domains and untrustability of connections, due to geographic distribution: communications between nodes have to be guaranteed, both by making it possible to identify partners during the sessions and by preserving the secrecy and integrity of the data exchanged. To this end specifications for message exchange, called security protocols, are defined on the basis of cryptographic algorithms. Even though carefully designed, protocols may have flaws, allowing malicious agents or *intruders* to violate security. An intruder gaining some control over the communication network is able to intercept or forge or invent messages. In this way the intruder may convince agents to reveal sensitive information (confidentiality's problems) or to believe it is one of the legitimate agents in the session (authentication's problems).

* Work partially supported by EU-project DEGAS (IST-2001-32072) and by Progetto MIUR Metodi Formali per la Sicurezza (MEFISTO).

Authentication is one of the main issues in security and it can have different purposes depending on the specific application considered. For example, *entity authentication* is related to the verification of an entity's claimed identity [20], while *message authentication* should make it possible for the receiver of a message to ascertain its origin [28]. In recent years there have been some formalizations of these different aspects of authentication (see, e.g., [1,8,14,16,17,21,27]). These formalizations are crucial for proofs of authentication properties, that sometimes have been automatized (see, e.g. [11,18,23,22, 25]).

A typical approach presented in the literature is the following. First, a protocol is specified in a certain formal model. Then the protocol is shown to enjoy the desired properties, regardless of its operating environment, that can be unreliable, and can even harbour a hostile intruder.

We use here basic calculi for modelling concurrent and mobile agents. In particular, we model protocols as systems of processes, called *principals* or *parties*. Using a pure calculus allows us to reason on authentication and security from an abstract point of view. Too often, security objectives, like authentication, are not considered in the very design phase and are instead approximately recovered after it. The ideal line underlying our approach relies on the conviction that security should directly influence the design of programming languages, because languages for concurrent and distributed systems do not naturally embed security.

In particular, we here slightly extend the spi calculus [1,2], a language for modelling concurrent and distributed agents, endowed with cryptographic primitives. We give this calculus certain kinds of semantics, exploiting the built-in mechanisms for authentication, introduced in [4]. Our mechanisms enable us to abstract from the various implementations/specifications of authentication, and to obtain idealized protocols which are "secure by construction". Our protocols, or rather their specifications can then be seen as a reference for proving the correctness of "real" protocols.

In particular, our first mechanism, called *partner authentication* [4], guarantees each principal A to engage an entire run session with the same partner B . Essentially, the semantics provides a way of "localizing" a channel to A and B , so that the partners accept sensitive communications on this localized channel, only. In particular, a receiver can localize the principal that sent him a message. Such a localization relies on the so-called *relative address* of A with respect to B . Intuitively, this represents the path between A and B in (an abstract view of) the network (as defined by the syntax of the calculus). Relative addresses are *not* available to the users of the calculus: they are used by the abstract machine of the calculus only, defined by its semantics.

Our solutions assume that the implementation of the communication primitives has a reliable mechanism to control and manage relative addresses. In some real cases this is possible, e.g., if the network management system filters every access of a user to the network as it happens in a LAN or in a virtual private network. This may not be the case in many other situations. However, relative addresses can be built by storing the actual address of processes in selected, secure parts of message headers (cf. IPsec [19]).

Also our second mechanism, called *message authentication* [6,4], exploits relative addresses: a datum belonging to a principal A is seen by B as "localized" in the local

space of A . So, our primitive enables the receiver of a message to ascertain its origin, i.e. the process that created it.

The above sketched primitives help us to give the *abstract* version of the protocol under consideration, which has the desired authentication properties “by construction”. A more *concrete* version of the protocol, possibly involves encryptions, nonces, signatures and the like. It gives security guarantees, whenever its behaviour turns out to be similar to that of the abstract specification. A classical process algebraic technique to compare the behaviour of processes is using some notion of equivalence: the intuition is that two processes have the same behaviour if no distinction can be detected by an external process interacting with each of them. The concrete version of a protocol is secure if its behaviour cannot be distinguished from the one of the abstract version. This approach leads to testing equivalence [10,7] and we shall follow it hereafter.

Our notion directly derives from the Non-Interference notion called NDC that has been applied to protocol analysis in [17,16,15]. Note also that the idea of comparing cryptographic protocol with secure-by-construction specifications is also similar to the one proposed in [1] where a protocol is compared with “its own” secure specification. We are indeed refining Abadi’s and Gordon’s approach [1]: the secure abstract protocol here is unique (as we will show in the following) and based on abstract authentication primitives. On the contrary, in [1] for each protocol one needs to derive a secure specification (still based on cryptography) and to use it as a reference for proving authentication.

The paper is organized as follows. The next section briefly surveys our version of the spi calculus. Section 3 intuitively presents our authentication primitives, Section 4 introduces our notion of correct implementation. Finally, Section 5 gives some applications.

2 The Spi Calculus

Syntax. In this section we intuitively recall a simplified version of the spi calculus [1, 2]. In the full calculus, terms can also be pairs, zero and successors of terms. Extending our proposal to the full calculus is easy. Our version of the calculus extends the π -calculus [24], with cryptographic primitives. Here, terms can be names, variables and can also be structured as pairs (M_1, M_2) or encryptions $\{M_1, \dots, M_k\}_N$. An encryption $\{M_1, \dots, M_k\}_N$ represents the ciphertext obtained by encrypting $M_1, \dots, M_k$ under the key N , using a shared-key cryptosystem such as DES [9]. We assume to have perfect cryptography, i.e. the only way to decrypt an encrypted message is knowing the corresponding key.

Most of the processes constructs should be familiar from earlier concurrent calculi: I/O constructs, parallel composition, restriction, matching, replication. We give below the syntax and, afterwards, we intuitively present the dynamics of processes. Terms and processes are defined according to the following BNF-like grammars.

$L, M, N ::=$ **terms**
 a, b, c, k, m, n *names*
 x, y, z, w *variables*
 $\{M_1, \dots, M_k\}_N$ *shared encryption*

$P, Q, R ::=$	processes
$\mathbf{0}$	<i>nil</i>
$\overline{M}\langle N \rangle.P$	<i>output</i>
$M(x).P$	<i>input</i>
$(\nu m)P$	<i>restriction</i>
$P P$	<i>parallel composition</i>
$[M = N]P$	<i>matching</i>
$!P$	<i>replication</i>
<i>case L of $\{x_1, \dots, x_k\}_k$ in P shared-key decryption</i>	

- The null process $\mathbf{0}$ does nothing.
- The process $\overline{M}\langle N \rangle.P$ sends the term N on the channel denoted by M (a name, or a variable to be bound to), provided that there is another process waiting to receive on the same channel. Then behaves like P .
- The process $M(x).P$ is ready to receive an input N on the channel denoted by M and to behave like $P\{N/x\}$, where the term N is bound to the variable x .
- The operator $(\nu m)P$ acts as a static declaration (i.e. a binder for) the name m in the process P that it prefixes. The agent $(\nu m)P$ behaves as P except that I/O actions on m are prohibited.
- The operator $|$ describes parallel composition of processes. The components of $P|Q$ may act independently; also, an output action of P (resp. Q) at any output port $\overline{M}$ may synchronize with an input action of Q (resp. P) at M . In this case, a silent action τ results.
- Matching $[M = N]P$ is an if-then operator: process P is activated only if $M = N$.
- The process $!P$ behaves as infinitely many copies of P running in parallel, i.e. it behaves like $P | !P$.
- The process *case L of $\{x_1, \dots, x_k\}_N$ in P* attempts to decrypt L with the key N . If L has the form $\{M_1, \dots, M_k\}_N$, then the process behaves as the process P , where each x_i has been replaced by M_i , i.e. as the process $P\{M_1/x_1, \dots, M_k/x_k\}$. Otherwise the process is stuck.

The operational semantics of the calculus is a labelled transition system, defined in the SOS, logical style. The transitions are represented as $P \xrightarrow{\tau} P'$, where the label corresponds to a silent or internal action τ that leads the process P in the process P' . To give the flavour of the semantics, we illustrate the dynamic evolution of a simple process S . For more details, see [4].

Example 1. In this example, the system S is given by the parallel composition of the replication (of process P) $!P$ and of the process Q .

$$\begin{aligned}
S &= !P | Q \\
P &= \overline{a}\langle \{M\}_k \rangle. \mathbf{0} \\
Q &= a(x). \text{case } x \text{ of } \{y\}_k \text{ in } Q' \\
Q' &= (\nu h)(\overline{b}\langle \{y\}_h \rangle. \mathbf{0} | R)
\end{aligned}$$

$!P$ represents a source of infinitely many outputs on a of the message M encrypted under k . Therefore it can be rewritten as $P | !P = \overline{a}\langle \{M\}_k \rangle. \mathbf{0} | !P$. So, we have the following part of computation:

$$S \xrightarrow{\tau} \mathbf{0} \mid !P \mid \text{case } \{M\}_k \text{ of } \{y\}_k \text{ in } Q' \xrightarrow{\tau} \mathbf{0} \mid !P \mid (\nu h)(\bar{b}(\{M\}_h). \mathbf{0} \mid R)$$

In the first transition, Q receives on channel a the message $\{M\}_k$ sent by P and $\{M\}_k$ replaces x in the residual of Q . In the second transition, $\{M\}_k$ can be successfully decrypted by the residual of Q , with the correct key k and M replaces y in Q' . The effect is to encrypt M with the key h , private to Q' . The resulting output $\bar{b}(\{M\}_h)$ can occur to be matched by some input in R .

3 Authentication Primitives

Before presenting our authentication mechanisms [4], it is convenient to briefly recall the central notion of *relative address* of a process P with respect to another process Q within a network of processes, described in our calculus. A relative address represents the path between P and Q in (an abstract view of) the network (as defined by the syntax of the calculus). More precisely, consider the abstract syntax trees of processes, built using the binary parallel composition as the main operator. Given a process R , the nodes of its tree (see e.g. Fig. 1) correspond to the occurrences of the parallel operator in R , and its leaves are the sequential components of R (roughly, those processes whose top-level operator is a prefix or a summation or a replication). Assuming that the left (resp.

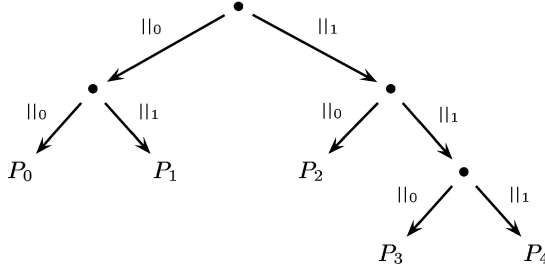


Fig. 1. The tree of (sequential) processes of $(P_0 \mid P_1) \mid (P_2 \mid (P_3 \mid P_4))$.

right) branches of a tree of sequential processes denote the left (resp. right) component of parallel compositions, then label their arcs with tag $\parallel_0$ (resp. $\parallel_1$).

Tecnically, relative addresses can be inductively built while deducing transitions, when a proved semantics is used [13,12], in which labels of transitions encode (a portion of) their deduction tree.

We recall the formal definition of relative addresses [5].

Definition 1 (relative addresses).

Let $\vartheta_i, \vartheta'_i \in \{\parallel_0, \parallel_1\}^*$, let ϵ be the empty string. Then, the set of relative addresses, ranged over by l , is

$$\mathcal{A} = \{\vartheta_0 \bullet \vartheta_1 : \vartheta_0 = \parallel_i \vartheta'_0 \Rightarrow \vartheta_1 = \parallel_{1-i} \vartheta'_1, i = 0, 1\}.$$

For instance, in Fig. 1, the address of P_3 relative to P_1 is $l = ||_0||_1 \bullet ||_1||_1 ||_0$ (read the path upwards from P_1 to the minimal common predecessor and reverse, then downwards to P_3). So to speak, the relative address points back from P_1 to P_3 .

Note that the relative address of P_3 with respect to P_1 is $||_1||_1 ||_0 \bullet ||_0||_1$ that we write also as l^{-1} . When two relative addresses l, l' both refer to the same path, exchanging its source and target, we call them compatible. Formally, we have the following definition.

Definition 2. A relative address $l' = \vartheta' \bullet \vartheta$ is compatible with l , written $l' = l^{-1}$, if and only if $l = \vartheta \bullet \vartheta'$.

We are now ready to introduce our primitives that induce a few modifications to the calculus surveyed above. Note that we separately present below the two primitives, but they can be easily combined, in order to enforce both kinds of authentication.

3.1 Partner Authentication

We can now intuitively present our first semantic mechanism, originally presented in [4]. Essentially, we bind sensitive inputs and outputs to a relative address, i.e. a process P can accept communications on a certain channel, say c , only if the relative address of its partner is equal to an *a priori* fixed address l . More precisely, channels may have a relative address as index, and assume the form c_l . Now, our semantics will ensure that P communicates with Q on c_l if and only if the relative address of P with respect to Q is indeed l (and that of Q with respect to P is l^{-1}). Notably, even if another process $R \neq Q$ possesses the channel c_l , R cannot use it to communicate with P , because relative addresses are not available to the users. Consequently, the hostile process R can never interfere with P and Q while they communicate, as the relative address of R with respect to Q (and to P) is not l (or l^{-1}).

Processes do not always know *a priori* which are the partners' relative addresses. So, we shall also index a channel with a variable λ , to be instantiated by a relative address, only. Whenever a process P , playing for instance the role of sender, has to communicate for the first time with another process S in the role, e.g. of server, it uses a channel c_λ . Our semantic rules will take care of instantiating λ with the address of P relative to S during the communication. From that point on, P and S will keep communicating for the entire session, using their relative addresses.

Suppose, for instance, that in Fig. 1 the process P_3 sends b along a_l and becomes P'_3 , i.e. is $\overline{a_l}(b).P'_3$ and that P_1 reads on a not yet localized channel a_λ a value, e.g. $a_\lambda(x).P'_1$; recall also that the relative address of P_3 with respect to the process P_1 is $l = ||_1||_1 ||_0 \bullet ||_0||_1$. Here P_3 knows the partner address, while P_1 does not. More precisely, for P_3 the output can only match an input executed by the process reachable from P_3 through the relative address l , while the variable λ will be instantiated, during the communication, to the address l^{-1} of the sender P_3 , with respect to the receiver P_1 . From this point on and for the rest of the protocol, P_1 can use the channel $a_{||_0||_1 \bullet ||_1||_1 ||_0}$ (and others that may have the form c_λ) to communicate with P_3 , only.

3.2 Message Authentication

Our second mechanism, called *message authentication*, originally presented in [6,4], enables the receiver of a message to ascertain its origin, i.e. the process that created it. Again it is based on relative addresses.

We illustrate this further extension originally modelled in [6] through a simple example. Suppose that P_3 in Fig. 1 is now $(\nu n)\bar{a}\langle n \rangle.P'_3$. It sends its private name n to $P_1 = a(x).P'_1$. The process P_1 receives it as $\|_1\|_0 \bullet \|_1\|_1\|_0 n = l^{-1}n$. In fact, the name n is enriched with the relative address of P_3 , its sender and creator, with respect to its receiver P_1 and the address l^{-1} acts as a reference to P_3 . Now suppose that P'_1 forwards to P_2 the name just received, i.e. $l^{-1}n$. We wish to maintain the identity of names, i.e., in this case, the reference to P_3 . So, the address l^{-1} will be substituted by a new relative address, that of P_3 with respect to P_2 , i.e. $\|_1\|_0 \bullet \|_0\|_0$. Thus, the name n of P_3 is correctly referred to as $\|_1\|_0 \bullet \|_0 n$ in P_2 . This updating of relative addresses is done through a suitable address composition operation (see [4] for its definition).

We can now briefly recall our second authentication primitive, $[LM \stackrel{\textcircled{a}}{=} l'N]$, akin to the matching operator. This “address matching” is passed only if the relative addresses of the two localized terms under check coincide, i.e. $l = l'$. For instance, if $P_3 = (\nu d)\bar{a}\langle d \rangle.P'_3$, $P_0 = (\nu b)\bar{a}\langle b \rangle$ and $P_1 = a(x).[x = \|_0\|_1 \bullet \|_1\|_1\|_0 d]P'_1$, then P'_1 will be executed only if x will be replaced with a name coming from P_3 , such as $\|_0\|_1 \bullet \|_1\|_1\|_0 n$. In fact, if P_1 communicates with P_0 , then it will receive b , with the address $\|_0\|_0 \bullet \|_1\|_1\|_0$ and the matching cannot be passed.

4 Implementing Authentication

We model protocols as systems of principals, each playing a particular role (e.g. sender or receiver of a message). We observe the behaviour of a system P plugged in any environment E , assuming that P and E can communicate each other on the channels they share. More precisely, E can listen and can send on these channels, possibly interfering with the behaviour of P .

A (specification of a certain) protocol, represented by P , gives security guarantees, whenever its behaviour it is not compromised by the presence of E , in a sense made clear later on.

For each protocol P , we present an *abstract* version of P , written using the above sketched primitives. We will show that this version has the desired authentication properties “by construction”, even in parallel with E . Then, we check the abstract protocol against a different, more *concrete* version, possibly involving standard cryptographic operations (e.g. encryptions, nonces). In other words, we compare their behaviour. The concrete version is secure, whenever it presents the same behaviour of the abstract version. We adopt here the notion of testing equivalence [10,7], where the behaviour of processes is observed by an external process, called tester. Testers are able to observe all the actions of systems, apart from the internal ones.

As a matter of fact, here we push a bit further Abadi and Gordon’s [1] idea of considering correct a protocol if the environment cannot have any influence on its continuation. More precisely, let $P_s = A_s|B_s$ be an abstract secure-by-construction protocol and $P = A|B$ be a (bit more) concrete (cryptographic) protocol. Suppose also that both B and B_s , after the execution of the protocol, continue with some activity, say B' . Then, we require that an external observer should not detect any difference on the behaviour of B' if an intruder E attacks the protocols. In other words, for all intruders E , we require that $A|B|E$ is equivalent to $A_s|B_s|E$. When this holds we say that P *securely implements* P_s . In doing this, we propose to clearly separate the observer, or tester T ,

from the intruder E . In particular, we let the tester T interact with the continuation B' , only. Conversely, we assume that the intruder attacks the protocol, only, and we do not consider how the intruder exploits the attacks for interfering on what happens later on. This allows us to completely abstract from the specific message exchange (i.e., from the communication) and focus only on the “effects” of the protocol execution.

This allows us to compare protocols which may heavily differ in the messages exchanged. In fact, as our authentication primitives provide secure-by-construction (abstract) protocols, the idea is to try to implement them by using, e.g., cryptography. We therefore adopt testing equivalence to formally prove that a certain protocol P implements an abstract protocol P' regardless of the particular message exchange.

We can keep message exchange apart from the rest of the protocol. In our model, protocol specifications are then seen as composed of two sequential parts: a message exchange part and a continuation part, kept separate by using different channels. As said above, the comparison we use focuses on the effects of the protocol execution on the continuation, i.e., on what happens after the protocol has been executed. In other words, the comparison is performed by making invisible the protocol message exchanges and the attacker activity. This is crucial as abstract protocols are never equivalent to their implementation if message exchanges were observed. Moreover, since authentication violations are easily revealed by observing the address of the received message, we can exploit our operator of address matching to this aim. In particular, in our notion of testing equivalence, testers have the ability of directly comparing message addresses (through address matching), thus detecting the origin of messages.

Our notion is such that if P' is a correct-by-construction protocol, specified through our authentication primitives, and P securely implements P' , then also the behaviour of P in every hostile environment, i.e. plugged in parallel with any other process, will be correct.

4.1 A Notion of Secure Implementation

We give here the formal definition of testing equivalence¹ directly on the spi calculus.

We write $P \xrightarrow{\bar{m}} (P \xrightarrow{m}, \text{resp.})$, whenever the process P performs an output (an input, resp.) on the channel m . When the kind of action is immaterial, we shall write $P \xrightarrow{\beta}$ and call β a *barb*.

A *test* is a pair (T, β) , where T is a closed process called *tester* and β is a *barb*. Then, a process P *exhibits* β (denoted by $P \downarrow \beta$) if and only if we have $P \xrightarrow{\beta}$, i.e. if P can do a transition on β . Moreover P *converges on* β (denoted by $P \Downarrow \beta$) if and only if $P \xrightarrow{\tau}^* P'$ and $P' \downarrow \beta$. Now, we say that a process P *immediately passes a test* (T, β) if and only if $(P \mid T) \downarrow \beta$. We also say that a process P *passes a test* (T, β) if and only if $(P \mid T) \Downarrow \beta$.

Our testers are processes that can directly refer to addresses in the address matching operator. As an example, a tester may be the following process $T = \text{observe}(z). [z \stackrel{\text{@}}{=} ||_1 ||_0 \bullet ||_1] \beta(x)$. A tester has therefore a global view of the network, because it has full knowledge of addresses, i.e., of the locations of processes. More importantly, this feature of the testers gives them the ability to directly observe authentication attacks. Indeed a

¹ Technically it is a *may-testing* equivalence.

tester may check if a certain message has been originated by the expected location. As an example, T receives a message on channel *observe* and checks if it has been originated at $\|_1\|_0\bullet\|_1$. Only in this case, the test (T, β) is passed, as the global process (T composed with the protocol) exhibits the barb β . We call $\mathcal{T}$ the set of tester processes.

Now we define the testing preorder $\leq$: a process P is in this relation with a process Q , when each time P passes a test (T, β) , then Q passes the test as well.

Definition 3. $P \leq Q$ iff $\forall T \in \mathcal{T}, \forall \beta : (P \mid T) \Downarrow \beta$ implies $(Q \mid T) \Downarrow \beta$.

As seen above, in our model, protocol specifications are composed of two parts: a message exchange part and a continuation part. Moreover, we assume that the attacker knows the channels that convey messages during the protocol. These channels are *not* used in continuations and can be extracted from the specification of the protocol itself. Note that the continuations may often use channels, that can also be transmitted during their execution, but never used to transmit messages.

We can now give our notion of implementation, where $C = \{c_1, \dots, c_n\}$ is the set of all the channels used by the protocols P and P' .

Definition 4. Let P and P' be two protocols that communicate over C . We say that P securely implements P' if and only if

$$\forall X \in \mathcal{E}_C : (\nu_{c_1}, \dots, (\nu_{c_n})(P \mid X) \leq (\nu_{c_1}, \dots, (\nu_{c_n})(P' \mid X))$$

where $\mathcal{E}_C$ is the set of processes that can only communicate over channels in C .

Note that the names of channels in C are restricted. Moreover, we require that X may only communicate through them. These assumptions represent some mild and reasonable constraints that are useful for the application of the testing equivalence.

These assumptions have both the effect of isolating all the attacker's activity inside the scope of the restriction $(\nu_{c_1}, \dots, (\nu_{c_n})$ and of making all the message exchanges that may be performed by P and P' not observable. As a consequence, we only observe what is done after the protocol execution: the only possible barbs come from the continuations. As we have already remarked, observing the communications part would distinguish protocols based on different message exchanges even if they provide the same security guarantees. Instead, we want to verify whether P implements P' , regardless of the particular underlying message exchange and of the possible hostile execution environment.

The definition above requires that when P and P' are executed in a hostile environment X , all the behaviour of P are also a possible behaviour for P' . So if P' is a correct-by-construction protocol, specified through some authentication primitives, and P securely implements P' , then also P is correct, being also a behaviour for the correct-by-construction protocol P' .

As anticipated in the Introduction, this definition directly derives from the NDC notion. In particular it borrows from NDC the crucial idea of not observing both the communication and the attacker's activity.

5 Some Applications

We show how our approach can be applied to study authentication and freshness. To exemplify our proposal we consider some toy protocols. Nevertheless, we feel that the ideas and the techniques presented could easily scale up to more complicate protocols.

5.1 A Single Session

Consider a simple single-session protocol where A sends a freshly generated message M to B and suppose that B requires authentication of the message, i.e., that M is indeed sent by A . We abstractly denote this as follows, according to the standard and informal protocol narration:

$$\begin{array}{l} (A \text{ freshly generates } M) \\ \text{Message 1} \quad A \xrightarrow{\text{auth}} B : M \end{array}$$

Note that, if B wants to be guaranteed that he is communicating with A , he needs as a reference some trusted information regarding A . In real protocols this is achieved, e.g., through a password or a key known by A only. We use instead the location of the entity that we want to authenticate. In order to do this, we specify this abstract protocol by exploiting our partner authentication primitive. The generation of a fresh message is simply modelled through the restriction operator νM of our calculus.

In order to allow the protocol parties to securely obtain the location of the entity to authenticate, we define a startup primitive that exchanges the respective locations in a trusted way. This primitive is indeed just a macro, defined as follows:

$$\text{startup}(t_A, A, t_B, B) \triangleq (\nu s)(\bar{s}_{t_A}\langle s \rangle.A \mid s_{t_B}(x).B)$$

where x does not occur in B and s does not occur in A and B . The restriction on s syntactically guarantees that communications on that channel cannot be altered by anyone else, except for A and B . This holds also when the process is executed in parallel with any possibly hostile environment E . Now, in process $\text{startup}(\lambda_A, A, \lambda_B, B)$, after the communication over the fresh channel s , variables λ_A and λ_B are securely bound to the addresses of B and A , respectively. More precisely, for each channel c_{λ_A} in A , λ_A is instantiated to the address of B w.r.t. A , while for each channel c_{λ_B} in B , λ_B is instantiated to the address of A w.r.t. B . So, on these channels, A and B can only communicate each other.

In particular, the following holds:

Proposition 1. *Consider the process $\text{startup}(\lambda_A, A, \lambda_B, B)$. Then, for all possible processes E , in any possible execution of $\text{startup}(\lambda_A, A, \lambda_B, B) \mid E$, the location variable λ_A (λ_B , resp.) can be only assigned to the relative address $\|_{0\bullet}\|_1$, of B with respect to A (to the relative address $\|_1\bullet\|_0$ of A with respect to B , resp.).*

Proof. By case analysis.

Now, we show an abstract specification of the simple protocol presented above:

$$\begin{aligned} P &= \text{startup}(\epsilon.\epsilon, A, \lambda_B, B) \\ A &= (\nu M)\bar{c}\langle M \rangle \\ B &= c_{\lambda_B}(z).B'(z) \end{aligned}$$

Technically, using $\epsilon.\epsilon$ in the place of t_A corresponds to having no localization for the channel with index t_A , e.g. $c_{\epsilon.\epsilon} = c$.

After the startup phase, B waits for a message z from the location of A : any other message coming from a different location cannot be received. In this way we model authentication.

Note that locating the output of M in A (as in $A' = (\nu M)\overline{c_{||_0 \bullet ||_1}}\langle M \rangle$) would give a secrecy guarantee on the message, because the process A would be sure that B is the only possible receiver of M .

Due to the partner authentication, this protocol is secure-by-construction. To see why, consider its execution in a possibly hostile environment, i.e. consider $P \mid E$. By Proposition 1, we directly obtain that λ_B is always assigned to the relative address of A w.r.t. B , i.e., $||_1 \bullet ||_0$. Thus, the semantic rules ensure that B can only receive a value z sent by A , on the located channel $c_{||_1 \bullet ||_0}$. Since A only sends one freshly generated message, we conclude that z will always contain a located name with address $||_1 \bullet ||_0$. This means that B always receives a message which is authentic from A .

As intuitively described in Section 3, the location of the channel c in process B guarantees a form of *entity authentication*: by construction, B communicates with the correct party A . Then, since A is following the protocol (i.e., is not cheating), we also obtain a form of *message authentication* on the received message, i.e., B is ensured that the received message has been originated by A .

To further clarify this we show the two possible execution sequences of the protocol:

$$\begin{aligned} P \mid E &= \text{startup}(\epsilon \bullet \epsilon, A, \lambda_B, B) \mid E \\ &= (\nu s)(\overline{s}\langle s \rangle.A \mid s_{\lambda_B}(x).B) \mid E \\ &\xrightarrow{\tau} (\nu s)((\nu M)\overline{c}\langle M \rangle \mid c_{||_1 \bullet ||_0}(z).B'(z)) \mid E \end{aligned}$$

There are now two possible moves. E may intercept the message sent by A (and then continue as E'):

$$\begin{aligned} &(\nu s)((\nu M)\overline{c}\langle M \rangle \mid c_{||_1 \bullet ||_0}(z).B'(z)) \mid E \xrightarrow{\tau} \\ &(\nu \bullet ||_0 ||_0 M)((\nu s)(\mathbf{0} \mid c_{||_1 \bullet ||_0}(z).B'(z)) \mid E') \end{aligned}$$

The way addresses are handled makes M to be received by E' as $||_1 \bullet ||_0 ||_0 M$, that is with address of A w.r.t. E . For the same reason, the restriction on M in the target of the transition become $(\nu \bullet ||_0 ||_0 M)$.

The other possible interaction is the one between A and B :

$$\begin{aligned} &(\nu s)((\nu M)\overline{c}\langle M \rangle \mid c_{||_1 \bullet ||_0}(z).B'(z)) \mid E \xrightarrow{\tau} \\ &(\nu s)(\nu \bullet ||_0 M)(\mathbf{0} \mid B'(|_1 \bullet ||_0 M)) \mid E \end{aligned}$$

It is important to observe that there is no possibility for E to make B accept a faked message, as B will never accept a communication from a location which is different from $||_1 \bullet ||_0$.

We now show how the abstract protocol above can be used as a reference for more concrete ones, by exploiting the notion of protocol implementation introduced in the previous section.

First, consider a clearly insecure protocol in which A sends M as plaintext to B , without any localized channel.

$$\begin{aligned} P_1 &= A_1 \mid B_1 \\ A_1 &= (\nu M)\bar{c}\langle M \rangle \\ B_1 &= c(z).B'(z) \end{aligned}$$

We can prove that P_1 does not implement P , by using testing equivalence. Consider a continuation that exhibits the received value z . So, let $B'(z) = \overline{observe}\langle z \rangle$, and consider the processes $(\nu c)(P \mid E)$ and $(\nu c)(P_1 \mid E)$, where $E = (\nu M_E)\bar{c}\langle M_E \rangle$ is an attacker which sends a fresh message to B , pretending to be A . Let the tester T be the process $\overline{observe}(z).[z \stackrel{@}{=} ||_1||_0 \bullet ||_1]\beta(y)$ which detects if z has been originated by E . Note that the only possible barb of the two processes we are considering is the output channel $\overline{observe}$. It is clear that $(\nu c)(P_1 \mid E)$ may pass the test (T, β) while $(\nu c)(P \mid E)$ cannot pass it, thus $(\nu c)(P \mid E) \not\leq (\nu c)(P_1 \mid E)$. In fact, P_1 can receive the value M_E on z with the address of B_1 w.r.t. E , that it is different from the expected $||_1||_0 \bullet ||_1$. This counter-example corresponds to the following attack:

Message 1 $E(A) \rightarrow B : M_E$ E pretending to be A

We show now that the following protocol, that uses cryptography is able to provide authentication of the exchanged message (in a single protocol session):

Message 1 $A \rightarrow B : \{M\}_{K_{AB}}$

K_{AB} is an encryption key shared between A and B . We specify this protocol as follows:

$$\begin{aligned} P_2 &= (\nu K_{AB})(A_2 \mid B_2) \\ A_2 &= (\nu M)\bar{c}\langle \{M\}_{K_{AB}} \rangle \\ B_2 &= c(z).case\ z\ of\ \{w\}_{K_{AB}}\ in\ B'(w) \end{aligned}$$

Here, A_2 encrypts M to protect it. Indeed, the goal is to prevent other principals from substituting for M a different message, as it may happen in P_1 . This is a correct way of implementing our abstract authentication primitive in a single protocol session. In order to prove that P_2 securely implements P , one has to show that every computation of $(\nu c)(P_2 \mid X)$ is simulated by $(\nu c)(P \mid X)$, for all $X \in \mathcal{E}_c$. This is indeed the case and P_2 gives entity authentication guarantees: B_2 can be sure that it is A the sender of the message. On the other hand, we can also have a form of message authentication, as far as the delivered message w is concerned, since our testers are able to observe the originator of a message through the address matching operator.

Proposition 2. P_2 securely implements P

Proof. We give a sketch of the proof. We have to show that every computation of $(\nu c)(P_2 \mid X)$ is simulated by $(\nu c)(P \mid X)$, for all $X \in \mathcal{E}_c$. To this purpose, we define a relation $\mathcal{S}$ which can be proved to be a barbed weak simulation. Barbed bisimulation [26] provides very efficient proof techniques for verifying may-testing preorder, and is defined as follows. A relation $\mathcal{S}$ is a barbed weak simulation if for $(P, Q) \in \mathcal{S}$:

- $P \downarrow \beta$ implies that $Q \Downarrow \beta$,
- if $P \xrightarrow{\tau} P'$ then there exists Q' s.t. $Q(-\xrightarrow{\tau})^* Q'$ and $(P', Q') \in \mathcal{S}$.

The union of all barbed simulation is represented by $\sqsubseteq$. Moreover, we say that a relation $\mathcal{S}$ is a barbed weak pre-order (denoted with $\approx$) if for $(P, Q) \in \mathcal{S}$ and for all $R \in \mathcal{T}$ we have $P \mid R \sqsubseteq Q \mid R$. It is easy to prove that $\approx \subseteq \leq_{may}$.

We now define a relation $\mathcal{S}$ as follows:

$$(\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\tilde{A} \mid B_2) \mid X) \mathcal{S} (\nu c)(P \mid X)$$

where either $\tilde{A} = A_2$ or $\tilde{A} = \mathbf{0}$. Moreover the key K_{AB} may appear in X only in a term $|_0|_0 \bullet |_1 \{M\}_{K_{AB}}$, possibly as a subterm of some other composed term.

The most interesting moves are the following:

- $\tilde{A} = A_2$, and

$$\begin{aligned} & (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((A_2 \mid B_2) \mid X) \\ & \xrightarrow{\tau} (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\mathbf{0} \mid B'(|_0 \bullet |_1 M)) \mid X) = F \end{aligned}$$

This is simulated as

$(\nu c)(P \mid Y) \xrightarrow{\tau} \xrightarrow{\tau} (\nu c)(\nu|_0 M) ((\mathbf{0} \mid B'(|_0 \bullet |_1 M)) \mid X) = G$. It is easy to see that $F \equiv G$, since K_{AB} is not free in $B'(w)$.

- $\tilde{A} = A_2$, and

$$\begin{aligned} & (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((A_2 \mid B_2) \mid X) \\ & \xrightarrow{\tau} (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\mathbf{0} \mid B_2) \mid X') \end{aligned}$$

Here X' intercepts the message which is exactly $|_0|_0 \bullet |_1 \{M\}_{K_{AB}}$. This is simulated by just idling. We indeed obtain that

$$(\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\mathbf{0} \mid B_2) \mid X') \mathcal{S} (\nu c)(P \mid X).$$

- $\tilde{A} = \mathbf{0}$ and

$$\begin{aligned} & (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\mathbf{0} \mid B_2) \mid X) \\ & \xrightarrow{\tau} (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) \\ & ((\mathbf{0} \mid \text{case } \theta \bullet \theta' \{N\}_{K_{AB}} \text{ of } \{w\}_{K_{AB}} \text{ in } B'(w)) \mid X') = F' \end{aligned}$$

By the hypothesis on X it must be $N = M$ and $\theta \bullet \theta' = |_0|_0 \bullet |_1$. Thus

$$F' \equiv (\nu c)(\nu|_0 K_{AB})(\nu|_0|_0 M) ((\mathbf{0} \mid B'(|_0 \bullet |_1 M)) \mid X')$$

This is simulated as for the first case above.

Since $(\nu c)(P_2 \mid X) \mathcal{S} (\nu c)(P \mid Y)$, we obtain the thesis.

5.2 Multiple Sessions

The version of the protocol P_2 is secure if we consider just one single session, but it is no longer such, when considering more than one session. We will see this, and we will

also see how to repair the above specification in order to obtain the same guarantees. Our first step is extending the *startup* macro to the multisession case:

$$m\_startup(t_A, A, t_B, B) \triangleq (\nu s)(\ !\bar{s}_{t_A}\langle s \rangle.A \mid !s_{t_B}(x).B \)$$

Two processes that initiate the startup by a communication over s are replicated through the “!” operator; so there are many pairs of instances of the sub-processes A and B communicating each other. Each pair plays a single session.

The following result extends Proposition 1 to the multisession case (note that here any replication originates a new instance of the two location variables). Intuitively, the proposition below states that, when many sessions are considered, our startup mechanism is able to establish different independent runs between instances of P and Q , where no messages of one run may be received in a different run. This is a crucial point that provides freshness, thus avoiding replay of messages from a different run.

Proposition 3. *Consider the process $startup(\lambda_A, A, \lambda_B, B)$. Then, for all possible processes E , in any possible execution of the location variable λ_A (λ_B , resp.) can be only assigned to the relative address of a single instance of B with respect to one instance of A (of a single instance of A with respect to one instance of B , resp.).*

Proof. By case analysis.

Actually, different instances of the same process are always identified by different instances of location variables. Therefore, two location variables, arising from two different sessions, never point to the same process.

We now define the extension of P to multisession as follows:

$$P^m = m\_startup(\epsilon \bullet \epsilon, A, \lambda_B, B)$$

Consider now the following execution:

$$\begin{aligned} P^m &= (\nu s)(\ !\bar{s}\langle s \rangle.A \mid !s_{\lambda_B}(x).B \) \mid E \\ &\xrightarrow{\tau} (\nu s)(\ (A \mid !\bar{s}\langle s \rangle.A) \mid (c_{||_0||_0} \bullet ||_1||_0(z).B'(z) \mid !s_{\lambda_B}(x).B) \) \mid E \\ &\xrightarrow{\tau} (\nu s)(\ (A \mid (A \mid !\bar{s}\langle s \rangle.A)) \mid (c_{||_0||_0} \bullet ||_1||_0(z).B'(z) \mid \\ &\quad \mid (c_{||_0||_1||_0} \bullet ||_1||_1||_0(z).B'(z) \mid !s_{\lambda_A}(x).B) \) \) \mid E \end{aligned}$$

Here, the first and second instances of B are uniquely hooked to the first and second instances of A , respectively. This implies that all the future located communications of such processes will be performed only with the corresponding hooked partner, even if they are performed on the same communication channel. Generally, due to non-determinism, instances of A and instances of B may hook in different order.

It is now straightforward proving a couple of properties about authentication and freshness, exploiting Proposition 3. They hold for protocol P^m and for all similar protocols, where multiple sessions arise from the replication of the same processes, playing the same roles. In the following, we use $B'(\vartheta \bullet \vartheta' N)$ to mean the continuation B' , where the variable z has been bound to the value $\vartheta \bullet \vartheta' N$, i.e. to a message N that has the relative address $\vartheta \bullet \vartheta'$ of its sender w.r.t. to its receiver.

Authentication: When the continuation of an instance of $B'(\vartheta \bullet \vartheta' N)$ is activated, $\vartheta \bullet \vartheta'$ must be the relative address of an instance of A with respect to the actual instance of B .

Freshness: For every pair of activated instances of continuations $B'(\vartheta \bullet \vartheta' N)$ and $B'(\tilde{\vartheta} \bullet \tilde{\vartheta}' N')$ it must be $\vartheta \neq \tilde{\vartheta}$, i.e., the two messages have been originated by two different instances of the process A .

We are now able to show that P_2 is not a good implementation when many sessions are considered, i.e. that P_2^m does not implement P^m . Consider:

$$P_2^m = (\nu K_{AB})(!A_2 \mid !B_2)$$

Let $B'(z) = \overline{observe}(z)$, and consider $E = c(x).\bar{c}\langle x \rangle.\bar{c}\langle x \rangle$. E may intercept the encrypted message and replay it twice. If we consider the tester $T = observe(x).observe(y).[x \stackrel{a}{=} y]\beta(x)$, we obtain that $(\nu c)(P_2^m \mid E)$ may pass the test (T, β) while $(\nu c)(P^m \mid E)$ never passes it. Indeed, in P_2^m the replay attack successfully performed, and B is accepting twice the same message:

Message 1.a	$A \rightarrow E(B) : \{M\}_{K_{AB}}$	E intercepts the message intended for B
Message 2.a	$E(A) \rightarrow B : \{M\}_{K_{AB}}$	E pretending to be A
Message 2.b	$E(A) \rightarrow B : \{M\}_{K_{AB}}$	E pretending to be A

Thus, we obtain that $(\nu c)(P_2^m \mid E) \not\leq (\nu c)(P^m \mid E)$.

We end this section by giving a correct implementation of the multisession authentication protocol P^m , which exploits a typical challenge-response mechanism to guarantee authentication:

Message 1	$B \rightarrow A : N$
Message 2	$A \rightarrow B : \{M, N\}_{K_{AB}}$

where N is a freshly generated nonce that constitutes the challenge. It can be formally specified as follows:

$$\begin{aligned} P_3^m &= (\nu K_{AB})(!A_3 \mid !B_3) \\ A_3 &= (\nu M)c(ns).\bar{c}\langle \{M, ns\}_{K_{AB}} \rangle \\ B_3 &= (\nu N)\bar{c}\langle N \rangle.c(x).case\ x\ of\ \{z, w\}_{K_{AB}}\ in\ [w = N]B'(z) \end{aligned}$$

The following holds.

Proposition 4. P_3^m securely implements P^m

Proof. The proof can be carried out in the same style of the one for Proposition 2.

Note that we are only considering protocols in which the roles of the initiator (or sender) and responder (or receiver) are clearly separated. If A and B could play both the two roles in parallel sessions, then the protocol above would suffer of a well-known reflection attack. Extending our technique to such a more general analysis is the object of future research.

References

1. M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". *Information and Computation*, 148(1):1–70, January 1999.
2. M. Abadi. "Secrecy by Typing In Security protocols". *Journal of the ACM*, 5(46):18–36, sept 1999.
3. M. Abadi, C. Fournet, G. Gonthier. Authentication Primitives and their compilation. In *Proceedings of Principles of Programming Languages (POPL'00)*, pp. 302–315. ACM Press, 2000.
4. C. Bodei, P. Degano, R. Focardi, and C. Priami. "Primitives for Authentication in Process Algebras". *Theoretical Computer Science* 283(2): 271–304, June 2002.
5. C. Bodei, P. Degano, and C. Priami. "Names of the π -Calculus Agents Handled Locally". *Theoretical Computer Science*, 253(2):155–184, 2001.
6. C. Bodei, P. Degano, R. Focardi, and C. Priami. "Authentication via Localized Names". In *Proceedings of the 12th Computer Security Foundation Workshop (CSFW12)*, pp. 98–110. IEEE press, 1999.
7. M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, August 1995.
8. M. Burrows, M. Abadi, and R. Needham. "A Logic of Authentication". *ACM Transactions on Computer Systems*, pp. 18–36, February 1990.
9. National Bureau of Standards. "Data Encryption Standard (DES)". FIPS Publication 46, 1977.
10. R. De Nicola and M.C.B. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
11. A. Durante, R. Focardi, and R. Gorrieri. "A Compiler for Analysing Cryptographic Protocols Using Non-Interference". *ACM Transactions on Software Engineering and Methodology*, vol. 9(4), pp. 488–528, October 2000.
12. P. Degano and C. Priami. "Enhanced Operational Semantics: A Tool for Describing and Analysing Concurrent Systems". To appear in *ACM Computing Surveys*.
13. P. Degano and C. Priami. "Non Interleaving Semantics for Mobile Processes". *Theoretical Computer Science*, 216:237–270, 1999.
14. F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. "Strand spaces: Why is a security protocol correct?" In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pp. 160–171, 1998. IEEE Press.
15. R. Focardi, R. Gorrieri, and F. Martinelli. "Message authentication through non-interference". In *Proceedings of International Conference in Algebraic Methodology and Software Technology*, LNCS 1816, pp.258–272, 2000.
16. R. Focardi, R. Gorrieri, and F. Martinelli. "Non Interference for the Analysis of Cryptographic Protocols". In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'00)*, LNCS 1853, Springer, 2000.
17. R. Focardi and F. Martinelli. "A Uniform Approach for the Definition of Security Properties". In *Proceedings of World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, pp. 794–813, Springer-Verlag, 1999.
18. R. Focardi, A. Ghelli, and R. Gorrieri. "Using Non Interference for the Analysis of Security Protocols". In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, DIMACS Center, Rutgers University, 1997.
19. R. Thayer, N. Doraswamy, and R. Glenn. RFC 2411: IP security document roadmap, November 1998.
20. International Organization for Standardization. Information technology – Security techniques – Entity authentication mechanism; Part 1: General model. ISO/IEC 9798–1, Second Edition, September 1991.

21. G. Lowe. "A Hierarchy of Authentication Specification". In *Proceedings of the 10th Computer Security Foundation Workshop (CSFW10)*. IEEE press, 1997.
22. G. Lowe. "Breaking and Fixing the Needham-Schroeder Public-key Protocol using FDR". In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, LNCS 1055, pp. 146–166, Springer-Verlag, 1996.
23. R. Kemmerer, C. Meadows, and J. Millen. "Three systems for cryptographic protocol analysis". *J. Cryptology*, 7(2):79–130, 1994.
24. R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes (I and II)". *Information and Computation*, 100(1):1–77, 1992.
25. J. C. Mitchell, M. Mitchell, and U. Stern. "Automated Analysis of Cryptographic Protocols Using Mur ϕ ". In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*, pp. 141–153. IEEE Computer Society Press, 1997.
26. Sangiorgi, D. "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.". PhD Thesis. University of Edinburgh, 1992.
27. S. Schneider. "Verifying authentication protocols in CSP". *IEEE Transactions on Software Engineering*, 24(9), Sept. 1998.
28. B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 1996. Second edition.

An Extensible Coloured Petri Net Model of a Transport Protocol for Packet Switched Networks

Dmitry J. Chaly and Valery A. Sokolov

Yaroslavl State University, 150000 Yaroslavl, Russia
{chaly,sokolov}@uniyar.ac.ru

Abstract. The paper deals with modelling and analysis of the Transmission Control Protocol (TCP) by means of Coloured Petri Nets (CPN). We present our CPN model and examples of how correctness and performance issues of the TCP protocol can be studied. We show a way of extension of this model for representing the Adaptive Rate Transmission Control Protocol (ARTCP). Our model can be easily configured and used as a basis for constructing formal models of future TCP modifications.

1 Introduction

The TCP/IP protocol suite works almost on all computers connected to the Internet. This protocol suite allows us to connect different computers running different operation systems. The TCP/IP suite has several layers, each layer having its own purpose and providing different services. Transmission Control Protocol (TCP) is the major transport layer protocol of the TCP/IP suite. It provides reliable duplex data transfer with end-to-end congestion control mechanisms. Since 1981, when the original TCP specification [1] had been published, there were many improvements and bug fixes of the protocol. The most important specification documents are: [2], containing many bug fixes and proposing the protocol standard; [3], contributed to TCP performance over large bandwidth $\times$ delay product paths and to provide reliable operation over very high-speed paths; [4], proposing selective acknowledgements (SACK) to cope with multiple segment losses; [6], which extends selective acknowledgements by specifying its use for acknowledging duplicate packets; [5], where standard congestion control algorithms are described; [8], proposing the Limited Transmit algorithm aimed to enhance TCP loss recovery.

Many studies have been contributed to the investigation of various aspects of TCP. Kumar [23] uses a stochastic model to investigate performance aspects of different versions of TCP, considering the presence of random losses on a wireless link. Fall and Floyd in [22] study the benefits of the selective acknowledgement algorithm. A Coloured Petri net model of the TCP protocol is presented in [19], but this version is very simplified and needs more accurate implementation of some algorithms (for example, retransmission time-out estimation). Another

deficiency of this model is its inability to represent simultaneous work of several TCP connections with different working algorithms without any essential reconstruction.

We use timed hierarchical Coloured Petri nets (CP-nets or CPNs) to construct an original CPN model of the TCP protocol of the latest standard specification (not for any TCP implementation). In this paper we also present an example of how our model can be extended without any essential reconstruction of the net structure to model the ARTCP protocol [9,10,11]. We use Design/CPN tool [16,18] to develop the model. Design/CPN tool and Coloured Petri Nets have shown themselves as a good formalism for modelling and analysis of the distributed systems and they have been used in a number of projects, such as [20,21]. We assume that the reader is familiar with basic concepts of high-level Petri nets [12,13,14,15,17].

2 Overview of the Model

Since the whole CP-net is very large, we will consider an example subnet and later we will give an overview of the model. One of the most important actions the protocol performs is the processing of the incoming segments. The subnet which models this aspect is shown in Figure 1. It has places (represented as circles) which are used to model various control structures of the protocol, and a transition (represented as a box) which models how these structures must be changed during execution of the model. The places hold markers which model a state of a given control structure at an instant of time. This is possible because each marker has a type (also called a colorset - represented as an italic text near a place). We distinguish markers which belong to different connections, so the model can be used to model a work of several connections simultaneously. Places and transitions are connected by arcs. Each arc has an expression (also called the arc inscription) written in CPN ML language. This language is a modification of the Standard ML language. An arc which leads from a place to a transition (input arcs) defines a set of markers must be removed from this place, and an arc which leads from a transition to a place (output arcs) defines a set of markers which must be placed to this place. Sometimes arc inscriptions may represent very complex functions. The declaration of these functions forms a very important part of the model, because they define how the protocol control structures will change. We place Standard ML code of the model into external files. Sometimes it is useful to use code segments when we need to calculate complex expressions (a code segment in Figure 1 is shown as a dashed box with the letter C. Note that we omit some code from the illustration). The code segment takes some variables as arguments (the *input* tuple) and returns some variables as a result (the *output* tuple). Result can be used in output arc inscriptions. The use of code segments helps us to calculate expressions only once.

The main motive of using a timed CP-net is the necessity of modelling various TCP timers. Value of the timeout is attached to a marker as a timestamp (in Figure 1 shown as @+ operator). For example, when we process a segment

sometimes we need to restart the retransmission time-out. In Figure 1 the value of the timeout is "stamped" to a marker which is placed to *RetrQueue* place. The marker with the timestamp can be used in transition execution iff it has the timestamp less than or equal to the value of the model global clock. Though we can ignore the timestamp of the marker (in Figure 1 shown as @ignore), for example, when we need to change the marker and recalculate its timestamp value.

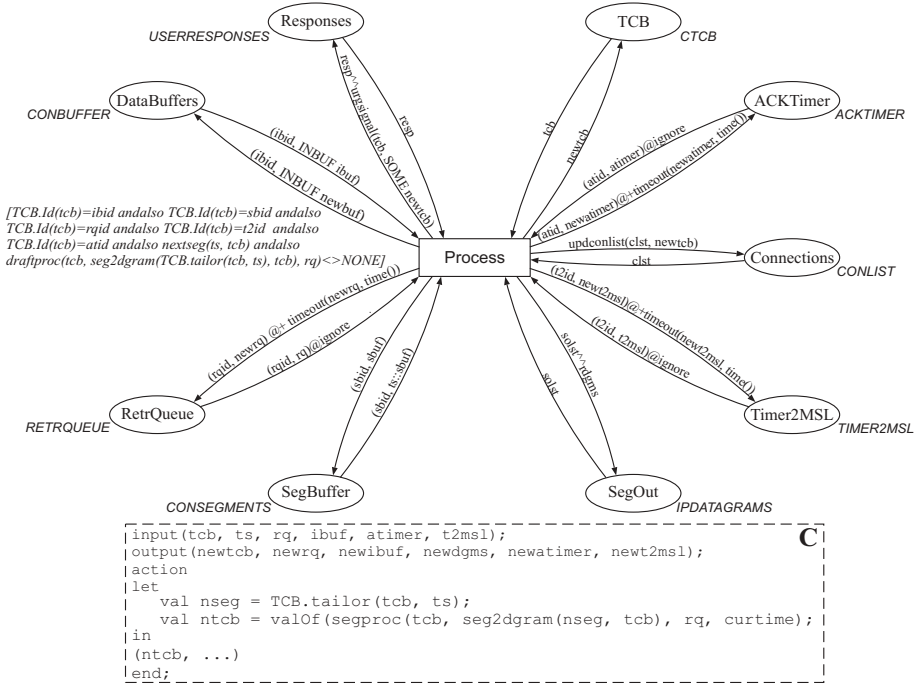


Fig. 1. The Processing page of the model

So we considered an example subnet of our model. Most other nets are not larger than the example but they model very complex aspects of the protocol work. To decompose the model, we represent it as an hierarchy of pages. The CP-net hierarchy of the TCP model is depicted in Figure 2. The hierarchy is represented as a tree, where nodes are pages, containing subnets which model different aspects of the protocol. The modelling of various actions, used by TCP in its work, takes place in subnets represented by leaf nodes (we can see that the Process page is a leaf node). Subnets represented by non-leaf nodes are used to divide the model in some reasonable manner and to deliver various parameters to the leaf subnets.

Since TCP is a medium between a user process and a network, we decided to divide the model into two corresponding parts as shown in Figure 2: the

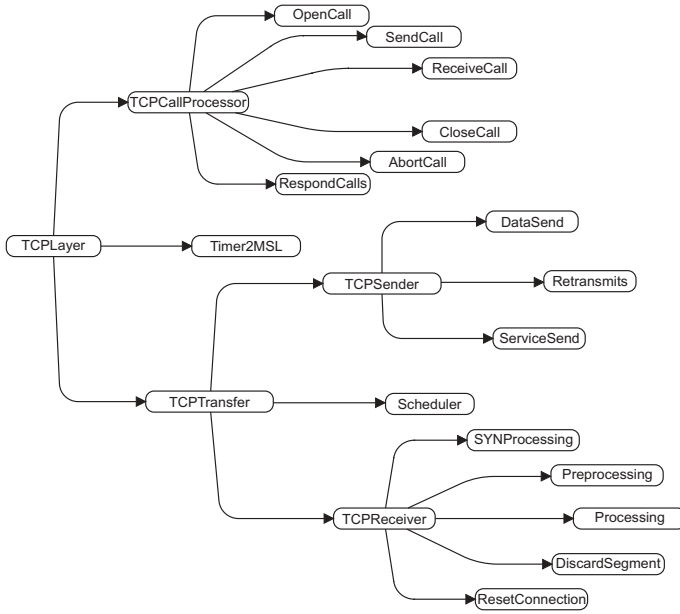


Fig. 2. The Hierarchy of the TCP/ARTCP model

part, which models the processing of various user calls (*TCPCallProcessor* page), and the part, which models the segment exchange (*TCPTransfer* page). The *Timer2MSL* page models the execution of the 2MSL (MSL – maximum segment life-time) timeout.

The page *TCPCallProcessor* has several subpages. All of them, except *RespondCalls* page, are used to model the processing of received user calls (for example, the page *OpenCall* models the processing of the user call OPEN), including error handling. Some user calls can be queued if the protocol can not process them immediately. This can happen, for instance, if a user send RECEIVE call to the protocol and the protocol does not have enough data at hand to satisfy that call. The page *RespondCalls* is used to model such a delayed user call processing. Note that we model a generic TCP-User interface given in [1], not an alternative (for example, Berkely Sockets interface).

The segment exchange part is modelled by subpages of the *TCPTransfer* page. It has a part dedicated to transmitting segments into a network (page *TCPSender*), the processing of the incoming segment part (page *TCPReceiver*) and the service page *Scheduler* which is used to model segment transmission with a given rate.

The page *TCPSender* has subpages that model transmitting a data segment (page *DataSend*), transmitting a service segment – acknowledgement or synchronizing connection segment (SYN-segment) for establishing a connection (page *ServiceSend*) and retransmission of segments predicted to be lost by a network (page *Retransmits*).

Incoming segment processing facility consists of the following parts: the processing of SYN-segments page (*SYNProcessing*); the initial segment preprocessing, used to discard, for example, old duplicate segments (page *Preprocessing*); discard non-acceptable segments which has a closed connection as destination (*DiscardSegment* page); the processing of in-order segments (page *Processing*); the processing of valid incoming reset segments, which are used to reset the connection (page *ResetConnection*).

The presented model meets the latest protocol specification standards referred to in the previous section. Our model has been developed to provide an ability of easy reconfiguring and tuning. It is possible to set many parameters used by various TCP standard documents just by setting appropriate variables in the ML code of the model.

3 Modification of the TCP Model

The extensibility principle, the basis of the model modification, is to add a new Standard ML code or to change the existing one. It is more suitable and less labour consuming than the change of the CP-net structure. The transmission control block (TCB) of the connection plays a very important role in the protocol. It contains almost all data to manage connection, for example, for a window management, for congestion control algorithms, for the retransmission time-out calculation and the other vital data for the protocol work. Better organization of this structure gives us a more suitable code for modification. Because the specification does not restrict the form of this structure we can organize it in the way we want. For example, our implementation of the transmission control block does not directly implement the standard congestion control algorithm but it holds a generic flow control structure. And when TCP tries to determine how much data a congestion control algorithm allows to send, the protocol does not determine it directly from the algorithm parameters, but asks the generic flow control structure to do this job. This structure is used to determine which congestion control algorithm is working and it asks the algorithm to calculate a needed value. This abstraction is very helpful because we must install implementation of a new algorithm there and change a small part of code.

So, if we change functions responsible for management of the TCB structure, we can dramatically change behaviour of the protocol. If we consider the example net in the Figure 1, we can see that the *segproc* function in the code segment is used to change the value of the marker which models the transmission control block. The segment processing function is the essential part of the protocol, and a good organization of this function is necessary. For example, the same function in the Linux operating system (kernel version 2.0.34) consists of about 600 lines of very sophisticated code. To make our task of the model modification easier we divided the segment processing algorithm into a number of stages. Each stage is implemented as a separate variable and consists of:

- A predicate function which defines if the segment must be processed at this stage;

- A processing function which defines how the protocol must process the segment at this stage;
- A response function which defines what segment must be transmitted in response (for example, if the protocol receives an incorrect segment, sometimes the protocol must send a specially formatted segment in response);
- A function which defines the next stage to which the segment must be passed.

The main benefits of such an implementation of the segment processing facility are: we can easily write a general algorithm of the segment processing and we get a code which is more suitable for modification. It can be noted that the initial segment processing stage (page *Preprocess*) use such a type of the segment processing. We just define an appropriate stage definition but use the same algorithm.

As an example of the model modification we can consider a number of steps to model the ARTCP congestion control algorithm.

First, we need an implementation of the ARTCP algorithm, which includes an implementation of the data structures which the algorithm uses and functions used to drive them. Since the algorithm uses TCP segment options, we must also add declaration of these options. Also we have to define functions which are used as an interface between generic flow control structure and our ARTCP algorithm implementation. For example, we must declare a function which define the amount of data the protocol allow to send.

Second, we must redefine the generic flow control structure in the way that it would use our ARTCP implementation. It includes "inserting" ARTCP congestion control structure into the generic flow control structure.

Third, we need to construct a scheduling facility to transmit segments with the rate specified by the ARTCP algorithm. This aspect is modelled with the *Scheduler* page and this was the only major addition to our CPN structure. The *Scheduler* page can also be useful to model other congestion control algorithms which manage the flow rate.

Another possibility of tuning the model is to create connections with different working algorithms. It is possible to enable or disable timestamps, window scaling (see [3] for details on algorithms), selective acknowledgements (see [4]). Also it is possible to study performance behaviour of a system consisting of several ARTCP-flows and several ordinary TCP-flows.

4 An Example of Model Analysis

In this section we present some examples of how our model can be analysed. For the analysis we used Design/CPN tool (ver 4.0.5). It has several built-in tool which are very useful: CP-net simulator which is used to simulate CP-nets, Occurrence Graph tool which is used to build state spaces of a model, Chart tool and some others.

The scheme of the network structure used in the analysis is illustrated in Figure 3. For the analysis we have constructed subnets to model data links and

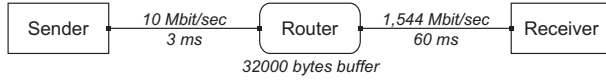


Fig. 3. Network structure used for the analysis

a simple router. We consider that links are error-free and the router has a finite-space buffer of 32000 bytes. The maximum segment size (MSS) of *Sender* and *Receiver* is equal to 1000 bytes.

Let us first consider an example of a discovered deadlock in the TCP protocol. The protocol has the recommended algorithm to define the amount of data to send, described in [2]. According to this algorithm, TCP can send data if:

- the maximum segment size can be sent;
- data are pushed and all queued data can be sent;
- at least a fraction of the maximum window can be sent (the recommended value of the fraction is $1/2$);
- data are pushed and the override time-out occurs.

Since the override time-out was not considered anywhere in the TCP standard documents, we do not implement it into our model. However, it does not affect the example below, since we assume that data are not pushed. The *Sender* tries to transfer 30000 bytes of data to the *Receiver*. The *Receiver* has a buffer where incoming data are stored. The capacity of the buffer at the receiving side is 16384 bytes. The receiving window is equal to it before data transmission is started. The window indicates an allowed number of bytes that the *Sender* may transmit before receiving further permission. The user process at the receiving side makes two calls to receive the data. Each call requests 16300 bytes of data. Data transfer process of the example is shown in Figure 4. The *Sender* will stop

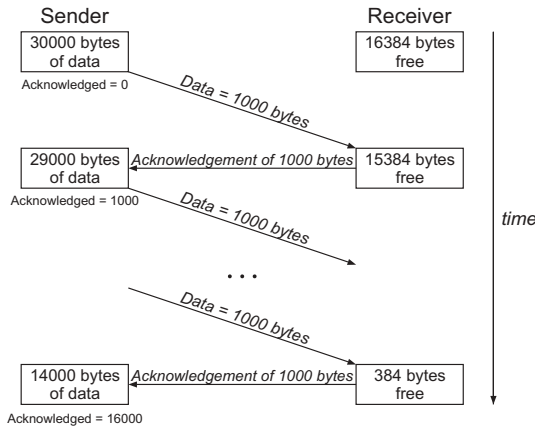


Fig. 4. Scheme of the deadlock

data transfer process after transferring 16 full-sized segments to the *Receiver*, since none of conditions to define the amount of data to be transferred is fulfilled. The maximum amount of data can not be transferred, since the *Receiver* has not enough space in the window to accept it (the window fraction can not be sent for the same reason). Conditions which deal with pushed data are not fulfilled, since we do not push data. This deadlock was discovered by building the state space in the Design/CPN occurrence graph tool. We propose to avoid this deadlock by imposing the following condition: the amount of data to send is equal to the remote window if all sent data are acknowledged, the remote window is less than MSS, the amount of buffered data is more than the remote window or equal to it.

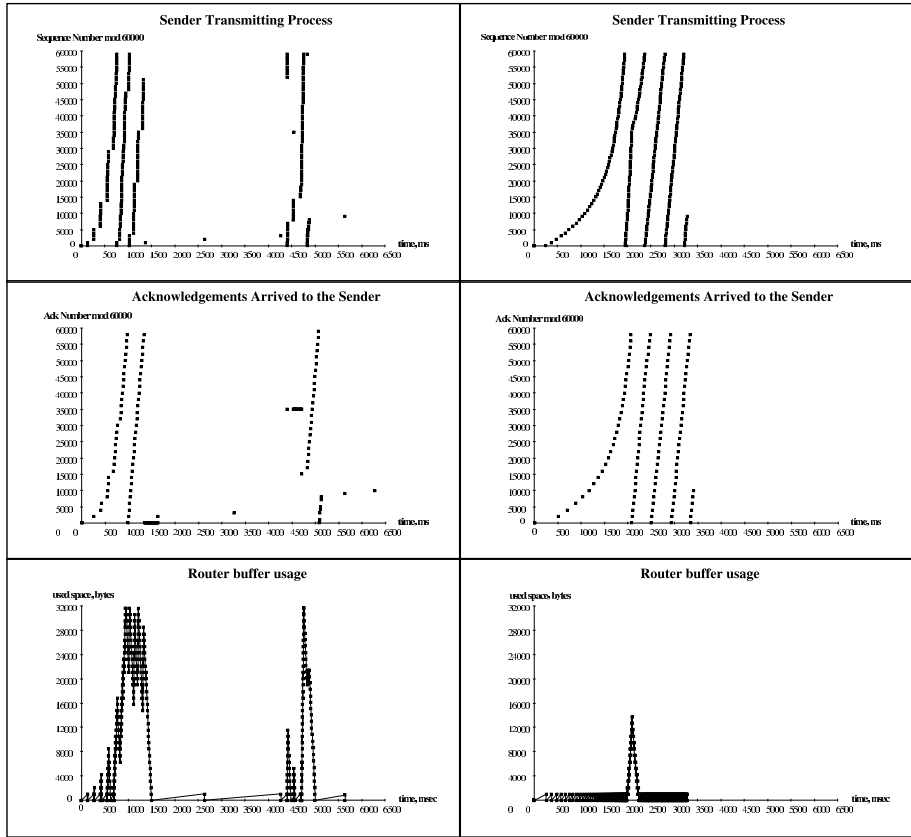


Fig. 5. Performance charts of the TCP protocol (*left*) and the ARTCP protocol (*right*)

Different kinds of performance measurements can be investigated by a simulation of our model. For simulations we used the Design/CPN simulator tool and the built-in chart facility for presentation of the results. To compare the TCP

protocol and the ARTCP protocol, we considered the same network structure as in the previous example, but in this case the *Sender* tries to transfer 250 000 bytes to the *Receiver*, and the *Receiver*'s buffer is 60000 bytes. The data transfer process will be completed when the *Sender* receives an appropriate acknowledgement. Figure 5 presents various kinds of measurements. The left side considers the standard TCP protocol and the right side considers the ARTCP protocol. The first pair of pictures shows how the *Sender* transmits segments into the network. Boxes which are not filled represent retransmission of segments predicted to be lost. The second pair of pictures shows how the *Sender* receives acknowledgements from the *Receiver*. The third pair of pictures shows the use of the router buffer space. We can see that the ARTCP protocol will complete data transfer in about approximately 3,5 seconds and the TCP protocol needs about 6,2 seconds (we consider here that there are no link errors and segments are lost only if the router buffer overflows). Also we can see that the ARTCP algorithm uses less router buffer space than the standard TCP. Thus, it is shown that our TCP/ARTCP model allows us to detect errors in the TCP protocol specification. An advantage of the ARTCP protocol over the standard TCP was also illustrated.

5 Conclusion

We have presented a timed hierarchical CPN model of the TCP protocol and have shown a way to reconfigure it into a mixed model of the TCP/ARTCP protocols. It should be noted that we model the specification of the protocol, and not an implementation. Some implementations can differ from the specification but our model can be reconfigured to represent them. The model can also be used for modelling and analysing future modifications of the TCP and the ARTCP.

Also, we have shown some examples of how the correctness and performance issues of the TCP can be investigated. Future research will be devoted to a more deep analysis of the TCP, particularly, of its modification – the ARTCP algorithm.

Our model can be used not only for the investigation of the TCP as it is, but also as a sub-model for the investigation of performance issues of the application processes which use a service provided by the TCP for communication.

In general, this approach is applicable to other transport protocols for packet switched networks.

References

1. Postel, J.: Transmission Control Protocol. RFC793 (STD7) (1981)
2. Braden, R. (ed.): Requirements for Internet Hosts – Communication Layers. RFC1122 (1989)
3. Jacobson, V., Braden, R., Borman, D.: TCP Extensions for High Performance. RFC1323 (1992)

4. Mathis, M., Mahdavi, J., Floyd, S., Romanow, A.: TCP Selective Acknowledgement Option. RFC2018 (1996)
5. Allman, M., Paxson, V., Stevens, W.: TCP Congestion Control. RFC2581 (1999)
6. Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M.: An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC2883 (2000)
7. Paxson, V., Allman, M.: Computing TCP's Retransmission Timer. RFC2988 (2000)
8. Allman, M., Balakrishnan, H., Floyd, S.: Enhancing TCP's Loss Recovery Using Limited Transmit. RFC3042 (2001)
9. Alekseev, I.V.: Adaptive Rate Control Scheme for Transport Protocol in the Packet Switched Networks. PhD Thesis. Yaroslavl State University (2000)
10. Alekseev, I.V., Sokolov, V.A.: ARTCP: Efficient Algorithm for Transport Protocol for Packet Switched Networks. In: Malyshkin, V. (ed.): Proceedings of PaCT'2001. Lecture Notes in Computer Science, Vol. 2127. Springer-Verlag (2001) 159–174
11. Alekseev, I.V., Sokolov, V.A.: Modelling and Traffic Analysis of the Adaptive Rate Transport Protocol. Future Generation Computer Systems, Number 6, Vol. 18. NH Elsevier (2002) 813–827
12. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 1. Basic Concepts. Monographs in Theoretical Computer Science. Springer-Verlag (1992)
13. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 2. Analysis Methods. Monographs in Theoretical Computer Science. Springer-Verlag (1995)
14. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 1. Practical Use. Monographs in Theoretical Computer Science. Springer-Verlag (1997)
15. Jensen, K., Rozenberg, G., (eds.): High-Level Petri Nets. Springer-Verlag (1991)
16. Christensen, S., Jørgensen, J.B., Kristensen, L.M.: Design/CPN – A Computer Tool for Coloured Petri Nets. In: Brinksma, E. (ed.): Proceedings of TACAS'97. Lecture Notes in Computer Science, Vol. 1217. Springer-Verlag (1997) 209–223
17. Coloured Petri Nets. University of Aarhus, Computer Science Department, World-Wide Web. <http://www.daimi.aau.dk/CPnets>.
18. Design/CPN Online. World-Wide Web. <http://www.daimi.au.dk/designCPN/>.
19. de Figueiredo, J.C.A., Kristensen, L.M.: Using Coloured Petri Nets to Investigate Behavioural and Performance Issues of TCP Protocols. In: Jensen, K. (ed.): Proceedings of the Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN (1999) 21–40
20. Clausen, H., Jensen, P.R.: Validation and Performance Ananlysis of Network Algorithms by Coloured Petri Nets. In: Proceedings of PNPM'93. IEEE Computer Society Press (1993) 280–289
21. Clausen, H., Jensen, P.R.: Ananlysis of Usage Parameter Control Algorithm for ATM Networks. In: Tohmè, S. and Casada, A. (eds.): Broadband Communications II (C-24). Elsevier Science Publishers (1994) 297–310
22. Fall, K., Floyd, S.: Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP. Computer Communication Review, 26(3):5–21 (1996)
23. Kumar, A.: Comparative Performance Ananlysis of Versions of TCP in a Local Network with a Lossy Link. IEEE/ACM Transactions on Networking. 6(4) (1998) 485–498

Parallel Computing for Globally Optimal Decision Making★

V.P. Gergel and R.G. Strongin

Nizhni Novgorod State University
Gagarin prosp., 23, Nizhni Novgorod 603950, Russia
{gergel, strongin}@unn.ac.ru

Abstract. This paper presents a new scheme for parallel computations on cluster systems for time consuming problems of globally optimal decision making. This uniform scheme (without any centralized control processor) is based on the idea of multidimensional problem reduction. Using same new multiple mappings (of the Peano curve type), a multidimensional problem is reduced to a family of univariate problems which can be solved in parallel in such a way that each of these processors shares the information obtained by the other processors.

1 Introduction

The investigation of different mathematical models in applications often involves the elaboration of estimations for the value that characterizes the given domain Q in the multidimensional Euclidean space R^N . Let us consider several typical examples of such problems.

As the first example we consider *the problem of integration* of the function $\varphi(y)$ over the domain Q , i.e. the problem of constructing the value

$$I = \int_Q \varphi(y) dy. \quad (1)$$

In some problems the domain Q can be described as an N -dimensional hyperinterval

$$D = \{y \in R^N : a_j \leq y_j \leq b_j, 1 \leq j \leq N\} \quad (2)$$

defined by the vectors $a = (a_1, \dots, a_N)$ and $b = (b_1, \dots, b_N)$. The coordinates of these vectors, satisfying the inequalities $a_j \leq b_j, 1 \leq j \leq N$, give the borders of the values for the components $y_j, 1 \leq j \leq N$, of the vector $y = (y_1, \dots, y_N)$. In more complicated cases the domain Q can be described as a set of points from D , satisfying the given system of *constraints-inequalities*

★ Supported in part by the Intel Research Grant "Parallel Computing on Multiprocessor and Multi-computer Systems for Globally Optimal Decision Making"

$$g_i(y) \leq 0, \quad 1 \leq i \leq m. \quad (3)$$

In this case the domain Q can be represented in the form:

$$Q = \{y \in D : g_i(y) \leq 0, 1 \leq i \leq m\}. \quad (4)$$

The second example is the problem of finding the point $y^* \in Q$ which is *the solution of the system of nonlinear equations*

$$q_i(y) = 0, \quad 1 \leq i \leq N, \quad (5)$$

where the domain Q is usually defined either in the form (2) or as (4).

The last example represents *the problem of nonlinear programming*, i.e. the problem of minimizing the function $\varphi(y)$ over the domain Q , which is denoted as

$$\varphi^* = \varphi(y^*) = \min\{\varphi(y) : y \in Q\}. \quad (6)$$

In this problem we consider the pair $(y^*, \varphi^* = \varphi(y^*))$, including the minimal value φ^* of the function $\varphi(y)$ over Q and the coordinate y^* of this value as a solution which is a characteristic of the domain Q . In the general case the function $\varphi(y)$ can have more than one minimum and (6) is called *the multiextremal or global optimization problem*.

The above examples (some more examples can be given) demonstrate *the existence of a wide class of important applied problems*, which require estimating a value (an integral, a global minimum, a set of nondominated solutions, etc.) by means of analyzing the behavior of the given vector-function

$$F(y) = (F_1(y), \dots, F_s(y)) \quad (7)$$

over the hyperinterval D from (2). The components of the vector-function (7) have different interpretation in the examples considered above. So, for instance, in the integration problem (1) over the domain Q from (4) they include both the integrated function $\varphi(y)$ and the left-hand sides of the constraints $g_i(y)$, $1 \leq i \leq m$; in the problem of searching for the solution of the system of non-linear equations (5) they describe both the left-hand sides $q_i(y)$, $1 \leq i \leq N$, of these equations and the above-mentioned left-hand sides of the constraints $g_i(y)$, $1 \leq i \leq m$ (if the search of the solution is executed in the domain Q from (4), but not in the whole space R^N), etc.

2 Finding Globally Optimal Solutions by Grid Computations

The next important question related to the class of problems of constructing estimations for multidimensional domains considered above concerns the manner in which the vector-function (7) is given in applications. As a rule, the researcher controls the operation which permits to calculate values of this function at chosen

points $y \in D$. It means that the problem of obtaining the sought estimation may be solved by analysing the set of vector values

$$Z^t = F(y^t), \quad y^t \in D, \quad 1 \leq t \leq T, \quad (8)$$

computed at the nodes of the grid

$$Y_T = \{y^t : 1 \leq t \leq T\}, \quad (9)$$

embedded into the hyperinterval D .

We consider this possibility for the problem (6) characterized by the vector-function (7) of the type

$$F(y) = (g_1(y), \dots, g_m(y), g_{m+1}(y) = \varphi(y)), \quad y \in D. \quad (10)$$

It is evident that knowing the set of values (8), we can construct the estimation

$$\tilde{\varphi}_T = \min \{ \varphi(y^t) : 1 \leq t \leq T, g_i(y^t) \leq 0, 1 \leq i \leq m \} \geq \varphi^*, \quad (11)$$

where φ^* is from (6). Here it is important to answer how the value (11) correlates with the sought value φ^* . In different applications the answer to this question is obligatory. Such problems are, in particular, the estimation of different risks under some conditions. For instance, if $\varphi(y)$ is a revenue under the conditions y , then the value (6) corresponds to the guaranteed income over the set of feasible variants (i.e. of variants from the set Q). In this last case the estimation of the value

$$\delta_T = \tilde{\varphi}_T - \varphi^*, \quad (12)$$

permits to determine the extent of overestimating the guaranteed revenue which is the implication of analyzing procedure based on comparison of some variants from the set (8). Similarly we may speak about the guaranteed durability (or about carrying capacity, etc.) over the set of variants.

A possible way to improve the estimation (11) is determined by the scheme of choosing the grid nodes (9) based on taking into account the following reason. In a wide set of problems arising in applications the limited variation Δy of the argument y generates restricted variations $\Delta F_i(y)$, $1 \leq i \leq s = m+1$ of the coordinate functions from (7), that is a consequence of limitations on energy changes inherent to real systems to be modelled. Certainly, real systems can generate resonance phenomena and various forms of impact actions which should be described as discontinuities of corresponding characteristics. In the case when characteristics reflected by the vector function $F(y)$ are not discontinuous the above-mentioned fact of the presence of restrictions for variations of the functions $F_i(y)$, $1 \leq i \leq s$, under limited variations of the argument y may be described, for example, by *Lipschitz condition*

$$|F_i(y'') - F_i(y')| \leq L_i \|y'' - y'\|, \quad y', y'' \in D, \quad 1 \leq i \leq s. \quad (13)$$

Suppose that the functionals of the problem (6) satisfy the conditions (13) and consider the case $Q = D$. Then, according to (3), (4) and (10), $m = 0$ and $F(y) = \varphi(y)$. Let $\Delta = \delta / L_1$, where L_1 is a Lipschitz constant corresponding to the function $\varphi(y)$ and $\delta > 0$ is a given real number. Let us construct a set $Y_T \in D$, possessing the property

$$(\forall y \in D) (\exists y' \in Y_T) \|y' - y\| \leq \Delta \quad (14)$$

and, hence, being a Δ -grid in D . Then, for the estimation (11) attained by means of comparison of values for the function $\varphi(y)$ at the nodes of the grid Y_T from (14) the following inequalities hold

$$\varphi^* \leq \tilde{\varphi}_T \leq \varphi^* + L_1 \Delta = \varphi^* + \delta. \quad (15)$$

From (15) we have $\delta_T \leq \delta$ for the value δ_T from (12). Therefore,

$$\lim_{\delta \rightarrow 0} \tilde{\varphi}_T = \varphi^*$$

and applying the scanning of all variants which are determined by the nodes of the grid Y_T from (14), we can achieve any given accuracy $\delta > 0$.

But the number of nodes of the Δ -grid (14), obviously, grows while the step Δ decreases, i.e. $T = T(\Delta)$ and $Y_T = Y_T(\Delta)$. Also the number of nodes of this grid grows exponentially when the dimension N of the domain D is increased. Indeed, for the uniform grid in the domain D from (2) with the step $\Delta > 0$, the estimation

$$T = T(\Delta) \approx \prod_{j=1}^N [(b_j - a_j) / \Delta]$$

is valid.

As a result, in different actual applied problems the implementation of the described uniform grid scheme becomes impossible because of extremely high requirements to computational resources. The main proposal aimed at increasing the efficiency of the analysis of complex problems is related to adopting a purposeful analysis of variants. In the course of this analysis, calculations made for some nodes of the grid permit to do without calculations for many other nodes. Thus, we are dealing with the use of essentially non-uniform grids. For instance, in the case of the above global minimization problem (6) it means that the grid should be more dense in the vicinity of the desired global minimum and noticeably less dense at some distance from the minimum point. In fact, it means that such a grid cannot be specified a priori (since the location of the global minimum is not known).

3 Improving the Computation Efficiency by Applying Non-uniform Grids

The main idea of reducing the volume of computations needed for estimating the characteristics of the function to be minimized consists in using irregular grids that

offer the same solution accuracy as the uniform ones. In the irregular case several initial nodes $y^1, \dots, y^t$, $t \geq 1$, of the grid (9) may be given a priori, however, all the remaining nodes are successively determined by a decision rule

$$y^{k+1} = G_k(y^1, \dots, y^k; Z^1, \dots, Z^k), \quad k \geq t, \quad (16)$$

where the values Z^l , $1 \leq l \leq k$, being arguments of the decision function G_k are the values introduced in (8) of the vector-function $F(y)$ from (7), computed at the nodes y^l , $1 \leq l \leq k$. Every concrete system of decision functions suggested for a certain problem class is based on some a priori assumptions regarding the vector-function $F(y)$ (see, for examples, [1-2,4,9]). This permits to use for making the non-uniform grids the information

$$\omega_k = \{y^1, \dots, y^k; Z^1, \dots, Z^k\}$$

accumulated during the process of problem investigation. Let us give an example of solving the problems of this type (6).

Example. Consider a class of two-dimensional test functions [4]

$$\varphi(y) = \left\{ \left(\sum_{i=1}^7 \sum_{j=1}^7 [A_{ij} a_{ij}(y) + B_{ij} b_{ij}(y)] \right)^2 + \left(\sum_{i=1}^7 \sum_{j=1}^7 [C_{ij} a_{ij}(y) + D_{ij} b_{ij}(y)] \right)^2 \right\}^{1/2}, \quad (17)$$

where

$$a_{ij}(y) = \sin(i\pi y_1) \sin(j\pi y_2), \quad b_{ij}(y) = \cos(i\pi y_1) \cos(j\pi y_2), \quad 0 \leq y_1, y_2 \leq 1,$$

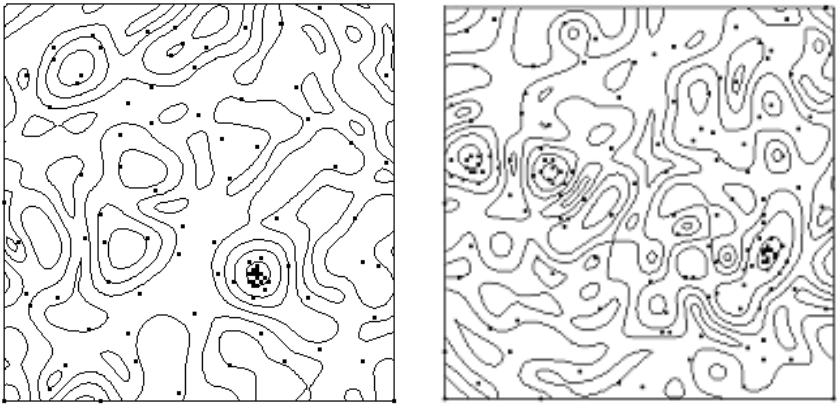


Fig. 1. Level curves of the test functions belonging to the set (17). The points of iterations executed in accordance with the decision rules of the multidimensional algorithm of the global search are marked as dark spots

and the values of the coefficients $A_{ij}, B_{ij}, C_{ij}, D_{ij}$ are chosen as realizations of the random variable uniformly distributed over the interval $[-1, 1]$. The problem's similarity to many applied problems has led to a sufficiently wide use of functions of this class for evaluating the global optimization algorithms developed. The level curves of two concrete functions are shown in Fig. 1.

The minimization of both functions taken with the reversed sign (for seeking the global maxima of functions), was carried out by the multidimensional algorithm of global search using maps of Peano type curves for the reduction of problem dimensionality (see, for instance, [4,9]). The points of the square D in which calculations of the minimized function values were executed (100 iterations for the function shown in the left half of the figure and 124 iterations for the second one) are marked in the figure by dark spots. In these examples the actual accuracy of problem solution exceeds the accuracy of the uniform grid method on the grid with the step of 0.01 (for every coordinate), for which 10^4 iterations are required. Thus, the use of a non-uniform grid speeds up the computations (in the example to be considered) almost by two orders of magnitude.

Concrete problems arising in applications are usually multidimensional. This fact makes it more difficult to generate efficient non-uniform grids by means of rules of sequential choice of the nodes using the information accumulated during the process of solving the problem. The matter is that the rules (16) realize an optimal choice by means of solving an auxiliary optimization problem. Since the chosen point y^{k+1} belongs to the multidimensional hyperinterval D from (2), the implementation of the rule (16) leads to solving a multidimensional optimization problem that becomes more difficult after every new trial (because of the increasing number of the values being arguments of the function G_k from (16)). Thus, the problem of choice of a current trial point becomes the problem like (6), solved in every step of the process and becoming more complicated from step to step.

A possible way to overcome these difficulties is to reduce multidimensional problems given over the hyperinterval D from (2) to some equivalent one-dimensional problems. The main idea of using such reductions is connected with the possibility to elaborate some efficient methods for optimization of complicated one-dimensional functions. In this approach, it is important that with such a reduction to a one-dimensional problem defined over an interval of the real axis x some property like Lipschitz condition (13) is kept. Remind that exactly this property allows to estimate the sought values (integrals, minima, etc.) by means of computations of vector-function values (9) at the nodes of a grid.

A possible scheme of such a reduction consists in the following (see, for example, [4,9]). Introduce a "standard" hypercube D^* with the edge of the length 1:

$$D^* = \{w \in R^N : -2^{-1} \leq w_j \leq 2^{-1}, 1 \leq j \leq N\} \quad (18)$$

and reduce the problem given over the hyperinterval to the problem given over the standard cube D^* from (18). This reduction is realized by a simple linear transformation of coordinates (with identical stretching coefficient for all coordinate axes):

$$w_j = (y_j - (a_j + b_j)/2)/\rho, \quad 1 \leq j \leq N, \quad (19)$$

where

$$\rho = \max \{b_j - a_j : 1 \leq j \leq N\}.$$

Moreover, introduce a supplementary constraint of the type

$$g_0(w) = \max \{|w_j| - (b_j - a_j)/2\rho : 1 \leq j \leq N\} \leq 0, \quad (20)$$

which enables to determine if the image (for the mapping being inverse to (19)) of the point $w \in D^*$ belongs to the hyperinterval D .

The next step consists in using the continuous single-valued correspondence $w(x)$ of *Peano type* curve for mapping the interval $[0,1]$ of the real axis x onto the standard hypercube D^* from (18). Then

$$D^* = \{w(x) : x \in [0,1]\},$$

and as the mapping $w(x)$ is continuous, we have, for example,

$$\min \{\varphi(w) : w \in D^*, g_i(w) \leq 0, 1 \leq i \leq m\} = \min \{\varphi(w(x)) : x \in [0,1], g_i(w(x)) \leq 0\}.$$

In a similar manner, other multidimensional problems considered above can be reduced to the equivalent one-dimensional problems.

Numerical methods for approximating Peano curves (with any given accuracy) and constructive methods of inverse mappings (being multiple) are described and substantiated in [4,5,9]. The latter work also contains C^{++} programs.

4 Parallel Computations in Globally Optimum Decision Making on Clusters

Multiprocessor clusters [3] allow to speed up solving problems by means of simultaneous computation of values (8) of the vector-function (7) at several nodes of the grid (9) to be used. In this situation each of $p > 1$ processors operates with a concrete point $y^l \in D, 1 \leq l \leq p$, for which this processor (using necessary software) determines the value of all or a part of all coordinate functions $F_i(y^l), 1 \leq i \leq s$. The possibility to confine oneself to the computation of only some part of coordinate functions can be connected, for example, with the fact that, according to (4), the violation of any inequality (3) identifies the node y^l given to the l -th processor as infeasible, i.e. $y^l \notin Q$.

In this case, the iteration at the point y^l may be implemented by determining successively the values of coordinate functions $F_i(y^l)$ and the computations at the point y^l terminate after first discovering a violated inequality from (3). In this

connection, the results of computations at the node $y^l \in D$ may be represented by a triad

$$\omega^l = (y^l, Z^l = (F_1(y^l), \dots, F_v(y^l)), v = v(y^l)), \quad 1 \leq v \leq s, \quad (21)$$

where the integer $v = v(y^l)$ is called *an index* of the point y^l . Such partial computations of values of coordinate functions take place, for example, when the index algorithms suggested in [4,6,9] for solving problems (6) are used. The computation of the triad (21) at the node y^l is called *a trial* at the point y^l and the triad (21) is called *the result of a trial* at the point y^l .

Write the information ω_k accumulated as an outcome of k trials in the form

$$\omega_k = \{\omega^1, \dots, \omega^k\} = \{y^1, \dots, y^k; Z^1, \dots, Z^k; v(y^1), \dots, v(y^k)\}, \quad k \geq 1. \quad (22)$$

It has to be noted that the scheme of reduction to one-dimensional problems described above makes it possible to build a unified database ω_k , in which instead of trial points $y^l \in D^*$ their pre-images $x^l \in [0,1]$ corresponding to mapping $w(x)$ are used. In this case for every triad a real index x^l is formed. It allows to order all triads in accordance with the values of this index. The insertion of new triads is implemented in such a manner that the ordering by index is retained. A possible option of database maintenance is described in [7]. It is important to note that the ordering of all the data in a one-dimensional scale simplifies the description and implementation of decision rules for the choice of trial points. The rules determine a new node $x^{k+1} \in [0,1]$, which is then mapped to the point $y^{k+1} = w(x^{k+1})$.

As it was mentioned above, we consider the problems for which the determination of values of the vector-function (7) is based on a numerical analysis of complex mathematical models demanding considerable computer resources. For such models characterized by sufficiently long time of trial realization, parallelizing computations by means of simultaneous execution of these computations on different processors is quite a substantiated approach. Since, as it was discussed above, the grids for which the nodes are generated in the course of solving the problem by sequential algorithms with the decision rules

$$y^{k+1} = G_k(\omega_k), \quad k \geq t, \quad (23)$$

can require essentially (by many orders of magnitude) less computations than the uniform grids, it is reasonable to parallelize trials at the nodes of grids produced by efficient sequential methods.

It is reasonable to organize solving the problem directly over the domain D , using corresponding generalizations of efficient sequential schemes for simultaneous generation (on the base of the information ω_k from (22)) several trial points in this region, i.e., the decision rule has to generate simultaneously $p > 1$ points of the following iterations

$$(y^{k+1}, \dots, y^{k+p}) = G_{k,p}(\omega_k), \quad k \geq t, \quad y^{k+l} \in D, 1 \leq l \leq p, \quad (24)$$

transmitted to individual processors for obtaining results (21). In [4,8-9] algorithms with decision rules (24) generalizing efficient sequential schemes (i.e., the schemes realizing fast compression of the grid in the vicinities of solution points of problems (6)) and characterized by low redundancy are suggested.

It has to be noted that the scheme (24) determines the points of the next p trials after obtaining the results of all the preceding trials and some processors will stand idle waiting for termination of functioning the other processors. This imperfection can be overcome by introducing the following asynchronous scheme.

Denote as $y^{k+1}(j)$ the point at which the processor with the number $j, 1 \leq j \leq p$, executes $(k+1)$ -th trial, where $k = k(j), 1 \leq j \leq p$. For compactness of description, introduce also a unified enumeration for all the points of the hyperinterval D at which the trials have already been completed. Using upper indices (like what we have done in (21) and (22)), define the set

$$Y_\theta = \{y^1, \dots, y^\theta\} = \bigcup_{j=1}^p \bigcup_{i=1}^{k(j)} y^k(j),$$

at the points of which the triads $\omega^l, 1 \leq l \leq \theta$, from (22) are given. These triads constitute the array of the results

$$\omega_\theta = \{\omega^l : 1 \leq l \leq \theta\}, \quad \theta = \sum_{j=1}^p k(j), \quad (25)$$

obtained by all the processors after θ trials terminated.

In this connection the choice of the point $y^{k+1}(j), k = k(j)$, of the next trial which has to be implemented on the j -th processor, may be done with the use of the information ω_θ from (25). Additionally, we may also take into account the information on the points $y^{k+1}(i), k = k(i)$, at which other processors ($1 \leq i \leq p, i \neq j$) execute trials. Denote the set of these points as

$$Y_\theta(j) = \bigcup_{i=1, i \neq j}^p y^{k(i)+1}(i). \quad (26)$$

As a result, the decision rule of the j -th processor may be designed in the form of the function

$$y^{k+1}(j) = G_{\theta,j}(\omega_\theta, Y_\theta(j)), \quad k = k(j), \quad 1 \leq j \leq p. \quad (27)$$

Note that according to the described scheme the information on the number of trials implemented by other processors is analyzed by the j -th processor at the moment after completion of the last trial. Therefore, by virtue of the assumption that the moments of choosing trial points are not synchronized for different processors, the values θ fixed by these processors at the same moment are not identical (i.e.,

$\theta = \theta(j)$). In fact, the suggested scheme (27) interprets the results of trials executed by other processors as the results obtained by the given processor (compare with the pure sequential rule (23)).

Some algorithms implementing parallelization schemes with decision rules (27) for solving problems (6) and problems of reconstruction of resulting dependences are adduced in [5,9]. The proposed approach to the development of such methods may be generalized to the case of parallelization of efficient sequential algorithms suggested in [9] for solving problems (5).

We can organize parallel trials in accordance with the scheme (27) so that every processor implements this rule of trial choice independently of the other ones. At the same time every processor uses the common information array ω_θ from (25) and the set (26), thus assuming interprocessor information exchange. Every processor after completing a trial is meant to send a message including the triad (21) obtained to all other processors. Moreover, after choosing a point for the new trial the processor informs of this choice all other processors. In the presence of the above-mentioned exchanges *every* processor generates a solution of the initial problem over the entire domain D (or Q). Consequently, the proposed scheme does not include any specially allocated (*leading, controlling*) processor that increases the reliability of functioning during possible hardware or software failures and in case of switching a group of processors over for serving other clients or other needs of a computational system. In the latter case (i.e., when the number p of the processors is decreased) there is no need to send any special information regarding this change of the value p . The fact of change of the value p may be brought to light due to the analysis of information required to form the sets (25), (26). It means that the value p can be variable.

Example. To demonstrate the efficiency of the parallel scheme proposed above let us give a more complicated example of a 5-dimensional problem with 5 constraints (see [9]). The minimized function in the example is defined by the expression:

$$\varphi(w) = \sin(xz) - (yv + zu) \cos(xy),$$

where the vector of parameters $w = (x, y, z, u, v)$ belongs to the hyperinterval:

$$-3 \leq x, y, z \leq 3, \quad -10 \leq u, v \leq 10,$$

and the constraints are as follows

$$g_1(w) = -(x + y + z + u + v) \leq 0,$$

$$g_2(w) = (y/3)^2 + (u/10)^2 - 1.4 \leq 0,$$

$$g_3(w) = 3 - (x+1)^2 - (y+2)^2 - (z-2)^2 - (v+5)^2 \leq 0,$$

$$g_4(w) = 4x^2 \sin x + y^2 \cos(y+u) + z^2 [\sin(z+v) + \sin(10(z-u)/3)] - 4 \leq 0,$$

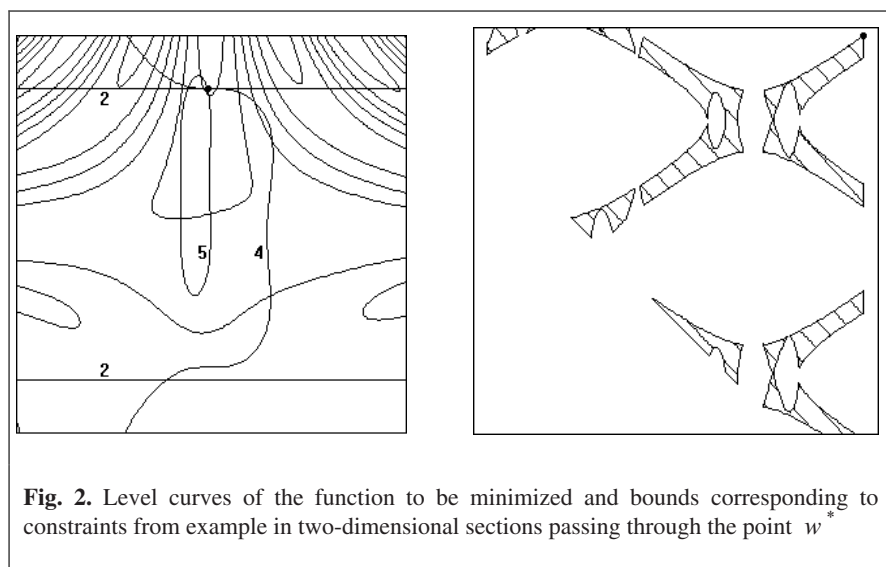


Fig. 2. Level curves of the function to be minimized and bounds corresponding to constraints from example in two-dimensional sections passing through the point w^*

$$g_5(w) = x^2 + y^2 [\sin((x+u)/3 + 6.6) + \sin((y+v)/2 + 9) + 0.9]^2 - 17 \cos^2(z+x+1) + 16 \leq 0.$$

The uniform grid (14) with the step of 0.01 (for every coordinate) contains about $8 \cdot 10^{14}$ nodes. As an experiment, the considered problem was solved by the method of scanning all nodes of this grid. Several tens of processors were used. For speeding up the computations, in the beginning of iteration at every node the simplest constraints were checked on (note that the violation of any constraint eliminates the node from the set of feasible points). Moreover, simple constraints were explicitly taken into account by reducing the part of the domain (2) in which the grid nodes were placed. As a result, the estimation

$$w' = (-0.06, 2.20, 2.40, 9.28, 9.63),$$

was found to which the value $\varphi(w') = -43.22$ corresponds. The local refinement of this estimation (with the accuracy of 0.0001 for every coordinate) gave some lesser value $\varphi^* = -43.2601$ achieved at the point

$$w^* = (0.0521, 2.2041, 2.3911, 9.2747, 9.6389).$$

As an illustration, Fig. 2 presents 2 two-dimensional sections of the domain D passing through the point w^* . The image of this point is marked as a dark circle in all subfigures. Level curves of the function minimized are also shown in every part of the Fig. 2.

The left-hand figure corresponds to the xy -section, i.e. it is an image of the square containing all points

$$w = (x, y, 2.391, 9.275, 9.639), \quad -3 \leq x, y \leq 3.$$

In this figure digits 2, 4, 5 are placed near the lines corresponding to the bounds of constraints with the same numbers (each digit is located in the part of the domain whose points satisfy the corresponding constraint). The section for the coordinate pair uv represented in the right-side subfigure is made in a different way: all the lines out of the section of the feasible domain are erased and it permits to demonstrate more visually the problem complexity.

The considered example has been also solved by the multidimensional index method that generated a non-uniform grid. The total number of iterations was equal to 59697. The found estimation

$$w'' = (-0.068, 1.962, 3.431, 9.833, 9.833)$$

is characterized by the value $\varphi(w'') = -42.992$, which was improved (after local refinement of the solution) to the value $\varphi(w^{**}) = -43.2985$ and, therefore, is not worse than the estimation obtained by the uniform grid method. Note that in our example the economy of iterations (due to applying irregular grids) constitutes many orders of magnitude.

5 Conclusions

The following results of this paper should be pointed out:

- An important class of time-consuming problems that can be effectively solved on clusters has been formed;
- Effective parallel numerical methods of globally optimum decision making have been presented. These methods are based on the fundamental idea of multidimensional problems reduction by using mappings of a Peano curve type;
- A new approach to parallel computing has been developed (without any centralized control). This approach is notable for high scalability and reliability. An implementation scheme has been also constructed (main computational procedures, information streams, modes of control synchronization).

References

1. Kushner H. J.: A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. Transactions of ASME, Ser. D. Journal of Basic Engineering 86, 1 (1964) 97–106
2. Piyavskij S. A.: An Algorithm for Finding the Absolute Minimum of a Function. Optimum Decision Theory 2. Inst. of Cybern. of the Acad. of Sci. of the Ukr. SSR, Kiev (1967) 13–24 (In Russian.)
3. Pfister G. P.: In Search of Clusters. Prentice Hall PTR, Upper Saddle River, NJ (1998)
4. Strongin R. G.: Numerical Methods in Multi-Extremal Problems (Information-Statistical Algorithms). Nauka, Moscow (1978) (In Russian.)

5. Strongin R. G.: Algorithms for Multi-extremal Mathematical Programming Problems Employing the Set of Joint Space-Filling Curves. *Journal of Global Optimization* **2**, (1992) 357–378
6. Strongin R. G., Markin D. L.: Minimization of Multiextremal Functions with Nonconvex Constraints. *Cybernetics* 16(4), (1986) 64–69 (In Russian.)
7. Strongin R.G., Gergel V. P.: On Realization of the Generalized Multidimensional Global Search Algorithm on a Computer. In: *Problems of Cybernetics. Stochastic Search in Optimization Problems*. Scientific Council of Academy of Sciences of USSR for Cybernetics, Moscow (1978) 59–66 (In Russian.)
8. Strongin R.G.: Sergeyev Ya.D.: Global Multidimensional Optimization on Parallel Computer. *Parallel Computing* **13** (1992) 1259–1273
9. Strongin R.G., Sergeyev Ya.D.: *Global Optimization with Non-convex Constraints: Sequential and Parallel Algorithms*. Kluwer Academic Publishers, Dordrecht (2000)

Parallelization of Alternating Direction Implicit Methods for Three-Dimensional Domains^{*}

V.P. Il'in, S.A. Litvinenko, and V.M. Sveshnikov

Computational physics, Institute of Computational Mathematics
and Mathematical Geophysics SB RAS,
Lavrentyev av., 6, 630090, Novosibirsk, Russia,
`ilin@comcen.nsk.su`

Abstract. The method of domain decomposition using a three-dimensional analogue of the Peaceman-Rachford algorithm is considered. In the proposed method, solution of a three-dimensional boundary value problem is performed with the use of iterations, each including two half-steps: at the first step “two-dimensional” problems are solved by the classical Peaceman-Rachford method in the planes perpendicular to the axis z , at the second – “one-dimensional” problems are solved on the lines parallel to the axis z . For the solution of one-dimensional problems, the block method of the even/odd reductions without “back step” is used, which is readily parallelized. Estimations of parallelization efficiency and results of numerical experiments on computers RM-600-E 30 (Siemens-Nixdorf) and MVS1000 (“Scientific Research Institute” Kvant - Moscow) for different grid domains and processor topologies are presented.

1 Introduction

The domain decomposition methods are an effective tool of parallelization in the solution of multi-dimensional boundary value problems by grid algorithms of finite differences, finite elements or finite volumes (see, for example [1] and the literature cited there). The essence of these methods is in partitioning the original domain into several subdomains with a subsequent solution of auxiliary problems in each of them on different processors with periodic data exchanges between them. In this case the computing complexity of algorithms as a whole increases, as it is required to carry out additional iterations in subdomains. In addition, inter-processor exchanges essentially contribute to the process of problem solution. In this connection, studying parallelization based on decomposition methods applying fast iterative Peaceman-Rachford methods, which are realized without increase of the whole volume of computation.

This paper deals with experimental estimations of the efficiency of parallelization of a three-dimensional analogue of the Peaceman-Rachford method. In a two-dimensional case, a similar problem was discussed in [1], [2].

In the proposed method, solution of a three-dimensional boundary value problem is performed with the use of iterations, each including two half-steps: at

^{*} The work was supported by the RFBR, N 01-01-00819 and N 02-01-01176

the first step “two-dimensional” problems are solved in the planes perpendicular to the axis z , at the second – “one-dimensional” problems are solved on the lines parallel to the axis z . Let us call these iterations external as opposed to the internal iterations which are performed by the classical Peaceman-Rachford method of solving alternating direction “one-dimensional” problems. The most efficient solution of the latter is with the use of the sweep method which is commonly accepted in sequential calculations. However the sweep directly is not efficiently parallelized.

In this connection, for the solution of one-dimensional problems, the block method of the even/odd reductions without “back step” is used, which is readily parallelized.

When carrying out the numerical experiment on multi-processor computers we are to do the following major actions in terms of computer costs needed for the solution of the problem (we suppose that all real values and arithmetics are presented with double precision):

- arithmetic-logical operations,
- the transmission of information between processors,
- initialization of the transmission.

If τ_a stands for the average time of carrying out one arithmetic operation with a floating point, and τ_c denotes the time of transmission of one number with double precision (requiring 8 bytes), τ_0 – the time of initialization of exchange, then the total CPU time of the synchronized multiprocessor computer system is estimated as

$$T = N_a\tau_a + N_i(\tau_0 + \tau_c N_c).$$

Here N_a is the number of arithmetic operations, implemented on one processor, N_i is the number of data exchanges and N_c is the average volume of one data transfer. Because $\tau_0 \geq \tau_c \geq \tau_a$, we must minimize the exchange number and try to superpose the data transfer and calculations in time.

In Section 2 we present the statement of the problem and the description of a three-dimensional analogue of the Peaceman-Rachford method. Section 3 presents the results of numerical experiments on computers RM-600-E 30 (Siemens-Nixdorf) and MVS-1000 (Scientific Research Institute “Kvant” - Moscow) for various grid domains and a different number of processors, their comparative analysis and discussion.

2 Statement of the Problem. Description of the Algorithm

2.1 Statement of the Problem

Assume it is required to find the function $\bar{u}(x, y, z)$, which is the solution of the boundary value problem

$$\begin{aligned} \mathcal{L}\bar{u}(x, y, z) &= g(x, y, z), & (x, y, z) &\in G, \\ \ell\bar{u}(x, y, z) &= 0, & (x, y, z) &\in \Gamma, \end{aligned} \tag{1}$$

where G is a computational domain, for simplicity taken as parallelepiped, Γ is its boundary ($G \cup \Gamma = \bar{G}$), $\mathcal{L}$ is elliptic differential operator, ℓ is operator of the boundary conditions. Nonstationary problems are also discussed here. Then in (1), $\mathcal{L}$ is a parabolic operator, ℓ is operator of boundary and initial conditions, and the sought for function $\bar{u}$ depends not only on spatial coordinates, but on time as well. Construct in $\bar{G}$ the parallelepiped grid Ω^h , formed by the intersection of the planes, perpendicular to the coordinate lines. On this grid, let us approximate problem (1) by the finite difference, finite element, or finite volume method [3]. As a result we obtain the system of seven-point (grid) linear algebraic equations:

$$\begin{aligned} & -a_{i,j,k}^1 u_{i-1,j,k} - a_{i,j,k}^2 u_{i+1,j,k} - a_{i,j,k}^3 u_{i,j-1,k} - a_{i,j,k}^4 u_{i,j+1,k} \\ & - a_{i,j,k}^5 u_{i,j,k-1} - a_{i,j,k}^6 u_{i,j,k+1} + a_{i,j,k}^0 u_{i,j,k} = f_{i,j,k}, \end{aligned} \quad (2)$$

$$i = 1, 2, \dots, I; \quad j = 1, 2, \dots, J; \quad k = 1, 2, \dots, K,$$

where $u_{i,j,k}$ is an approximate value of the sought for function $\bar{u}(x_i, y_j, z_k)$ at the grid nodes, $a_{i,j,k}^m$, $m = 0, 1, \dots, 6$ and $f_{i,j,k}$ are known values, I, J, K is the number of grid nodes along the coordinates x, y, z , respectively. Let us write down (2) in the matrix form

$$Au = f,$$

where A is a square matrix of order $I \times J \times K$, and $u = \{u_{i,j,k}\}$, $f = \{f_{i,j,k}\}$ are the sought for and the specified vectors, respectively.

In case of nonstationary problems, when using implicit in time approximations, the systems of the form (2) should be solved at each time step, and estimations of the parallelization efficiency, considered below, remain valid. We will not dwell here on the parallelization of explicit methods for parabolic differential equations, because they are not associated with solution to algebraic systems and have their own problems in providing stability and minimization of communication losses.

With the characteristic current sizes of a grid of several millions (or tens and even hundreds millions) nodes, the application of direct method of solving the band systems of the form (2) (if only they have not a specific simple form) appears to be non-efficient due to high demands to resources the number of arithmetic operations and memory capacity are about $I^3 J^3 K$ and $I^2 J^2 K$ respectively.

The use of fast iterative algorithms with preconditioning (methods of incomplete factorization, see[5]) and acceleration by conjugate gradients or Chebyshev parameters enables us to decrease the estimations in question up to $O(IJK^{3/2})$ and $O(IJK)$, respectively.

If for solution of the high order system (2) we apply the "classical" method of domain decomposition, this will be equivalent to application of the Shwartz iterative alternating method with overlapping of neighbouring subdomains equal to the characteristic mesh size h , assuming I, J, K to be values of the same order $O(h^{-1})$. This is responsible for the number of "external" iterations of order $O(h^{-1})$, which inevitably brings about considerable extra computer costs. It seems unreasonable to use the proposed algorithm on one-processor computers.

2.2 Three-Dimensional Analogue of the Peaceman-Rachford Method

Let us consider an analogue of the Peaceman-Rachford iterative implicit method for a three-dimensional case. When solving system (2) it formally represents an algebraic problem which is in the calculation of a sequence of the vectors

$$\begin{aligned} u^{n-1/2} &= u^{n-1} - \omega_z (A_{xy} u^{n-1/2} + A_z u^{n-1} - f), \\ u^n &= u^{n-1/2} - \omega_z (A_{xy} u^{n-1/2} + A_z u^n - f), \end{aligned} \quad (3)$$

where $n = 1, 2, \dots$, ω_z – is the iterative “external” numerical parameter, A_{xy} and A_z – are matrices defined as

$$\begin{aligned} A_{xy} + A_z &= A, \\ (A_{xy} u)_{i,j,k} &= -a_{i,j,k}^1 u_{i-1,j,k} + a_{i,j,k}^{0,x} u_{i,j,k} - a_{i,j,k}^2 u_{i+1,j,k} - \\ &\quad - a_{i,j,k}^3 u_{i,j-1,k} + a_{i,j,k}^{0,y} u_{i,j,k} - a_{i,j,k}^4 u_{i,j+1,k}, \\ (A_z u)_{i,j,k} &= -a_{i,j,k}^5 u_{i,j,k-1} + a_{i,j,k}^{0,z} u_{i,j,k} - a_{i,j,k}^6 u_{i,j,k+1}, \\ a_{i,j,k}^{0,x} + a_{i,j,k}^{0,y} + a_{i,j,k}^{0,z} &= a_{i,j,k}^0. \end{aligned} \quad (4)$$

Relations (3), (4) represent the vector expression of the Peaceman-Rachford alternating direction implicit (ADI) method where each n -iteration is performed in two half-steps. Let us consider the realizations of each of them on the componentwise level in more detail. At the first half-step, K “two-dimensional” systems of order IJ are solved

$$(E + \omega_z A_{xy}) u^{n-1/2} = f^{xy}, \quad (5)$$

and at the second – IJ “one-dimensional” systems of order K ,

$$(E + \omega_z A_z) u^n = f^z. \quad (6)$$

Hereinafter E is a unit matrix, and the right-hand sides

$$f^{xy} = \{f_{i,j,k}^{xy}\}, f^z = \{f_{i,j,k}^z\}$$

are calculated as

$$\begin{aligned} f_{i,j,k}^{xy} &= \omega_z f_{i,j,k} + \omega_z a_{i,j,k}^5 u_{i,j,k-1}^{n-1} + \omega_z a_{i,j,k}^6 u_{i,j,k+1}^{n-1} + (1 - \omega_z a_{i,j,k}^{0,z}) u_{i,j,k}^{n-1}, \\ f_{i,j,k}^z &= \omega_z f_{i,j,k} + \omega_z a_{i,j,k}^1 u_{i-1,j,k}^{n-1/2} + \omega_z a_{i,j,k}^2 u_{i+1,j,k}^{n-1/2} + (1 - \omega_z a_{i,j,k}^{0,x}) u_{i,j,k}^{n-1/2} \\ &\quad + \omega_z a_{i,j,k}^3 u_{i,j-1,k}^{n-1/2} + \omega_z a_{i,j,k}^4 u_{i,j+1,k}^{n-1/2} + (1 - \omega_z a_{i,j,k}^{0,y}) u_{i,j,k}^{n-1/2}. \end{aligned}$$

“Two-dimensional” problem (5) in turn is also solved by the Peaceman-Rachford method:

$$\begin{aligned} v^{\bar{n}-1/2} &= v^{\bar{n}-1} - \omega_{xy} (A_x v^{\bar{n}-1/2} + A_y v^{\bar{n}-1} - f^{xy}), \\ v^{\bar{n}} &= v^{\bar{n}-1/2} - \omega_{xy} (A_x v^{\bar{n}-1/2} + A_y v^{\bar{n}} - f^{xy}), \\ v^0 &= u^{n-1}, \end{aligned} \quad (7)$$

where $\bar{n} = 1, 2, \dots$, is the iterative ω_{xy} “internal” numerical parameter, and A_x, A_y are three-diagonal matrices defined by the relations

$$\begin{aligned} A_x + A_y &= E + \omega_z A_{xy}, \\ (A_x u)_{i,j,k} &= -\omega_z a_{i,j,k}^1 u_{i-1,j,k} - \omega_z a_{i,j,k}^2 u_{i+1,j,k} + (1/2 + \omega_z a_{i,j,k}^{0,x}) u_{i,j,k}, \\ (A_y u)_{i,j,k} &= -\omega_z a_{i,j,k}^3 u_{i,j-1,k} - \omega_z a_{i,j,k}^4 u_{i,j+1,k} + (1/2 + \omega_z a_{i,j,k}^{0,y}) u_{i,j,k}. \end{aligned}$$

Both half-steps in algorithm (7), which realizes the first half-step of algorithm (3), require inversions of three-diagonal matrices to be performed by the sweep method.

Let us consider in more detail the second half-step of algorithm (3). The system of equations (6) represents IJ independent “one-dimensional” systems, which are solved by algorithms of the same type to be described below. Let us label them with the indices (i, j) , $i = 1, 2, \dots, I$; $j = 1, 2, \dots, J$ and write down the (i, j) -th system in the block form

$$\begin{aligned} -A_l \bar{u}_{l-1} + B_l \bar{u}_l - C_l \bar{u}_{l+1} &= \bar{f}_l, \\ l &= 1, 2, \dots, L, \quad A_1 = C_L = 0, \end{aligned} \quad (8)$$

where L is a specified integer, $\bar{u}_l$, $\bar{f}_l$ are vectors of the dimension $p = K/L$: $\bar{u}_l = \{u_m^l\}$, $\bar{f}_l = \{f_m^l\}$; $u_m^l = u_{i,j,(l-1)p+m}$, $f_m^l = f_{i,j,(l-1)p+m}^z$, $m = 1, 2, \dots, p$; A_l , B_l , C_l are square matrices of order p , B_l being three-diagonal, and A_l , C_l having one non-zero entry a_l , c_l in the upper right and in the lower left angles, respectively.

To solve system (8) we exploit the block method of even-odd reduction without “back step”, see [4]. If $L = 2^R$ (this is the case we consider in this paper), then R stages of the reduction are carried out. At r -th stage a_l , c_l are recalculated as follows:

$$a_l^{(r)} = \left(a_l \hat{b}_{p,1}^{l-} a_{l_r}^- \right)^{(r-1)}, \quad c_l^{(r)} = \left(c_l \hat{b}_{1,p}^{l+} a_{l_r}^+ \right)^{(r-1)}, \quad (9)$$

as well as entries of the matrix B_l :

$$b_{1,1}^{(r)} = (b_{1,1} - a_l \hat{b}_{p,p}^{l-} c_{l_r}^-)^{(r-1)}, \quad b_{p,p}^{(r)} = (b_{p,p} - c_l \hat{b}_{1,1}^{l+} a_{l_r}^+)^{(r-1)}, \quad (10)$$

and, finally, the right-hand side is also recalculated:

$$(f_1^l)^{(r)} = (f_1^l + a_l w_p^{l-})^{(r-1)}, \quad (f_p^l)^{(r)} = (f_p^l + c_l w_1^{l+})^{(r-1)}. \quad (11)$$

Here $l_r^{+,-} = l \pm 2^{r-1}$, $b_{t,s}^l$, $t, s=1, 2, \dots, p$ are entries of the matrix B_l , $\hat{b}_{t,s}^l$ are entries of the inverse matrix B_l^{-1} , also, the following notations are introduced:

$$w_p^l = \sum_{k=1}^p \hat{b}_{p,k}^l f_k^l, \quad w_1^l = \sum_{k=1}^p \hat{b}_{1,k}^l f_k^l.$$

With inadmissible values $l_r^- < 1$, $l_r^+ > L$ the corresponding calculations in formulas (9)-(11) are not done. Entries of the matrices B_l , B_l^{-1} and components of the right-hand side $\bar{f}_l$, not included into formulas (9)-(11), remain unchanged. At the last R -th stage of the reduction we obtain L independent subsystems

$$B_l^{(R)} \bar{u}_l = \bar{f}_l^{(R)}, \quad l = 1, 2, \dots, L, \quad (12)$$

each being solved by the sweep method.

Let us write down the system of “one-dimensional” three-point equations in the form:

$$\begin{aligned} -\alpha_i v_{i-1} + \beta_i v_i - \gamma_i v_{i+1} &= \eta_i, \\ i &= 1, 2, \dots, p, \quad \alpha_1 = \gamma_p = 0, \end{aligned}$$

where $\alpha_i, \beta_i, \gamma_i$ are known coefficients and v_i are sought for values. The sweep method for its solution is realized by the formulas

$$\begin{aligned} \theta_i &= (\beta_i - \alpha_i \delta_{i-1})^{-1}, \quad \delta_i = \gamma_i \theta_i, \quad \varkappa_i = (\eta_i + \alpha_i \varkappa_{i-1}) \theta_i, \\ i &= 1, 2, \dots, p, \quad v_i = \delta_i v_{i+1} + \varkappa_i, \quad i = p, p-1, \dots, 1. \end{aligned} \quad (13)$$

To find entries of the inverse matrix $\hat{b}_{1,p}, \hat{b}_{p,p}$ let us solve the system of equations

$$B_l v = \bar{\xi}_p \quad (14)$$

with the right-hand side $\bar{\xi}_p = (0, 0, \dots, 0, 1)'$ in relation to the unknown vector $v = \{v_i, i = 1, 2, \dots, p\}$. In this case, in (13) the volume of arithmetic operations, carried out by the formulas

$$\begin{aligned} \varkappa_1 &= \varkappa_2 = \dots = \varkappa_{p-1} = 0, \quad \varkappa_p = \theta_p, \quad v_p = \varkappa_p, \\ v_i &= \delta_i v_{i+1}, \quad i = p-1, p-2, \dots, 1. \end{aligned}$$

essentially decreases.

Finding $\hat{b}_{1,1}, \hat{b}_{p,1}$ results in the solution of (14) with the right-hand side $\bar{\xi}_p = (1, 0, \dots, 0)'$, which simplifies in (13) the calculation of $\varkappa_i$:

$$\varkappa_1 = \theta_1, \quad \varkappa_i = \alpha_i \varkappa_{i-1} \theta_i, \quad i = 1, 2, \dots, p.$$

Solving (14) with the right-hand side $\bar{f}_l$, we obtain

$$w_p^l = v_p, \quad w_1^l = v_1.$$

Each of L systems (12) is formed and solved on a separate processor. To realize formulas (9)-(11) at each step of reduction it is necessary to make an exchange between processors by values, combined in the two groups:

$$D^l = (a_l, c_l, \hat{b}_{1,1}^l, \hat{b}_{1,p}^l, \hat{b}_{p,1}^l, \hat{b}_{p,p}^l), \quad W^l = (w_1^l, w_p^l).$$

The values D^l are independent of an iteration, and W^l should be recalculated at each iteration, as they depend on $\bar{f}^l$.

To conclude this section let us note that the discussed “external” Peaceman-Rachford method with optimal values of the iterative parameter ω_z demands, for attaining sufficient accuracy, carrying out about $O(h^{-1})$ iterations, and if this method is supplemented with the Chebyshev acceleration method or the conjugate gradient method, which are readily parallelized and inessentially increase the computer costs, the number of iterations decreases up to $O(h^{-1/2})$. As for

the internal iterations (7) along the horizontal planes, the required number of iterations decreases up to $O(h^{-1/2})$ due to the improvement of the condition number of the algebraic systems to be solved, and with a supplementary use of the acceleration – up to $O(h^{-1/4})$. Thus, the total volume of computation in the proposed two - step method is of order $O(h^{-4.5})$, and with the use of accelerations at both levels – $O(h^{-3.75})$. It should be also mentioned that for the commutative the operators A_x, A_y, A_z the number of iterations can be decreased due to the use of optimal sequences of iterative parameters, however this can be done only when solving the original boundary value problems of a very special type.

3 Estimations of Parallelization Efficiency

Let us represent the grid Ω^h in the form

$$\Omega^h = \bigcup_{l=1}^L \Omega_l^h,$$

where L is the number of subgrids Ω_l^h , each having the same number of nodes IJp (I, J, p is the number of nodes along x, y, z respectively). Each subgrid is set in correspondence with one processor, which carries out solution of “two-dimensional” problems in the horizontal planes and solution of reduced “one-dimensional” problems, defined on a certain subgrid. The calculation of coefficients of the difference equations a_{ijk}^m , $m = 0, 1, \dots, 6$, on the subgrid Ω_l^h is done by the l -th processor. The time of calculation of coefficients is not included into the below-presented estimations, as its contribution is inessential at a sufficiently large number of iterations.

When carrying out calculations on l -processor at each external iteration it is required to make exchanges with neighbouring processors by values of the sought for vector, which are on extreme planes of a difference grid. The program, realizing these calculations is constructed so that calculations are done simultaneously with data exchanges.

As a whole, the solution of the original system of equations on each processor can be written down as sequence of the following steps.

1. Before the beginning of iterations, the auxiliary values for all R stages of reduction along z are calculated and stored as digital arrays:
 - θ_k, δ_k are sweep coefficients for $k = 1, 2, \dots, p$ in carrying out sweep along z ;
 - a_l, c_l are coefficients in formulas (9) – (11).
2. The first half-step of external iterations is carried out, including:
 - initialization of exchanges by the values u_{ijk} , which are on the extreme planes of a grid, with neighbouring processors;
 - solution of “two-dimensional” problems by the Peaceman-Rachford method for internal planes, not waiting for completion of exchanges;
 - solution of “two-dimensional” problems upon completion of exchanges.

3. The second half-step of external iterations is carried out, implementing:
 - R stages of reduction, on each of them the values W^l being calculated for all lines of a grid, the exchange by these values with respective processors being conducted;
 - solution to system of equations (12).

As criteria of parallelization efficiency, acceleration of computation and the efficiency coefficient are considered

$$P = T_1/T_L, \quad Q = P/L,$$

where T_L is the time of solution of the problem on L processors. Dependencies of these values on the number of processors and the number of grid nodes can be estimated on the basis of the following qualitative analysis of one external iteration, since parallelization of each of them is similar.

Realization of one external iteration on one processor (without parallelization) demands carrying out a number of arithmetic operations per one grid node, equal to

$$Y_1^1 = c_1 + c_2N + c_3,$$

where c_1 is the number of arithmetic operations for calculation of the right-hand sides f^{xy} in (7), c_2 is the number of operations for carrying out one internal iteration, N is the number of internal iterations, c_3 is the number of operations for calculation of the sweep coefficients of the form (13). These calculations are done during the time

$$T_1^1 = Y_1^1 IJK\tau_a.$$

Note that

$$T_1 = T_1^1 n_z,$$

where n_z is the number of external iterations.

Let us now evaluate the time $T_{l,L}^1$ of carrying out one external iteration on l -processor with parallelization on L processors (it should be mentioned that the number of subdomains is also equal to L and $l = 1, 2, \dots, L$) with allowance for communication losses. Note that

$$T_L = \frac{n_z}{L} \sum_{l=1}^L T_{l,L}^1.$$

Let us call “internal” the processor which one having both neighbouring processors, and we call “boundary” the processor having only one neighbour. These notions are introduced as related to a certain stage of the computational algorithm, for example, when carrying out reduction the processor can be internal on one step and boundary on another step. The total number of initializations of exchanges (or simply exchanges) on one external iteration on l -th processor is equal to

$$Z_l = Z_l^I + Z_l^R,$$

where Z_l^I is the number of exchanges needed for calculation of f^{xy} in (7), and Z_l^R is the number of exchanges made in the course of realization of formulas

(9) – (11) of the reduction method. On each internal processor 4 exchanges (2 receipts + 2 sendings), are initialized in both cases, while 2 exchanges (1 receipts + 1 sending) are initialized on the extreme processor. Then

$$Z_l^I = \begin{cases} 4 & \text{for } 1 < l < L, \\ 2 & \text{for } l = 1, L, \end{cases}$$

$$Z_l^R = \begin{cases} 4R - 2 & \text{for } R \leq l \leq L - R + 1, \\ 4R - 2(r + 1) & \text{for } l = R - r, L - R + 1 + r, \quad r = 1, 2, \dots, R - 1, \end{cases}$$

and the time needed for exchanges is equal to

$$T_{l,L}^{1,Z} = Z_l \tau_0 + (Z_l^I + 2Z_l^R) I J p \tau_c.$$

The number of arithmetic operations in the considered calculation can be expressed as

$$Y_{l,L}^1 = Y_1^1 + \sum_{r=1}^L c_r^l,$$

where c_r^l are the same as above, and c_4 is the number of operations for forming systems (12) at one stage of reduction. The time, spent for their execution, equals

$$T_{l,L}^{1,I} = Y_{l,L}^1 I J p \tau_a.$$

For the sought for time $T_{l,L}^1$ holds the estimation as follows:

$$T_{l,L}^1 \leq T_{l,L}^{1,Z} + T_{l,L}^{1,I}.$$

For the acceleration coefficients P by virtue of the latter equalities we have

$$P \leq \eta L$$

Here

$$\eta = \frac{1}{1 + \frac{c_4 R}{Y_1^1}}$$

where $c_4 = \min c_r^l$. Hence, the coefficient $\eta \leq 1$, and decreases with L increasing. Therefore, even in an ideal case, when the number of arithmetic operations is such that against their background exchanges succeed to complete, i.e., $T_{l,L}^1 = T_{l,L}^{1,I}$, we have $P \leq L$, the equality sign being reached at $L = 1$.

Consider the results of numerical experiments carried out on the solution of a model Dirichlet problem for the Laplace equation

$$\Delta u = 0, \quad u|_F = 1$$

in the cube $0 < x < 1, 0 < y < 1, 0 < z < 1$. The calculation was made on the grids $\Omega^{h,k}, k = 1, 2, \dots$ with a various number of nodes. On these grids, the original boundary value problem was approximated by the finite difference

method. The initial approximation was selected as $u^0 = 0$. The iteration process was terminated when attaining the preset number of external iterations n_z . In each external iteration, the fixed number n_{xy} of internal iterations $n_z = 100$ was carried out. The completion of the iteration process upon the fixed number of iterations and not upon reaching the specified accuracy is due to the following considerations. First, for the evaluation of the parallelization efficiency, in fact, the time of execution of one iteration is needed. Second, on a dense grid, the number of iterations can be large, which brings about unreasonable computer costs. The experiments were carried out with $n = 100$ $n_{xy} = 10$.

Let us note that when conducting calculations on one processor, the reduction in fact is not used, since the solution of "one-dimensional" systems is performed by the sweep method at all the stages of the algorithm.

The calculation results with the use of the proposed algorithm on various grids and with a different number of processors of the computer system MVS-100-E-30 are presented in Table 1.

Table 1. The Peaceman-Rachford method (RM 600)

L	Crite- rion	$I \times J \times K$		
		$16 \times 16 \times 16$	$32 \times 32 \times 32$	$64 \times 64 \times 64$
1	T_1	9.52	83.53	740.77
2	T_N	5.25	44.28	373.5
	P	1.81	1.89	1.98
	Q	0.90	0.94	0.99
	T_N	2.91	22.73	189.03
4	P	3.27	3.67	3.92
	Q	0.82	0.92	0.98
	T_N	2.25	12.62	100.72
8	P	4.24	6.61	7.35
	Q	0.53	0.83	0.92

From the results obtained we can make the following conclusions.

1. Exchanges between processors essentially increase the time of solving a problem, which is especially noticeable on the grid with a small number of nodes.
2. With an increase of the number of processors, the parallelization efficiency, as was expected, drops, which is explained by computer costs needed for parallel computations. However, with a large number of processors, attaining the coefficient $Q > 0.8$ can be considered to be a good enough result.

The results of numerical experiments on MVS-1000 of solution of the same problem are shown in Table 2. As is seen from this Table, the efficiency indices do not behave so monotonously as in the previous Table. This is, apparently, due to a large volume of cash-memory on MVS-1000, and in this connection, its increasing role in calculations on a sparse grid.

On the whole, for the problems with a sufficiently large number of nodes with different of parallelization efficiency is approximately equal to $Q \approx 0.6$, which is quite reasonable.

Table 2. The Peaceman-Rachford (MVS-1000)

L	Crite- rion	$I \times J \times K$				
		16x16x16	32x32x32	64x64x64	128x128x128	256x256x256
1	T_1	0.55	4.26	43.47	398.48	5696.10
2	T_N	0.30	4.11	34.91	352.87	4980.42
	P	1.81	1.03	1.25	1.13	1.13
	Q	0.90	0.52	0.62	0.56	0.56
4	T_N	0.18	2.07	17.19	153.37	2605.23
	P	2.97	2.05	2.53	2.60	2.19
	Q	0.74	0.51	0.63	0.65	0.55
8	T_N	0.16	0.97	9.94	78.97	1138.73
	P	3.45	4.39	4.37	5.05	5.05
	Q	0.43	0.55	0.55	0.63	0.63

To conclude, let us note that the rate of convergence of external iterations can be considerably increased with the help of the Chebyshev acceleration or the conjugate gradient method. However this should not essentially influence on the parallelization efficiency coefficients which are the main objective of our study, since the required additional calculations are a small part of computer costs needed for one iteration, and are readily parallelized.

References

1. Il'in V.P., Sveshnikov V.M. : Experimental estimations of parallelization efficiency and iterative methods. Proceedings ICM & MG, series: Numerical Mathematics, Vol. 6, Novosibirsk (1998) 58–70.
2. Il'in V.P., Sveshnikov V.M.: Estimations of parallelization efficiency of domain decomposition methods. Avtometriya, N 1 (2002) 31–39 .
3. Il'in V.P.: Numerical methods of solution of electrophysics problems. Edn. Nauka, Moscow (1985).
4. Il'in V.P.: Parallel implicit methods of alternate variables. Zh. vych. mat. & mat.fiz., Vol. 37, N 8 (1997) 899–907.
5. Il'in V.P.: Methods of finite differences and finite volumes. Edn. IM SB RAS, Novosibirsk (2000).

Interval Approach to Parallel Timed Systems Verification

Yuri G. Karpov and Dmitry Sotnikov

St.Petersburg State Polytechnical University
Karpov@dcn.nord.nw.ru, DSotnikov@yandex.ru

Abstract. In this paper we present interval approach to hybrid system verification. When studying a system behavior we suggest keeping track of the minimum and maximum possible value of each continuous variable as well as the possible differences between variable pairs. We demonstrate that the approach is natural for parallel timed systems, use it for temporal logic verification and parameter analysis.

1 Introduction

With the pervasion of computer control almost in all spheres of modern life verification of those systems became a task of ever-growing importance. In majority of cases these systems are hybrid (include both digital discrete controllers and continuous environment), parallel (several components act simultaneously) and time-dependent.

For a system of interacting hybrid automata (at least timed automata) the dominant issue in verification is the computational complexity of verification algorithms because of the state explosion that we experience when dealing with concurrent systems. The direct approach to verification problem based on reachability analysis is impossible since its state space is infinite [2]. In some cases, composing the system into a single hybrid automaton, and then converting it into a so-called region graph by splitting the continuous space into finite number of equivalent regions can overcome this. However this problem is PSPACE complete-the number of states in the resultant finite region graph is enormous.

In this paper we present an efficient algorithm for reachability analysis and verification of a set of parallel timed automata that requires much less computation in comparing to region graph construction. The algorithm constructs equivalent classes of global state space of a parallel composition of timed automata by associating additional information that comprised interrelations of all active timers of the system with each global state.

The structure of the paper is as follows. Section 2 introduces a sample parallel timed system and sample TCTL formulas to be verified for it. Section 3 provides a brief overview of the traditional region graph approach for timed system verification and verify the sample system from section 2. Section 4 is the main section of the paper. It introduces our approach to parallel system execution and verification via organizing a continuous diversity of parallel timed automata into

time intervals. We show that the approach is much more efficient and illustrative, and enables parametric analysis. Section 5 takes down the limitations that were set on verified systems in section 4 and shows that our approach can be used for systems comprised of rectangular automata. Conclusion contains brief discussion of the proposed approach.

2 Sample Parallel System

In this section we introduce the sample verification problem that we use as a through example.

2.1 Asynchronous Logical Circuit

As an example we examine an asynchronous logical circuit with feedback connectors. This sample is borrowed from [3]. The circuit shown in Fig. 1 consists of three elements: a switch E , inverting element $I(x) = \neg x$, and *Sheffer stroke* element $A(x_1, x_2, x_3) = \neg(x_1 \wedge x_2 \wedge x_3)$.

The state Q of the system at each moment of time is a vector of the outputs of the three elements: $\langle Q_1, Q_2, Q_3 \rangle$.

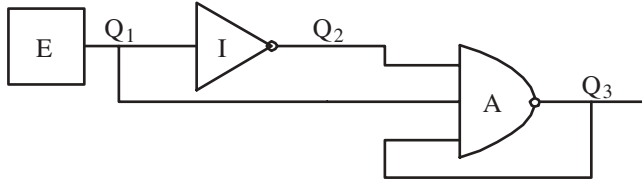


Fig. 1. A sample circuit with feedback connectors

The timed models of the system elements are shown in Fig. 2. Each element has a delay that it needs to adjust to the new inputs. The delays are shown as time intervals associated with transitions to new states. Outputs of the elements are associated with their states. The switch E changes its output to 1 exactly after 1 time unit. It takes from 4 to 5 time units for the inverter I to change its state. The Sheffer function delay is $[1, 2]$ for transition from 1 to 0 and $[2, 3]$ to the opposite direction.

Initially, the system resides in a stable state $\langle Q_1, Q_2, Q_3 \rangle = \langle 0, 1, 1 \rangle$. Then, the switch E spontaneously inverts its output Q_1 to 1. The problem is to examine properties of the system evolution to a new stable state.

For sample verification, we will try to verify two CTL formulas for this system:

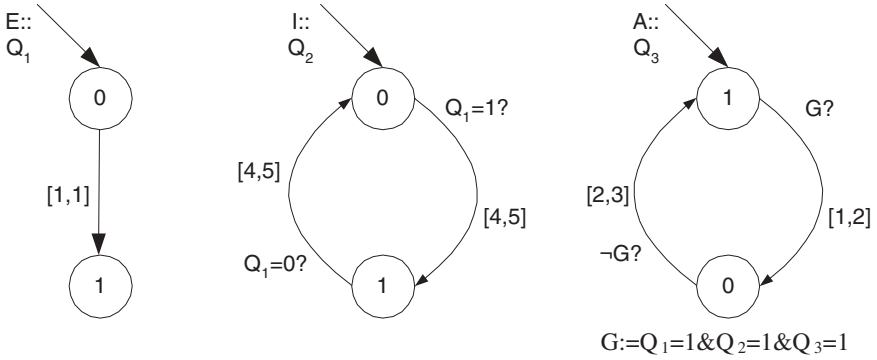


Fig. 2. Models of the elements

- $\varphi_1 = AF(Q = \langle 1, 0, 1 \rangle)$: All executions lead to the new stable state $\langle 1, 0, 1 \rangle$;
- $\varphi_2 = EG(Q_3 = 1)$: There is an execution along which $Q_3 = 1$ holds everywhere, which means that along this trajectory Q_3 does not get changed at all.

2.2 Verification without Timing Considerations

The behavior of the system Fig. 1 depends on the delays after which each of the elements responds to its new stimuli. Fig. 3 shows possible trajectories if the delays are assumed to be arbitrary (i.e. time is not taken into consideration). Asterisks (*) mark unstable states (i.e. the output of a correspondent element does not match its inputs and is about to get changed).

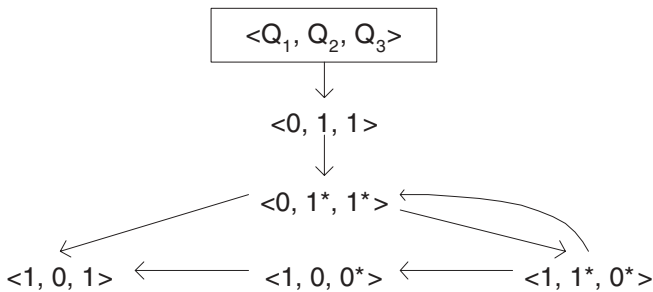


Fig. 3. Possible trajectories if timing is not considered

Let's have a look at the formulas introduced in 2.1:

- $\varphi_1 = AF(Q = \langle 1, 0, 1 \rangle)$: because there is an execution that loops infinitely between states $\langle 1, 1, 1 \rangle$ and $\langle 1, 1, 0 \rangle$, φ_1 does not hold;
- $\varphi_2 = EG(Q_3 = 1)$: the system may move along the trajectory: $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 0, 1 \rangle$. In this execution the component Q_3 does not get changed at all. So φ_2 is true.

The subsequent sections will show that neither of these results is true if time is considered.

3 Hybrid System Verification

This section gives a brief overview of the 'traditional' approach to hybrid automata verification. Its more detailed discussions are easy to find elsewhere, e.g. in [5].

One of the approaches to hybrid automata verification is solving the reachability problem for a finite automaton built for equivalent continuous space regions. In general the problem of hybrid system verification is unsolved ([4]). However, in most cases any hybrid automaton can be safely approximated by a rectangular automaton. It has been shown that rectangular automata are bisimilar to timed automata.

3.1 Definitions

A timed automaton is a hybrid automaton $H = (Q, X, Init, f, Inv, E, G, R)$, where

- Q is a set of discrete variables, $Q = \{q_i, \dots, q_m\}$;
- $X = \{x_i, \dots, x_m\}$, $X = R^n$ —a set of continuous variables;
- $Init = \{\{q_i\} \times \widehat{Init_{q_i}}\}_{i=1}^m$ where $Init_{q_i} \in \Phi(X)$ —a set of initial states;
- $f(q, x) = (1, \dots, 1)$ for all (q, x) —vector of $\dot{x}$ in each q ;
- $Inv(q) = X$ for all $q \in Q$ —state invariants;
- $E \subset Q \times Q$ —interstate transitions;
- $G(e) = \hat{G}_e$, where $G_e \in \Phi(X)$, for all $e = (q, q') \in E$ —transition guards, and
- For all e , $R(e, x)$ either leaves x_i unaffected or resets it to 0.

The set $\Phi(X)$ of clock constraints is a set of finite logical expressions defined inductively by $\delta \in \Phi(X)$ if $\delta := (x_i \leq c) \mid (x_i \geq c) \mid \neg \delta_1 \mid \delta_1 \wedge \delta_2$, where $\delta_1, \delta_2 \in \Phi(X)$, $x_i \in X$ and $c \geq 0$ is a rational number.

3.2 Automata Parallel Composition

For analysis purposes, automata constituting a system are normally composed into a single hybrid automaton. Their variable spaces get merged, their locations multiplied, synchronous transitions are merged, and interleaving transitions co-exist. Thus, to verify a system of hybrid automata we just need to verify a hybrid automaton that is a result of parallel composition of the system components.

3.3 Region Graphs

For finite automata the verification task is always solvable because of the finite state-space of the result of their composition. For a hybrid automaton this direct approach is impossible since its state space is infinite. Composing the system into a single hybrid automaton, and then converting it into a so-called region graph by splitting the continuous space into multiple equivalent regions can overcome this.

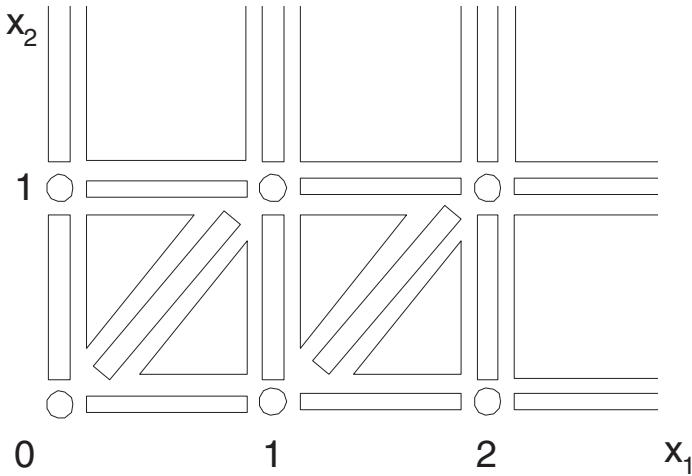


Fig. 4. Splitting continuous space into regions

It has been proved that for a single-rate clock automaton with integer constraints the continuous space can be split into equivalent regions as shown in Fig. 4 (in this particular figure the state space is constituted by two variables x_1 and x_2 . For x_1 the biggest constant to which it is ever compared is 2, for x_2 it is 1.)

3.4 Verification

Once an equivalent (or at least approximating) finite automaton is constructed the verification is straightforward. Unfortunately this approach scales badly. This problem is PSPACE complete: the number of states in the resultant finite automaton has up to $m(n!)(2^n) \prod_{i=1}^n (2c_i + 2)$ discrete states (m - the number of discrete states in timed automaton, n - the number of continuous variables, c_i - the largest constant with which x_i is compared).

If verification of temporal properties requires introducing additional clocks the state space grows even more as all components of the $m(n!)(2^n) \prod_{i=1}^n (2c_i + 2)$ product increase.

3.5 Sample Verification

Let us use this approach to verify the system specified in 2.1. The timed automaton for the system is shown in Fig. 5.

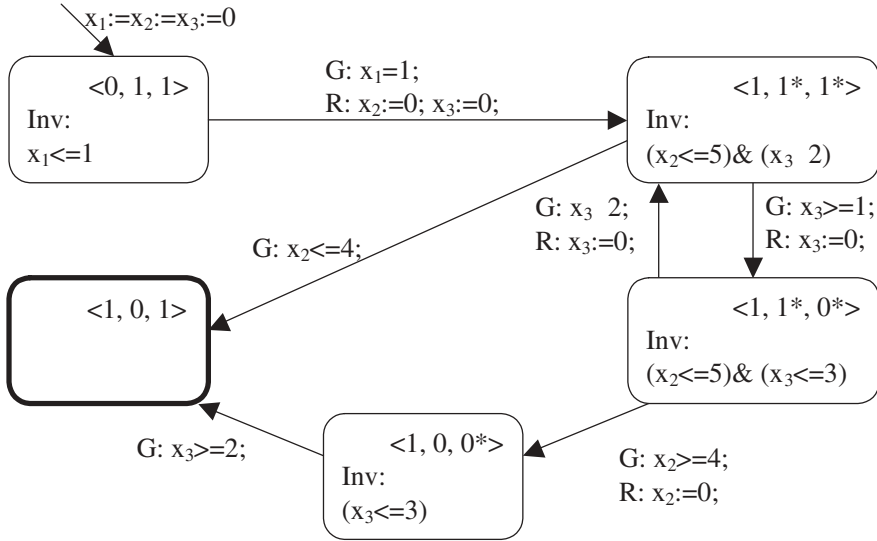


Fig. 5. Timed automaton

To verify the automaton we have to split each location into equivalent regions. The full state space of the automaton has states. For the automaton from Fig. 5 this gives $5 * 3! * 23 * (2 * 1 + 2) * (2 * 5 + 2) * (2 * 3 + 2) = 92,160$. However, in practice we don't have to consider all the states. Instead, we can start from the initial state $(\langle 0, 1, 1 \rangle, x_1 = x_2 = x_3 = 0)$ and draw only states reachable from it. The a part of the graph can be found in Fig. 6. The whole resultant region graph has 95 states.

Let's have a look at the formulas introduced in 2.1:

- $\varphi_1 = AF(Q = \langle 1, 0, 1 \rangle)$: The region graph has no loops, all its possible trajectories end up in $\langle 1, 0, 1 \rangle$. This proves that the system does arrive into a stable state after the switch changes its output along all its executions. So φ_1 is true.

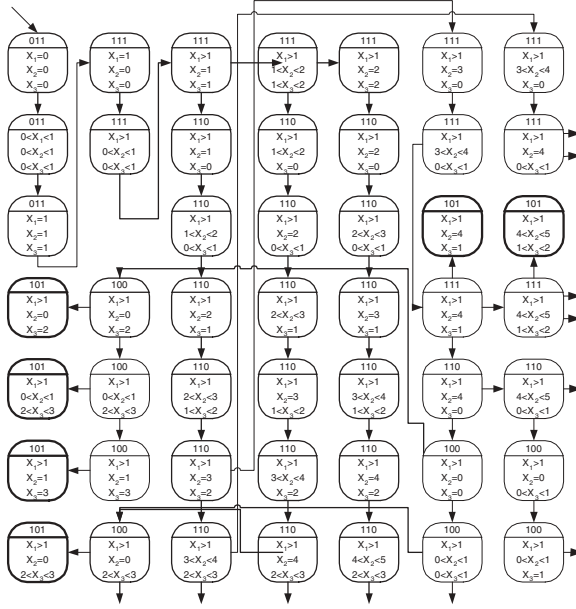


Fig. 6. Verification with Region Graph (part)

- $\varphi_2 = EG(Q_3 = 1)$: All possible trajectories pass states in which $Q_3 = 0$, trajectory $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 0, 1 \rangle$ is impossible. So Q_3 is sensible to the switch of E and φ_2 is false.

Above results are opposite to ones received in 2.2 when time was not taken into consideration.

While this approach does allow for the verification tasks we need, it is far from being efficient.

4 Parallel Interval Executions

4.1 Interval Approach

We suggest verifying a parallel timed system without preliminary explicitly combining it into a single automaton and splitting time space into regions. Here we reuse some results of [1]. For brevity we omit proves of some assertions borrowed from this paper.

The main idea of our approach is to construct a time-abstract transition system G_A for a given timed automata A by grouping those states of A together, which are equivalent according to any temporal formula. To do this we keep track of possible time differences between all clocks in A and use these inter-clock time intervals to determine which transition in A can fire next.

While the approach can be used for any timed automaton (see section 5) we will first demonstrate it on a simplified class of timed automata, which we call 'interval automata':

1. Each interval automaton has only one clock x associated with it (a continuous variable);
2. The clock rate is 1, no matter what is the current location: $f(q, x) = 1$;
3. Each transition τ has a clock condition $I_\tau = [l_\tau, u_\tau]$ (guard): $l_\tau \leq x \leq u_\tau$. The lower and upper bounds are positive reals. The lower or the upper time limit, or both are optional—the lower one may be lifted by assuming $l_\tau = 0$; the upper one - by setting $u_\tau = +\infty$. If no limits are set, $x \in [0, +\infty)$. The upper bound can be considered to be the state invariant: e.g. if Inv is $x \leq 3$ and transition condition is $x \geq 2$ the merged transition condition will be $2 \leq x \leq 3$.
4. The clock of each automaton gets reset to zero each time a transition is taken. Thus, a transition τ is taken not earlier than l_τ and not later than u_τ after the moment the automaton enters into its current state.

Above restrictions let us demonstrate the approach more clearly. With them, we can represent each interval automaton as a set of locations and delayed transitions, as a sample automaton shown in Fig. 7. In section 5 we'll show that our approach can be used for a model of traditional timed automaton without these restrictions.

Interval timed automaton in Fig. 7 has three locations: q^0 , q^1 , and q^2 and three transitions marked with a, b, c . It has only one clock x with the rate 1. The clock gets reset with each transition. Transitions a , b and c have clock constraints: $g^a : 2 \leq x \leq 3$; $g^b : 3 \leq x \leq 4$; $g^c : x \leq 5$.

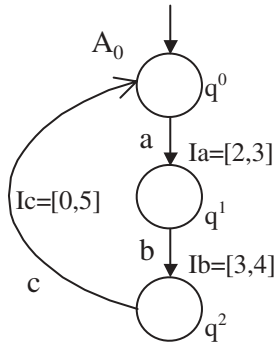


Fig. 7. Sample Timed Automaton

The usual way to define the semantics of an interval automaton A is to associate a transition system S_A with it. A state of S_A is an element of $Q \times R^+$

(R^+ is a set of nonnegative reals). In a pair (s, t) from $Q \times R^+$ s is a location of A and t is a current time value. There are two types of transitions in S_A : time transition $(s, t) \xrightarrow{\delta} (s, \tau + \delta), 0 \leq \delta \leq u_\tau$, and a location-switch transition $(s, t) \xrightarrow{\tau} (q, t)$.

An *automaton execution* is a sequence of pairs of locations and time moments when the automaton arrives at the correspondent locations, for example $\sigma = (q_0, t_0)(q_1, t_1) \dots (q_n, t_n) \in (Q \times T)^*$. Such form of string is in fact a concise representation of transitions of the system $S_A : (q_0, t_0) \xrightarrow{t_1 - t_0} (q_0, t_1) \xrightarrow{\tau_1} (q_1, t_1) \dots (q_{n-1}, t_{n-1}) \xrightarrow{t_n - t_{n-1}} (q_{n-1}, t_n) \xrightarrow{\tau_n} (q_n, t_n)$.

In any execution the automaton A_0 in Fig. 7 starts in q^0 at t_0 , then it can stay in q^0 not more than 3 but not less than 2, then in state q^1 it should stay not more than 4 but not less than 4 time units. From q^2 it can proceed immediately to q^0 and should do that no later than in 5 time units. A sample execution of A_0 with $t_0=0$ is $\sigma_1 = (q^0, 0)(q^1, 2.5)(q^2, 6.7)(q^0, 6.7)(q^1, 8.7)(q^2, 12) \dots$

The execution σ_1 of automaton A_0 is a 'point' execution in the sense that each transition is specified by the actual time value of its occurrence. In general, a time automaton can have infinite number of point executions, since each time interval has infinite number of points. Obviously transition system S_A and its point executions are useless for verification purposes because no matter how many point executions prove that the system behaves 'properly' we cannot be sure that there is no 'bad' trajectory among the infinite number of executions we have not checked.

Since time in our interval timed automaton is defined as intervals it is natural to use time intervals to specify all possible time moments when an event (a transition to a new location) can occur.

An *interval automaton execution* is $\rho = (q_0, I_0)(q_1, I_1) \dots (q_n, I_n) \in (Q \times T \times T)^*$, i.e. a set of pairs of locations and time intervals ($I_i = [t_{li}, t_{ui}]$) when the automaton arrives at the locations. A pair (q_i, I_i) denotes that an automaton could arrive to the state q_i at any moment $t_{li} \leq t \leq t_{ui}$.

For verification purposes we will study interval executions that are dense (i.e. all points in the trajectory belong to the possible automaton's point executions) and full (i.e. all possible point executions belong to that interval execution).

It is very simple to calculate interval executions for a given interval automaton:

Suppose we have an interval execution: $\rho = (q_0, I_0)(q_1, I_1) \dots (q_n, I_n)$ and an active (the one that can be executed) transition $\tau_{n,n+1}$ from q_n to q_{n+1} with associated time interval $I_\tau = [l_\tau, u_\tau]$. Then, the interval execution ρ can be appended with (q_{n+1}, I_{n+1}) , where $I_{n+1} = I_n + I_\tau = [l_n + l_\tau, u_n + u_\tau]$. The proof is pretty obvious.

All interval executions of the sample interval automaton A_0 in Fig. 7 started at $t_0 = 0$ are prefixes of the sequence:

$$(q^0, [0, 0])(q^1, [2, 3])(q^2, [5, 7])(q^0, [5, 12])(q^1, [7, 15]) \dots$$

4.2 Sets of Interval Automata

When several interval automata are combined in a system they are executed in parallel possibly interacting via handshake or setting and checking event flags. If automata A and B form a system, their parallel composition $A \times B$ is a timed automaton which has $|Q_A \times Q_B|$ locations and infinite number of states.

We can associate a transition system $S_{\mathbf{A}}$ with a timed automaton $\mathbf{A} = A_1 \times A_2 \times \dots \times A_n$ pretty much the same way we defined above for one interval automaton, but $S_{\mathbf{A}}$ is useless for verification. To verify temporal properties of $\mathbf{A} = A_1 \times A_2 \times \dots \times A_n$ we'll construct a time-abstract transition system $G_{\mathbf{A}}$ whose transitions are labeled only with the symbols in the set of transitions $Q_{A_1} \times Q_{A_2} \times \dots \times Q_{A_n}$ and nodes are labeled by names of locations of $\mathbf{A}$. Our goal is to define an equivalence relation on the state-space of $\mathbf{A}$ that groups together all states with the same locations in which every temporal formula has the same truth value. More formally, the transition relation of $G_{\mathbf{A}}$ is the relation $\Rightarrow$: for states q and q' in the transition system $S_{\mathbf{A}}$ marked with location names L and L' , and a transition label τ of $\mathbf{A}$, there exist nodes N and N' in $G_{\mathbf{A}}$, marked with L and L' , such that $N \Rightarrow^\tau N'$ holds in $G_{\mathbf{A}}$ iff there exists a state q'' in $S_{\mathbf{A}}$ and a time value $\delta \in R^+$ such that $q \xrightarrow{\delta} q'' \xrightarrow{\tau} q'$ holds in $S_{\mathbf{A}}$. If all A_i are interval automata, the number of nodes in $G_{\mathbf{A}}$ is finite.

To construct $G_{\mathbf{A}}$ we have to define all possible nodes and transitions between them.

4.3 Determining the Next Possible Event

As in the previous section we will compute trajectories of $G_{\mathbf{A}}$ iteratively. Suppose that parallel interval automata has reached some global state, how can we determine which transition may fire next?

Let's study an example in Fig. 8. In the figure, τ_1 and τ_2 are two transitions of two interval automata A and B . If we have constructed a node N of $G_{A \times B}$ with the location $\langle s^1, q^1 \rangle$, which transition will be taken next from N ?

The answer τ_1 may seem obvious. However, it is not necessarily true. Suppose A arrived at state s_1 at the moment $t_1 = 10$, and B arrived at q_1 at $t_2 = 6$. In that case, at the moment $t = 10$ automaton B should already be in the location q_2 because according to τ_2 it cannot wait for more than 4 time units after its arrival in q_1 at $t_2 = 6$, while A has to wait at least till $t = 11$ before transition τ_1 fires.

Thus, to know which transitions can fire next we should know when each of the transitions became enabled. So we have to associate with any node of $G_{A \times B}$ information about arrival time to the correspondent location of each component timed automaton. So the next state for a suffix $\dots (\langle s_0, q_1 \rangle, 6) (\langle s_1, q_1 \rangle, 10) \dots$ of a point execution of $A \times B$ may be only $(\langle s_1, q_2 \rangle, 10)$.

Note: Associating the absolute arrival time for each location is sufficient for constructing possible point executions of timed automata. However to construct the time-abstract transition system $G_{\mathbf{A}}$ we need to keep with every node of $G_{\mathbf{A}}$ time interval between all arrival events. Let e_1 be the event of entering A into

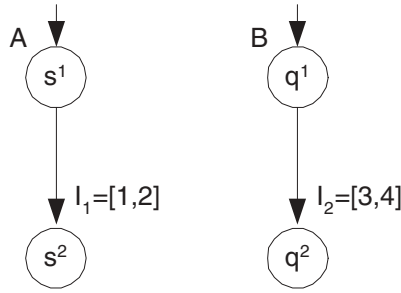


Fig. 8. Parallel execution

s_1 and e_2 be the event of entering B into q_1 . Let $I_{e_1 e_2} = [l_{e_1 e_2}, u_{e_1 e_2}]$ be the interval from event e_1 till event e_2 (thus, $l_{e_1 e_2}$ is the minimal time span that can pass from event e_1 till event e_2 , and $u_{e_1 e_2}$ is the maximal one). We can consider $I_{e_1 e_2}$ to be the relative interval of time during which event e_1 can occur if A is taken as the starting point ($t = 0$). Obviously, $I_{e_1 e_2} = -I_{e_2 e_1}$.

Suppose events e_1 and e_2 have occurred in the following time intervals: $I_{e_1} = [l_{e_1}, u_{e_1}]$ and $I_{e_2} = [l_{e_2}, u_{e_2}]$. Then, the relative time interval between events e_1 and e_2 can be calculated as: $I_{e_1 e_2} = [l_{e_2} - u_{e_1}, u_{e_2} - l_{e_1}]$.

The proof is obvious. $l_{e_1 e_2}$ is the minimal time possible from event e_1 till event e_2 . $l_{e_1 e_2}$ is the earliest time moment when event e_2 can happen, minus the latest time moment when event e_1 can happen. Considerations for $u_{e_1 e_2}$ are similar.

If we construct a node N of the transition system $G_{A \times B}$ marked with location names $\langle s_1, q_1 \rangle$, and we know relative interval $I_{e_1 e_2}$ of arriving automata A and B into their locations s_1 and q_1 , as well as transition delay intervals I_{τ_1} and I_{τ_2} , then we can answer the question: which transition may fire next? Obviously, a transition may fire before all other enabled transitions if the earliest time it can fire is not later than the latest deadline of all other transitions.

Let's formally write the statement above for a situation shown in Fig. 9, a). If intervals I_{e_1} and I_{e_2} are known, then we can calculate the condition of firing both τ_1 and τ_1 :

- Transition τ_1 may fire if $l_{e_1} + l_{\tau_1} \leq u_{e_2} + u_{\tau_2}$ or $l_{\tau_1} \leq u_{e_2} - l_{e_1} + u_{\tau_2}$ and thus, if $l_{\tau_1} \leq u_{e_1 e_2} + u_{\tau_2}$
- Transition τ_2 may fire if $l_{\tau_2} \leq u_{e_2 e_1} + u_{\tau_1}$

Note that the formulas show that information about time interval between the transition-enabling events ($I_{e_1 e_2} = [l_{e_1 e_2}, u_{e_1 e_2}]$) and time intervals for the transition delays is sufficient to determine conditions that define which transition may fire next. In Fig. 9, b) we present it as information in a node of a graph $G_{A \times B}$.

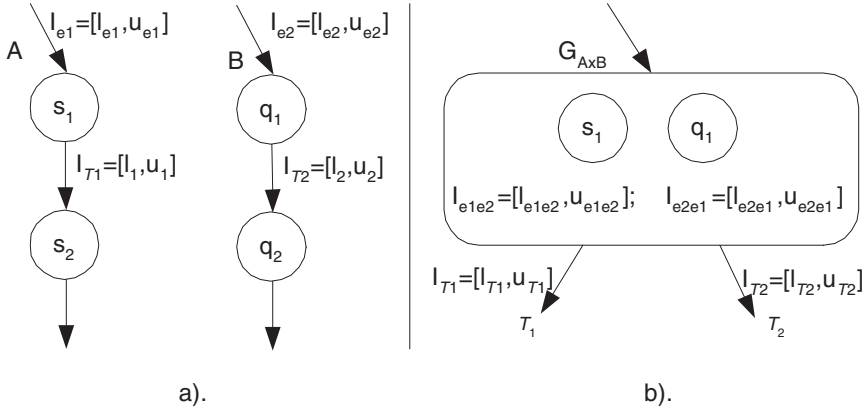


Fig. 9. Calculating the next transition

4.4 Recalculating the Inter-event Time Intervals

The inter-event intervals for all the current automata locations in a node of G_A let us to determine which transition may be taken next. Now all we need to construct G_A is to learn how to update the inter-event intervals once a transition is taken and one (or more) automaton changes its location.

Let's get back to our previous example in Fig. 9, a).

If automata A and B were in locations s_1 and q_1 with inter-event interval $I_{e_1e_2}$, and an event e_3 of firing transition $\tau_2 : q_1 \xrightarrow{\tau_2} q_2$ with the interval $I_{\tau_2} = [l_{\tau_2}, u_{\tau_2}]$ has occurred, the time interval between events e_1 and e_3 can be calculated by the following formula: $I_{e_1e_3} = [l_{e_1e_2} + l_{\tau_2}, u_{e_1e_2} + u_{\tau_2}]$.

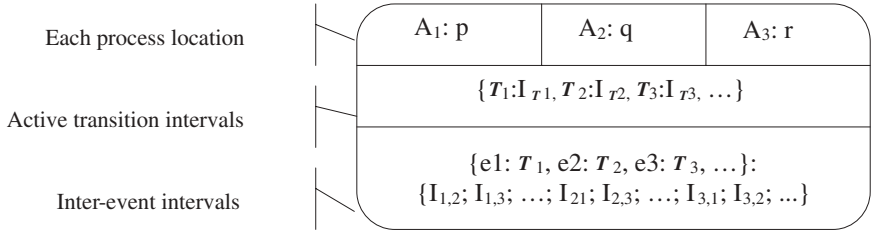
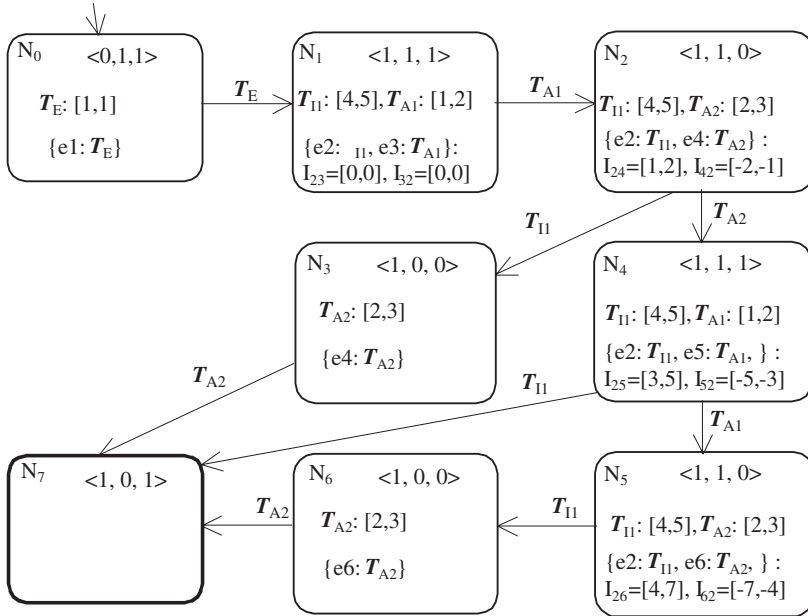
Obviously, $l_{e_1e_3}$ is the minimal span between e_1 and e_2 , plus the minimal time it takes to move from q_1 to q_2 , i.e. $l_{e_1e_3} = l_{e_1e_2} + l_{\tau_2}$. The proof for $u_{e_1e_3}$ is similar.

4.5 The Sample Verification

Return to the problem of verification of the sample system Fig. 1 using the approach suggested above. To make the verification process more illustrative we will show the following information for each node of the transition system G_A (Fig. 10).

The upper row contains current locations of all component interval automata. The second row contains all transitions enabled in this node and their intervals. The third row contains all events, enabling transitions and inter-event time intervals.

Transition τ_i may fire in this node iff for each enabling event e_j in this node, $j \neq i, l_{\tau_i} \leq u_{e_i e_j} + u_{\tau_j}$.

**Fig. 10.** A node of G_A **Fig. 11.** Transition system $G_{E \times I \times A}$

For simplicity we will show all this data in every node of G_A . In fact this is redundant because time intervals are symmetric: $I_{e_i e_j} = -I_{e_j e_i}$.

Transition system $G_{E \times I \times A}$ is presented in Fig. 11. The system starts from location $\langle 0, 1, 1 \rangle$. The only event in the starting node N_0 is its start event e_1 , enabling the only transition τ_E . From N_0 we iteratively construct next nodes by using formulas from section 4.2. Each time a transition is taken the inter-state intervals are recalculated with formulas from section 4.4.

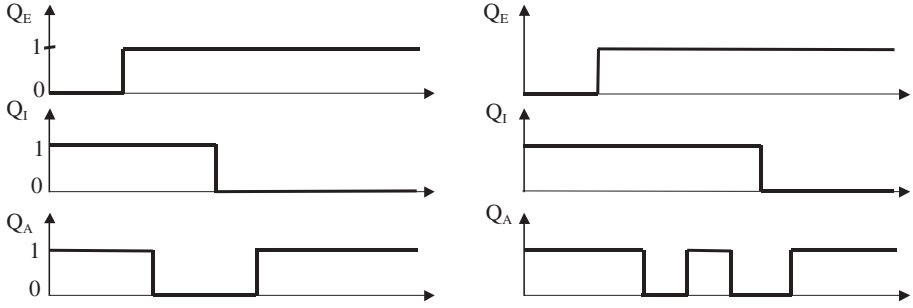


Fig. 12. Two different behaviors of $E \times I \times A$

The transition system $G_{E \times I \times A}$ lets us verify any reachability property and, more that, any property of the timed automaton $E \times I \times A$, expressed in a formula of a temporal logic.

Let's have a look at the formulas introduced in 2.1:

- $\varphi_1 = AF(Q = \langle 1, 0, 1 \rangle)$: The graph $G_{E \times I \times A}$ has no loops, all its possible trajectories end up in $\langle 1, 0, 1 \rangle$. This proves that the system does arrive at a stable state after the switch changes its output. So φ_1 is true.
- $\varphi_2 = EG(Q_3 = 1)$: All possible trajectories pass states in which $Q_3 = 0$, trajectory $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 0, 1 \rangle$ is impossible. So φ_2 is false.

The results are similar to the ones we have got examining the region graph in 3.5. Both approaches allow us to verify the timed automaton $E \times I \times A$. However the graph $G_{E \times I \times A}$ is significantly smaller (it has only 8 nodes) and easier to compute and analyze.

It is easy to see that only 3 trajectories are possible in $G_{E \times I \times A}$:

1. $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 1, 0 \rangle \rightarrow \langle 1, 0, 0 \rangle \rightarrow \langle 1, 0, 1 \rangle$
2. $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 1, 0 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 1, 0 \rangle \rightarrow \langle 1, 0, 0 \rangle \rightarrow \langle 1, 0, 1 \rangle$
3. $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 1, 0 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 0, 1 \rangle$

In Fig. 12 two time diagrams for two first above behavior trajectories of $E \times I \times A$ are presented. They show qualitative sequence of state changes of three elements in the Fig. 1 circuit.

If we do need to know the time when an event happens in the system this can easy be calculated as well, since we know the possible delays for each transition and the order in which transitions are taken. For example, for the first trajectory the time intervals for each state will be as shown in Table 1.

So the transition system gives a possibility to verify properties, expressed in Timed Temporal Logic formulas as well. Another advantage of the approach is that it is much easier to perform parameter analysis for the system. For example, we saw that $\varphi_2 = EG(Q_3 = 1)$ is evaluated to false in our system. All possible

Table 1. Time from the earliest time when the system can arrive at a state, till the latest time when the state can be left

State	Time
$\langle 0, 1, 1 \rangle$	$[0, 1]$
$\langle 1, 1, 1 \rangle$	$[1, 3]$
$\langle 1, 1, 0 \rangle$	$[2, 6]$
$\langle 1, 0, 0 \rangle$	$[5, 6]$
$\langle 1, 0, 1 \rangle$	$[5, +\infty]$

trajectories pass states in which $Q_3 = 0$. From Fig. 11 it is clear, that if $l_{\tau_2} \leq 2$ then the system can go along $\langle 0, 1, 1 \rangle \rightarrow \langle 1, 1, 1 \rangle \rightarrow \langle 1, 0, 1 \rangle$ because $l_{\tau_{I1}} \leq u_{2,3} + u_{\tau_{A1}}$. Thus, we found the parameter value ($l_{\tau_2} \leq 2$) for which φ_2 starts evaluating to **true** if other delay intervals remain unchanged.

5 Lifting Limitations

In section 4 we demonstrated the interval approach for systems composed of simplified timed automata. In this section we will show that these restrictions can be safely lifted and so the approach may be easily expand to the class of rectangular automata systems.

The limitations we used were:

1. Each interval automaton has only one clock x associated with it (a continuous variable);
2. The clock rate is 1, no matter what is the current location: $f(q, x) = 1$;
3. Each transition τ has a clock condition $I_\tau = [l_\tau, u_\tau]$ (guard): $l_\tau \leq x \leq u_\tau$. The lower and upper bounds are positive reals. The lower or the upper time limit, or both are optional—the lower one may be lifted by assuming $l_\tau = 0$; the upper one - by setting $u_\tau = +\infty$. If no limits are set, $x \in [0, +\infty)$. The upper bound can be considered to be the state invariant: e.g. if Inv is $x \leq 3$ and transition condition is $x \geq 2$ the merged transition condition will be $2 \leq x \leq 3$.
4. The clock of each automaton gets reset to zero each time a transition is taken. Thus, a transition τ is taken not earlier than l_τ and not later than u_τ after the moment the automaton enters into its current state.

Limitation 2 will obviously be the first to fall. If a clock rate is different from 1 it can be set to 1 by scaling the guard and invariant conditions.

For lifting the rest of the limitations let us briefly reconsider the essence of the approach to design a transition system G_A which represent all executions of interval automata composition $\mathbf{A} = A_1 \times A_2 \times \dots \times A_n$.

It was shown that a transition system G_A was constructed by the following rules:

1. We maintain inter-automata time intervals in each node of G_A that give the earliest and the latest time when an automaton may arrive the current state after another automaton arrived to its current state.
2. We use these intervals to determine whether one transition may precede another.
3. Each time a transition is taken and one of the automaton changes its location we recalculate the inter-automata intervals. See section 4.4.

As one can see from the formulas in section 4.3 ($l_{\tau_A} \leq u_{AB} + u_{\tau_B}$), the intervals are used simply to scale one clock comparatively to another. Once u_{AB} is added to u_{τ_B} , it has the same starting point as A and those can be compared. That is the key to lifting all other limitations of the interval approach.

Instead of maintaining intervals between the times when automata change their locations we need to calculate the intervals between the clocks' starting points. (Of course when each automaton has just one clock and it gets reset whenever a location gets changed, these two approaches are identical.)

For timed automata the interval approach should be reformulated as follows:

1. We maintain inter-clock time intervals that give the earliest and the latest time when one clock got reset last time compared to the other.
2. To decide which transition may fire next, we use the inter-clock intervals to rescale all the enabled transitions' conditions to one starting point.
 - a) We consider all invariant conditions in all locations:
 - i. Scale all clocks to a single starting point
 - ii. Calculate the latest time when the invariant starts evaluating to false
 - iii. Get the earliest latest time among the locations.
 - b) For all enabled transitions we:
 - i. Scale all clocks
 - ii. Calculate the earliest time when the transition can get executed
 - iii. If that earliest calculated time is not more than the earliest latest invariant time (calculated in 2.a.iii) this transition can get executed earlier than other transitions.
3. Each time a transition is taken the inter-clock intervals get recalculated both:
 - a) Because a clock can get reset or
 - b) Because the condition for the transition executed to true only for a subinterval of the inter-clock interval.

With this approach the limitations from 4.1 can be lifted.

Unfortunately, in general case the formulas for recalculation do get more complicated. Providing these formulas for a general case of timed automata requires further research. It seems that the interval approach could be integrated into hybrid system simulation tools (e.g. [8]) to reuse the tools' calculus capabilities to keep track of the changing inter-clock intervals.

Also, we want to note that the interval approach seems quite natural for rectangular automata (when all the possible limits and clock rates are specified by the maximal and minimal limits). It seems that rectangular automata can be

verified directly by employing the interval approach, thus not requiring doubling the number of clocks in the system as the regional approach does.

Finally, it is worth mentioning that the approach is similar to clock zones and difference-bound matrices introduced in [7]. Difference-bound matrices provide an efficient way to verify timed automata. To some extent the interval approach from this paper may be considered to be an extension to the difference-bound matrices approach as it imposes less limitations on the system.

6 Conclusion

We discussed here an approach to verification of a set of communicating timed systems. The methodology described in this paper is suitable for finding errors in time-dependant computer programs, communication protocols and asynchronous circuits. It seems that the transition system G_A for a set of timed automata suggested here is much simpler than usually constructed region graph for the parallel composition of automata. One additional advantage of the approach is that it can be used to perform parameter analysis for the system—to find values of time limits in clock constraints which guarantee desirable properties of the system.

References

1. Y. Karpov, A. Borshev "Analysis of Parallel Real-Time Programs", In "System Informatics", Vol.5, 1997, Novosibirsk (In Russian)
2. R. Alur, D. Dill "A theory of timed automata", Theoretical Computer Science, vol. 126, pp. 183–235, 1994
3. Lewis H. R. "A logic of concrete time intervals." Fifth Annu. IEEE Symp. on Logics in Computer Science. – IEEE, 1990. – P.380–389
4. T. Herzinger, P. Kopke, A. Puri and P. Varaiya, "What's decidable about hybrid automata", 27th Annual Symposium on the Theory of Computing, STOC'95, pp. 373–382, ACM Press
5. John Lygeros, Shankar Sastry, "Hybrid Systems: Modeling, Analysis & Control"
6. E. A. Emerson, E. Clarke. "Design and synthesis of synchronization skeletons using branching-time temporal logic." Proc. Workshop on Logic of Programs. Lecture Notes in Computer Science 131, Springer-Verlag, 1981
7. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, Automatic Verification Methods for Finite State Systems, LNCS 407, pp. 197–212. Springer-Verlag, 1989.
8. XJ Technologies AnyLogic, <http://www.xjtek.com/products/anylogic/>

An Approach to Assessment of Heterogeneous Parallel Algorithms

Alexey Lastovetsky and Ravi Reddy

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
{Alexey.Lastovetsky, manumachu.reddy}@ucd.ie

Abstract. The paper presents an approach to the performance analysis of heterogeneous parallel algorithms. As a typical heterogeneous parallel algorithm is just a modification of some homogeneous one, the idea is to compare the heterogeneous algorithm with its homogeneous prototype, and to assess the heterogeneous modification rather than to analyse the algorithm as an isolated entity. A criterion of optimality of heterogeneous parallel algorithms is suggested. A parallel algorithm of matrix multiplication on heterogeneous clusters is used to demonstrate the proposed approach.

1 Introduction

Heterogeneous networks of computers are a promising distributed-memory parallel architecture. In the most general case, a heterogeneous network includes PCs, workstations, multiprocessor servers, clusters of workstations, and even supercomputers. Unlike traditional homogeneous parallel platforms, the heterogeneous parallel architecture uses processors running at different speeds. Therefore, traditional parallel algorithms, which distribute computations evenly across parallel processors, will not balance the load of different-speed processors of the heterogeneous network. Faster processors will quickly perform their portions of computation and will wait for slower ones at points of synchronisation.

A natural approach to the problem is to distribute data across processors unevenly so that each processor performs the volume of computation proportional to its speed. Several authors have applied this approach to data parallel algorithms based on the two-dimensional block-cyclic distribution [1-4].

The methods of the performance analysis of homogeneous parallel algorithms are well studied. They are based on a number of models of parallel computers, including the parallel random access machine (PRAM) [5], the bulk-synchronous parallel model (BSP) [6], and the LogP model [7]. All the models assume a parallel computer to be a homogeneous multiprocessor. The PRAM is the most simplistic model. It assumes that all processors work synchronously and that interprocessor communication is free. The BSP allows processors to work asynchronously and models latency and limited bandwidth. Finally, the LogP is the most realistic model among them. It characterizes a parallel machine by the number of processors (P), the communication bandwidth (g), the communication delay (L), and the communication overhead (o). The LogP model has been successfully used for the performance analysis of parallel algorithms

for (homogeneous) supercomputers. The theoretical analysis of a homogeneous parallel algorithm is normally accompanied by a relatively small number of experiments on a homogeneous parallel computer system. The purpose of these experiments is to demonstrate that the analysis is correct, and the analysed algorithm is really faster than its counterparts.

Theoretical performance analysis of heterogeneous parallel algorithms is much more difficult task than that of homogeneous ones. While some research efforts have been made in this direction [8-9], there is no adequate and practical model of heterogeneous networks of computers yet, which would be able to predict the execution time of heterogeneous parallel algorithms with satisfactory accuracy. The problem of optimal heterogeneous data distribution has proved NP-complete even for such a simple linear algebra kernel as matrix multiplication on heterogeneous networks [4]. Therefore, most practical heterogeneous parallel algorithms are sub-optimal. A typical approach to assessment of a heterogeneous parallel algorithm is its experimental comparison with some homogeneous counterpart on one or several heterogeneous platforms. Different heterogeneous algorithms are also compared mostly experimentally. Due to the complex and irregular nature of heterogeneous networks, such experimental assessment of heterogeneous parallel algorithms is not as convincing as for homogeneous ones. One can easily argue that the demonstration of the advantage of one algorithm over other algorithm on one or several heterogeneous networks does not prove that the situation will not change if you run the algorithms on other networks of computers, with the different relative speed of processors, and the different structure and speed of the communication network.

In this paper, we present a new approach to the performance analysis of heterogeneous parallel algorithms. As a typical heterogeneous parallel algorithm is just a modification of some homogeneous one, the idea is to compare the heterogeneous algorithm with its homogeneous prototype, and to assess the heterogeneous modification rather than analyse the algorithm as an isolated entity. Namely, we propose to compare the efficiency demonstrated by the heterogeneous algorithm on a heterogeneous network with the efficiency demonstrated by its homogeneous prototype on a homogeneous network having the same aggregate performance as the heterogeneous one.

This paper is structured as follows. In Section 2, we briefly formulate our approach to assessment of heterogeneous parallel algorithm. Then we demonstrate how to apply this approach to the assessment of a concrete heterogeneous parallel algorithm. For this purpose we use an algorithm of matrix multiplication on heterogeneous networks based on the heterogeneous matrix distribution proposed in [3]. In Section 3, we describe a block cyclic algorithm of parallel matrix multiplication on homogeneous platforms. In Section 4, we introduce its heterogeneous modification. In Section 5, we assess this heterogeneous algorithm by comparing the efficiency demonstrated by this algorithm on a heterogeneous network with the efficiency demonstrated by its homogeneous prototype on a homogeneous network, which has the same aggregate performance as the heterogeneous one. We show that the heterogeneous algorithm is very close to the optimal one. In Section 6, we present some results of experiments with this application, which in particular confirm our theoretical analysis.

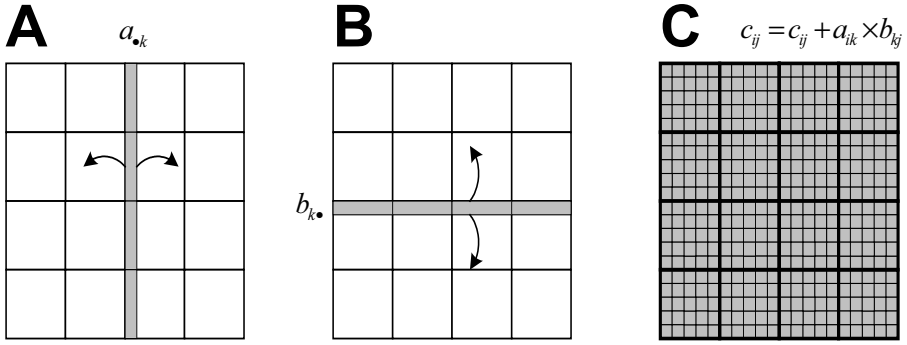


Fig. 1. One step of the algorithm of parallel matrix multiplication based on two-dimensional block distribution of matrices A, B, and C. First, the pivot column $a_{\bullet k}$ of $r \times r$ blocks of matrix A (shown shaded grey) is broadcast horizontally, and the pivot row $b_{k\bullet}$ of $r \times r$ blocks of matrix B (shown shaded grey) is broadcast vertically. Then, each $r \times r$ block c_{ij} of matrix C (also shown shaded grey) is updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$.

2 Assessment of Heterogeneous Algorithms

We propose to assess heterogeneous algorithms as follows. Typically, a heterogeneous algorithm is just a modification of some homogeneous one. Therefore, our proposal is to compare the heterogeneous algorithm with its homogeneous prototype and assess the heterogeneous modification rather than analyse the algorithm as an isolated entity.

Our basic postulate is that the heterogeneous algorithm cannot be more efficient than its homogeneous prototype. It means that the heterogeneous algorithm cannot be executed on the heterogeneous network faster than its homogeneous prototype on the *equivalent* homogeneous network. A homogeneous network of computers is equivalent to the heterogeneous network if

- Its communication characteristics are the same;
- It has the same number of processors;
- The speed of each processor is equal to the average speed of processors of the heterogeneous network.

The heterogeneous algorithm is considered *optimal* if its efficiency is the same as that of its homogeneous prototype.

3 Block Cyclic Algorithm of Parallel Matrix Multiplication on Homogeneous Platforms

Consider the following algorithm of parallel multiplication of two dense square $n \times n$ matrices A and B on a p -processor MPP:

- The A , B , and C matrices are identically partitioned into p equal $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ squares, so that each row and each column contain $\sqrt{p}$ squares (for simplicity, we assume that p is a square number and n is a multiple of $\sqrt{p}$). There is one-to-one mapping between these squares and the processors. Each processor is responsible for computing its C square (see Figure 1).
- Each element in A , B , and C is a square $r \times r$ block and the unit of computation is the updating of one block, i.e., a matrix multiplication of size r . For simplicity, we assume that $\sqrt{p}$ is a multiple of r .
- The algorithm consists of $\frac{n}{r}$ steps. At each step k ,
 - A column of blocks (the pivot column) of matrix A is communicated (broadcast) horizontally (see Figure 1);
 - A row of blocks (the pivot row) of matrix B is communicated (broadcast) vertically (see Figure 1);
 - Each processor updates each block in its C square with one block from the pivot column and one block from the pivot row, so that each block c_{ij} ($i, j \in \{1, \dots, \frac{n}{r}\}$) of matrix C will be updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ (see Figure 1).

Thus, after $\frac{n}{r}$ steps of the algorithm, each block c_{ij} of matrix C will be

$$c_{ij} = \sum_{k=1}^{\frac{n}{r}} a_{ik} \times b_{kj} ,$$

i.e., $C = A \times B$.

Consider this algorithm from the processor point-of-view. The processors of the MPP executing the algorithm are arranged into a two-dimensional $m \times m$ grid $\{P_{ij}\}$, where $m = \sqrt{p}$ and $i, j \in \{1, \dots, m\}$. At each step k of the algorithm,

- The pivot column $a_{\bullet k}$ is owned by the column of processors $\{P_{iK}\}_{i=1}^m$ and the

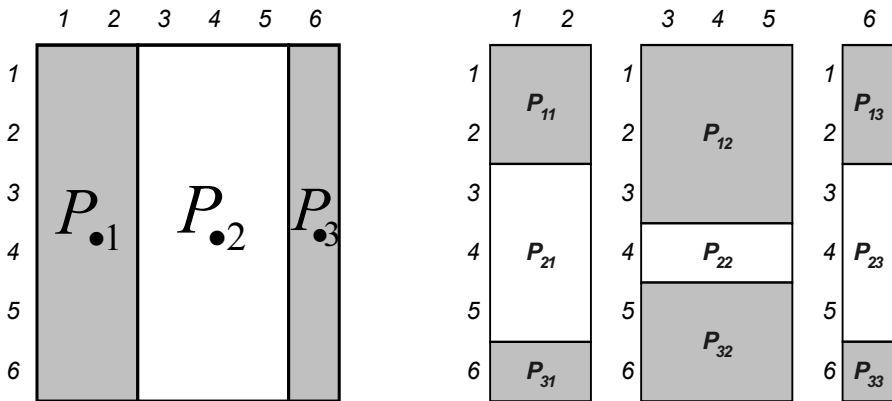
pivot row $b_{k\bullet}$ is owned by the row of processors $\{P_{Ki}\}_{i=1}^m$, where $K = \left\lceil \frac{r \times k}{\frac{n}{m}} \right\rceil$.

- Each processor P_{iK} (for all $i \in \{1, \dots, m\}$) horizontally broadcasts its part of the pivot column $a_{\bullet k}$ to processors $P_{i\bullet}$.
- Each processor P_{Kj} (for all $j \in \{1, \dots, m\}$) vertically broadcasts its part of the pivot row $b_{k\bullet}$ to processors $P_{\bullet j}$.
- Each processor P_{ij} receives the corresponding part of the pivot column and pivot row and uses them to update each $r \times r$ block of its C square.

Note that at each step k , each processor P_{ij} participates in two collective communication operations: a broadcast involving the row of processors $P_{i\bullet}$ and a broadcast involving the column of processors $P_{\bullet j}$. Processor P_{iK} is the *root* for the first broadcast, and processor P_{Kj} is the root for the second. As r is usually much less than m , in most cases at next step $k+1$ of the algorithm processor P_{iK} will be again the root of the broadcast involving the row of processors $P_{i\bullet}$, as well as processor P_{Kj} will be the root of the broadcast involving the column of processors $P_{\bullet j}$. Therefore, at step $k+1$, the broadcast involving the row of processors $P_{i\bullet}$ cannot start until processor P_{iK} completes this broadcast at step k . Similarly, the broadcast involving the column of processors $P_{\bullet j}$ cannot start until processor P_{Kj} completes that broadcast at step k . The root of the broadcast communication operation completes when its communication buffer can be reused. Typically, the completion means that the root has sent out the contents of the communication buffer to all receiving processors.

Thus, there is strong dependence between successive steps of the parallel algorithm, which hinders parallel execution of the steps. If at successive steps of the algorithm the broadcast operations involving the same set of processors had different roots, they could be executed in parallel. As a result, more communications would be executed in parallel and more computations and communications would be overlapped.

In order to break the dependence between successive steps of the algorithm, the way, in which matrices A , B and C are distributed over the processors, can be modified. The modified distribution is called a *two-dimensional block cyclic distribution* and can be summarized as follows:



(a) Partition between processor columns.

(b) Partition inside each processor column.

Fig. 2. Example of two-step distribution of a 6×6 generalized block over a 3×3 processor

grid. The relative speed of processors is given by matrix $s = \begin{pmatrix} 0.11 & 0.25 & 0.05 \\ 0.17 & 0.09 & 0.08 \\ 0.05 & 0.17 & 0.03 \end{pmatrix}$. (a)

At the first step, the 6×6 square is distributed in a one-dimensional block fashion over processors columns of the 3×3 processor grid in proportion $0.33 : 0.51 : 0.16 \approx 2 : 3 : 1$. (b) At the second step, each vertical rectangle is distributed independently in a one-dimensional block fashion over processors of its column. The first rectangle is distributed in proportion $0.11 : 0.17 : 0.05 \approx 2 : 3 : 1$. The second one is distributed in proportion $0.25 : 0.09 : 0.17 \approx 3 : 1 : 2$. The third is distributed in proportion $0.05 : 0.08 : 0.03 \approx 2 : 3 : 1$.

- Each element in A , B , and C is a square $r \times r$ block.
- The blocks are scattered in a cyclic fashion along both dimensions of the $m \times m$ processor grid, so that for all $i, j \in \{1, \dots, \frac{n}{r}\}$ blocks a_{ij}, b_{ij}, c_{ij} will be mapped to processor P_{IJ} so that $I = (i - 1) \bmod m + 1$ and $J = (j - 1) \bmod m + 1$.

The algorithm is easily generalized for an arbitrary two-dimensional processor arrangement.

The two-dimensional block cyclic distribution is a general-purpose basic decomposition in parallel dense linear algebra libraries for MPPs such as ScaLAPACK [10]. The block cyclic distribution has been also incorporated in the HPF language [11].

4 Block Cyclic Algorithm of Parallel Matrix Multiplication on Heterogeneous Platforms

In an MPP, all processors are identical. Therefore, the load of the processors will be perfectly balanced if each processor performs the same amount of work. As all $r \times r$ blocks of the C matrix require the same amount of arithmetic operations, each processor executes an amount of work, which is proportional to the number of $r \times r$ blocks that are allocated to it, and, hence, proportional to the area of its rectangle. Therefore, to equally load all processors of the MPP, a rectangle of the same area must be allocated to each processor.

In a heterogeneous cluster, processors perform computations at different speeds. To balance the load of the processors, each processor should execute an amount of work that is proportional to its speed. In case of matrix multiplication, it means that the number of $r \times r$ blocks, which are allocated to each processor, should be proportional to its speed. Let us modify the two-dimensional block cyclic distribution to satisfy the requirement.

Suppose that the relative speed of each processor P_{ij} is characterised by a real positive number, s_{ij} , so that $\sum_{i=1}^m \sum_{j=1}^m s_{ij} = 1$. Then, the area of the rectangle allocated to processor P_{ij} should be $s_{ij} \times n^2$.

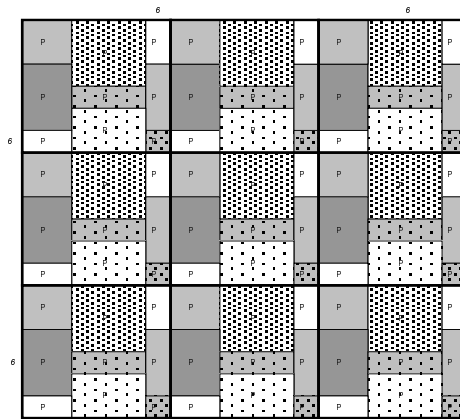
The homogeneous two-dimensional block cyclic distribution partitions the matrix into *generalized blocks* of size $(r \times m) \times (r \times m)$, each partitioned into $m \times m$ blocks of the same size $r \times r$, going to separate processors. The modified, *heterogeneous*, distribution also partitions the matrix into generalized blocks of the same size, $(r \times l) \times (r \times l)$, where $m \leq l \leq \frac{n}{r}$. The generalized blocks are identically partitioned into m^2 rectangles, each being assigned to a different processor. The main difference is that the generalized blocks are partitioned into unequal rectangles. The area of each rectangle is proportional to the speed of the processor that stores the rectangle.

The partitioning of a generalized block can be summarised as follows:

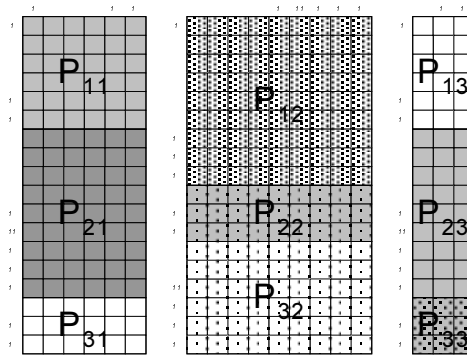
- Each element in the generalized block is a square $r \times r$ block of matrix elements. The generalized block is a $l \times l$ square of $r \times r$ blocks.
- First, the $l \times l$ square is partitioned into m vertical slices, so that the area of the j -th

slice is proportional to $\sum_{i=1}^m s_{ij}$ (see Figure 2(a)). It is supposed that blocks of the j -

th slice will be assigned to processors of the j -th column in the $m \times m$ processor grid. Thus, at this step, we balance the load *between* processor columns in the $m \times m$ processor grid, so that each processor column will store a vertical slice whose area is proportional to the total speed of its processors.



(a) Heterogeneous block cyclic distribution over 3x3 grid.



(b) Data distribution from processor point-of-view.

Fig. 3. A matrix with 18×18 blocks is distributed over a 3×3 processor grid. The relative

speed of processors is given by matrix $s = \begin{pmatrix} 0.11 & 0.25 & 0.05 \\ 0.17 & 0.09 & 0.08 \\ 0.05 & 0.17 & 0.03 \end{pmatrix}$. The numbers on the

left and on the top of the matrix represent indices of a row of blocks and a column of blocks, respectively. (a) Each labelled (shaded and unshaded) area represents different rectangles of blocks, and the label indicates at which location in the processor grid the rectangle is stored – all rectangles labelled with the same name are stored in the same processor. Each square in a bold frame represents different generalised blocks. (b) Each processor has the number of blocks approximately proportional to its relative speed,

$$\begin{pmatrix} 6 \times 6 & 9 \times 9 & 6 \times 3 \\ 9 \times 6 & 3 \times 9 & 9 \times 3 \\ 3 \times 6 & 6 \times 9 & 3 \times 3 \end{pmatrix} \approx \begin{pmatrix} 0.11 & 0.25 & 0.05 \\ 0.17 & 0.09 & 0.08 \\ 0.05 & 0.17 & 0.03 \end{pmatrix}.$$

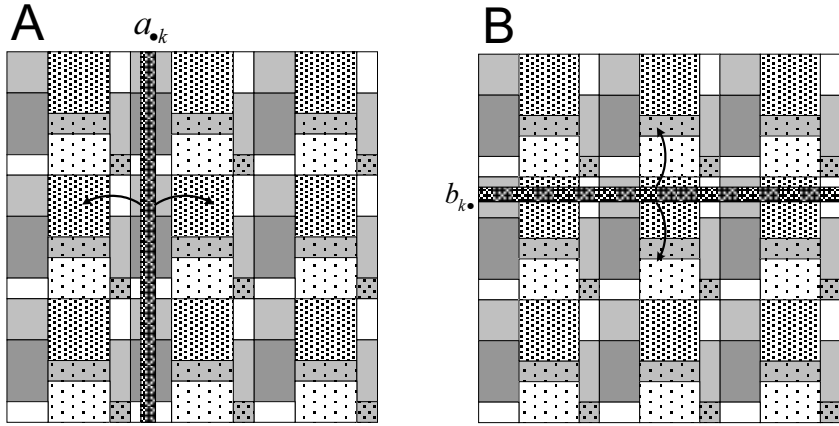


Fig. 4. One step of the algorithm of parallel matrix-matrix multiplication based on heterogeneous two-dimensional block distribution of matrices A, B, and C. First, each $r \times r$ block of the pivot column $a_{\bullet k}$ of matrix A (shown shaded dark grey) is broadcast horizontally, and each $r \times r$ block of the pivot row $b_{k\bullet}$ of matrix B (shown shaded dark grey) is broadcast vertically. Then, each $r \times r$ block c_{ij} of matrix C is updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$.

- Then, each vertical slice is partitioned independently into m horizontal slices, so that the area of the i -th horizontal slice in the j -th vertical slice is proportional to s_{ij} (see Figure 2(b)). It is supposed that blocks of the i -th horizontal slice in the j -th vertical slice will be assigned to processor P_{ij} . Thus, at this step, we balance the load of processors *within* each processor column independently.

Figure 3(a) illustrates the heterogeneous two-dimensional block cyclic distribution from the matrix point-of-view.

Figure 3(b) shows this distribution from the processor point-of-view. Each rectangle represents the total area of blocks allocated to a single processor.

Figure 4 depicts one step of the algorithm of parallel matrix-matrix multiplication on a heterogeneous $m \times m$ processor grid. Note that the total volume of communications during execution of this algorithm is exactly the same as that for a homogeneous $m \times m$ processor grid. Indeed, at each step k of both algorithms,

- Each $r \times r$ block a_{ik} of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to $m-1$ processors;
- Each $r \times r$ block b_{kj} of the pivot row of matrix B is sent vertically from the processor, which stores this block, to $m-1$ processors.

The size l of a generalized block is an additional parameter of the heterogeneous algorithm. The range of the parameter is $[m, \frac{n}{r}]$. The parameter controls two conflicting aspects of the algorithm:

- The accuracy of load balancing.
- The level of potential parallelism in execution of successive steps of the algorithm.

The greater is this parameter, the greater is the total number of $r \times r$ blocks in a generalized block, and, hence, the more accurately this number can be partitioned in a proportion given by positive real numbers. Therefore, the greater is this parameter, the better the load of processors is balanced. On the other hand, the greater is this parameter, the stronger the dependence between successive steps of the parallel algorithm is, which hinders parallel execution of the steps.

Consider two extreme cases. If $l = \frac{n}{r}$, the distribution provides the best possible balance of the load of processors. At the same time, the distribution turns into a pure two-dimensional block distribution resulting in the lowest possible level of parallel execution of successive steps of the algorithm.

If $l=m$, then the distribution is identical to the homogeneous distribution, which does not bother about load-balancing at all. At the same time, it provides the highest possible level of parallel execution of successive steps of the algorithm. Thus, the optimal value of this parameter lies in between of these two, being a result of trade-off between load-balancing and parallel execution of successive steps of the algorithm.

The algorithm is easily generalized for an arbitrary two-dimensional processor arrangement.

5 Assessment of the Heterogeneous Algorithm

Let us compare the heterogeneous algorithm presented in Section 4 with its homogeneous prototype presented in Section 3. We assume that parameters n , m and r are the same. Then, both algorithms consist of $\frac{n}{r}$ successive steps.

At each step, equivalent communication operations are performed by each of the algorithms, namely:

- Each $r \times r$ block of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to $m-1$ processors;
- Each $r \times r$ block of the pivot row of matrix B is sent vertically from the processor, which stores this block, to $m-1$ processors.

Thus, the per-step communication cost is the same for both algorithms.

If l is big enough, then at each step each processor of the heterogeneous network will perform the volume of computation approximately proportional to its speed. In this case, the per-processor computation cost will be approximately the same for both algorithms.

Thus, the per-step cost of the heterogeneous algorithm will be approximately the same as that of the homogeneous one. So the only reason for the heterogeneous algorithm to be less efficient than its homogeneous prototype is the lower level of potential overlapping of communication operations at successive steps of the algorithm. Obviously, the bigger is the ratio between the maximal and minimal

processor speed, the lower is this level. Note that if the communication layer serializes data packages (for example, plain Ethernet), then the heterogeneous algorithm has approximately the same efficiency as the homogeneous one. Therefore, in that case the presented heterogeneous algorithm is the optimal modification of its homogeneous prototype. In section 6, we present some experimental results that allow us to estimate the significance of the additional dependence between successive steps of the algorithm if the communication layer allows multiple data packages.

6 Experimental Results

This algorithm of parallel matrix multiplication on heterogeneous clusters was implemented in the mpC language [8].

This section presents some results of experiments with this application. All presented results are obtained for $r = 8$ and generalized block size $l = 9$, which have appeared optimal for both homogeneous and heterogeneous block cyclic distributions.

A small heterogeneous local network of 9 different Solaris and Linux workstations is used in the experiments presented in Figures 5 and 6. The relative speed of the workstations is as follows: 46, 46, 46, 46, 46, 46, 46, 84, and 9. We measure their relative speed with the core computation of the algorithm (updating of a matrix). The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers.

For experiments presented in Figure 7, we use the same heterogeneous network and a homogeneous local network of 9 Solaris with the following relative speeds: 46, 46, 46, 46, 46, 46, 46, 46, 46. The two sets of workstations share the same network equipment. Note that the aggregate performance of the processors of the heterogeneous network is practically the same as that of the homogeneous one.

Figure 5 shows the comparison of the execution times of 3 parallel algorithms of matrix multiplication:

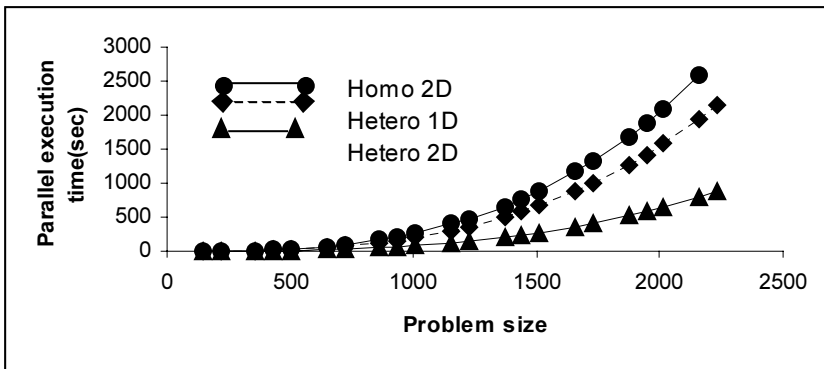


Fig. 5. Execution times of the heterogeneous and homogeneous 2D algorithms and the heterogeneous 1D algorithm. All algorithms are performed on the same heterogeneous network.

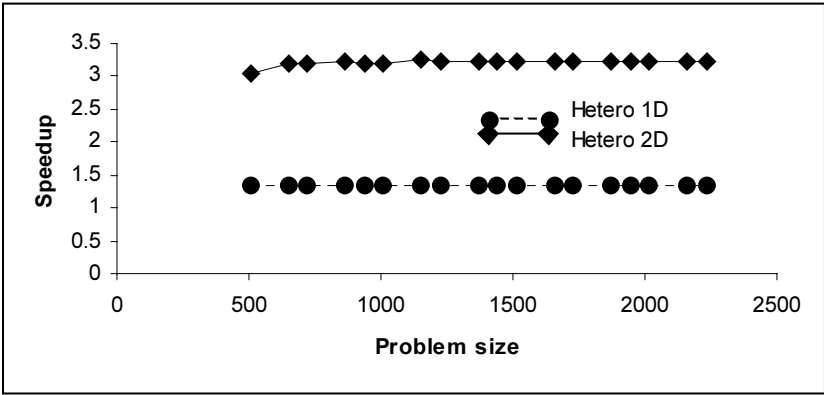


Fig. 6. The speedup of the heterogeneous 1D and 2D algorithms compared to the homogeneous 2D block cyclic algorithm. All algorithms are performed on the same heterogeneous network.

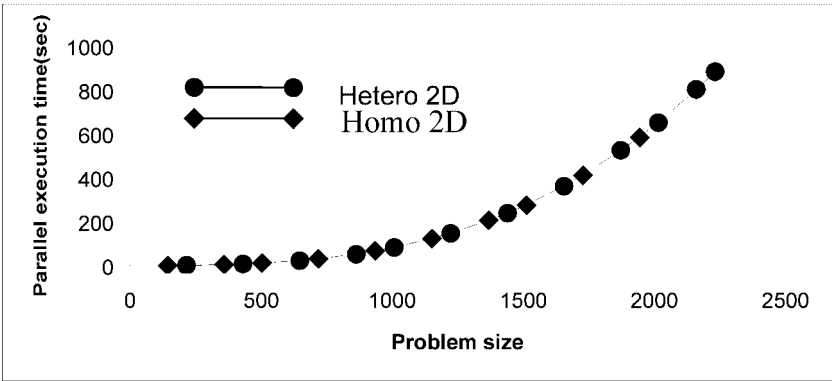


Fig. 7. Execution times of the 2D heterogeneous block cyclic algorithm on a heterogeneous network and of the 2D homogeneous block cyclic algorithm on a homogeneous network. The networks have approximately the same aggregate power of processors and share the same communication network.

- The algorithm based on 2D heterogeneous block cyclic distribution;
- The algorithm based on 1D heterogeneous block cyclic distribution;
- The algorithm based on 2D homogeneous block cyclic distribution.

One can see that the 2D heterogeneous algorithm is almost twice faster than the 1D heterogeneous algorithm and almost 3 times faster than the 2D homogeneous one.

Figure 6 shows the speedup demonstrated by the heterogeneous algorithms compared to the homogeneous one.

Figure 7 shows the comparison of the execution times of the 2D heterogeneous block cyclic algorithm performed on the heterogeneous network and the 2D homogeneous block cyclic algorithm performed on the homogeneous network. One can see that the algorithms demonstrate practically the same speed, but each on its

network. As the two networks are practically of the same power, we can conclude that the heterogeneous algorithm is very close to the optimal heterogeneous modification of the basic homogeneous algorithm. The experiment shows that the additional dependence between successive steps introduced by the heterogeneous modification has practically no impact on the efficiency of the algorithm. This may be explained by the following two factors:

- The speedup due to the overlapping of communication operations performed at successive steps of the algorithm is not very significant;
- The speed of processors in the heterogeneous network does not differ too much. Actually, the network is moderately heterogeneous. Therefore, for this particular network, the additional dependence between steps is very weak.

Thus, for reasonably heterogeneous networks, the presented heterogeneous algorithm has proved to be very close to the optimal one significantly accelerating matrix multiplication on such platforms compared to its homogeneous prototype.

References

1. Crandall, P., Quinn, M.: Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network. In: Proceedings of the Second International Symposium on High Performance Distributed Computing. Spokane WA USA (1993) 42–49
2. Kaddoura, M., Ranka, S., Wang, A.: Array Decomposition for Nonuniform Computational Environments. *Journal of Parallel and Distributed Computing* 3 (1996) 91–105
3. Kalinov, A., Lastovetsky, A.: Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *Journal of Parallel and Distributed Computing* 61 (2001) 520–535
4. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 12 (2001) 1033–1051
5. Fortune, S., Wyllie, J.: Parallelism in Random Access Machines. In: Proceedings of the 10th Annual Symposium on Theory of Computing. San Diego CA USA (1978) 114–118
6. Valiant, L.G.: A Bridging Model for Parallel Computation. *Communications of the Association for Computing Machinery* 33 (1990) 103–111
7. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauer, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation. In: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego CA USA (1993)
8. Lastovetsky, A.: Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Computing* 28 (2002) 1369–1407
9. Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. In: Proceedings of 16th International Parallel and Distributed Processing Symposium. IEEE Computer Society, CD-ROM/Abstracts Proceedings, Fort Lauderdale FL USA (2002)
10. Blackford, L., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance. In: Proceedings of the 1996 ACM/IEEE Supercomputing Conference. IEEE Computer Society, CD-ROM/Abstracts Proceedings, Pittsburgh PA USA (1996)
11. High Performance Fortran Language Specification, Version 2.0. High Performance Fortran Forum (1997)

A Hierarchy of Conditions for Asynchronous Interactive Consistency

Achour Mostefaoui¹, Sergio Rajsbaum², Michel Raynal¹, and Matthieu Roy¹

¹ IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

rajsbaum@matem.unam.mx

² Instituto de Matematicas, UNAM, Ciudad Universitaria, D.F. 04510, Mexico

{achour|raynal|mroy}@irisa.fr

Abstract. The *condition based* approach consists in identifying sets of input vectors, called *conditions*, for which it is possible to design a protocol solving a distributed computing problem despite failures. In a recent work we have applied the condition based approach to the *interactive consistency* (IC) problem (the agreement problem where the processes have to agree on the vector of proposed values), and provided a characterization of the conditions that allow us to solve it in presence of up to f_c process crashes and f_e erroneous proposals. We have shown that these conditions correspond exactly to error correcting codes, where the errors can be erasures or modified values. Here, we investigate this set of conditions from a complexity perspective, and show that it actually consists of a hierarchy of classes of conditions, $\mathcal{C}_{f_c, f_e}^{[\delta]}$, where δ is the *degree* of the condition ($0 \leq \delta \leq f_c$), each class being contained in the previous one (intuitively, the value $f_c - \delta$ represents the “*difficulty*” of a class).

Keywords: Asynchronous Shared Memory System, Atomic Register, Condition, Crash Failure, Erroneous Value, Error-Correcting Code, Fault-Tolerance, Hamming Distance, Interactive Consistency.

1 Introduction

Context of the paper. Agreement problems are among the most important problems one has to solve when designing or building reliable applications on top of asynchronous systems prone to failures [1,10]. *Consensus* is the most famous of these problems: each process proposes a value and each correct process has to decide a value (termination) such that there is a single decided value (agreement) and that value has been proposed by a process (validity). *Interactive consistency* [14] is another important agreement problem that has first been introduced in the context of synchronous systems where processes can suffer Byzantine failures [14]: each process proposes a value and the correct processes have to decide the same vector such that the i -th entry of the vector contains the value proposed by the process p_i if p_i is correct. Practical applications of the interactive consistency problem can be found in [9] (a simple application concerns the detection of the termination of a distributed program in presence of process failures).

The most fundamental result associated with the agreement problems is the so-called FLP theorem that states that consensus cannot be solved in an asynchronous distributed system if processes (even only one) can fail by crashing [6]. This impossibility result has challenged researchers who have investigated several ways of circumventing it, such as considering weaker versions of the problem (e.g., [2,4]) or stronger versions of the system (e.g., [3,5]). We have recently proposed a new approach to address the consensus problem. It consists in identifying sets of input configurations (each one represented by a vector) for which the problem is solvable [11,12,13]. Let an *input vector* be an array whose i -th entry contains the value proposed by p_i . A set of input vectors defines a *condition*. Our main results concerning the consensus problem are the following. (1) A characterization of the set of conditions that allow us to solve consensus, and a condition-based protocol that works for any such condition [11]. (2) The statement of a hierarchy of consensus conditions such that the stronger the condition, the less costly the corresponding protocol. (3) A general weight-based method to define consensus conditions [13].

Content of the paper. Very recently, we have considered the condition-based approach to address the interactive consistency problem [7]. More precisely, we have provided a characterization of the set of conditions that allow us to solve interactive consistency in presence of process crashes and erroneous proposals (so, we also consider “value domain” faults [15], where a process can propose a value a while its input was actually b). In a surprising and interesting way, it is shown that these conditions do exactly correspond to error correcting codes: a condition is actually a set of codewords where the errors can be erasures or modified values. Consequently, any error correcting code defines a condition that solves the interactive consistency problem, and so, information theory provides systematic methods to define conditions suited to interactive consistency.

This paper continues our investigation of the condition-based approach to solve interactive consistency, from a complexity perspective. Similarly to what we have done for the consensus problem [12], we show here that the set of interactive consistency conditions defines a hierarchy of classes of conditions such that the stronger the condition, the less costly the corresponding condition-based interactive consistency protocol. To attain this goal, the hierarchy is expressed in term of a *degree* notion δ ($0 \leq \delta \leq f_c$), and a class of conditions is denoted $\mathcal{C}_{f_c, f_e}^{[\delta]}$ (where f_c and f_e denote the maximum number of processes that can crash or propose erroneous values, respectively). In a very interesting way, it appears that the generic condition-based protocol that we have introduced for the hierarchy of conditions associated with the consensus problem [12], can as well solve IC with any condition of the hierarchy. When instantiated with a condition $C \in \mathcal{C}_{f_c, f_e}^{[\delta]}$, this protocol (designed for a shared memory model) uses $(2n+1) \lceil \log_2(\lceil (f_c - \delta)/2 \rceil + 1) \rceil$ shared memory read/write operations per process in its wait-free synchronization part. In that sense, the value $f_c - \delta$ represents the “*difficulty*” of the class $\mathcal{C}_{f_c, f_e}^{[\delta]}$: the smaller $f_c - \delta$, the more efficient the protocol. As this protocol can also be used to solve consensus, this shows

a strong algorithmic correlation linking consensus and IC: IC is harder than consensus in the sense it requires stronger conditions, and not in the sense it requires a more costly protocol (in terms of communication). So, this paper is on the foundations of reliable distributed computing.

Organization of the paper. The paper is made up of four sections. Section 2 presents the computation model and the condition-based approach for the interactive consistency problem. Section 3 defines the hierarchy of interactive consistency conditions. Finally, Section 4 concludes the paper. Table 1 recapitulates some of our results related to the condition-based approach and indicates the contribution of the current paper.

Table 1. Synthetic Presentation of Results Related to the Condition-Based Approach

	Charac. of the Conditions Basic Cond-Based Protocol	Hierarchy	Definition of Conditions
Consensus	[7,11]	[12]	Weight-based [13]
Int. Consistency	[7]	This paper	Error Correcting Codes [7]

2 Condition-Based Interactive Consistency

2.1 Computation Model

We consider a standard asynchronous system made up of $n > 1$ processes, $p_1, \dots, p_n$, that communicate through a communication medium and where at most f_c , $1 \leq f_c < n$, processes may crash (for more details, see any textbook devoted to distributed computing [1,8,10]). The communication medium can be a shared memory made up of single-writer, multi-reader atomic registers, or a communication network.

2.2 Interactive Consistency

As indicated in the Introduction, the *Interactive Consistency* (IC) problem has initially been defined in the context of synchronous systems prone to Byzantine failures [14]. In the context of asynchronous systems prone to process crash failures, it is defined as follows. A universe of values $\mathcal{V}$ is assumed, together with a default value $\perp$ not in $\mathcal{V}$, that represents an undefined value. Each process p_i proposes a value $v_i \in \mathcal{V}$, and has to decide a vector D_i whose i -th entry is in $\mathcal{V} \cup \{\perp\}$, such that the following properties are satisfied:

- IC-Agreement. No two different vectors are decided.
- IC-Termination. A process that does not crash decides.

- **IC-Validity.** Any decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i does not crash.

So, the IC problem consists in providing the processes with the same vector made up of a value per process, the validity of each value being defined from the behavior of the corresponding process. Unfortunately, as noted in the Introduction, even in a computation model where at most one process can fail only by crashing, this problem has no solution (if IC was solvable, consensus would be). It follows that it cannot be solved either in the model considered in this paper. (Interestingly, it has been shown that, in asynchronous message passing systems in which processes can fail only by crashing, IC and the problem that consists in building a perfect failure detector [3] are equivalent problems [9], which means that any solution to one of them can be used to solve the other).

2.3 Condition-Based Interactive Consistency

The values proposed during each execution form an n -entry vector of $\mathcal{V} \cup \{\perp\}$ with at most f_c undefined ($\perp$) entries. Let $\mathcal{V}_{f_c}^n$ denote the set of all such vectors; thus $\mathcal{V}_{f_c}^n$ is the set of all possible input configurations. The *condition-based* approach for the IC problem has been introduced in [7]. It consists in defining subsets of $\mathcal{V}^n$ for which there exists a protocol that solves the IC problem *at least* when the input vector belongs to this subset or can represent one of its vectors. More precisely, as indicated in the Introduction, in addition to process crashes, we consider also “value domain” faults [15], where a process proposes a value a while it was supposed to propose another value b . Such a process is *value-faulty*. At most f_e processes are value-faulty. We assume $f_c + f_e < n$. (Let us notice that, as in an execution a process proposes a single value, a value-faulty process is not a Byzantine process.) So, we are interested in protocols that tolerate at most f_c process crashes and f_e erroneous proposals.

Remark. The notion of “correct/faulty” with respect to crash is related to an execution, as it is not known in advance if a process will crash. Similarly, the notion of “correct/faulty” with respect to a proposed value is also related to an execution. If $D[i] = v_i$, where D is the decided vector and v_i is the value proposed by p_i , then p_i is value-correct, otherwise it is value-faulty. *End of remark.*

Notations:

- Let $I \in \mathcal{V}^n$, $J \in \mathcal{V}_{f_c}^n$. $d_{\Delta}(I, J)$ = number of corresponding non- $\perp$ entries that differ in I and J .
- If I is a vector, $\mathcal{I}_{f_c, f_e}$ denotes the ball centered at I such that: $\mathcal{I}_{f_c, f_e} = \{J \in \mathcal{V}_{f_c}^n : d_{\Delta}(I, J) \leq f_e\}$.
- For vectors $J_1, J_2 \in \mathcal{V}_{f_c}^n$, $J_1 \leq J_2$ if $\forall k : J_1[k] \neq \perp \Rightarrow J_1[k] = J_2[k]$ (J_2 “contains” J_1).
- $\#_x(J)$ = number of entries of J whose value is x (with $x \in \mathcal{V} \cup \{\perp\}$).
- $d(J_1, J_2)$ = number of entries in which J_1 and J_2 differ (Hamming distance).
- If $C \subset \mathcal{V}^n$ is a condition, $\mathcal{C}_{f_c, f_e}$ is defined as $\mathcal{C}_{f_c, f_e} = \cup_{I \in C} \mathcal{I}_{f_c, f_e}$.

We say that an (f_c, f_e) -fault tolerant protocol solves the interactive consistency problem for the condition C (CB_IC problem) if, for any input vector J , the protocol satisfies:

- CB_IC-Agreement. No two different vectors are decided.
- CB_IC-Validity. If $J \in \mathcal{C}_{f_c, f_e}$, then the decided vector D is such that $J \in \mathcal{D}_{f_c, f_e}$ with $D \in C$.
- CB_IC-Termination. If (1) $J \in \mathcal{C}_{f_c, f_e}$ and at most f_c processes crash, or (2.a) a process decides, or (2.b) no process crashes, then every crash-correct process decides.

The agreement property states that there is a single decision, even if the input vector is not in C , guaranteeing “safety” always. The termination property requires that the processes that do not crash must decide at least when the circumstances are “favorable.” Those are (1) when the input could have belonged to C , as explained above, (provided there are no more than f_c crashes during the execution), and (2) under normal operating conditions. The aim of the validity property is to eliminate trivial solutions by relating the decided vector and the proposed vector. It states that, when the proposed vector belongs to at least one ball defined by the condition, the center of such a ball is decided, which is one of the possible actual inputs that could have been proposed [7].

Let us consider an ideal system, namely a system where there is neither crash nor erroneous proposal ($f_c = f_e = 0$). In that case, it is trivial to design a protocol that works for the condition made up of all vectors of $\mathcal{V}^n$. In that case, IC and CB_IC confuse and the decided vector is always the proposed vector. As soon as there are failures, the condition including all possible input vectors fails solve the problem (as indicated before, if it was the case, it would also solve consensus). Hence, some price has to be paid if we want to solve interactive consistency without augmenting the underlying system with appropriate devices (such as, for example, failure detectors). This price is related to the possibility of crashes and erroneous proposals we want to cope with. It is clearly formulated (1) in the statement of the termination property (that does not require termination when there are more than f_c crashes or when there are crashes and the input vector is too far from the condition), and (2) in the statement of the validity property (that does not require a vector of the condition to be decided if the input vector is too far from the condition).

Basically, the underlying idea of the CB_IC problem is that the processes are assumed to collectively propose an input vector I belonging to the condition C , and then get it and decide. As crashes and erroneous proposals can occur, the specification of CB_IC precisely states the situations in which a vector has to be decided and which is then the decided vector.

It is shown in [7] that the set of conditions that solve the CB_IC problem is exactly the set of error correcting codes. This not only establishes a strong link relating error correcting codes and distributed computing, but also provides an easy way to define conditions suited to the CB_IC problem.

3 A Hierarchy of Classes of IC Conditions

This section defines and investigates the hierarchy $\mathcal{C}_{f_c, f_e}^{[f_c]} \subset \dots \mathcal{C}_{f_c, f_e}^{[\delta]} \subset \dots \subset \mathcal{C}_{f_c, f_e}^{[1]} \subset \mathcal{C}_{f_c, f_e}^{[0]}$ of condition classes that allow solving the interactive consistency problem. The parameter δ ($0 \leq \delta \leq f_c$) is called the *degree* of the class. (When we consider a condition $C \in \mathcal{C}_{f_c, f_e}^{[\delta]}$, δ is also called the *degree* of C .)

3.1 Acceptability of a Condition

As shown in [11,7], a condition can be defined in two equivalent ways, called acceptability and legality. The first is interesting to define protocols, while the second is more useful to prove impossibility results. Here, we extend these definitions to take into account a parameter δ (degree) that allows us to define a hierarchy of conditions ([7] does not consider the degree notion, and so, implicitly considers the only case $\delta = 0$).

Given a condition C and two values f_c and f_e , acceptability is an operational notion defined in terms of a predicate P and a function S that have to satisfy some properties in order that a protocol can be designed. Those properties are related to termination, validity and agreement, respectively. The intuition for the first property is the following. The predicate P allows a process p_i to test if a decision can be computed from its view (the vector it can build from the proposed values it knows). Thus, P returns true at least for all those input vectors J such that $J \in \mathcal{I}_{f_c, f_e}$ for $I \in C$.

- Property $T_{C \rightarrow P}$: $I \in C \Rightarrow \forall J \in \mathcal{I}_{f_c, f_e} : P(J)$.

The second property is related to validity.

- Property $V_{P \rightarrow S}$:

$$\forall J \in \mathcal{V}_{f_c}^n : P(J) \Rightarrow S(J) = I \text{ such that } I \in C \wedge J \in \mathcal{I}_{f_c, f_e}.$$

The last property concerns agreement. Given an input vector I , if two processes p_i and p_j get the views $J1$ and $J2$, such that $P(J1)$ and $P(J2)$ are satisfied, these processes have to decide the same vector, from $J1$ for p_i and $J2$ for p_j , whenever the following holds.

- Property $A_{P \rightarrow S}^{[\delta]}$: $\forall I \in \mathcal{V}^n : \forall J1, J2 \in \mathcal{V}_{f_c}^n : J1 \leq I, J2 \leq I :$

$$P(J1) \wedge P(J2) \wedge ((J1 \leq J2) \vee (\#_{\perp}(J1) + \#_{\perp}(J2) \leq f_c + \delta))$$

$$\Rightarrow S(J1) = S(J2).$$

Definition 1. A condition C is (f_c, f_e, δ) -acceptable if there exist a predicate P and a function S satisfying the properties $T_{C \rightarrow P}$, $V_{P \rightarrow S}$ and $A_{P \rightarrow S}^{[\delta]}$.

The following results are proved in [11]. (1) The set of conditions C for which there exists a pair (P, S) satisfying the properties $T_{C \rightarrow P}$, $V_{P \rightarrow S}$ and $A_{P \rightarrow S}^{[0]}$ is the largest set of conditions for which an interactive consistency protocol does

exist. (2) This set is the set of error correcting codes. As a consequence, error correcting code theory provides systematic ways to define conditions and their (P, S) pair. Let us assume a code/condition defined by a check matrix A . We have (let us remind that $\text{syndrome}(I) = A I^T$):

- $C = \{I \text{ such that } \text{syndrome}(I) = 0\}$,
- $P(J) = \exists I : \text{ such that } J \in \mathcal{I}_{f_c, f_e} \wedge \text{syndrome}(I) = 0$,
- $S(J) = I \text{ such that } J \in \mathcal{I}_{f_c, f_e} \wedge \text{syndrome}(I) = 0$.

3.2 Legality of a Condition

While acceptability is an operational notion, legality is a combinatorial notion useful to analyze a condition using a geometrical representation.

Definition 2. A condition C is (f_c, f_e, δ) -legal if for all distinct $I_1, I_2 \in C$, $d(I_1, I_2) \geq 2f_e + f_c + \delta + 1$.

The following theorem (Theorem 1) is important for the condition-based approach applied to interactive consistency. It states that, for any degree δ , acceptability and legality are actually equivalent notions. This theorem is based on the following lemma:

Lemma 1. Let C be an (f_c, f_e, δ) -acceptable condition. Then, for any I in C , S is constant on the ball $\mathcal{I}_{f_c, f_e}$ and $S(\mathcal{I}_{f_c, f_e}) = \{I\}$.

Proof The proof is made in two parts: we first show that $S(I) = I$ for any I in C , then we show that this can be extended to any view in the ball centered at I .

Part 1: proof of $S(I) = I$.

Let C be an (f_c, f_e, δ) -acceptable condition. Let us assume that $I \in C$ and $S(I) = I_0 \neq I$.

Using the validity property $V_{P \rightarrow S}$ shows that $I_0 \in C$ and $I \in \mathcal{I}_{f_c, f_e}$. Since $I \neq I_0$, the two balls $\mathcal{I}_{f_c, f_e}$ and $\mathcal{I}_{f_c, f_e}$ are different; let J in $\mathcal{I}_{f_c, f_e} \setminus \mathcal{I}_{f_c, f_e}$. The termination property $T_{C \rightarrow P}$ instantiated with J and I ensures that $P(J)$ holds. The validity property applied to J gives $S(J) \neq I_0$ (since $J \notin \mathcal{I}_{f_c, f_e}$, by definition).

Let us construct the following chain of vectors. Let J_1 be the vector obtained by replacing the $\perp$ entries in J by the corresponding entries in I . Let J_2 be the view obtained from J_1 by replacing up to f_c entries that differ in J_1 and I by $\perp$. For $i \geq 1$, let J_{2i+1} be the vector obtained by replacing the $\perp$ entries in J_{2i} by the corresponding entries in I , and J_{2i} be the view obtained from J_{2i-1} by replacing up to f_c entries that differ in J_{2i-1} and I by $\perp$. There exists an i_0 such that $J_{i_0} = I$.

The following holds by construction of the chain : (1) $J_1 \geq J, J_1 \geq J_2, J_3 \geq J_2, J_3 \geq J_4 \cdots J_{2i+1} \geq J_{2i}, J_{2i+1} \geq J_{2i+2}$ and (2) $\forall i, J_i \in \mathcal{I}_{f_c, f_e}$.

The termination property shows that P holds for any J_i of this chain, and the agreement property (applied to J_{2i} and J_{2i-1} and to J_{2i} and J_{2i+1}) ensures that $S(J) = S(J_1) = \dots = S(J_{i_0}) = S(I)$.

But $S(I) = I0 \neq I$ (initial assumption), and the definition of J yields to $S(J) \neq I0$. Hence a contradiction.

Part 2: proof of $S(\mathcal{I}_{f_c, f_e}) = \{I\}$.

Let I in C , and J in $\mathcal{I}_{f_c, f_e}$. In a similar way, let us construct the chain $(J_i)_i$ such that:

- $J_0 = J$
- J_{2i+1} is obtained by replacing every $\perp$ entry in J_{2i} by the corresponding entry in I
- J_{2i} is obtained from J_{2i-1} by replacing up to f_c entries that differ in J_{2i-1} and I by $\perp$.

Agreement and termination applied to the chain shows that S is constant on the chain. Since there exists an i_0 such that $I = J_{i_0}$, we can conclude that $S(J) = S(J_0) = S(J_{i_0}) = S(I)$. The first part of the lemma shows that $S(I) = I$, and finally, we get that for any J in the ball centered at I , $S(J) = I$. $\square_{\text{Lemma 1}}$

Theorem 1. *A condition C is (f_c, f_e, δ) -acceptable iff it is (f_c, f_e, δ) -legal.*

Proof $\Rightarrow$ direction:

Let C be an (f_c, f_e, δ) -acceptable condition. Let $I1$ and $I2$ be two distinct vectors in C such that $d(I1, I2) \leq 2f_e + f_c + d$. Without loss of generality, let us assume that $I1$ and $I2$ differ only in the $2f_e + f_c + d$ first indices. From these two vectors, let us construct two vectors $J1$ and $J2$ as follows:

$i \in$	$J1[i] =$	$J2[i] =$
$1..f_e$	$I1[i]$	$I1[i]$
$f_e + 1..2f_e$	$I2[i]$	$I2[i]$
$2f_e + 1..2f_e + \left\lfloor \frac{f_e + \delta}{2} \right\rfloor$	$I1[i]$	$\perp$
$2f_e + \left\lfloor \frac{f_e + \delta}{2} \right\rfloor + 1..2f_e + f_c + d$	$\perp$	$I2[i]$
$2f_e + f_c + d + 1..n$	$I1[i](= I2[i])$	$I1[i]$

Since (1) $J1$ (resp. $J2$) is in $\mathcal{I}_{f_c, f_e}^{[\delta]}$ (resp. in $\mathcal{I}_{f_c, f_e}^{[\delta]}$), and (2) $I1$ and $I2$ belong to C , the $\text{TC}_{C \rightarrow P}$ property implies that P holds for both $J1$ and $J2$.

By construction of $J1$ and $J2$, $\#_{\perp}(J1) + \#_{\perp}(J2) \leq f_c + \delta$; hence, by $A_{P \rightarrow S}^{[\delta]}$, $S(J1) = S(J2)$.

Let us now apply the previous lemma on vectors $I1$ and $J1$ (resp. $I2$ and $J2$), and obtain that $S(I1) = S(J1) = I1$ (resp. $S(I2) = S(J2) = I1$).

Therefore, the following holds: $I1 = S(I1) = S(J1) = S(J2) = S(I2) = I2$, i.e. $I1 = I2$. It follows that any two distinct vectors of C are distant of at least $f_c + 2f_e + \delta + 1$.

$\Leftarrow$ direction:

Let C be an (f_c, f_e, δ) -legal condition. Since, for every pair of vectors $I1, I2$ of C , $d(I1, I2) \geq f_c + 2f_e + \delta + 1$, the two balls $\mathcal{I}_{f_c, f_e}$ and $\mathcal{I}_{2f_c, f_e}$ do not intersect.

Therefore, for any J in $\mathcal{V}_{f_c}^n$, if there exists an I in C such that $J \in \mathcal{I}_{f_c, f_e}$, then let $P(J)$ be true and $S(J) = I$. Otherwise, let $P(J)$ be false.

The properties $T_{C \rightarrow P}$ and $V_{P \rightarrow S}$ hold by definition of P and S .

For the proof of the $A_{P \rightarrow S}^{[\delta]}$, let an I in $\mathcal{V}^n$, and two views $J1$ and $J2$ in $\mathcal{V}_{f_c}^n$ such that $J1 \leq I$, $J2 \leq I$, $P(J1)$ and $P(J2)$. Let $I1 = S(J1)$ and $I2 = S(J2)$. Let us notice that $d(J1, I1) \leq f_e + \#_{\perp}(J1)$ by the definition of $I1$. There are two cases:

If $J1 \leq J2$, then $d(I1, I2) \leq 2f_e + \#_{\perp}(J1) \leq 2f_e + f_c$. Since C is (f_c, f_e, δ) -legal, it implies that $I1 = I2$, hence $S(J1) = S(J2)$.

If $\#_{\perp}(J1) + \#_{\perp}(J2) \leq f_c + \delta$, $d(I1, I2) \leq 2f_e + \#_{\perp}(J1) + \#_{\perp}(J2) \leq 2f_e + f_c + \delta$, thus showing that $I1 = I2$, i.e. $S(J1) = S(J2)$. $\square_{Theorem 1}$

3.3 The Hierarchy

This section describes the hierarchy of conditions induced by the previous definitions, and some of its properties.

Definition 3. Let the class $\mathcal{C}_{f_c, f_e}^{[\delta]}$ be the set of all the (f_c, f_e, δ) -acceptable conditions.

The next theorem shows that these classes form a hierarchy of conditions.

Theorem 2.

$$\mathcal{C}_{f_c, f_e}^{[f_c]} \subset \mathcal{C}_{f_c, f_e}^{[f_c-1]} \subset \mathcal{C}_{f_c, f_e}^{[f_c-2]} \subset \dots \subset \mathcal{C}_{f_c, f_e}^{[0]}.$$

Proof These containments follow directly from the definition of the legality and Theorem 1. It is easy to check that these containments are strict using the definition of legality. For example, let C be the (f_c, f_e, δ) -legal condition made up of two vectors: the vector $I1$ with all entries equal to 1, and the vector $I2$ with the first $f_c + 2f_e + \delta$ entries equal to 0, the others equal to 1. We have $d(I1, I2) = f_c + 2f_e + \delta$. It follows that $C \in \mathcal{C}_{f_c, f_e}^{[\delta-1]}$ and $C \notin \mathcal{C}_{f_c, f_e}^{[\delta]}$. $\square_{Theorem 2}$

The definition of a condition involves three parameters, namely f_c , f_e and δ . The simple linear form of the legality definition provides the following “trading” theorem.

Theorem 3.

$$\mathcal{C}_{f_c+\alpha, f_e}^{[\delta-\alpha]} = \mathcal{C}_{f_c, f_e}^{[\delta]} \quad (1)$$

$$\mathcal{C}_{f_c, f_e+\alpha}^{[\delta-2\alpha]} = \mathcal{C}_{f_c, f_e}^{[\delta]} \quad (2)$$

$$\mathcal{C}_{f_c, f_e+\alpha}^{[\delta]} = \mathcal{C}_{f_c+2\alpha, f_e}^{[\delta]} \quad (3)$$

Proof These equalities follow directly from Theorem 1 and elementary calculus. Namely, (1) and (2) from the fact that $f_c+2f_e+\delta+1 = (f_c+\alpha)+2f_e+(\delta-\alpha)+1 = f_c+2(f_e+\alpha)+(\delta-2\alpha)+1$. And (3) from the fact that $f_c+2(f_e+\alpha)+\delta+1 = (f_c+2\alpha)+2f_e+\delta+1$. $\square_{Theorem 3}$

3.4 A Simple Example

Let a system made up of $n = 6$ processes, and let $\mathcal{V} = \{0, 1\}$ the set of values that can be proposed by the processes. Let us consider the following two conditions:

- $C1$ is defined as follows:

$$C1 = \{V \in \mathcal{V}^6 \mid \#_1(V) \text{ is even} \}.$$

The condition $C1$ includes 2^{n-1} (32) vectors. Its minimal Hamming distance is 2. It follows that (1) $C1$ is $(1, 0)$ -legal, i.e., $C1 \in \mathcal{C}_{1,0}^{[0]}$; and (2) $C1$ is not $(2, 0)$ -legal, i.e., $C1 \notin \mathcal{C}_{2,0}^{[0]}$.

- Let us now consider the condition $C2$ made up of the following 8 vectors:

$$\begin{array}{cccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}$$

Its minimal Hamming distance is 3, hence (trivially, $C2 \in \mathcal{C}_{1,0}^{[0]}$), $C2 \in \mathcal{C}_{2,0}^{[0]}$, which is equivalent (Theorem 3) to $C2 \in \mathcal{C}_{1,0}^{[1]}$, and $C2 \in \mathcal{C}_{0,1}^{[0]}$.

It follows that both $C1$ and $C2$ can cope with $f_c = 1$ crash and no erroneous proposal ($f_e = 0$). Moreover, $C2$ can also cope either with $f_c = 2$ crashes and no erroneous proposal ($f_e = 0$), or with no crash ($f_c = 0$) and $f_e = 1$ erroneous proposal. Finally, when used in a system with $f_c = 1$ crash and $f_e = 0$, the condition $C2$ generates a protocol more efficient than a protocol designed for $C1$, as shown in the next section. This exhibits a tradeoff relating the cost of a CB-IC protocol and the number of vectors defining the condition it uses: the smaller the condition, the more efficient the protocol when the input vector does belong to the condition¹.

4 Conclusion

This paper has addressed the interactive consistency problem in the context of the condition-based approach. It has shown that the set of conditions that solve the interactive consistency problem defines a hierarchy, each class $\mathcal{C}_{f_c, f_e}^{[\delta]}$ of the hierarchy being associated with a parameter δ , such that the value $f_c - \delta$ represents the “*difficulty*” of a class. Interestingly, the generic condition-based protocol initially designed for the hierarchy of consensus conditions [12] can as well be used with the hierarchy of interactive consistency conditions. When the communication medium is a shared memory, the cost of this protocol is $(2n + 1) \lceil \log_2(\lceil (f_c - \delta)/2 \rceil + 1) \rceil$ shared memory accesses. As this protocol can also be used to solve consensus, it shows that the difference between IC and consensus lies only in the condition they require: interactive consistency is harder than consensus in the sense it requires stronger, conditions (i.e., conditions including less input vectors).

¹ But when the condition is smaller, it includes less vectors, and so the protocol can converge less often.

References

1. Attiya H. and Welch J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. McGraw-Hill (1998), 451
2. Ben-Or M.: Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83), Montréal (1983), 27–30
3. Chandra T. and Toueg S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, Vol. 43(2) (1996), 225–267
4. Chaudhuri S.: More Choices Allow More Faults: set Consensus Problems in Totally Asynchronous Systems. Information and Computation, Vol. 105 (1993), 132–158
5. Dwork C., Lynch N. and Stockmeyer L.: Consensus in the Presence of Partial Synchrony. Journal of the ACM, Vol. 35(2) (1988), 288–323
6. Fischer M.J., Lynch N.A. and Paterson M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, Vol. 32(2) (1985), 374–382
7. Friedman R., Mostefaoui A., Rajsbaum S., Raynal M.: Distributed Agreement and its Relation with Error-Correcting Codes. In: Proc. 16th Symposium on Distributed Computing (DISC'02), Lecture Notes in Computer Science, Vol. 2508. Springer-Verlag, Berlin Heidelberg New York (2002), 63–87
8. Garg V.K.: Elements of Distributed Computing. Wiley (2002), 423
9. Hélyar J.-M., Hurfin M., Mostéfaoui A., Raynal M. and Tronel F.: Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. IEEE Trans. on Parallel and Distributed Systems, Vol. 11(9) (2000), 897–910
10. Lynch N.A.: Distributed Algorithms. Morgan Kaufmann Pub. (1996), 872
11. Mostefaoui A., Rajsbaum S. and Raynal M.: Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. In: Proc. 33rd ACM Symposium on Theory of Computing (STOC'01), ACM Press, Hersonissos, Crete (July 2001), 153–162
12. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M.: A Hierarchy of Conditions for Consensus Solvability. In: Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01), ACM Press, Newport (RI), (August 2001), 151–160
13. Mostefaoui A., Rajsbaum S., Raynal M., Roy M.: Efficient Condition-Based Consensus. In: 8th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'01), Carleton Univ. Press (June 2001), 275–291
14. Pease L., Shostak R. and Lamport L.: Reaching Agreement in Presence of Faults. Journal of the ACM, Vol. 27(2) (1980), 228–234
15. Powell D.: Failures Mode Assumptions and Assumption Coverage. Proc. 22th IEEE Fault-Tolerant Computing Symposium (FTCS'92), IEEE Society Press, Boston (MA) (1992), 386–395

Associative Parallel Algorithms for Dynamic Edge Update of Minimum Spanning Trees*

Anna S. Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics,
Siberian Division of Russian Academy of Sciences,
pr. Lavrentieva, 6, Novosibirsk, 630090, Russia anep@ssd.sccc.ru

Abstract. In this paper we propose two associative parallel algorithms for the edge update of a minimum spanning tree when an edge is deleted or inserted in the underlying graph. These algorithms are represented as the corresponding procedures implemented on a model of associative parallel systems of the SIMD type with vertical data processing (the STAR-machine). We justify correctness of these procedures and evaluate their time complexity.

1 Introduction

Dynamic graph algorithms are designed to handle graph changes. They maintain some property of a changing graph more efficiently than recomputation of the entire graph with a static algorithm after every change. We will consider the edge update of a minimum spanning tree (MST) of an undirected graph with n vertices and m edges. This problem involves reconstructing a new MST from the current one when an edge is deleted or inserted or its weight changes.

Sequential algorithms for edge updating an MST have been presented in [1,4,12]. In [2], a general technique, called sparsification, for designing dynamic graph algorithms is provided. In [10], the edge update problem is studied by means of a CREW PRAM model. The corresponding parallel algorithms take $O(\log n)$ time and use $O(n^2)$ processors. In [9], parallel algorithms for updating an MST under a batch of edge insertions or edge deletions are described using a CREW PRAM model.

In this paper, we propose associative parallel algorithms for dynamic edge update of an MST of an undirected graph represented as a list of triples (edge vertices and the weight). Our model of computation (the STAR-machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with bit-serial (vertical) processing and simple processing elements (PEs). Such an architecture performs data parallelism at the base level, provides massively parallel search by contents, and allows one the use of two-dimensional tables as basic data structure [11]. For dynamic edge update of an MST, we use, in particular, a matrix of tree paths consisting of m rows and n columns. Its every

* This work was supported in part by the Russian Foundation for Basic Research under Grant N 03-01-00399

i -th column saves the tree path from the root v_1 to vertex v_i . We show how to perform local changes in this matrix along with the edge update of an MST. In [8], a static associative parallel algorithm for finding an MST starting at a given vertex of a graph takes $O(n \cdot \log n)$ time assuming that each elementary operation of the STAR-machine (its microstep) takes one unit of time. Associative parallel algorithms for dynamic edge update of an MST are represented as the corresponding STAR procedures that take $O(h \cdot \log n)$ time each, where h is the number of vertices whose tree paths change after an edge update.

2 Model of Associative Parallel Machine

We define the model as an abstract STAR-machine of the SIMD type with vertical processing and simple single-bit PEs. To simulate the access data by contents, we use some *typical* operations for associative systems first presented in Staran [3]. Many contemporary associative systems employ bit-serial and word-parallel processing because it permits the use of a low-cost standard memory and chips [5].

The model consists of the following components:

- a sequential control unit (CU), where programs and scalar constants are stored;
- an associative processing unit consisting of p single-bit PEs;
- a matrix memory for the associative processing unit.

The CU broadcasts an instruction to all PEs in unit time. All active PEs execute it simultaneously while inactive PEs do not perform it. Activation of a PE depends on the data employed.

Input binary data are loaded in the matrix memory in the form of two-dimensional tables, where each data item occupies an individual row and it is updated by a dedicated PE. The rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed.

The associative processing unit is represented as h vertical registers, each consisting of p bits. A vertical register can be regarded as a one-column array that maintains the entire column of a table. Bit columns of tabular data are stored in the registers which perform the necessary bitwise operations.

The STAR-machine run is described by means of the language STAR [6] being an extension of Pascal. To simulate data processing in the matrix memory, we use data types **slice** and **word** for the bit column access and the bit row access, respectively, and the type **table** for defining the tabular data. Assume that any variable of the type **slice** consists of p components. For simplicity, let us call “slice” any variable of the type **slice**.

Let X , Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following elementary operations for slices:

SET(Y) sets all components of the slice Y to '1'; CLR(Y) sets all components of Y to '0'; $Y(i)$ selects the i -th component of Y ; FND(Y) returns the ordinal number of the first (the uppermost) '1' of Y ; NUMB(Y) returns the number of components '1' in the slice Y .

In the usual way we introduce the predicate $\text{SOME}(Y)$ and the bitwise Boolean operations $X \text{ and } Y$, $X \text{ or } Y$, $\text{not } Y$, $X \text{ xor } Y$.

Let T be a variable of the type **table**. We use the following two operations:

$\text{ROW}(i, T)$ returns the i -th row of the matrix T ; $\text{COL}(i, T)$ returns the i -th column of T .

Remark 1. Note that the STAR statements are defined in the same manner as for Pascal. They will be used for presenting our procedures.

We will employ the following three basic procedures implemented on the STAR-machine [7]. They use a global slice X to mark by '1' *positions* of rows which will be processed.

The procedure $\text{MATCH}(T, X, v, Z)$ defines in parallel positions of the given matrix T rows which coincide with the given pattern v written in binary code. It returns the slice Z , where $Z(i) = '1'$ if and only if $\text{ROW}(i, T) = v$ and $X(i) = '1'$.

The procedure $\text{MIN}(T, X, Z)$ defines in parallel positions of the given matrix T rows, where minimum elements are located. It returns the slice Z , where $Z(i) = '1'$ if and only if $\text{ROW}(i, T)$ is the minimum element in T and $X(i) = '1'$.

The procedure $\text{MAX}(T, X, Z)$ is defined by analogy with $\text{MIN}(T, X, Z)$.

As shown in [7], the basic procedures run in $O(k)$ time each, where k is the number of columns in T .

3 Finding MST along with Tree Paths

Let $G = (V, E)$ denote an *undirected graph*, where V is a set of vertices and E is a set of edges. Let w denote a function that assigns a weight to every edge. We assume that $V = \{1, 2, \dots, n\}$, $|V| = n$, and $|E| = m$.

A *path* from v_1 to v_k in G is a sequence of vertices $v_1, v_2, \dots, v_k$, where $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. If $v_1 = v_k$, then the path is called a *cycle*.

A *minimum spanning tree* $T = (V, E')$ is a connected acyclic subgraph of G , where $E' \subseteq E$ and the sum of weights of the corresponding edges is minimal.

Let every edge (u, v) be matched with the triple $\langle u, v, w(u, v) \rangle$. Note that vertices and weights are written in binary code. In the STAR-machine matrix memory, a graph is represented as association of matrices *left*, *right*, and *weight*, where every triple $\langle u, v, w(u, v) \rangle$ occupies an individual row, and $u \in \text{left}$, $v \in \text{right}$, and $w(u, v) \in \text{weight}$. We will also use a matrix *code*, whose every i -th row saves the binary representation of vertex v_i . Let us agree to use a slice Y for the matrix *code*, a slice S for the list of triples, and a slice T for the MST.

In [8], we have proposed an associative version of the Prim-Dijkstra algorithm for finding an MST starting at a given vertex v . The corresponding procedure MSTPD returns a slice T , where *positions* of edges belonging to the MST are marked by '1'. Dynamic graph algorithms require, in particular, a fast method for finding a tree path between any pair of vertices. To this end, by means of *minor* changes in the procedure MSTPD, we build an MST along with a matrix M , whose every i -th column saves *positions* of edges belonging to the tree path from vertex v_1 to vertex v_i . The corresponding procedure MSTPaths returns the slice T and the matrix of tree paths M . It runs as follows. Initially, the

procedure sets zeros in the first column of M and saves the root v_1 . By analogy with MSTPD, at every iteration, it defines both the *position* of the current edge (say, γ) and the corresponding *new* vertex v_k being included in the fragment T_s . Moreover, it defines end-point v_l of γ included in T_s *before* this iteration. The tree path from v_1 to v_k is obtained by adding the *position* of γ to the tree path from v_1 to v_l defined before. This path is written in the k -th column of M .

Its correctness is proved by induction on the number of tree edges.

Without loss of generality, we will assume that initially a minimum spanning tree is always given along with the matrix of tree paths.

4 Auxiliary Procedures

Here, we propose a group of auxiliary procedures being used for dynamic edge update of an MST T and a matrix of tree paths M .

The procedure $\text{EdgePos}(\text{left}, \text{right}, \text{code}, T, i, j, l)$ returns the *position* l of an edge having end-points v_i and v_j . It runs as follows. First, the procedure defines binary codes *node1* and *node2* of vertices v_i and v_j , respectively. Then, it determines whether this edge has a form $(\text{node1}, \text{node2})$ or $(\text{node2}, \text{node1})$. Finally, the edge *position* in the graph representation is defined.

The next procedures explore the case, when an edge (say, γ) is deleted from the MST T . Then its position l is marked by '0' both in the slice T and in every tree path of the matrix M that includes the edge γ . Moreover, the vertices, whose tree paths include this edge, will form a connected component (say, $Y1$), because after deleting γ none of them can be reachable from the root v_1 .

The procedure $\text{CompVert}(l, M, Y1)$ returns the slice $Y1$ for the matrix *code* to save vertices not reachable from the root v_1 after deleting an edge from T . It runs as follows. The procedure first selects the l -th row in the matrix M , where the deleted edge is located. While this row is non-empty, it defines the current vertex v_j and saves its position in the slice $Y1$.

The procedure $\text{OldRoot}(\text{left}, \text{right}, \text{code}, Y1, l, \text{del})$ returns end-point v_{del} of an edge located in the l -th row. It runs as follows. The procedure determines end-points of the edge and selects the vertex that belongs to the connected component $Y1$ after deleting the edge from the MST.

The procedure $\text{NewRoot}(\text{left}, \text{right}, \text{code}, M, Y1, k, \text{ins}, W)$ returns vertex v_{ins} of an edge located in the k -th row and a slice W to save positions of edges belonging to the *new* tree path from v_1 to v_{ins} after the edge insertion in the MST. It runs as follows. The procedure determines vertex v_{ins} in the same manner as vertex v_{del} . The slice W is obtained by adding the edge position k to the tree path from v_1 to the other end-point of the edge written in the corresponding column of the matrix M .

The procedure $\text{ConEdges}(\text{left}, \text{right}, \text{code}, S, Y1, Q)$ returns the slice Q to save *positions* of edges having a *single* end-point from $Y1$. It runs as follows. By means of two slices, the procedure accumulates *positions* of edges whose left (respectively, right) end-point belongs to $Y1$. Disjunction of these slices determines *positions* of edges having at least one end-point from $Y1$, while their

conjunction defines *positions* of edges whose both end-points belong to $Y1$. Knowing disjunction and conjunction of these slices, it determines the slice Q .

Correctness of procedures *EdgePos*, *CompVert*, *OldRoot* and *NewRoot* is evident. Correctness of *ConEdges* is established by contradiction.

5 Updating Tree Paths

Let a new MST be obtained from the underlying one by deleting an edge (say, γ) located in the l -th position and inserting an edge (say, δ) located in the k -th position. Let $Y1$ be a connected component of G obtained after deleting γ . Let v_{del} and v_{ins} be end-points of the corresponding edges γ and δ that belong to $Y1$. Let P be a slice that saves positions of tree edges joining v_{ins} and v_{del} .

Let us agree, for convenience, that a tree path from v_1 to any vertex v_s is denoted by p_s before updating the MST and by p'_s after updating the MST.

The algorithm determines *new* tree paths for all vertices from $Y1$. It starts at vertex v_{ins} . Note that p'_{ins} (the slice W) is obtained in the procedure *NewRoot*.

The algorithm carries out the following stages.

At the *first* stage, make a copy of the matrix of tree paths M , namely $M1$. The matrix $M1$ will save tree paths *before* updating the current MST. Write p'_{ins} in the corresponding column of M . Mark vertex v_{ins} by '0' in the slice $Y1$. Then fulfil the statement $r := ins$.

While P is a non-empty slice, repeat stages 2 and 3.

At the *second* stage, determine vertices not belonging to P that form a subtree of the MST with the root v_r if any. For every $v_j \neq v_r$ from this subtree, compute p'_j as follows:

$$p'_j := (p_j \text{ and } (\text{not } p_r)) \text{ or } p'_r \quad (1)$$

Write p'_j in the corresponding column of M . Mark v_j by '0' in the slice $Y1$.

At the *third* stage, select position i of an edge from P incident on vertex v_r . Then define its end-point (say, v_q) being adjacent with v_r . The new tree path p'_q is obtained by writing '1' in the i -th bit of p'_r . Now, write p'_q in the corresponding column of M . Mark the edge position i by '0' in the slice P and vertex v_q by '0' in the slice $Y1$. Finally, perform the statement $r := q$.

At the *fourth* stage, since P is an empty slice, the vertices marked by '1' in the slice $Y1$ form a subtree of the MST with the root v_r just determined. For every $v_j \neq v_r$ from this subtree, define p'_j using formula (1). Write p'_j in the corresponding column of M . Then mark vertex v_j by '0' in the slice $Y1$.

The algorithm terminates when slices P and $Y1$ become empty.

It is implemented on the STAR-machine as procedure *TreePaths* which uses the following input parameters: matrices *left*, *right*, and *code*, vertices v_{ins} and v_{del} , and the number of vertices n . It returns the matrix M for the new MST and slices W , $Y1$, and P . Initially, the slice W saves the new tree path from v_1 to v_{ins} , the slice P saves positions of edges from the tree path joining v_{ins} and v_{del} , and the slice $Y1$ saves vertices whose tree paths will be recomputed.

Correctness of this algorithm is checked by induction on the number of edges belonging to the slice P .

Now, we illustrate the run of the procedure `TreePaths`.

Let a new MST be obtained from the underlying one after deleting the edge (v_4, v_8) and inserting a new edge (v_7, v_{14}) as shown in Figures 1 and 2. Here, the connected component Y_1 consists of vertices $v_8, v_9, \dots, v_{18}$; $del = 8$ and $ins = 14$.

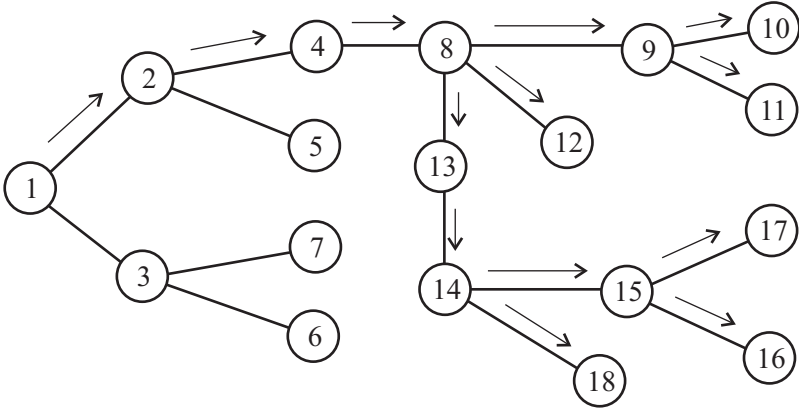


Fig. 1. MST before deleting the edge $(4,8)$

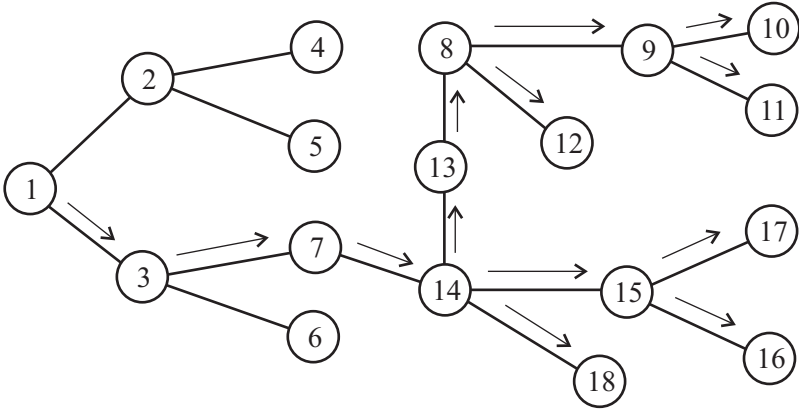


Fig. 2. MST after inserting the edge $(7,14)$

The algorithm starts at vertex v_{14} . Then the new tree paths are recomputed for vertices v_{15}, v_{16}, v_{17} , and v_{18} from the subtree rooted at v_{14} . Further, a new tree path is first defined for v_{13} and then for v_8 . Finally, new tree paths are recomputed for vertices v_9, v_{10}, v_{11} , and v_{12} from the subtree rooted at v_8 .

6 Associative Parallel Algorithm for Edge Deletion

Let v_i and v_j be end-points of an edge being deleted from T . The algorithm runs as follows. It first determines the deleted edge position and excludes it from further consideration. Then, it defines the connected component $Y1$ whose vertices are not reachable from root v_1 after deleting this edge. Further, it determines the position of the minimum weight edge joining two connected components and saves it in T . Finally, tree paths for vertices from $Y1$ are recomputed.

Now, we present the procedure DelEdge.

```

procedure DelEdge(left,right,weight: table; code: table;
  Y: slice(code); i,j,n: integer; var S,T: slice(left);
  var M: table);
var P,Q,W,X,Z: slice(left);
  Y1,Y2: slice(code);
  k,l,h,r,ins,del: integer;
1. Begin EdgePos(left,right,code,T,i,j,l);
/* Knowing end-points, we define the edge position. */
2.   T(1):= '0'; S(1):= '0';
3.   CompVert(l,M,Y1);
/* By means of the slice Y1, we save vertices not reachable
   from  $v_1$  after deleting the edge from  $T$ . */
4.   r:=NUMB(Y); h:=NUMB(Y1);
5.   if h≤r/2 then ConEdges(left,right,code,S,Y1,Q)
/*Positions of edges joining two connected components are saved
   in the slice Q. */
6.   else begin Y2:=Y and (not Y1);
7.             ConEdges(left,right,code,S,Y2,Q)
8.           end;
9.   MIN(weight,Q,X);
10.  k:=FND(X);
/* We define the position of an edge inserted in  $T$ . */
11.  T(k):= '1';
12.  OldRoot(left,right,code,Y1,l,del);
13.  NewRoot(left,right,code,M,Y1,k,ins,W);
14.  X:=COL(ins,M); Z:=COL(del,M);
15.  P:=X xor Z;
/* In the slice P, we save the positions of edges that belong
   to the path joining vertices  $v_{del}$  and  $v_{ins}$ . */
16.  TreePaths(left,right,code,n,ins,del,M,P,W,Y1)
17. End;
```

Remark 2. By Lemma 1 from [1], if an edge is deleted from a given MST, then each of the resulting components is a minimum spanning tree induced by its vertices.

Claim 1. *Let an undirected graph G be given as a list of triples, a matrix code save binary representations of vertices, and a slice Y save positions of vertices. Let v_i and v_j be end-points of an edge deleted from the minimum spanning tree T . Then the procedure DelEdge returns the current slice S for the graph G , the current MST T , and the current matrix of tree paths M .*

Sketch of the proof. We first prove that the procedure DelEdge returns the current MST T . This is proved by contradiction. Let all assumptions of the claim be true. However, the spanning tree obtained from the given T after deleting the edge with end-points v_i, v_j and adding a new edge is not a minimum spanning tree. We will show that this contradicts to execution of the procedure DelEdge.

Really, on performing lines 1–3, the deleted edge position l is marked by '0' in slices T and S , and vertices not reachable from v_1 after deleting this edge are marked by '1' in the slice $Y1$.

On performing lines 4–8, *positions* of edges joining two connected components are marked by '1' in the slice Q . Since $Y1$ and $Y2$ include the same set of edges with a single end-point in them, the smaller of these components is used to determine Q . On fulfilling lines 9–11, the minimum weight edge joining connected components is defined and its position is included in T . Therefore, taking into account Remark 2, we obtain the current MST. This contradicts to the assumption.

Now, we check that the procedure DelEdge returns the current matrix of tree paths M . On performing lines 12–15, we determine vertices v_{del} and v_{ins} from $Y1$, the *new* tree path joining v_1 and v_{ins} , and a tree path joining v_{del} and v_{ins} . On performing line 16, the new tree paths for all vertices from $Y1$ are written in the matrix M . $\square$

Let us evaluate time complexity of DelEdge. We first note that in the worst case the procedures ConEdges and TreePaths take $O(h \cdot \log n)$ time each, where h is the number of vertices in the connected component $Y1$. Other auxiliary procedures take $O(\log n)$ time each. Therefore, DelEdge takes $O(h \cdot \log n)$ time. The factor $\log n$ arises due to the use of MATCH. In [8], the procedure MSTPD for finding an MST of an undirected graph takes $O(n \cdot \log n)$ time on the STAR-machine having no less than m PEs.

7 Associative Parallel Algorithm for Edge Insertion

As shown in [1], if a new edge is added to G , then the new MST is obtained by adding the new edge to the current MST and deleting the largest edge in the cycle created. Here, we propose an associative parallel algorithm for dynamic updating the current MST after insertion of an edge in the underlying graph G .

Let v_i and v_j be end-points of an edge being inserted in G . The algorithm runs as follows. It first determines the *position* k of an edge being added to G . Then, it defines *positions* of tree edges joining end-points of this edge. Further, it determines *position* l of the maximum weight edge in the cycle created.

If $k \neq l$, the algorithm carries out the following steps. First, it sets '0' in the l -th position of the slice T and '1' in its k -th position. Then, it defines the connected component $Y1$ whose vertices are not reachable from v_1 after deleting an edge from T . Finally, it recomputes tree paths for vertices from $Y1$.

Let us present the procedure InsertEdge.

```

procedure InsertEdge(left,right,weight: table; code: table;
    i,j,n: integer; var T: slice(left); var M: table);
var X,Z: slice(left);
    Y1: slice(code);
    k,l: integer;
1. Begin EdgePos(left,right,code,T,i,j,k);
/* We define the position of the edge being inserted in G. */
2. X:=COL(i,M); Z:=COL(j,M);
3. X:=X xor Z;
/* In the slice X, we save positions of tree edges joining  $v_i$  and  $v_j$ . */
4. X(k):='1';
5. MAX(weight,X,Z);
6. if Z(k)='0' then
7.     begin l:=FND(Z);
/* We define the position of the maximum weight edge in the cycle. */
8.         T(l):='0'; T(k):='1';
9.         CompVert(l,M,Y1);
10.        OldRoot(left,right,code,Y1,l,del);
11.        NewRoot(left,right,code,M,Y1,k,ins,W);
12.        X:=COL(ins,M); Z:=COL(del,M);
13.        P:=X xor Z;
14.        TreePaths(left,right,code,n,ins,del,M,P,W,Y1)
15.    end;
16. End;
    
```

Correctness of the procedure InsertEdge is established in the same manner as for the procedure DelEdge.

8 Conclusions

In this paper, we have proposed two associative parallel algorithms for the dynamic edge update of an MST in an undirected graph G represented as a list of triples. As a model of parallel computation, we have used the STAR-machine that simulates the run of associative parallel systems of the SIMD type with vertical data processing. For the dynamic edge update of an MST, we have used, in particular, a matrix of tree paths consisting of m rows and n columns. We have shown that initially the MST of the underlying graph is built along with the matrix of tree paths. We have also proposed a new associative parallel algorithm to perform local changes in the matrix of tree paths each time after deletion or insertion of an edge in G . Let us enumerate main advantages of the proposed

algorithms. First, after deleting an edge from the MST, the corresponding connected components are easily determined. Second, to define positions of edges joining two connected components, the smaller of them is used. Third, by means of the current matrix of tree paths, we easily define positions of edges forming a cycle after adding a new edge to G . Fourth, by means of the basic procedures MAX and MIN, we easily determine both the maximum weight edge in the cycle created and the minimum weight edge joining two connected components.

We are planning to explore associative parallel algorithms for dynamic updates of a batch of edges and for the dynamic vertex update of a minimum spanning tree.

References

1. Chin, F., Houck D.: Algorithms for Updating Minimum Spanning Trees. In: J. of Computer and System Sciences, Vol. 16 (1978) 333–344
2. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, A.: Sparsification – A Technique for Speeding Up Dynamic Graph Algorithms. In: J. of the ACM, Vol. 44, No. 5 (1997) 669–696
3. Foster, C.C.: Content Addressable Parallel Processors. Van Nostrand Reinhold Company, New York (1976)
4. Frederickson, G.: Data Structure for On-line Updating of Minimum Spanning Trees. In: SIAM J. Comput., Vol. 14 (1985) 781–798
5. Krikelis, A., Weems, C.C.: Associative Processing and Processors. IEEE Computer Society Press, Los Alamitos, California, (1997)
6. Nepomniaschaya, A.S.: Language STAR for Associative and Parallel Computation with Vertical Data Processing. In: Mirenkov, N.N. (ed.): Proc. of the Intern. Conf. “Parallel Computing Technologies”, World Scientific, Singapore (1991) 258–265
7. Nepomniaschaya, A.S., Dvoskina, M.A.: A Simple Implementation of Dijkstra’s Shortest Path Algorithm on Associative Parallel Processors. In: Fundamenta Informaticae, IOS Press, Vol. 43 (2000) 227–243
8. Nepomniaschaya, A.S.: Comparison of Performing the Prim-Dijkstra Algorithm and the Kruskal Algorithm on Associative Parallel Processors. In: Cybernetics and System Analysis, Kiev, Naukova Dumka, No. 2 (2000) 19–27 (in Russian. English translation by Plenum Press)
9. Pawagi, S., Kaser, O.: Optimal Parallel Algorithms for Multiple Updates of Minimum Spanning Trees. In: Algorithmica, Vol. 9 (1993) 357–381
10. Pawagi, S., Ramakrishnan, I.V.: An $O(\log n)$ Algorithm for Parallel Update of Minimum Spanning Trees. In: Inform. Process. Lett., Vol. 22 (1986) 223–229
11. Potter, J.L.: Associative Computing: A Programming Paradigm for Massively Parallel Computers. Kent State University, Plenum Press, New York and London (1992)
12. Spira, P., Pan, A.: On Finding and Updating Spanning Trees and Shortest Paths. In: SIAM J. Comput., Vol. 4 (1975) 375–380

The Renaming Problem as an Introduction to Structures for Wait-Free Computing

Michel Raynal

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
raynal@irisa.fr

Abstract. The aim of this introductory survey paper is twofold: to be an introduction to wait-free computing and present the renaming problem. “*Wait-free*” means that the progress of a process depends only on it, regardless of the other processes (that can progress slowly or even crash). It is shown that the design of wait-free algorithms rests on the definition and the use of appropriate data/control structures. To illustrate such structures, the paper considers the *renaming problem* where the processes have to acquire new names from a small bounded space despite possible process crashes. Two renaming algorithms are presented. The first is a protocol due to Moir and Anderson; it is based on a grid of splitters. The second is due to Attiya and Fouren; it is based on a network of reflectors. It appears that splitters and reflectors are basic data/control structures that permit to define switching networks well-suited to wait-free computing.

Keywords: Atomic register, Concurrency, Fault-tolerance, Non-blocking synchronization, Process crash, Attiya-Fouren’s reflector, Renaming problem, Shared memory system, Lamport-Moir-Anderson’s splitter, Wait-free computation.

1 Introduction

A *concurrent object* is a data structure shared by asynchronous concurrent processes. An implementation of a concurrent object is *wait-free* if it guarantees that any process will complete any operation in a finite number of steps, regardless the execution speed of the other processes. This means that a process terminates in a finite number of steps, even if the other processes are very slow or even stop taking steps completely. The “*wait-free*” property is a very desirable property when one has to design concurrent objects that have to cope with processes that can encounter unexpected delays (e.g., due to swapping or scheduling policy) or prematurely crash. Wait-free computing has first been introduced by Lamport [10], and then developed by several authors (e.g., [16]). A theory of wait-free computing is described in [8].

Wait-free computing rules out many conventional synchronization techniques such as busy waiting, conditional waiting or critical sections. That is an immediate consequence of the fact that the arbitrary delay of a single process within

the critical section would make the progress of the other processes dependent on its speed, or could even prevent them from progressing in case it crashes within the critical section. It is also important to notice that some synchronization problems cannot be solved in a wait-free manner (that is the case of a process that has to wait a signal from some other process in order to progress).

This paper is an introduction to wait-free computing, with a particular emphasis on data structures that allows the implementation of wait-free operations. Basically, the aim of the paper is to show that the design of wait-free operations relies on the “discovery” of appropriate data/control structures. To this end, the paper addresses the *renaming problem* and presents two such structures, each allowing the design of a wait-free solution to this problem.

The *renaming problem* has been introduced in [2] in the context of unreliable asynchronous message-passing systems. Then, it has received a lot of attention in the context of shared memory systems (e.g., [1,3,4,5,13,14]). Informally, it consists in the following. Each of the n processes that define the system has a distinct name taken from an unbounded domain. The processes have to cooperate to choose new names from a name space of size M such that no two processes get the same name. (A simple application of the renaming problem is when the processes perform a computation whose time complexity is dependent on the size of their name space. By first using a renaming algorithm to reduce their name space, the time complexity can be made independent of the original name space [14].)

The renaming problem is trivial when no process can commit a crash failure. Differently, it has been shown that there is no solution to the M -renaming problem when $M < n + f$, where f is an upper bound on the number of processes that can crash [9]. As noticed previously, several renaming protocols have been designed for shared memory systems. Basically, the processes compete to acquire new (distinct) names. The net effect of process asynchrony and process crashes creates an uncertainty on the system state that a renaming protocol has to cope with. The fact that, additionally, the solution has to be wait-free makes the problem far from being trivial.

To illustrate wait-free solutions to the M -renaming problem, the paper considers two algorithms. The first one, due to Moir and Anderson [14], solves the problem for $M = n(n + 1)/2$. It is based on a grid of *splitters*, a new data/control structure specially suited to wait-free computing. (This data structure has been initially used by Lamport to solve fast mutual exclusion [11]. It has then been identified as a basic object by Moir and Anderson.) The second algorithm, due to Attiya and Fouren [4], is more intricate. It solves the problem for $M = 2n - 1$. (Let us notice that this value of M is optimal for a wait-free solution, as “wait-free” means that f can be as high as $n - 1$, and there is no solution for $M < n + f$). This algorithm is based on a network of *reflectors* (a data/control structure introduced by Attiya and Fouren). Interestingly, it appears that splitters and reflectors are basic structures from which it is possible to design appropriate “switching networks” through which

the processes navigate in a way-free manner (thereby cooperating in an implicit way) to eventually produce their results.

The paper is made up of five sections. Section 2 introduces the computation model. Then, Sections 3 and 4 presents Moir-Anderson's algorithm, and Attiya-Fouren's algorithm, respectively. Section 5 concludes the paper. For completeness, an appendix provides a solution to the renaming problem in a message-passing system.

2 Computation Model and the Renaming Problem

Computation model. We consider a standard asynchronous shared memory system with n processes ($n > 1$), where at most f ($0 \leq f \leq n - 1$) may crash. A nonfaulty (or correct) process is a process that never crashes. A faulty process executes correctly (i.e., according to its specification) until it crashes. After having crashed, a process executes no operation (i.e., its state is no longer modified).

The shared memory consists of multi-writer/multi-reader atomic registers (also named shared variables). A process p_i can have local variables: those are private in the sense p_i is the only process that can read or write them. The index i associated with the process p_i is only used for notational convenience; more precisely, a process p_i does not know its index i . For more details on the computation model see any standard textbook [5,12].

The renaming problem. Let us assume that the n processes have arbitrarily large (and distinct) initial names $id_1, \dots, id_n \in [0..N - 1]$, where $n \ll N$. In the M -renaming problem, the processes are required to get new names in such a way that the new names belong to the set $\{0, \dots, M - 1\}$, $M \ll N$, and no two processes get identical names [2]. More formally, the problem is defined by the three following properties:

- **Termination.** Each correct process decides a new name.
- **Validity.** A decided name belongs to $[0..M - 1]$.
- **Agreement.** No two processes decides the same name.

This formulation corresponds to the *one-time* renaming problem. In the *long-lived renaming* problem [14], the processes are provided with two operations, namely `get_name` and `release_name`. A process can use a name only during a finite period and then releases it; a name that has been released can be used again by another process. Long-lived renaming is actually a resource allocation problem (the names are the resources) that has to be solved by a wait-free algorithm.

Remark. Let us observe that any solution to the *consensus* problem [6,7] allows to solve the one-time renaming problem as follows. First, the consensus solution is used to provide an *atomic broadcast* protocol (as shown in [6]). Then, each process

uses the atomic broadcast to send a message. As the atomic broadcast ensures that all processes get these messages in the same order, a process considers x as its new name if the x -th message it receives is its own message. Interestingly, this consensus-based solution provides an algorithm where $M = n$ (this does not contradict the $M \geq n + f$ requirement as consensus requires additional assumptions to be solved [6,7,15]).

3 Moir-Anderson's Protocol

Moir-Anderson's wait-free solution is surprisingly simple and elegant. It uses a building block called *splitter*. As noted in the Introduction, this building block has been initially introduced by Lamport to provide fast mutual exclusion [11].

A basic building block. A *splitter* is a concurrent object that assigns one out of three values (*stop*, *down* or *right*) to a local variable $move_i$ of the invoking process p_i . A splitter is characterized by the following global property: if x processes access the splitter, then at most one receives the value *stop*, at most $x - 1$ receive the value *down*, and at most $x - 1$ receive the value *right* (Figure 1).

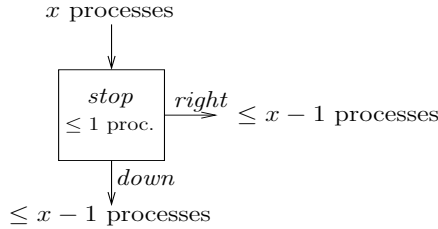


Fig. 1. A Splitter

A wait-free implementation of a splitter is described in Figure 2. Its internal state is represented by two shared global variables: X (whose content will be process ids) and Y (a boolean initialized to *false*). It can be accessed by the function `splitter()` that returns a value to the invoking process.

As there is no loop, the splitter is trivially wait-free. Let us assume that x processes access the splitter object. Let us first observe that, due to the initialization of Y , not all of them can get the value *right* (for a process to obtain *right*, another process has to first set Y to *true*). Let us now consider the last process that executes line 1. If it does not crash, this process cannot get the value *down* (due to line 4), hence, not all processes can get the value *down*. Finally, no two process can get the value *stop*. Let p_i be the first process that finds $X = id_i$ at line 4 (consequently, p_i gets the value *stop* if it does not crash). This means that no process p_j has modified X while p_i was executing the lines

```

function splitter ()
(1)   $X \leftarrow id_i$ ;
(2)  if  $Y$  then return (right)
(3)      else  $Y \leftarrow true$ 
(4)          if  $(X = id_i)$  then return (stop)
(5)              else return (down)
(6)  endif      endif

```

Fig. 2. A Wait-Free Implementation of a Splitter

1-4. It follows that any $p_j \neq p_i$ that will modify X (at line 1) will find $Y = true$ (at line 2), and consequently cannot get the value *stop*.

A process that moves right is actually a *late* process: it arrived late at the splitter and found $Y = true$. Differently, a process that moves down is actually a *slow* process: it set $Y \leftarrow true$ but was not quick enough during the period that started when it updated X (line 1), and ended when it read X (line 4). At most one process can be neither late nor slow, it is *on time* (and gets *stop*).

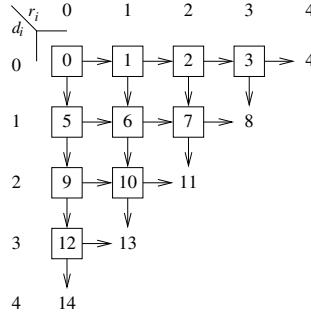


Fig. 3. A Renaming Grid

A grid of renaming splitters. The elegance and simplicity of Moir-Anderson's solution consists in a grid made up of $n(n-1)/2$ renaming splitters (Figure 3 depicts such a grid for $n = 5$). A process p_i first enters the left corner of the grid. Then, it moves along the grid according the values it obtains from the splitters (*down* or *right*) until it gets the value *stop*. Finally, it considers as its new name the value associated with the splitter where it stopped. The property attached to each splitter ensures that no two processes stop at the same splitter.

The resulting Moir-Anderson protocol is described in Figure 4 (its writing is inspired from [4]). Process p_i invokes `splitter( $d_i, r_i$ )` to access the splitter identified $[d_i, r_i]$ in the grid (the shared global variables $X[d_i, r_i]$ and $Y[d_i, r_i]$, initialized to *false*, are associated with this splitter). It follows from the property of the splitters that no process takes more than $(n-1)$ iteration steps. It is relatively

easy to see that the worst case time complexity is $4(n - 1)$ (maximum number of shared memory accesses). An assertional proof of the protocol can be found in [14].

```

(1)  $d_i \leftarrow 0$ ;  $r_i \leftarrow 0$ ;  $move_i \leftarrow down$ ;
(2) while ( $move_i \neq stop$ ) do
(3)    $move_i \leftarrow \text{splitter}(d_i, r_i)$ ;
(4)   case ( $move_i = right$ ) then  $r_i \leftarrow r_i + 1$ 
(5)     ( $move_i = down$ ) then  $d_i \leftarrow d_i + 1$ 
(6)     ( $move_i = stop$ ) then exit_loop
(7)   endcase
(8) endwhile
(9) return ( $n \times d_i + r_i - (d_i(d_i - 1)/2)$ )
      % the new name is the position  $[d_i, r_i]$  in the grid %

```

Fig. 4. Wait-Free $n(n + 1)/2$ Renaming

4 Attiya-Fouren's Protocol

The aim of Attiya and Fouren's protocol is to solve the $(2n - 1)$ -renaming problem, i.e., provide an algorithm that is optimal with respect to the value of M (the size of the new name space). To attain this goal, the protocol is based on a new appropriate data structure (*reflector*) [4]. The price that the algorithm has to pay to attain this goal, lies in its time complexity which is $O(N)$ (where N is the size of the initial space name).

A basic building block. Differently from a splitter, a *reflector* has two entrances in_0 and in_1 , each connected to two exits, in_0 connected to up_0 and $down_0$, and in_1 connected to up_1 and $down_1$, respectively (see Figure 5). A process entering a reflector on entrance in_x leaves it on exit up_x or $down_x$. The goal is to allow a process to change its direction according to its entrance and the fact another process has already accessed the reflector. The rules associated with a reflector are the following:

- (1) If a single process enters the reflector, it leaves it on a down exit,
- (2) If two processes enter the reflector each on a different entrance, at most one of them leaves it on a down exit.

Let us notice that it is possible that several processes enter the reflector and leave it on upper exits. A very simple implementation of a reflector is described in Figure 6; it rests on a 2-entry boolean array *visited*[0..1] (each entry being initialized to *false*). It is easy to see that a process p_i proceeds from in_y to $down_y$, except if another process has already passed (or is currently passing) through the other entrance of the reflector, in which case p_i proceeds to up_y .

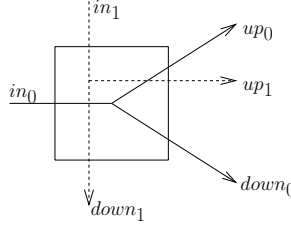


Fig. 5. A Reflector

```

function reflector (entrance  $y : 0, 1$ )
(1)   $visited[y] \leftarrow true$ ;
(2)  if ( $\neg visited[1 - y]$ ) then return ( $down_y$ )
(3)  else return ( $up_y$ )  endif

```

Fig. 6. An Implementation of a Reflector

A network of reflectors. Similarly to the previous protocol, the idea is to provide a network of base objects that, when traversed by processes, ensures that these processes will get new distinct names (from a small space name) at the end of their traversal. But differently from the previous protocol where all the processes enter the grid at the same splitter, here each process enters an associated reflector. The clever idea introduced by Attiya and Fouren lies in the introduction and the use of reflectors: their network is such that at most one process enter a reflector on each of its entrances. This allows the corresponding reflector to manage the name assignment conflict between these two processes.

More specifically, the network designed by Attiya and Fouren consists of N columns, numbered from 0 to $N - 1$, left to right (remind that $[0..N - 1]$ is the initial space name). Column c contains $2c - 1$ reflectors numbered $c, c - 1, \dots, 0, \dots, -(c - 1), -c$ from top to bottom. The idea of the protocol is for a process p_i , whose initial name $id_i = c \in [0..N - 1]$, to traverse the network from left to right starting from the reflector $R[c, c]$ until a reflector $R[x, N - 1]$ of the last column, and then consider as its new name the last row x on which it arrived.

In order the new name space be bounded by $M = 2n - 1$, the traversal has to ensure that processes cannot terminate with their last rows too far apart one from the other. If x_1 and x_2 are the two extreme rows on which processes terminate, we must have $|x_1 - x_2| \leq 2n - 1$. To attain this goal, the reflectors are connected as follows. Considering a reflector $R[r, c]$, we have:

- The exit up_0 is connected to entrance in_0 of $R[r + 1, c + 1]$,
- The exit up_1 is connected to entrance in_0 of $R[r, c + 1]$,
- The exit $down_0$ is connected to entrance in_0 of $R[r - 1, c + 1]$,
- For the $down_1$ exit there are two cases. If $r > -c$, $down_1$ is connected to entrance in_1 of $R[r - 1, c]$ (to allow a process to descend along its column). Otherwise $r = -c$, and $R[-c, c]$ is the last reflector of column c ; $down_1$ it then is connected to entrance in_0 of $R[-(c + 1), c + 1]$.

These connections are depicted on the right part of Figure 7. The resulting network of reflectors is depicted in its left part (the point whose coordinates are (r, c) corresponds to the reflector $R[r, c]$).

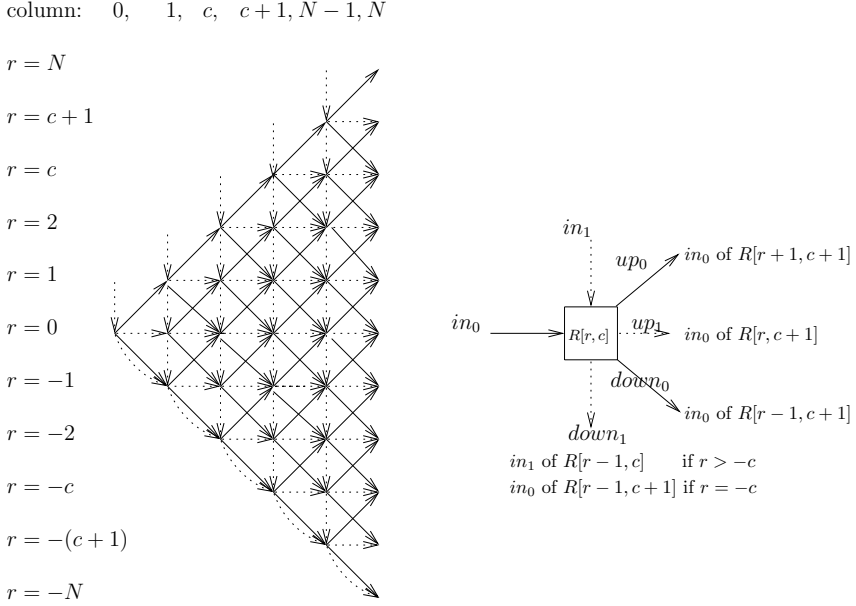


Fig. 7. The Network of Reflectors

Process p_i , with initial name $id_i = c \in [0..N-1]$, starts from entrance in_1 of the reflector $R[c, c]$. Then, it descends through column c (i.e., from $down_1$ of reflector $R[x, c]$ to the entrance in_1 of reflector $R[x-1, c]$) until it attains a reflector $R[x, c]$ that has already been visited (or is currently visited) by another process. In that case, p_i moves right through up_1 of $R[x, c]$ to the in_0 entrance of the reflector $R[x, c+1]$. If it attains the last reflector of column c (namely, the reflector $R[-c, c]$) without having traversed a reflector already visited, p_i progresses to the in_0 entrance of reflector $R[-(c+1), c+1]$. When, it attains column $N-1$, p_i terminates and considers its last row number as its new name. Attiya-Fouren's protocol is described in Figure 8 where $\text{reflector}(r, c, y)$ stands for the invocation of the reflector $R[r, c]$ on entrance in_y .

The fundamental and noteworthy property resulting from the net effect of each reflector and the way they are connected, is the following [4]. Let S_c be the set of processes whose initial names belong to $[0..c]$. Those are the processes that start in columns $\leq c$. The processes in S_c (that do not crash) enter column $c+1$:

- (1) on *distinct rows*, and
- (2) among the lowest $2|S_c| - 1$ ones.

```

(1)  $c_i \leftarrow id_i; r_i \leftarrow id_i;$ 
(2) while ( $c_i = id_i$ ) do
(3)    $exit \leftarrow \text{reflector}(c_i, r_i, 1);$ 
(4)   if ( $exit = up_1$ ) then  $c_i \leftarrow c_i + 1$ 
(5)   else  $r_i \leftarrow r_i - 1;$ 
(6)   if  $r_i < -c_i$  then  $c_i \leftarrow c_i + 1$  endif
(7)   endif
(8) endwhile;
(9) while ( $c_i < N$ ) do
(10)   $exit \leftarrow \text{reflector}(c_i, r_i, 0);$ 
(11)   $c_i \leftarrow c_i + 1;$ 
(12)  if ( $exit = up_0$ ) then  $r_i \leftarrow r_i + 1$ 
(13)  else  $r_i \leftarrow r_i - 1$ 
(14)  endif
(15) endwhile;
(16) return ( $r_i$ )      %  $0 \leq r_i + N \leq 2(n-1)$  %

```

Fig. 8. Wait-Free $(2n - 1)$ -Renaming

It follows from this property that at most one process enters a reflector on each entrance, and the processes that attain column $N - 1$, decide distinct names and those names belong to $[-N.. -N + 2(n - 1)]$ (a simple translation provides non-negative names).

As an exercise and to get a better insight into this property the reader can consider the following two “extreme” cases.

- The processes execute the renaming protocol one after the other, in the increasing order of their initial names¹. In that case, the process with the lowest initial name gets $-N$ as new name, the process with the second lowest initial name gets $-N + 1$ as new name, the process with the third lowest initial name gets $-N + 2$ as new name, etc., If no process crashes, the process with the largest initial name will get new name $-N + (n - 1)$. This is a case where the actual name range is the smallest. Such a scenario is depicted in the left part of Figure 9, where the initial names of p_1 , p_2 and p_3 are respectively a , b and c with $a < b < c$. If, sequentially, p_1 gets first its name, then p_2 and finally p_3 , the paths followed by p_1 , p_2 and p_3 are the ones indicated in bold, dash and dash-dot, respectively, providing the new names $-N$ to p_1 , $-N + 1$ to p_2 , and $-N + 2$ to p_3 . The reflectors that are circled are the ones where the processes conflict. As we can see, p_1 and p_2 conflict. Similarly, p_2 and p_3 conflict, but p_1 and p_3 do not conflict.

- The processes execute the renaming protocol one after the other, in the decreasing order of their initial names. In that case, the process with the largest initial name gets $-N$ as new name, the process with the second largest initial

¹ “One after the other” means that a process starts requesting a name only after the previous one has got its new name. Using the terminology introduced in Section 3, the second process is *late* with respect to the first, etc.

name gets $-N + 2$ as new name, the process with the third largest initial name gets $-N + 4$ as new name, etc. If no process crashes, the process with the lowest initial name will get new name $-N + 2(n - 1)$. This is a case where the actual name range is the largest. Such a scenario is depicted in the right part of Figure 9.b, where the initial names of p_1 , p_2 and p_3 are respectively c , b and a with $a < b < c$. If p_1 is the first to execute the protocol, followed by p_2 , and then p_3 , the paths followed by p_1 , p_2 and p_3 are the ones indicated in bold, dash and dash-dot, respectively, providing the new names $-N$ to p_1 , $-N + 2$ to p_2 , and $-N + 4$ to p_3 .

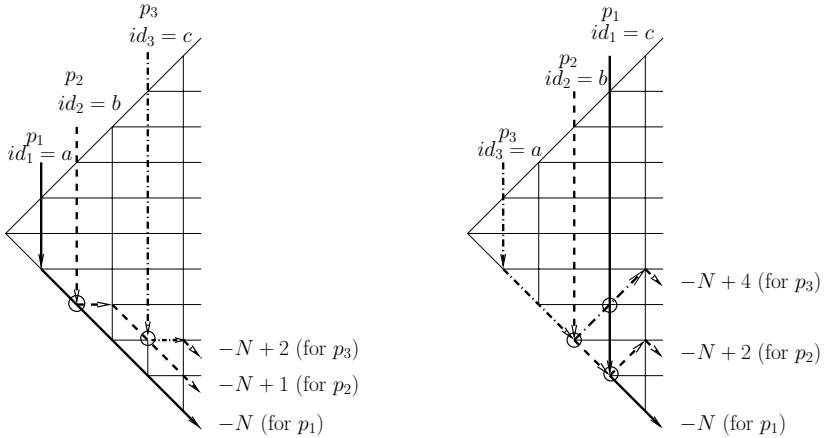


Fig. 9. Paths in the Network of Reflectors

These two extreme cases help better understand how name conflicts are handled by the protocol. (The other possible cases are combinations of these two scenarios.) The network of reflectors actually ensures that the number of conflicts between processes to get names is upper bounded by $2n - 1$.

5 Conclusion

The aim of this paper was to introduce a few techniques encountered in wait-free computing. Wait-free computing is particularly attractive as it naturally copes with process crashes: the progress of a process cannot be prevented by the other processes. To this end, the paper has considered a particular problem, namely, the M -renaming problem in a set of n processes, and has presented two wait-free renaming protocols (M is the size of the new name space). Both protocols are based on a network of wait-free base objects appropriately interconnected.

The first protocol uses a grid of $O(n^2)$ splitters and solves the M -renaming problem for $M = n(n + 1)/2$. Its time complexity is $O(n)$. The second protocol

uses a network of $O(N^2)$ reflectors and solves the M -renaming problem for $M = 2n - 1$. Its time complexity is $O(N)$. (N the size of the initial name space.)

It is interesting to notice the tradeoff among these protocols. The first has a smaller network size ($O(n^2)$), but generates more conflicts in name assignment, which results in a greater value for M . The second uses a bigger network size ($O(N^2)$), generates less conflicts in name assignment, which results in an optimal value for M , namely, $2n - 1$. Actually, the range of the new name space is equal to the number of conflicts in name assignment allowed by the corresponding protocol. Each protocol addresses this issue in its own way: all processes share the same entry (splitter) in Moir-Anderson's protocol, while (at the price of a bigger network) each process has its own entry (reflector) in Attiya-Fouren's protocol.

The reader interested in the renaming problem will find other protocols in the list of references. A protocol is *adaptive* if its time complexity depends only of the number $k \leq n$ of processes participating in the protocol. Differently from n , such a parameter k is unknown in advance and may change from one execution to another. The reader interested in adaptive long-lived renaming can consult [3] (where another appropriate data structure is introduced, namely, a *sieve*).

References

1. Afek Y. and Merritt M., Fast, Wait-Free $(2k - 1)$ -Renaming. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, ACM Press, pp. 105–112, Atlanta (GA), 1999.
2. Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524–548, 1990.
3. Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2k - 1)$ -Renaming. *Proc. Symposium on Distributed Computing (DISC'00)*, Springer-Verlag Lecture Notes in Computer Science, Vol. 1914, pp. 149–163, Toledo (Spain), 2000.
4. Attiya H. and Fouren A., Adaptive and Efficient Algorithms for Lattice Agreement and Renaming. *SIAM Journal of Computing*, 31(2):642–664, 2001.
5. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, McGraw-Hill, 451 pages, 1998.
6. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
7. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
8. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
9. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858–923, 1999.
10. Lamport L., Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811, 1977.
11. Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
12. Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

13. Moir M., Fast, Long-Lived Renaming Improved and Simplified. *Science of Computer Programming*, 30:287–308, 1998.
14. Moir M. and Anderson J.H., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25:1–39, 1995.
15. Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM Symp. on Theory of Computing (STOC'01)*, ACM Press, pp. 153–162, 2001.
16. Peterson G.L., Concurrent Reading while Writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

Appendix: Renaming in Message-Passing Systems

This appendix considers the renaming problem in message passing systems. It has a “completeness” motivation, and is for readers interested in the renaming problem.

As noticed in the Introduction, the renaming problem has first been introduced in the context of unreliable asynchronous distributed systems [2]. The problem was to find a non-trivial agreement problem that can be solved in presence of up to $f < n/2$ faulty processes². [2] states the problem, analyzes it, and provides several message-passing renaming protocols. This appendix presents one of these protocols which solves the M -renaming problem in presence of up to $f < n/2$ process crashes, for $M = (n - f/2)(f + 1)$. ([2] presents another protocol that, under the same assumptions, provides $M = n + f$. Unfortunately, that protocol is much more intricate.) Let us also observe that, as processes have to wait for messages, message-passing protocols are not wait-free.

In this protocol each process p_i manages a set V_i containing the initial names (id_j) it knows from the other processes. (Remind that process indexes are used for exposition, they are not known by the processes. Initially, a process knows only n and its initial name id_i .) Each time it learns new initial names, p_i propagates them to the other processes. To this end it uses the “**broadcast** $new(V_i)$ ” operation which is a shorthand for “**send** $new(V_i)$ to all processes (including itself)” (notice that a process can crash in the middle of such a statement). When it receives a message $new(V)$, there are three cases.

- $V \subset V_i$ (with $V \neq V_i$). In that case, p_i learns nothing. It simply discards the message.
- $V - V_i \neq \emptyset$. In that case, p_i learns new initial names. It updates accordingly V_i and consequently issues **broadcast** $new(V_i)$.
- $V = V_i$. In that case p_i learns that one more process knows the same set of initial names as it knows. So, p_i manages a counter ct_i to count the number of processes that know the same set V as it knows.

² Differently from the consensus problem that cannot be solved in presence of even a single process crash in purely asynchronous systems [7].

As f processes can crash, p_i cannot expect to receive the same set V from more than $n - f$ processes (including itself). So, when $ct_i = (n - f)$, p_i decides its new name. It is the pair $\langle |V_i|, \text{rank of } id_i \text{ in } V_i \rangle$. The protocol is described in Figure 10.

```

 $V_i \leftarrow \{id_i\}; ct_i \leftarrow 0; decided_i \leftarrow false;$ 
broadcast new( $V_i$ );
while ( $\neg decided_i$ ) do
  wait until receive new( $V$ );
  case ( $V \subset V_i$ )      then  $V$  carries old information: discard it
    ( $V = V_i$ )        then % one more process knows exactly the same %
       $ct_i \leftarrow ct_i + 1;$ 
      if ( $ct_i = n - f$ )
        then %  $V_i$  is stable %
          let  $v = |V_i|; r = \text{rank of } id_i \text{ in } V_i;$ 
          new name =  $\langle v, r \rangle;$ 
           $decided_i \leftarrow true$ 
        endif
      ( $V - V_i \neq \emptyset$ ) then %  $p_i$  learns initial names %
        % Let  $p_j$  be the sender of new( $V$ ) %
        case ( $V_i \subset V$ ) then %  $p_j$  knows  $V_i \cup V$  %
           $ct_i \leftarrow 1$ 
           $\neg(V_i \subset V)$  then %  $p_j$  doesn't know  $V_i \cup V$  %
             $ct_i \leftarrow 0$ 
        encase;
         $V_i \leftarrow V_i \cup V;$ 
        broadcast new( $V_i$ )
      endcase
    endwhile;
  while ( $true$ ) do
    wait until receive new( $V$ );
     $V_i \leftarrow V_i \cup V;$ 
    broadcast new( $V_i$ )
  endwhile

```

Fig. 10. A Message-Passing Renaming Protocol

Let a set V be *stable* if a process received $n - f$ copies of it (so, this process decides its new name from this set). A main property of the protocol is the following: Stable sets are totally ordered (by inclusion). This follows from the fact that if $V1$ is stable for p_i (i.e., p_i has received $new(V1)$ from $n - f$ processes) and $V2$ is stable for p_j (i.e., p_j has received $new(V2)$ from $n - f$ processes), then due the assumption $2f < n$, there is at least one process p_k from which p_i has received $new(V1)$ and from which p_j has received $new(V2)$. So, $V1$ and $V2$ are values taken by the set variable V_k . As a set variable V_k can only increase, it

follows that $V1 \subseteq V2$ or $V2 \subseteq V1$. This property allows to conclude no two decided names are the same.

Let us notice that a set V_i contains at most n initial names. So, a process sends its set V_i at most n times. It follows that the algorithm terminates, and its message complexity is bounded by $O(n^3)$. The proof that each correct process decides follows from the fact that each set V_i can only increase and has an upper bound (whose value V_max depends on the execution).

As indicated, the size of the new name space is $M = (n - f/2)(f + 1)$. This come from the following observation [2]. A new name is a pair $\langle v, r \rangle$. Due to the protocol text, we trivially have $n - f \leq v \leq n$. Moreover, r is the rank of the deciding process p_i in the set V_i containing v values. It follows that $1 \leq r \leq v$. Consequently the number of possible decisions is $M = \sum_{x=n-f}^{x=n} x = (n - f/2)(f + 1)$. A fixed mapping from the $\langle v, r \rangle$ pairs to $[1..M]$ can be used to get integer names.

It is important to notice that a process that decides a new name has to continue receiving and sending messages to help the other processes to decide. This help is necessary to deal with situations where some very slow processes start participating in the protocol after some other processes have already decided. It is shown in [2] that there is no renaming protocol if a process is required to stop just after deciding its new name. That is the price required by process coordination to solve the renaming problem. When we look at the shared memory protocols described in Sections 3 and 4, the result of the process coordination is recorded in the shared variables of the grid of splitters (or the network of reflectors). As there is no such shared memory in the message-passing context, the processes have to “simulate” it by helping each other. (In a practical setting, a secondary storage -e.g., a disk- shared by the processes can be used to eliminate the second **while** loop.)

Graph Partitioning in Scientific Simulations: Multilevel Schemes versus Space-Filling Curves^{*}

Stefan Schamberger¹ and Jens-Michael Wierum²

¹ University of Paderborn, Germany
schaum@upb.de, <http://www.upb.de/>

² Paderborn Center for Parallel Computing, Germany
jmwie@upb.de, <http://www.upb.de/pc2/>

Abstract. Using space-filling curves to partition unstructured finite element meshes is a widely applied strategy when it comes to distributing load among several computation nodes. Compared to more elaborated graph partitioning packages, this geometric approach is relatively easy to implement and very fast. However, results are not expected to be as good as those of the latter, but no detailed comparison has ever been published. In this paper we will present results of our experiments comparing the quality of partitionings computed with different types of space-filling curves to those generated with the graph partitioning package Metis.

Keywords: FEM graph partitioning, space-filling curves

1 Introduction

Finite Elements (FE) are often used to numerically approximate solutions of a Partial Differential Equation (PDE) describing physical processes. The domain on which the PDE has to be solved is discretized into a mesh of finite elements, and the PDE itself is transformed into a set of linear equations defined on these elements [1], which can then be solved by iterative methods such as Conjugate Gradient (CG). Due to the very large amount of elements needed to obtain an accurate approximation of the original problem, this method became a classical application for parallel computers. The parallelization of numerical simulation algorithms usually follows the Single-Program Multiple-Data (SPMD) paradigm: Each processor executes the same code on a different part of the data. This means that the mesh has to be split into P subdomains (where P is the number of processors) and each subdomain is then assigned to one of the processors. Since iterative solution algorithms mainly perform local operations, i.e. data dependencies are defined by the mesh, the parallel algorithm only requires communication at the partition boundaries. Hence, the efficiency depends on two factors: An equal distribution of the data (work load) on the processors and

^{*} This work was partly supported by the German Science Foundation (DFG) project SFB-376 and by the IST Program of the EU under contract number IST-1999-14186 (ALCOM-FT).

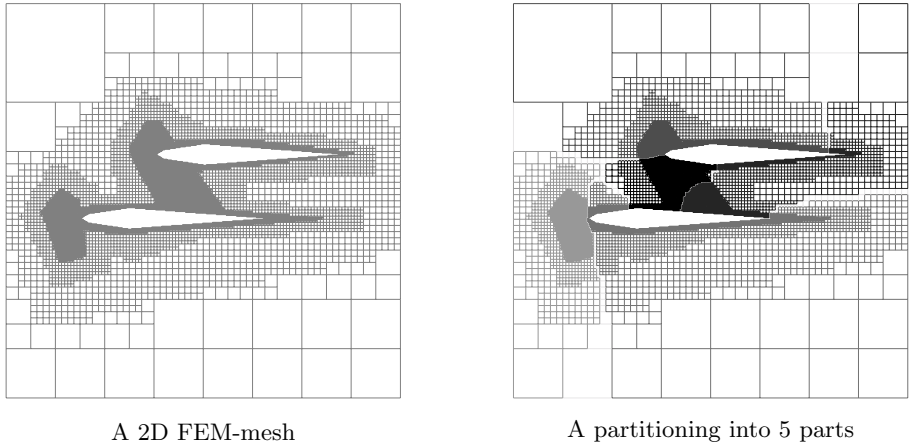


Fig. 1. Example: Applying a library to partition the 2D FEM-mesh “biplane.9” into 5 parts

a small communication overhead achieved by minimizing the number of edges between different partitions.

In practice, mainly two distinct approaches are applied to take care of this problem: Advanced partitioning tools based on sometimes quite complicated heuristics and more simplistic methods based on geometric approaches. Comparisons between these approaches have been undertaken and results are presented for example in [2,3]. However, since these publications consider a large number of partitioning approaches, the presentation of the results are somewhat comprehensive.

Especially space-filling curves, one of the geometric approaches, have not been compared extensively to other methods yet. To better understand what their advantages and disadvantages over elaborated heuristics like multilevel methods are, we will present more detailed results here. The rest of this paper is organized as follows: In the next section we will give a brief overview of the two graph partitioning approaches compared in this paper. In section 3, we define the types of space-filling curves used for our evaluations and also present some of their properties. Section 4 shows how we performed the experiments. The results are presented in section 5.

2 Related Work

Because the graph partitioning problem is known to be NP-complete, a number of heuristics have been developed and implemented in several graph partitioning libraries like Metis [4], Jostle [5], Chaco [6] or Party [7,8]. They usually follow the multilevel approach: In every level vertices of the graph are matched and a new, smaller graph with a similar structure is generated, until only a small

graph, sometimes only with P vertices, is left. The partitioning problem is then solved for this small graph and vertices in higher levels are partitioned according to their representatives in lower levels. Additionally, to improve the partition quality a local refinement phase is applied in every level. In most cases, this refinement is based on the Fiduccia-Mattheyses method [9], a run-time optimized version of the Kernighan-Lin (KL) algorithm [10]. However, also the Helpful-Set method [11] has been shown to produce good results. But since its current implementation in Party is designed (and reliable) for bisection only, we will restrict our comparison to Metis which uses a KL like algorithm. Figure 1 shows an FEM graph and its partitioning into 5 parts computed with Metis.

Another, widely applied approach to partition a mesh are geometric methods. The one we consider in this paper is based on space-filling curves. The vertices of the FE mesh are sorted by a certain recursive scheme covering the whole domain. Then, the now linear array of vertices is split into equal sized parts, each representing a partition. In contrast to partitioning heuristics, this method only works if vertex coordinates are present. It is also clear, that the quality of the generated partitioning does not excel the quality of the former mentioned elaborated heuristics since except for the coordinates any other information provided by the graph is simply ignored. This can especially be observed if the FE domain has holes which are not handled well by techniques covering the whole coordinate space. On the other hand, not relying on any other information than coordinates can be seen as a big advantage because memory requirements and run-time do decrease a lot. Furthermore, these kinds of algorithms are relatively easy to implement and provide information useful for realizing cache efficient calculations. Therefore, they are often closely coupled with the FE application. Since different kinds of space-filling curves exist, the ones used here are defined in the next section.

3 Partitioning with Space-Filling Curves

Space-filling curves are geometric representations of bijective mappings $M : \{1, \dots, N^m\} \rightarrow \{1, \dots, N\}^m$. The curve M traverses all N^m cells in the m -dimensional grid of size N . They have been introduced by Peano and Hilbert in the late 19th century [12]. An (historic) overview on space-filling curves is given in [13].

3.1 Computation of Space-Filling Curves

In figures 2-4, the recursive construction of space-filling curves is illustrated exemplarily for the Hilbert curve. Figure 2 shows the refinement rule splitting each quadrant into four subparts. The order within the quadrants has the same u-like basic pattern with some subparts being reflected and rotated. A possible algorithm calculating this refinement is sketched in figure 4. The HILBERT-function separates all given nodes in the interval $[first, last[$ into four sections and processes them recursively. The separators for the four sections are *firstcut*, *midcut*,

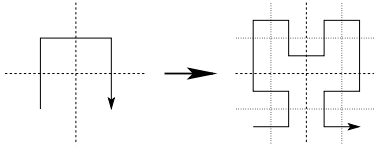


Fig. 2. Refinement rule of the Hilbert curve.

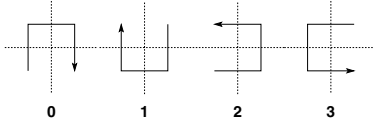


Fig. 3. Enumeration of basic pattern types of the Hilbert curve.

HILBERT (*first*, *last*, *type*):

```

orient1 = (type ÷ 2) ? x : y
orient2 = (type ÷ 2) ? y : x
dir1 = (type%2==0) ? ascend : descend
dir2 = (type%2==0) ? descend : ascend

midcut = SPLIT (dir1, orient1, first, last)
firstcut = SPLIT (dir2, orient1, first, midcut)
thirdcut = SPLIT (dir2, orient2, midcut, last)

HILBERT (first, firstcut, (type+2)%4)
HILBERT (firstcut, midcut, type)
HILBERT (midcut, thirdcut, type)
HILBERT (thirdcut, last, 3-type)

```

Fig. 4. Algorithm sketch for the recursive calculation of the Hilbert curve.

and *thirdcut*. The SPLIT-operation sorts all nodes in the specified interval according to an *orientation* (x- or y-axis) and a *direction* (ascend or descend), returning the index which represents the geometric separator. The orientation and direction are determined from the basic pattern *type* of the Hilbert curve to be generated. The given algorithm fits to the enumeration of pattern types printed in figure 3.

An overview on the indexing schemes evaluated in this paper is plotted in figure 7 (top row). In addition to the older indexing schemes of Hilbert, Lebesgue, and Sierpiński we have examined the $\beta\Omega$ -indexing. The refinement rules for all curves can be found in [13,14].

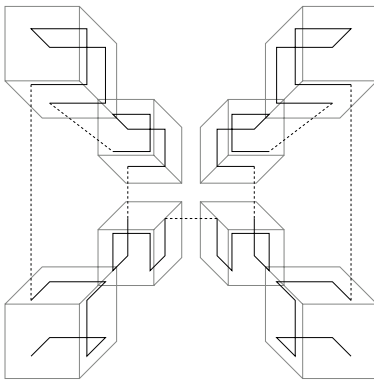


Fig. 5. 3-dimensional Hilbert curve used for evaluations.

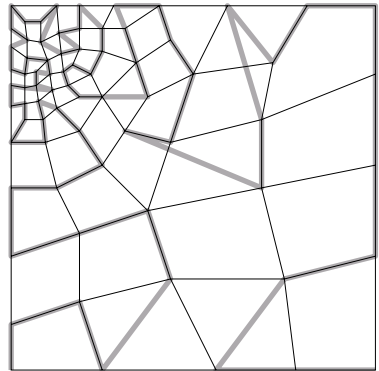


Fig. 6. Hilbert order within an irregular graph.

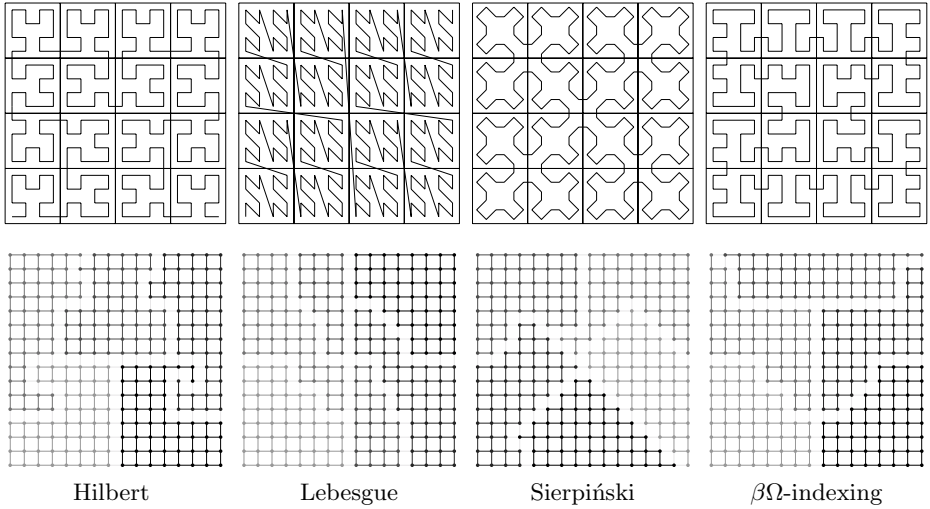


Fig. 7. The four evaluated curves. Top: Structure after 4 refinement steps. Bottom: The induced partitionings on a 16×16 -grid.

While the extension to 3-dimensional space is obvious and unique for the Lebesgue curve, there are 1536 structurally different possibilities for curves with the Hilbert property [15]. In this paper the evaluations are based on the 3-dimensional definition sketched in figure 5 showing two refinement steps. For Sierpiński and $\beta\Omega$ -indexing, no 3-dimensional versions are known.

3.2 An Example: Partitioning the 16×16 -Grid

The bottom row of figure 7 shows the partitioning of a 16×16 -grid into 5 parts using space-filling curves. The edge cuts for the four indexing schemes are 65 (Hilbert), 64 (Lebesgue), 66 (Sierpiński), and 58 ($\beta\Omega$ -indexing). For comparison, the edge cuts obtained with Metis are 63 (kmetis, direct k-partitioning) and 46 (pmetis, recursive partitioning). Thus, the edge cut for the partitionings based on the indexing schemes is 26 % to 44 % higher than the one of the solution computed by pmetis.

Although this example is not representative for the overall quality (especially for irregular graphs), it shows some of the specific disadvantages of indexing schemes. In case of the Hilbert curve, the endings of the partitions are sometimes spiral. The partitions induced by the Lebesgue curve are not connected even in regular grids.¹ On regular grids, the Sierpiński curve shows a weakness, since here the diagonal geometric separators lead to a high edge cut during recursive

¹ None of the indexing schemes can guarantee connected partitions in irregular graphs, but the probability of disconnected partitions is much higher for the Lebesgue curve for all graphs.

construction, even if the partitions are quite compact. Furthermore, the endings of the partitions are sometimes of a slightly spiral structure. The $\beta\Omega$ -indexing is based on the same u-like base pattern as the Hilbert curve but uses different refinement rules to reduce the spiral effects at the partition endings.

3.3 Partitioning Irregular Graphs

For the indexing of the vertices of irregular graphs the space is split recursively until each subsquare (or subcube) contains at most one vertex. The order of the vertices is given by the order of the subspaces. The Hilbert curve for an irregular graph is presented in figure 6. In the regular grid, the curves of Hilbert, Sierpiński, and the $\beta\Omega$ -indexing only connects vertices which are connected in the graph. This observation does not hold for an irregular graph.

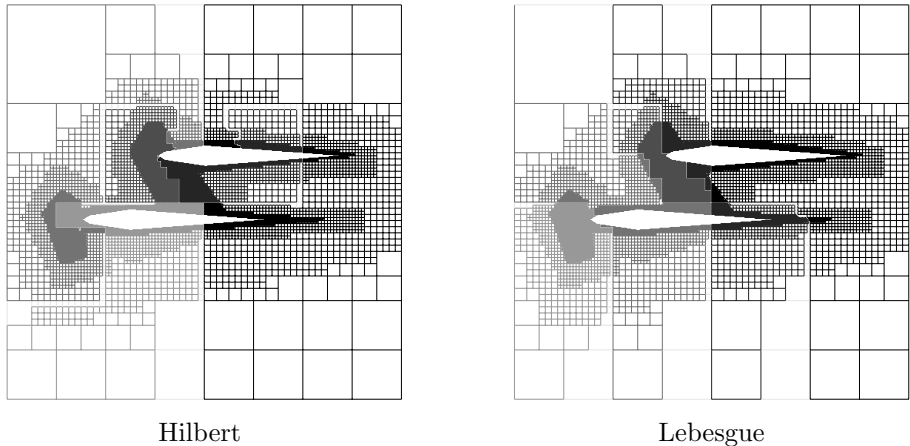


Fig. 8. Partitioning example for *biplane.9* using space-filling curves.

Figures 8 and 9 show the partitionings of the larger irregular graph “biplane.9” (cf. table 1 in section 4) into five parts using the evaluated indexing schemes. The resulting edge cuts are 627 (Hilbert), 611 (Lebesgue), 863 (Sierpiński), and 615 ($\beta\Omega$ -indexing). For comparison, the edge cuts obtained using Metis are 299 (kmetis, cf. figure 1) and 302 (pmetis).

Due to the holes in the graph, space-filling curves and all other geometric partitioning heuristics may lead to disconnected partitions. Spiral effects can be observed again for the Hilbert and Sierpiński based partitioning and in a slightly reduced form for the $\beta\Omega$ -indexing. On the other hand, for the Lebesgue curve there is a larger number of disconnected partitions. The Sierpiński curve results in the worst partitioning because its recursive definition is based on triangles which fits badly to a graph dominated by axis aligned edges.

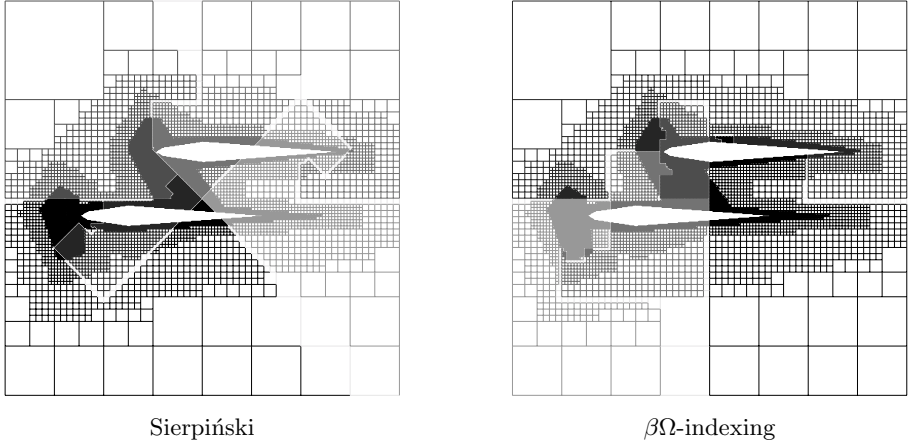


Fig. 9. Partitioning example for *biplane.9* using space-filling curves.

3.4 Analytical Results

In [16,17], it is shown that the partitioning based on connected space-filling curves is “quasi optimal” for regular grids and special types of adaptively refined grids:

$$\text{edge cut} \leq C \cdot \left(\frac{|V|}{P} \right)^{(d-1)/d}, \quad (1)$$

where $|V|$ denotes the number of vertices, P the number of partitions, and d the dimension of the graph. The constant C depends on the type of the curve.

Some constants have been determined for 2-dimensional regular grids in worst case [18]. The quality of a partition based on the Lebesgue curve is bounded by

$$7.348 < 3 \cdot \sqrt{6} - \varepsilon \leq C_{\max}^{\text{Lebesgue}} \leq \frac{384}{\sqrt{2730}} < 7.350. \quad (2)$$

A lower bound for the Hilbert curve is

$$7.442 < 12 \cdot \sqrt{\frac{5}{13}} \leq C_{\max}^{\text{Hilbert}}. \quad (3)$$

It follows that the Lebesgue curve is better than the Hilbert curve in worst case analysis for the regular grid, despite its disconnected partitions. Combined with the fact that the Sierpiński curve and the $\beta\Omega$ -indexing have larger lower bounds, Lebesgue turns out to be the best of the four evaluated indexing schemes in this case.

In average case, partitions based on the Lebesgue curve are bounded by

$$C_{\text{avg}}^{\text{Lebesgue}} \leq \frac{10}{\sqrt{3}} < 5.774. \quad (4)$$

Upper bounds for the Hilbert curve (5.56) and the $\beta\Omega$ -indexing (5.48) can be extracted from experimental results [14]. Compared to the optimal partition, a square with a boundary of $4 \cdot \sqrt{|V|/P}$, the decrease in quality is about 85 % in worst case and 40 % in average case.

4 The Test Environment

4.1 Used Metrics

Several metrics have been proposed to value the quality of results provided by partitioning algorithms. The first and probably most common one is the *edge cut*. Given a graph $G = (V, E)$ and a partitioning $\pi : V \rightarrow P$, the edge cut is defined straight forward as

$$\text{edge cut} = |\{(u, v) \in E \mid \pi(u) \neq \pi(v)\}|$$

As described in [19], this metrics has some flaws when applied to FEM partitioning, since it does not model the real communication costs. Therefore, a more exact metrics can be obtained by counting the number of *boundary vertices*, that is those vertices connected via an edge with a vertex from a different partition

$$\text{boundary vertices} = |\{v \in V \mid \exists u \in V : (u, v) \in E \wedge \pi(u) \neq \pi(v)\}|$$

In practice however, the edge cut is still the most widely used metrics. Furthermore, the results obtained for the boundary metrics and other metrics are very similar to the ones obtained based on the edge cut metrics. Thus, we will restrict our presentations to the latter one.

Another important factor for load balancing is the amount of required resources, namely time and memory. Since the goal of the load balancing process is to reduce the overall computation time, it is important that it does only consume very little time itself. To measure time and memory, we implemented a more precise clock and a memory counter into Metis 4.0. The overhead produced thereby has been tested to be negligible. All experiments have been performed on a Pentium III 850 MHz, 512 MB RAM system.

4.2 Evaluated Graphs

As usual for heuristics the obtained results also depend on the input instances. For our experiments, we used a set of established FEM graphs that can be obtained via the Internet [20] and have already been used in other work [5, 7, 21]. Table 1 lists the graphs that we used in this paper to describe our results in more detail.

Table 1. Graphs used in our experiments.

graph	$ V $	$ E $	$\deg_{\min}$	$\deg_{\text{avg}}$	$\deg_{\max}$	Comments
grid100x100	10000	19600	2	3.96	4	2-dim. grid
airfoil1	4253	12289	3	5.78	9	2-dim. triangle FEM (holes)
biplane.9	21701	42038	2	3.87	4	2-dim. square FEM (holes)
stufe.10	24010	46414	2	3.87	4	2-dim. square FEM
shock.9	36476	71290	2	3.91	4	2-dim. square FEM
dime20	224843	336024	2	2.99	3	2-dim. dual FEM (holes)
pwt	36519	144794	0	7.93	15	3-dim. FEM
ara	62032	121544	2	3.92	4	3-dim. FEM
rotor	99617	662431	5	13.30	125	3-dim. FEM
wave	156317	1059331	3	13.55	44	3-dim. FEM
hermes_all	320194	3722641	4	23.25	56	3-dim. FEM

5 Experimental Results

5.1 Quality

The first graph we chose for our test set is a 100×100 -grid. Figure 10 displays the edge cut obtained with kmetis and the four types of space-filling curves described in section 3.

With an increasing number of partitions the total edge cut also rises in all cases. The cut size calculated with Metis is the smallest, with some exceptions at powers of 2 where space-filling curves fit perfectly into a grid, followed by $\beta\Omega$ -Indexing, Hilbert, Lebesgue and then Sierpinski. The interesting aspect in this figure is, that the gap between Metis and the space-filling curves increases, but on the other hand the relative difference decreases.

This becomes more obvious if the cut sizes achieved with space-filling curves are shown in relation to the ones obtained with Metis (figure 11). Starting with bisection, the results produced with Metis are up to almost twice as good as those obtained with space-filling curves. This head start decreases more or less constantly down to a factor of 1.3, where at the already mentioned powers of 2 space-filling curves are approximately 10 percent better than Metis. Among the space-filling curves the $\beta\Omega$ -indexing and Hilbert perform best while Sierpiński and Lebesgue produce partitions with a slightly higher edge cut.

Unfortunately, since the partitioning problem is NP-complete, no optimal solutions are known for large graphs. Thus, if the difference between both methods decreases, it is an open question whether this is due to an improvement of the partitionings induced by space-filling curves or a quality reduction of the results obtained with Metis.

Another way of normalizing the results is shown in figure 12. Here, the obtained edge cut is plotted in relation to the number of vertices of a partition as described in equation 1. As shown in equation 4, the theoretical upper bound of

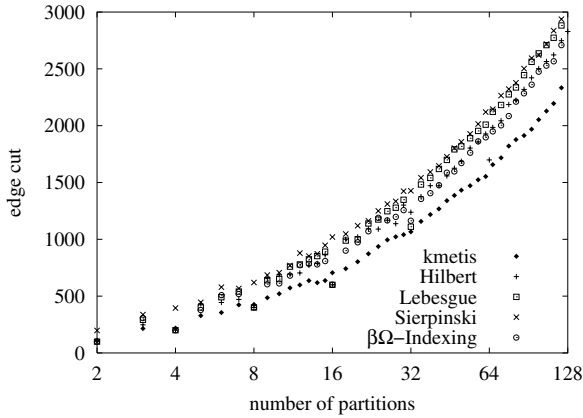


Fig. 10. Total edgecut obtained for “grid100x100” using different space-filling curves

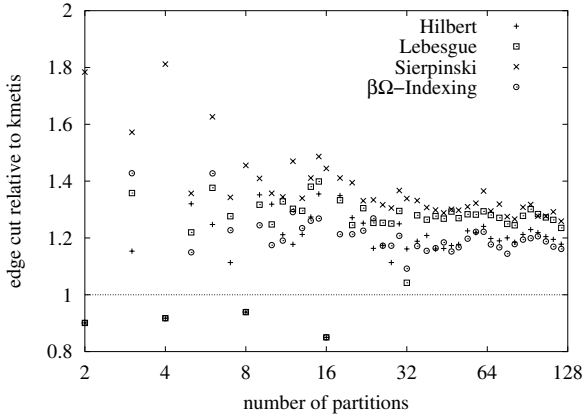


Fig. 11. “Grid100x100”: Quality of partitionings compared to kmetis.

a partitioning induced by the Lebesgue curve is about 5.8 in the average case. This bound also holds in this experiment where results get closer to it for higher numbers of partitions.

Figure 13 shows the results for another 2-dimensional graph. In contrast to the grid, the “dime20” consists of an irregular structure and also contains two holes. Therefore, we do not expect as good results as the ones obtained for the grid. While this expectation turned out to be true, this is mainly the case for a small number of partitions. Compared to the grid, the head start of Metis decreases even more with an increasing number of partitions, reaching a factor of 2 in the interesting range of partition sizes. Furthermore, among the space-filling curves the Sierpiński curve performs best for this graph, followed by the

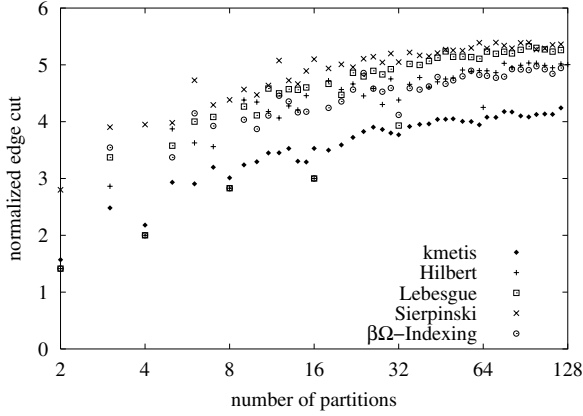


Fig. 12. Edge cut of partitions normalized to volume of partitions ($\sqrt{|V|/P}$).

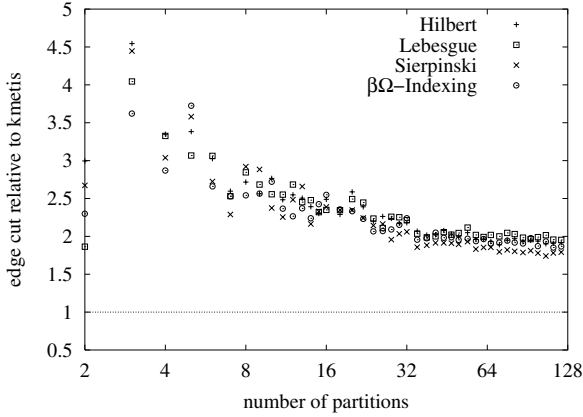


Fig. 13. Quality of partitionings compared to Metis for graph “dime20”.

$\beta\Omega$ -indexing, the Hilbert curve and then Lebesgue, which produces an about 15 percent worse edge cut than the Sierpiński curve.

In figure 14, more results obtained applying the Hilbert curve on the other 2-dimensional graphs from table 1 are presented. The same observations made for the “dime20” graph can also be made here. For a small number of partitions Metis outperforms the space-filling curves by quite a large factor, this time ranging up to 3.5. But with an increasing number of partitions, the edge cut obtained using space-filling curves gets closer to the one calculated by Metis. For the graphs included here, a factor of less than 1.5 is reached.

The 3-dimensional extensions of the Hilbert and Lebesgue curves show a similar behavior as their 2-dimensional counterparts. Figure 15 gives the results obtained for the “pwt” graph. While the overall picture is similar to the 2-

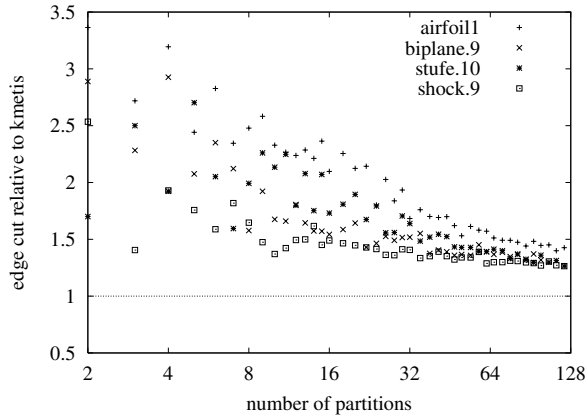


Fig. 14. Quality of partitionings compared to Metis for different 2d graphs.

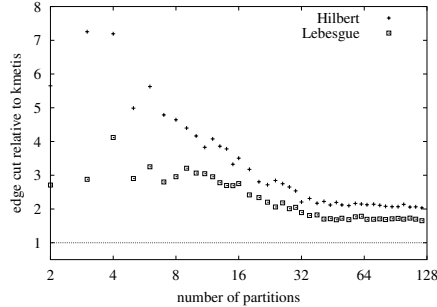


Fig. 15. Quality of partitionings of graph “pwt” compared to Metis.

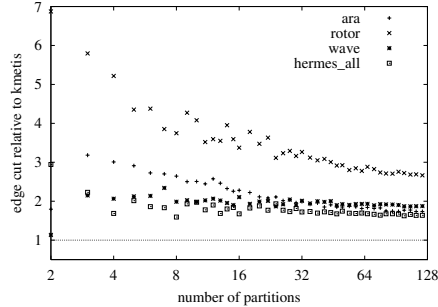


Fig. 16. Quality of partitionings compared to Metis for different 3d graphs using the Hilbert curve.

dimensional ones, the difference between both curves is much larger, with the Hilbert curve producing an up to 7 times worse result than Metis. On the other hand, the Lebesgue scheme does start with only a factor of about 3. Nevertheless, if more than 32 partitions are desired this factor decreases down to 2.2 and 1.8, respectively.

We combined the results from the experiments with the graphs “ara”, “rotor”, “wave”, and “hermes.all” in figure 16, displaying the solution quality obtained by using the Hilbert scheme. The space-filling curves perform quite well again, producing edge cuts less than twice as large as those from Metis from 16 partitions on. An exception to this is the “rotor” graph, where only a factor of 3 to 4 can be achieved with the Hilbert scheme.

Tables 2 and 3 summarize our observations for 16 and 64 partitions, respectively. In the 2-dimensional case and for 16 partitions the Lebesgue curve

Table 2. Edge cut obtained for 16 partitions.

graph	kmetis	pmetis	Hilbert	Lebesgue	Sierpiński	$\beta\Omega$ -indexing
grid100x100	660	706	600	600	1020	600
airfoil1	555	574	1204	1215	1125	1276
biplane.9	800	812	1253	1235	1593	1285
stufel.10	759	723	1251	1245	1503	1389
shock.9	1208	1233	1837	1675	2050	1736
dime20	1311	1330	3310	3125	3176	3390
pwt	2992	2933	10281	8073		
ara	4652	4666	10602	9914		
rotor	24477	23863	80523	96889		
wave	48183	48106	101493	83661		
hermes_all	119219	119170	199542	256865		

Table 3. Edge cut obtained for 64 partitions.

graph	kmetis	pmetis	Hilbert	Lebesgue	Sierpiński	$\beta\Omega$ -indexing
grid100x100	1543	1599	1699	1713	2141	1748
airfoil1	1528	1572	2430	2501	2265	2408
biplane.9	1906	2023	2867	2888	3229	2844
stufel.10	2268	2303	3286	3036	3087	3427
shock.9	2902	2889	3873	3915	4355	3761
dime20	3655	3670	7465	7510	6846	7419
pwt	9015	9310	19458	15767		
ara	9034	9405	16546	16378		
rotor	52190	53623	151820	184376		
wave	94342	97010	180764	141013		
hermes_all	241771	249959	420313	459234		

produces the best results for our graphs. Thus, the discontinued structure of the Lebesgue curve described in 3 results in better partitionings than the connected, but spiral ones of e.g. the Hilbert scheme. However, in the case of 64 partitions this advantage diminishes and no curve is clearly superior. Due to the different structure of the Sierpiński curve, it either produces very good or very bad results compared to the other curves (e.g. on graphs “airfoil1” and “biplane.9”). The edge cut of the partitionings induced by space-filling curves is about twice as large as the one obtained with Metis. For 64 partitions however, this value decreases down to a factor of 1.6. For 3-dimensional graphs, no space-filling curve performs clearly better than the other, neither for 16 nor for 64 partitions. Compared to Metis, the relative difference here is 2.5 and 2.1 for partition numbers of 16 and 64, respectively.

5.2 Resources

As mentioned before, the goal of load balancing is the reduction of the overall computation time. Therefore, the time spend on partitioning itself should also be minimized. Figure 17 shows the results of our experiments performed on the

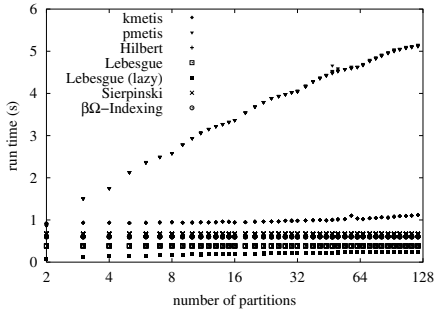


Fig. 17. Run-time of space-filling curves and Metis needed to partition “dime20”.

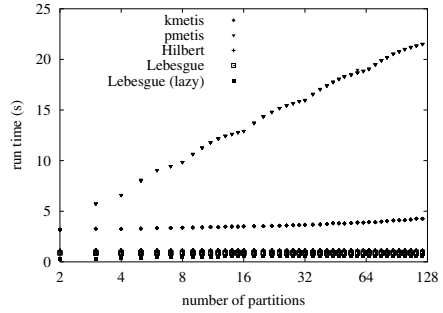


Fig. 18. Run-time of space-filling curves and Metis needed to partition “hermes_all”.

“dime20” graph. All space-filling curves need much less time for their computation than either kmetis or even the recursive partitioner pmetis. This is even more the case if the computation of the ordering is interrupted as soon as all partitionings have been determined, listed as *lazy* in plots 17 and 18. On the other hand, if full ordering information is available, it is easy to decompose the graph into any other given number of partitions very quickly.

Considering the memory consumption, Metis is outperformed even more by all space-filling curves. In case of the “dime20” graph, Metis requires about 42 MByte whereas space-filling curves only consumes 3.5 MByte for the graph and additional 2 KByte for the recursive descending. Partitioning the “hermes_all” graph, this gap even widens to 220 MByte vs. 5 MByte.

6 Conclusions

As expected Metis produces better results concerning the edge cut than space-filling curves do. This is not surprising since space-filling curves only rely on vertex coordinates rather than on any connectivity information between the vertices; the information that edges do provide is simply ignored. However, the gap between the solution quality of both approaches is not too large. In most cases, applying Metis does not result in more than a 30 to 50 percent decrease in edge cut for a decent number of partitions. This factor decreases further with an increasing number of partitions. Moreover, space-filling curves save both, a lot of time and a lot of memory.

Finally, it depends on the application which method is suited best: If memory is a concern, space-filling curves are definitely superior. Looking at run-time we can say, that if some additional communication overhead does not slow down the application too much, space-filling curves have to be considered as a partitioning alternative.

References

1. G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann, San Francisco, 1994.
2. Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500, 2000.
3. K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editor, *The Sourcebook of Parallel Computing*. Morgan Kaufmann, 2002. to appear.
4. George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
5. C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIJSSC: SIAM Journal on Scientific and Statistical Computing*, apparently renamed *SIAM Journal on Scientific Computing*, 22, 2000.
6. B. Hendrickson and R. Leland. The chaco user’s guide — version 2.0, 1994.
7. R. Preis and R. Diekmann. Party - a software library for graph partitioning. *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71, 1997.
8. Robert Preis. The PARTY graphpartitioning-library – user manual – version 1.99.
9. C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Design Automation Conference*, May 1984.
10. B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.
11. R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. In *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–73. AMS Publications, 1995.
12. David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.
13. H. Sagan. *Space Filling Curves*. Springer, 1994.
14. Jens-Michael Wierum. Definition of a new circular space-filling curve – $\beta\Omega$ -indexing. Technical Report TR-001-02, Paderborn Center for Parallel Computing, <http://www.upb.de/pc2/>, 2002.
15. Jochen Alber and Rolf Niedermeier. On multi-dimensional hilbert indexings. In *Computing and Combinatorics*, number 1449 in LNCS, pages 329–338, 1998.
16. Gerhard Zumbusch. On the quality of space-filling curve induced partitions. *Zeitschrift für Angewandte Mathematik und Mechanik*, 81, SUPP/1:25–28, 2001.
17. Gerhard Zumbusch. Load balancing for adaptively refined grids. Technical Report 722, SFB 256, University Bonn, 2001.
18. Jan Hungershöfer and Jens-Michael Wierum. On the quality of partitions based on space-filling curves. In *International Conference on Computational Science ICCS*, volume 2331 of LNCS, pages 36–45. Springer, 2002.
19. Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.
20. C. Walshaw. The graph partitioning archive. <http://www.gre.ac.uk/c.walshaw/partition/>.
21. R. Battiti and A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, April 1999.

Process Algebraic Model of Superscalar Processor Programs for Instruction Level Timing Analysis

Hee-Jun Yoo and Jin-Young Choi

Theory and Formal Methods Lab., Dept. of Computer Science and Engineering,
Korea University, Seoul, Korea(ROK), 136-701
{hyoo, choi}@formal.korea.ac.kr
<http://formal.korea.ac.kr>

Abstract. This paper illustrates a formal technique for describing timing properties and resource constraints of pipelined out of order superscalar processor instructions at a high level. The degree of parallelism depends on the multiplicity of hardware functional units as well as data dependencies among instructions. Thus, the timing properties of a superscalar program are difficult to analyze and predict. We describe how to model the instruction level architecture of a superscalar processor using ACSR and how to derive the temporal behavior of an assembly program using ACSR laws. Our approach is to model superscalar processor registers as ACSR resources, instructions as ACSR processes, and use ACSR priorities to achieve maximum possible instruction-level parallelism.

1 Introduction

Many methods have been explored to improve computer execution speed. Superscalar processor is one of such methods, but different from others in which superscalar processors depend on instruction-level parallelism. Superscalar processors realize instruction-level parallelism by replicating functional hardware and by overlapping instruction execution stage in pipeline[5]. Consequently, multiple instructions can be issued and executed simultaneously in superscalar processors. So, performance of superscalar processors may vary according to their applications or programs as well as hardware structures. Herein lie the difficulties with superscalar processors. To acquire maximum-level parallelism, the sequence of instructions, that is, programs must be optimized to be executed in parallel as well as hardware structure. Specially in time critical applications, exact execution cycle must be verified. The formal methods we suggest can be used to verify such time critical system, and also used at the stage of designing superscalar processors at high level or optimizing programs to be executed in superscalar processors.

Previous attempts[4] had modeled only small parts of instruction set. In this paper, we include conditional branch instructions to that and extend out-of-order superscalar pipelined method that can find instruction pair in searchable instructions at any cycle regardless of order. This approach is to augment the ISA(Instruction Set Architecture) level[7] description with timing properties and

resource constraints using a formal technique based on process algebra. Most of other approaches[1][6] for this field were focused on Worst Case Execution Time and analyzed processor that was issued only one instruction per a cycle. But, our approach considers how processor can find maximum executable instructions at the cycle.

The rest of the paper is organized as follows: in section 2, we introduce basic syntax of ACSR. In section 3, we describe in-order instructions modeling of our approach and demonstrate how a ToyP program can be translated into ACSR and its execution simulated. In section 4, describe out-of-order case, Section 5 summarizes the paper and describes plans for future work.

2 ACSR(Algebra of Communicating Shared Resources)

ACSR includes the concepts of time, resource, priority that are needed in concurrency theory. ACSR can be used in specifying and verifying real-time systems. The execution of an ACSR process is defined by a labelled transition system. For example, a process P has a following behavior. P denotes ACSR process. Following example represents a labelled transition system.

$$P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} P_3 \xrightarrow{\alpha_3} \dots \xrightarrow{\alpha_{n-1}} P_n$$

The detailed descriptions and semantics of ACSR can be found in [3]. The syntax of ACSR processes containing actions is as follows :

$$\begin{aligned} P ::= & NIL \mid A : P \mid (a.n).P \mid P + Q \mid P \parallel Q \\ & \mid [P]_I \mid P[f] \mid P \setminus F \mid P \setminus H \mid recX.P \mid P \end{aligned}$$

3 Modeling for In-Order Case

We use the ToyP system as [4] with same assumption. The detailed descriptions and assumptions of system can be found in [4]. We define additional process Done and operator *next* as follows :

Definition 3.1 Process Done, which represents the end of instruction execution, is defined as follows :

$$Done \stackrel{def}{=} recX.\emptyset : X$$

The process done has the following identity property with respect to the parallel composition operator $\parallel$: For every process P , $P \parallel Done = P$. We introduce one binary operator, which is used to model the issuing of instructions in a consecutive cycle. Resource J is represented for searching branch instruction.

Definition 3.2 For any P, Q , the *next* operator is defined as follows :

$$P \text{ next } Q \stackrel{\text{def}}{=} P \parallel \{J\}:Q$$

Instruction modeling converts five instructions to ACSR process that consumes register resource. Definition 3.3 is described for instruction modeling and Definition 3.4 is execution modeling.

Definition 3.3 In-Order Instructions in ACSR

$$\begin{array}{lll} \text{Add} & Ri, Rj, Rk & \stackrel{\text{def}}{=} \sum_{1 \leq l \leq 6} \sum_{1 \leq m \leq 6} \{Ri, Rj_l, Rk_m\}:\text{Done} \\ \text{Mov} & Ri, Rj & \stackrel{\text{def}}{=} \sum_{1 \leq l \leq 6} \{Ri, Rj_l\}:\text{Done} \\ \text{Load} & Ri, Rj, \#c & \stackrel{\text{def}}{=} \sum_{1 \leq l \leq 6} \{Ri, Rj_l\}:\{Ri\}:\text{Done} \\ \text{Store} & Ri, Rj, \#c & \stackrel{\text{def}}{=} \sum_{1 \leq l \leq 6} \sum_{1 \leq m \leq 6} \{Ri_l, Rj_m\}:\{Ri_l\}:\text{Done} \\ \text{Jump} & \#c & \stackrel{\text{def}}{=} \{J\}:\text{Insts}(c) \end{array}$$

Definition 3.4 Execution modeling for In-Order Instruction of ToyP processor

$$\begin{aligned} \text{Insts}(\text{PC}) & \stackrel{\text{def}}{=} \emptyset : \text{Insts}(\text{PC}) + \text{Super-Insts}(\text{PC}) \\ \text{Super-Insts} & \stackrel{\text{def}}{=} (\text{Mem}(\text{PC}) \parallel \text{Mem}(\text{PC}+4) \parallel \text{Mem}(\text{PC}+8)) \text{ next } \text{Insts}(\text{PC}+12) \\ & \quad + (\text{Mem}(\text{PC}) \parallel \text{Mem}(\text{PC}+4) \text{ next } \text{Insts}(\text{PC}+8)) \\ & \quad + \text{Mem}(\text{PC}) \text{ next } \text{Insts}(\text{PC}+4) \\ & \quad + \text{Mem}(\text{PC}) \end{aligned}$$

Ordinarily, Add and Mov instruction take an instruction cycle for execution. But, related memory instruction, Load and Store, need two cycles. Jump instruction could not take any other instruction, and PC(Program Counter) indicates branching destination. The whole system of ToyP is composed of a finite set R that consists of registers with the same priority. An action is defined by a subset of R . It takes one cycle time. But, resource is consisted of several sub-resources. Therefore, we need to represent how many sub-resources are used by access to a resource. $\{(r, i)\}$ represents the action that consumes a cycle with priority i with any register $r \in R$. For simplicity, priority is omitted assuming all priority has the same value in action and action $\emptyset$ (also $\{\}$) represents empty action in a time unit.

4 Modeling for Out-of-Order Case

1. Add R1, R1, R1
2. Load R2, R3₁, #8
3. Mov R4, R1₁
4. Store R4₁, R3₂, #4
5. Add R5, R5, R6₁

The reason is that instruction 1 and 2 could not execute simultaneously, because register R1 is monopolized by instruction 3. But, previous algorithms could not find executable pairs with high parallelism. Such strategy of pipelined instruction arrangement is called in-order form. But, we could not have high parallelism. In opposition to a previous case, we call this method as out-of-order pipelined superscalar method that can find parallel executable instruction pair of searchable instruction in any cycle, regardless of order (that is, method that could execute instruction 1 and 3 simultaneously in the instruction set). Though out-of-order superscalar method is a more optimal than in-order, it requires more complex micro processor circuit. Real commercial micro processors add special buffer between instruction cache and pipeline that bring set instruction numbers in pipeline with reordering after moving instruction to buffer. Definition 4.1 is described for out-of-order Jmp instruction modeling. The remainder instructions are same as in-order case.

Definition 4.1 Out-of-Order Jmp Instructions in ACSR

$$\text{Jmp } \#c \stackrel{\text{def}}{=} \text{Issue}(c, c+4, c+8)$$

Execution modeling consists of two processes. One is $\text{Issue}(\text{PC}_1, \text{PC}_2, \text{PC}_3)$ that detects branch instruction and proceeds to next execution step. The other is $\text{Exec}(\text{PC}_1, \text{PC}_2, \text{PC}_3)$ that finds and executes more large number of executable instructions simultaneously in the reordering buffer. Parameters $\text{PC}_1, \text{PC}_2, \text{PC}_3$ are instruction address indexes of reordering buffer and indicate the memory address of stored instruction.

Definition 4.2 Execution modeling for Out-of-Order instruction of ToyP

$$\begin{aligned} \text{Issue}(\text{PC}_1, \text{PC}_2, \text{PC}_3) &\stackrel{\text{def}}{=} \{B\} : (\text{Issue}(\text{PC}_1, \text{PC}_2, \text{PC}_3) + \text{EXEC}(\text{PC}_1, \text{PC}_2, \text{PC}_3)) \\ \text{EXEC}(\text{PC}_1, \text{PC}_2, \text{PC}_3) &\stackrel{\text{def}}{=} \text{Mem}(\text{PC}_1) \parallel \text{Mem}(\text{PC}_2) \parallel \text{Mem}(\text{PC}_3) \\ &\quad \parallel \text{Issue}(\text{PC}_3+4, \text{PC}_3+8, \text{PC}_3+12) \\ &\quad + \text{Mem}(\text{PC}_1) \parallel \text{Mem}(\text{PC}_2) \parallel \text{Issue}(\text{PC}_3, \text{PC}_3+4, \text{PC}_3+8) \\ &\quad + \text{Mem}(\text{PC}_1) \parallel \text{Mem}(\text{PC}_3) \parallel (\text{Mem}(\text{PC}_1) \parallel \text{Mem}(\text{PC}_3)) \setminus [\emptyset, f_R] \\ &\quad \parallel \text{Issue}(\text{PC}_2+4, \text{PC}_3+4, \text{PC}_3+8) \\ &\quad + \text{Mem}(\text{PC}_1) \parallel \text{Issue}(\text{PC}_2, \text{PC}_3, \text{PC}_3+4) \\ &\quad + \text{Mem}(\text{PC}_1) \end{aligned}$$

$$\begin{aligned} \text{Program} &\stackrel{\text{def}}{=} [\text{Issue}(0, 4, 8)]_{R \cup B} \\ f_R &= \{Si_j \setminus Ri_j \mid 1 \leq i \leq 6, 1 \leq j \leq 6\} \end{aligned}$$

Process Exec has six choices according to the number of executable simultaneous instructions and order. First line describes the case that three instructions can be executed simultaneously. Second, third, and fourth line describe for case that two instructions can be executed simultaneously. Fifth line is the case that the most front instruction in buffer can be executed, and the last case is executed

branch instruction. So, Exec process is closed at the set of all register resource and branch detecting resource(B) in program definition. It can be executed where there is more executable instruction.

5 Conclusion

This paper illustrates a formal technique for describing the timing properties and resource constraints of pipelined superscalar processor instructions at a high level. We use simple virtual superscalar process ToyP. Process is a set of register resources. Instruction is represented by ACSR process that shares a part of register resources at each cycle. ToyP program is also represented by an indexed set of ACSR processes. We could analyze timing properties of ToyP program by simplifying this ACSR process using the ACSR law. As the results of our approach, we can find maximal executable pairs for instruction set at a cycle. Thus, we could obtain the highest parallelism of ToyP processor programs. All ACSR specifications are tested by VERSA[2] that is the verification tool of ACSR. We detect data hazard and obtain maximal executable pairs for given instruction set that is generated by parallel composition of instruction and execution modeling in VERSA. Our future work is extending our approach to the real superscalar process.

References

1. A. Colin and I. Puaut.: Worst case execution time analysis technique for a processor with branch prediction. *Real-Time Systems*, 18(2/3).(2000) 249–274
2. D. Clarke.: VERSA : Verification, Execution and Rewrite System For ACSR, Technical Report. University of Pennsylvania (1994)
3. I. Lee, P. Brmond-Groire, and R. Gerber.: A Process Algebraic Approach to the Specification and Analysis of Resources-bound Real-time Systems. Technical Report MS-CIS-93-08, Univ. of Pennsylvania(1993). To appear in *IEEE Proceedings*. (1994)
4. J. Y. Choi, I. Lee, and I. Kang, Timing Analysis of Superscalar Processor Programs Using ACSR, *IEEE Real-Time Systems Newsletter*, Volume 10, No. 1/2, (1994)
5. M. Johnson, *Superscalar Microprocessor Design*. Prentice-Hall, (1991)
6. S. -K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, (1996) 230–240
7. T. Cook, P. Franzon, E. Harcourt, T. Miller, System-Level Specification of Instruction Sets, In *Proc. of the International Conference on Computer Design*, (1993)

Optimization of the Communications between Processors in a General Parallel Computing Approach Using the Selected Data Technique

Hervé Bolvin¹, André Chambarel¹, Dominique Fougere², and Petr Gladkikh³

¹ Laboratory of Complex Hydrodynamics, Faculté des Sciences, 33, rue Louis Pasteur
F-84000 AVIGNON

`andre.chambarel@univ-avignon.fr`

² Laboratory of Modeling in Mechanics L3M, La jetée, Technopôle de Château-Gombert
8, rue F. Joliot Curie

F-13451 - Marseille cedex 20

`fougere@l3m.univ-mrs.fr`

³ Supercomputer Software Department

ICM and MG SB RAS

Pr. Lavrentiev 6, 630090 Novosibirsk, Russia

Abstract. A large variety of problems that are out of reach of single processor computer capabilities. Many approaches are offered today to get round this. Each of these has its own strengths and weaknesses : a compromise has to be found. We will introduce a general parallel computing method for engineering problems dedicated to all users. We have searched an easy method for code development. A technique of data selection (Selected Data Technique – SDT) is used for the determination of the data dedicated to each processor. Several problems associated with the communication times are posed and solutions are proposed in accordance with the number of processors. This method is applied to very large CPU cost problems, particularly the unsteady problems or steady problems using an iterative method. So the domain of potential applications is very wide. The SDT-parallelization is performed by an expert system called AMS (Automatic Multi-grid System) included in the software. This new concept is a natural way for the standardization of parallel codes. An example is presented hereafter.

1 Introduction

A first approach of the parallel computing method for engineering problems by a Finite Element Method is presented under reference [1]. A technique of data selection (Selected Data Technique – SDT) is used for the determination of the data dedicated to each processor. The main problem concerns the communication time between the processors, and several solutions are proposed. This method is applied to very large

CPU cost problems, particularly the unsteady problems or steady problems using an iterative method. So the domain of potential applications is very wide.

An application of the Automatic Multi-grid System in this software development is the easy parallel computing which seems to be a natural way of performing intensive computation. Our purpose is to carry out parallel algorithms without modifying the object structure of the solvers and the data structure. To answer this requirement, we use a selected data method resulting in suitable load balancing thanks to the determination of lists of elements. This technique is independent from the geometry, and can be applied in general cases. This new concept is a natural way for the standardization of parallel codes. In fact, parallelization is here applied to the resolution of the nonlinear system by “matrix free” algorithms. The examples are performed with distributed memory computers associated to MPI technology.

2 Structure of Code

The code is based on usage of three classes corresponding to the functional blocks of the Finite Element Method [2]. With these classes we built three objects that are connected by a single heritage. So the transmission of the parameters between these objects is defined by a “list” technique.

We use efficient C++ Object-Oriented Programming for the Finite Element code called FAFEMO (Fast Adaptive Finite Element Modular Object [2]).

3 Principle of Parallelization

The main part of the CPU cost corresponds to the computing of elementary matrices and the time-step updating. In the example of an unsteady problem, the analytical discretization of the equations with the Finite Element Method gives the following scalar product [3] [4]:

$$\sum_{NE} \langle \delta ue \rangle . ([me] . \left\{ \frac{due}{dt} \right\} + [ke] \{ue\} - \{fe\}) = 0 \quad \text{with} \quad NE = 1 .. ne$$

We use the “matrix free” technique and we only consider the elementary residuum

$$\sum_{NE} \langle \delta ue \rangle . ([me] . \left\{ \frac{due}{dt} \right\} - \{\psi e\}) = 0$$

$\{\psi e\}$:

If n is the number of processors, we select a list of elements N_k :

$$\bigcup_{k=1}^n N_k = NE \quad \text{and} \quad N_i \cap N_j = \emptyset \quad \text{for } i \neq j$$

The various elementary matrices can be assembled in a global matrix by a classical Finite Element process [3]. We thus obtain :

$$\sum_{k=1}^n [M_k] = [M] \quad \text{global mass matrix}$$

$$\sum_{k=1}^n \{\Psi_k\} = \{\Psi\} \quad \text{global residuum}$$

So we have a correct load balancing if the list of elements is similar for each processor. The definition of arrays depends on the technology. For example if we have a shared memory only the global matrices are memorized for the full problem. The communications between the processors only exist at the end of the time step. Each processor builds his part of the differential system and the above algorithm allows the updating of the solution $\{U\}$. A semi-implicit algorithm is used [3]:

```

 $t_n = 0$ 
while ( $t_n \leq t_{\max}$ )
    {
        for  $j = 1$  to  $p$     $\{\Delta U_n^i\}_j = \Delta t_n \cdot [M_n^i]^{-1} \cdot \{\Psi_j(U_n + \alpha \cdot \Delta U_n^{i-1}, t_n + \alpha \cdot \Delta t_n)\}$ 
        {
             $i = 1, 2, \dots$    until  $\|\Delta U_n^i - \Delta U_n^{i-1}\| \leq \text{tolerance}$ 
        }
         $\{U_{n+1}\} = \{U_n\} + \{\Delta U_n\}$ 
         $t_{n+1} = t_n + \Delta t_n$ 
    }
end while
    
```

where α is the upward time-parameter.

We use a general technique for the easy diagonalization of the global mass matrix [5].

4 Technique of Parallelization

One of most important quality criteria of the parallelization method is speedup, which is usually defined as follows:

$$S(n) = \frac{T(1)}{T(n)}$$

where

$S(n)$ - speedup for a n -processor computer;

$T(n)$ - execution time of certain programs on an n -processor computer.

The "dream speedup" is n , it is the highest possible performance boost obtainable on a multi processor system. Real $S(n)$ is less than n due to synchronization and communication expenses between program processes running on different processor nodes.

Instead of $S(n)$ we can consider the following "parallelization efficiency" :

$$E(n) = \frac{S(n)}{n}$$

This characteristic determines how the method works with large scale computers, which have big number of processors.

Let's give a formal description of such a kind of parallelization.

4.1 First Approach

But as a first approach, in each time step we communicate the full matrices M_k and ψ_k to the main processor to update the differential system solution. The sequential problem can thus be summarized as follows:

```
for (j = 0; j < n_element; j++)
{ finite element process .... }
```

This parallelization process can be described by the following patterns. The AMS expert system chooses the elements dedicated to each processor. First we present a sequential shearing of the element list [1].

$$active\_element[i][j] = \begin{bmatrix} 1 & 1 & .. & 0 & 0 & .. & 0 & 0 & ... \\ 0 & 0 & .. & 1 & 1 & .. & 0 & 0 & ... \\ 0 & 0 & .. & 0 & 0 & .. & 1 & 1 & ... \\ 0 & 0 & .. & 0 & 0 & .. & 0 & 0 & ... \end{bmatrix}$$

```
for (i = 0; i < n_processor; i++)
for (j = 0; j < n_element; j++)
if active_element[i][j] { finite element process .... }
```

This is, in fact, the principle to use low memory; we do not hope to store the Boolean matrix above. For this aim we define a boolean function, as follows, and the sequence of corresponding code can be written:

```
for (i = 0; i < n_processor; i++)
for (j = 0; j < n_element; j++)
if boolean[i][j] { finite element process .... }
```

boolean[i][j] is true if element *j* is dedicated to processor *i*.

Another example is presented. In this case we distribute the elements to each processor as playing-cards around a table. The code sequence can be written as follows :

$$active\_element[i][j] = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \dots \end{bmatrix}$$

for ($i = 0; i < n\_processor; i++$)
 for ($j = 0; j < n\_element; j++$)
 if $active\_element[i][j] \{ \text{finite element process } \dots \}$

As in the preceding example a Boolean function is used.

In accordance with the algorithm used the time process can be defined by three values:

- T_a is the assembling procedure time,
- T_c is the communication time for sending partial matrices and receiving current solution,
- T_s is the updating time of the solution.

In the finite element process time T_a is preponderant. Under these conditions we can estimate the speedup of the method. Execution time in our case can be expressed as

$$T(n) = \frac{T_a}{n} + (n-1)T_c + T_s$$

T_a is the time necessary for assembling global problem data structures, using a sequential algorithm. Namely, here we consider the time required to assemble a global value of $\psi(U,t)$ and $[M]$ the global mass matrix. Here we assume that all processors sent their partial sums to a single “main” processor, which in turn calculates the global matrices and calculates a solution for the next time step. The communication scheme in this case looks like this:

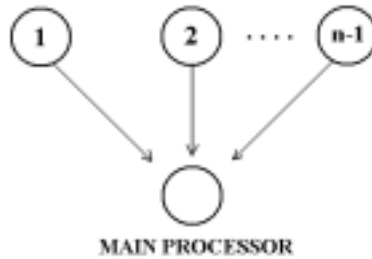


Fig. 1. First approach of the communications.

Let us assume that for large values of n we have

$$T(n) \approx nT_c + T_s$$

It is obvious that for large scale computers only communication and solution time give considerable contributions. Thus this method will have good scalability and speedup if the matrices are not too big.

Speedup and efficiency in this case are:

$$S(n) = \frac{T_a + T_s}{\frac{T_a}{n} + (n-1).T_c + T_s} \qquad E(n) = \frac{T_a + T_s}{T_a + n.(n-1).T_c + n.T_s}$$

In extreme cases, we notice that:

$$n \rightarrow +\infty \Rightarrow E(n) \rightarrow 0$$

thus this method has limited scalability. In practice, the main part of time resides in T_a . This method is valid in accordance with the communication efficiency and the T_a value. An example of this method is presented in the table 1 below:

Table 1.

CLUSTER 25 x (2 x Athlon 1.5 GHz, 1 Gb)
Linux RedHat 7.1 , MPICH, gcc
Network Gigabit ethernet with switch

PROCESSORS	SPEED UP
2	1.97
4	3.73
6	5.27
8	6.79
10	8.02

With the characteristics of this example it is possible to determine times T .

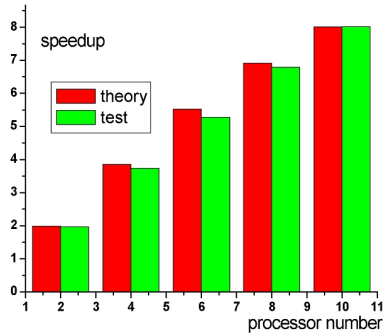


Fig. 2. Validation of the speedup estimation.

For this example we obtain the following relative values:

$$T_a = 99,5 \% \quad T_c = 0,25 \% \quad T_s = 0,25 \%$$

Under these conditions it is possible to estimate the efficiency of the parallelization process for the determination of a reasonable number of processors.

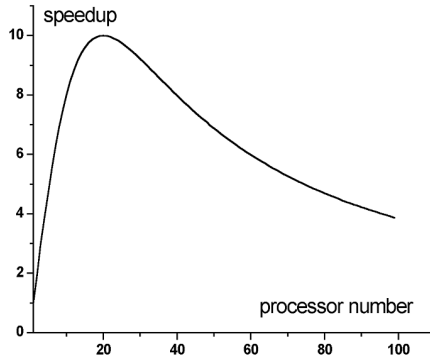


Fig. 3. Optimal value of the speedup.

Figures 3 and 4 show the simulation of the speedup and the efficiency for a larger number of processors. So we can see the number of processors accessible by this method, which sustains satisfactory speedup results only for computational systems of no more than a few tens of processors. If we hope to use a very large number of processors we must optimize the process above. In this case the code development is particularly easy.

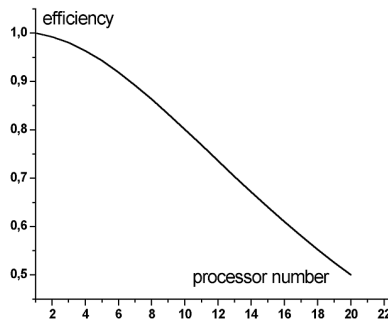


Fig. 4. Limitation of the efficiency.

4.2 Second Approach

In order to reduce communication costs we can implement matrices summation, using binary tree schemes [6]. An example is presented in Figure 5. This gives a logarithmic law for communication time in relation with the number of processors, and in this case we have the speedup estimate :

$$S(n) = \frac{T_a + T_s}{\frac{T_a}{n} + T_c \cdot \log_2 n + T_s}$$

Communication scheme in this case is as follows:

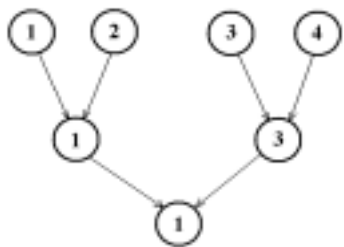


Fig. 5. First optimization of communications.

Note that there is no actual necessity for messages from i-th processor to i-th processor. Here is a graph of efficiency and speedup in a case of identical characteristic times:

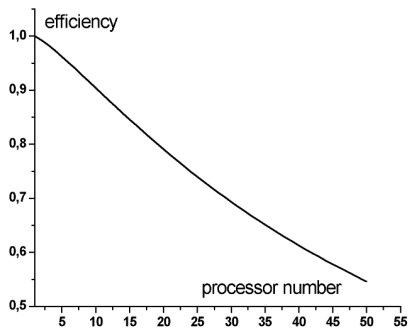


Fig. 6. A best efficiency.

Table 2.

CLUSTER 25 x (2 x Athlon 1.5 GHz, 1 Gb)
Linux RedHat 7.1 , MPICH, gcc
Network Gigabit ethernet with switch

PROCESSORS	SPEED UP
2	1.98
4	3.88
6	5.70
8	7.35
10	9.01

4.3 Optimization of the Message Size

To update the solution of the finite elements' algorithm, the techniques above send vectors in the full-size finite elements' space. It is possible to send only the unknowns dedicated to the processor concerned. So if the element lists associated to a processor are the same size then the communication time is approximately a constant. We define a linear operator A_k associated to each processor k and for the example of residuum ψ , we can write:

$$\{\Psi_k\} = [A_k] \cdot \{\Psi_k^*\}$$

So processor k only sends vector $\{\psi_k^*\}$ and linear operator A_k is known in the finite element process. In practice we define an integer function as follows:

$$\{\Psi_k^*\}_j = \{\Psi_k\}_{f(j)} \quad \text{with} \quad i = f(j), j = 1, \dots, \text{size\_message}$$

Under these conditions speedup and efficiency can be written:

$$S(n) = \frac{T_a + T_s}{\frac{T_a}{n} + T_c + T_s} \quad E(n) = \frac{T_a + T_s}{T_a + n.T_c + n.T_s}$$

and we present the corresponding graphs:

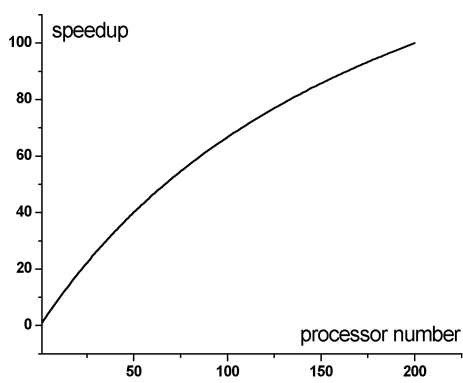


Fig. 7. Speedup with size message optimization.

Tests are performed on the same computer as with the first method, and we obtain the following speedup:

Table 3.

CLUSTER 25 x (2 x Athlon 1.5 GHz, 1 Gb)
 Linux RedHat 7.1 , MPICH, gcc
 Network Gigabit ethernet with switch

PROCESSORS	SPEED UP
2	1.98
4	3.90
6	5.83
8	7.65
10	9.50

5 Application

This method is well adapted to the problems of wave propagation when using a finite element technique. Depending on the choice of technology it is possible to use up to 100 processors with acceptable efficiency. Figure 8 shows the case of an

electromagnetic wave which propagates out of a wave guide. We present only pictures with 2 processors because with a larger number they become ‘unreadable’.

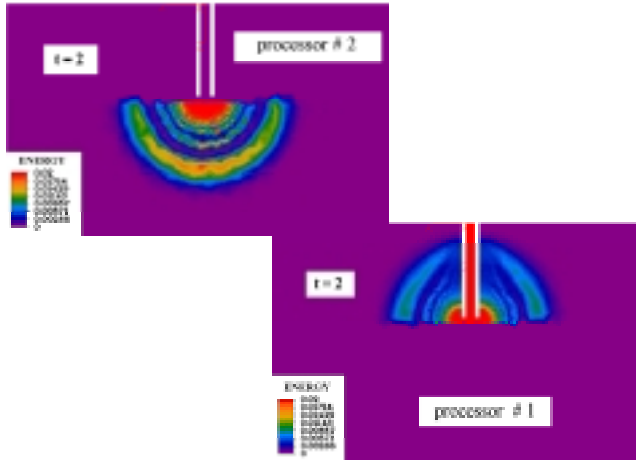


Fig. 8. Results with 2 processors.

6 Conclusion

An easy method of parallel computing is proposed to solve engineering problems. It consists in using a coherent set of techniques. In this context the implementation of the low sized solvers concerned is very easy. The SIMD architecture associated with the MPI-C++ library is used. So we have an efficient method for the parallelization of differential systems resulting from the Finite Element Method. We particularly notice the low size memory and the good load balancing.

We present here a set of techniques based on the method SDT associated with the Finite Element Method. Different techniques are possible in accordance with the number of processors. A good load balancing is associated with the CPU time and the main problem of this parallelization method is the communication time between processors [8]. In all cases the development of the code is very easy owing to Object-Oriented Programming. This method can be used with SIMD or MIMD technology, both with distributed memory computers and shared memory computers. Its general character allows its use by non specialists.

Acknowledgment. The authors would like to thank Ralph Beisson for his collaboration about the English composition of this paper.

References

1. Chambarel, A., Bolvin, H.: Application of the parallel computing technology to a wave front model using the Finite Element method. *Lecture Notes in Computer Science*, Vol. 2127, Springer-Verlag (2001) 421–427
2. Chambarel, A., Onuphre, E.: Finite Element software based on Object Programming. International Conference of the twelfth I.A.S.T.E.D., Annecy France (May 18–20, 1994)
3. Chambarel, A., Ferry, E.: Finite Element formulation for Maxwell's equations with space dependent electric properties. *Revue européenne des Eléments Finis*, Vol. 9, n° 8 (2000) 941–967
4. Laevsky, Y.M., Banushkina, P.V., Litvinenko, S.A., Zotkevich, A.A.: Parallel algorithms for non-stationary problems: survey of new generation of explicit schemes. *Lecture Notes in Computer Science*, Vol. 2127, Springer-Verlag (2001) 442–446
5. Bernardin L.: Maple on a Massively Parallel, Distributed Memory Machine. PASC0 97, Second Int. Sym. on Parallel Symbolic Computation, Maui, Hawaii (July 20–22, 1997), 217–222
6. Gresho, P.M.: On the theory of semi-implicit projection methods for viscous incompressible flow and its implementation via a finite element method that also introduces a nearly consistent mass matrix. *Int. J. Numer. Meth.Fluids*, Vol. 11 (1990) 621–659
7. Chambarel, A., Fougère, D.: A general parallel computing approach using the Finite Element method and the object-oriented programming by selected data technique. 6th International Conference, PACT 2001, Novosibirsk, Russia, (September 3–7, 2001)
8. Hempel, R., Calkin R., Hess, R., Joppich, W., Keller, U., Koike, N., Oosterlee, C.W., Ritzdorf, H., Washio, T., Wypior, P., Ziegler, W.: Real applications on the new parallel system NEC Cenju-3. *Parallel Computing*, Vol. 22 (1996) 131–148

Load Imbalance in Parallel Programs^{*}

Maria Calzarossa, Luisa Massari, and Daniele Tessera

Dipartimento di Informatica e Sistemistica,
Università di Pavia,
I-27100 Pavia, Italy,
`{mcc,massari,tessera}@unipv.it`

Abstract. Parallel programs experience performance inefficiencies as a result of dependencies, resource contentions, uneven work distributions and loss of synchronizations among processors. The analysis of these inefficiencies is very important for tuning and performance debugging studies. In this paper we address the identification and localization of performance inefficiencies from a methodological viewpoint. We follow a top down approach. We first analyze the performance properties of the programs at a coarse grain. We then study the behavior of the processors and their load imbalance. The methodology is illustrated on a study of a message passing computational fluid dynamic program.

1 Introduction

The performance achieved by a parallel program is the result of complex interactions between the hardware and software resources involved in its execution. The characteristics of the program, that is, its algorithmic structure and input parameters, determine how it can exploit the available resources and the allocated processors. Hence, tuning and performance debugging of parallel programs are challenging issues [11].

Tuning and performance debugging typically rely on an experimental approach based on instrumenting the program, monitoring its execution and analyzing the performance measures either on the fly or post mortem. Many tools have been developed for this purpose (see e.g., [1], [2], [5], [12], [13], [14]). These tools analyze the behavior of the various activities of a program, e.g., computation, communication, synchronization, by means of visualization and statistical analysis techniques. Their major drawback is that they fail to assist users in mastering the complexity inherent in the analysis of parallel programs.

Few tools focus on the analysis of parallel programs with the aim of identifying their performance bottlenecks, that is, the code regions critical from the performance viewpoint. The Poirot project [6] proposed a tool architecture to automatically diagnose parallel programs using a heuristic classification scheme.

^{*} This work has been supported by the Italian Ministry of Education, University and Research (MIUR) under the FIRB Programme, by the University of Pavia under the FAR Programme and by the Italian Research Council (CNR).

The Paradyn Parallel Performance tool [9] dynamically instruments the programs to automate bottleneck detection during their execution. The Paradyn Performance Consultant starts a hierarchical search of the bottlenecks, defined as the code regions of the program whose performance metrics exceed some pre-defined thresholds. The automated search performs a stack sampling [10] and a pruning of the search space based on historical performance and structural data [7].

In this paper we address the analysis of the performance inefficiencies of parallel programs from a methodological viewpoint. We study the behavior and the performance properties of the programs with the aim of detecting the symptoms of performance problems and localizing where they occurred. Our methodology is based on the definition of performance metrics and on the use of a few criteria able to explain the performance properties of the programs and the inefficiencies due to load imbalance among the processors.

The paper is organized as follows. Section 2 introduces the metrics and criteria for the evaluation of the overall behavior of parallel programs. Section 3 focuses on the analysis of the behavior of the allocated processors. An application of the methodology is presented in Section 4. Finally, Section 5 concludes the paper and outlines guidelines towards the integration of our methodology into a performance analysis tool.

2 Performance Properties

Tuning and debugging the performance of a parallel program can be seen as an iterative process consisting of several steps, dealing with the identification and localization of inefficiencies, their repair and the verification and validation of the achieved performance.

As already stated, our objective is to address the performance analysis process by focusing on the identification and localization of performance inefficiencies. We follow a top down approach in which we first characterize the overall behavior of the program in terms of its activities, e.g., computation, communication, synchronization, memory accesses, I/O operations. We then analyze the various code regions of the program, e.g., loops, routines, code statements, and the activities performed within each region.

The characterization of the performance properties and inefficiencies of the program is based on the definition of various criteria. In this section, we define the criteria that identify the dominant activities and the dominant code regions of the program. Next section is dedicated to the identification of inefficiencies due to dissimilarities in the behavior of the processors.

The performance of a parallel program is characterized by timings parameters, such as, wall clock times, as well as counting parameters, such as, number of I/O operations, number of bytes read/written, number of memory accesses, number of cache misses. Note that, not to clutter the presentation, in what follows we focus on timings parameters.

Let N denote the number of code regions of the parallel program, K the number of its activities, and P the number of allocated processors. t_{ijp} ($i = 1, 2, \dots, N$; $j = 1, 2, \dots, K$; $p = 1, 2, \dots, P$) is the wall clock time of processor p in the activity j of the code region i . t_{ij} ($i = 1, 2, \dots, N$; $j = 1, 2, \dots, K$) is the wall clock time of the activity j in the code region i , that is:

$$t_{ij} = \frac{1}{P} \sum_{p=1}^P t_{ijp} .$$

Similarly, t_i ($i = 1, 2, \dots, N$) is the wall clock time of the code region i , T_j ($j = 1, 2, \dots, K$) is the wall clock time of the activity j , and T is the wall clock time of the whole program.

A preliminary characterization of the performance of a parallel program is based on the breakdown of its wall clock time T into the times T_j , ($j = 1, 2, \dots, K$) spent in the various activities. The activity with the maximum T_j is defined as the dominant, that is, “heaviest”, activity of the program, and could correspond to a performance bottleneck.

The analysis of the code regions is aimed at identifying the portions of the code where the program spends most of its time. The region with the maximum wall clock time, i.e., the heaviest region, might correspond to an inefficient portion of the program or to its core.

A refinement of this analysis is based on the breakdown of the wall clock time t_i into the times t_{ij} spent in the various activities. It might be difficult to understand which activity better explains the behavior and the performance of the program. We can identify the code region characterized by the maximum time in the dominant activity of the program. Moreover, for each activity j we can identify the worst and the best code regions, that is, with the maximum and minimum t_{ij} , respectively. This analysis results in a large amount of information. Hence, it is useful to summarize the properties of the program by identifying patterns or groups of regions characterized by a similar behavior. Clustering techniques [4] work for this purpose. Each code region i is described by its wall clock times t_{ij} and is represented in a K -dimensional space. Clustering partitions this space into groups of code regions with homogeneous characteristics such that the candidates for possible tuning are identified.

3 Processor Dissimilarities

The coarse grain analysis of the performance properties of parallel programs is followed by a fine grain analysis that focuses on the behavior of the processors with the objective of studying their load imbalance.

Load balancing is an ideal condition for a program to achieve good performance by fully exploiting the benefits of parallel computing. Programming inefficiencies might lead to uneven work distributions among processors. These distributions then lead to poor performance because of the delays due to loss of synchronization, dependencies and resource contentions among the processors.

Our methodology analyzes whether and where a program experienced poor performance because of load imbalance. For this purpose, we study the dissimilarities in the behavior of the processors with the aim of identifying the symptoms of uneven work distributions. In particular, we study the spread of the t_{ijp} 's, that is, the wall clock times spent by the various processors to perform activity j within code region i . As a first step, we need to define the metrics that detect and quantify dissimilarities and the criteria that assess their severity.

The metrics for evaluating the dissimilarities rely on the majorization theory [8], which provides a framework for measuring the spread of data sets. Such a theory is based on the definition of indices for partially ordering data sets according to the dissimilarities among their elements. The theory allows the identification of the data sets that are more spread out than the others. Dissimilarities can be measured by different indices of dispersion, such as, variance, coefficient of variation, Euclidean distance, mean absolute deviation, maximum, sum of the elements of the data sets. The choice of the most appropriate index of dispersion depends on the objective of the study and on the type of physical phenomenon to be analyzed. In our study, the index of dispersion has to measure the spread of the times spent by the processors to perform a given activity with respect to the perfectly balanced condition, where all processors spend exactly the same amount of time. The Euclidean distance between the time of each processor and the corresponding average is then well suited for our purpose.

Once the metrics to quantify dissimilarities have been defined, it is necessary to select the criteria for their ranking. The choice of the most appropriate criterion to assess the severity of the load imbalance among processors depends on the level of details required by the analysis. Possible criteria are the maximum of the indices of dispersion, the percentiles of their distribution, or some predefined thresholds.

The analysis of dissimilarities can then be summarized by the following steps:

- standardization of the wall clock times;
- computation of the indices of dispersion;
- ranking of the indices of dispersion.

Note that as the indices of dispersion have to provide a relative measure of the spread of the wall clock times, the first step of the methodology deals with a standardization of the wall clock times of each code region. As we will see, the standardized times are such that they sum to one, that is, they are obtained by dividing the wall clock times by the corresponding sum.

The second step of the methodology deals with the computation of the various indices of dispersion. In particular, our analysis focuses on three different views, namely, processor, activity, and code region. These views provide complementary insights into the behavior of the processors as they correspond to the different perspectives used to characterize a parallel program.

Once the indices of dispersion have been computed for the various views, their ranking allows us to identify processors, activities and code regions characterized by large dissimilarities which could be chosen as candidates for performance tuning.

3.1 Processor View

Processor view is aimed at analyzing the behavior of the processors across the activities performed within each code region with the objective of identifying the most frequently imbalanced processor. We describe the dissimilarities of each code region with P indices of dispersion ID_P_{ip} , one for each processor. These indices are computed as the Euclidean distance between the times spent by processor p on the various activities performed within code region i and the average time of these activities over all processors:

$$ID\_P_{ip} = \sqrt{\sum_{j=1}^K (\tilde{t}_{ijp} - \tilde{T}_{ij})^2}.$$

Note that the $\tilde{t}_{ijp}$'s are obtained by standardizing the t_{ijp} 's over the sum of the times spent by each processor in the various activities performed within a given code region. $\tilde{T}_{ij}$ denotes the corresponding average.

From the various indices of dispersion, we can identify the processors that have been most frequently imbalanced and imbalanced for the longest time.

3.2 Activity View

Activity view analyzes dissimilarities within the activities performed by the processors across all the code regions with the objective of identifying the most imbalanced activity. We first quantify the dissimilarities in the times spent by the various processors to perform a given activity within a code region. Let ID_{ij} be the index of dispersion computed as the Euclidean distance between the times spent by the various processors to perform activity j within code region i and their average. We then summarize the ID_{ij} 's to identify and localize the activity characterized by the largest load imbalance.

ID_A_j is the relative measure of the load imbalance within the activity j and is obtained as the weighted average of the ID_{ij} 's. The weights represent the fractions of the overall wall clock time accounted by activity j within code region i , that is, $\frac{t_{ij}}{T_j}$. As activities with large dissimilarities might have a negligible impact on the overall performance of the program because of their short wall clock time, we scale the index of dispersion ID_A_j according to the fraction of the program wall clock time accounted by the activity itself, namely:

$$SID\_A_j = \frac{T_j}{T} ID\_A_j.$$

The scaled indices of dispersion SID_A_j allow us to identify the activities characterized by large dissimilarities and accounting for a significant fraction of the wall clock time of the program.

3.3 Code Region View

Code region view analyzes the dissimilarities with respect to the various activities performed by the processors within each region with the objective of identifying the most imbalanced region. The computation of the dissimilarities is based on the ID_{ij} 's defined in the activity view. ID_{C_i} is a relative measure of the load imbalance within code region i , and is obtained as the weighted average of the ID_{ij} 's with respect to $\frac{t_{ij}}{t_i}$, that is, the fraction of the wall clock time of the code region accounted by activity j . As in the activity view, we scale the index of dispersion ID_{C_i} with respect to the fraction of the program wall clock time accounted by code region i , i.e., $\frac{t_i}{T}$, and we obtain the scaled index SID_{C_i} .

4 Application Example

In this section we illustrate our methodology on the analysis of the performance inefficiencies of a message passing computational fluid dynamic code. We focus on an execution of the program on $P = 16$ processors of an IBM Sp2. The measurements refer to 7 code regions corresponding to the main loops of the program. Moreover, within each region, four activities have been measured, namely, computation, point-to-point communications (i.e., `MPI_SEND`, `MPI_RECV`), collective communications (i.e., `MPI_REDUCE`, `MPI_ALLTOALL`), and synchronizations among processors (i.e., `MPI_BARRIER`). In what follows, we identify the loops of the application with a number, from 1 to 7.

Table 1 presents the wall clock time of each loop with the corresponding breakdown into the wall clock times of its activities. By profiling the program, that is, by looking where the time is spent, we notice that the heaviest loop, that is, loop 1, accounts for about 27% of the overall wall clock time. This loop, which corresponds to the core of the program, is characterized by the longest time in computation, that is, the dominant activity of the program, as well as in collective communications and synchronizations, whereas it does not perform any point-to-point communication. The loop which spends the longest time in point-to-point communications is loop 3. Moreover, only three loops perform synchronizations.

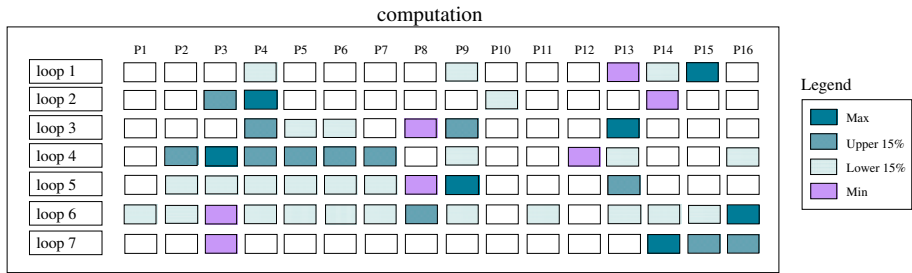
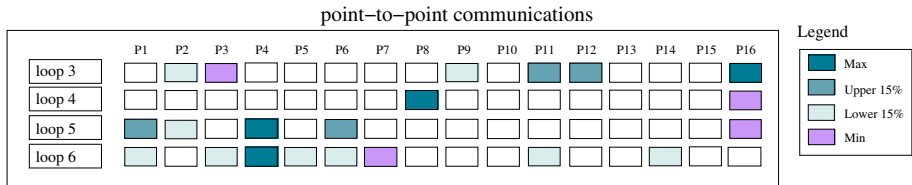
For a more detailed analysis of the behavior of the loops we applied the *k-means* clustering algorithm [4]. Each loop is described the wall clock times it spent in the various activities. Clustering yields a partition of the loops into two groups. The heaviest loops of the program, that is, loops 1 and 2, belong to one group, whereas the remaining loops belong to the second group.

To gain better insights into the performance properties of the program and to study the dissimilarities in the processor behavior, we analyzed the wall clock times spent by the processors to perform the various activities. Figures 1 and 2 show the patterns of the times spent in computation and point-to-point communications activities, respectively. The patterns are plotted for each loop separately, namely, each row refers to one loop. Different colors are used to highlight the patterns.

Table 1. Overall wall clock time, in seconds, of the loops and corresponding breakdown

loop	wall clock time				
	overall	computation	point-to-point	collective	synchronization
1	19.051	12.24	-	6.75	0.061
2	14.22	7.90	-	6.32	-
3	10.90	5.22	5.68	-	-
4	10.54	8.03	2.51	-	-
5	9.041	7.53	0.07	1.43	0.011
6	0.692	0.36	0.33	-	0.002
7	0.31	0.28	-	0.03	-

The four colors used in the figures refer to the maximum and minimum values of the wall clock times of the loop and to values belonging to the lower and upper 15% intervals of the range of the wall clock times, respectively. Note that the diagrams plot only the loops performing the activity shown by the diagram itself.

**Fig. 1.** Patterns of the times spent by the processors in computation**Fig. 2.** Patterns of the times spent by the processors in point-to-point communications

As can be seen, the behavior of the processors within and across the various loops and activities is quite different. By analyzing the patterns shown in Figure 1, we notice that the times spent in computation by five out of 16 processors executing loop 4 belong to the upper 15% interval, whereas on loop 6 the times of 11 out of 16 processors belong to the lower 15% interval. From Figure 2 we can notice

that the behavior of the processors executing point-to-point communications is very balanced.

These figures provide some qualitative insights into the behavior of the processors, whereas they lack in providing any quantitative description of their dissimilarities. To quantify the dissimilarities, we standardized the wall clock times and computed the indices of dispersion as defined in Section 3. From the analysis of the processor view, we have discovered that processor 1 is the most frequently imbalanced as it is characterized by the largest values of the index of dispersion on two loops, namely, loops 3 and 7. Processor 2 is imbalanced for the longest time. This processor is the most imbalanced on one loop only, namely, loop 1, with an index of dispersion equal to 0.25754 and a wall clock time equal to 15.93 seconds.

For the analysis of the activity and code region views, we have computed the indices of dispersion ID_{ij} presented in Table 2. As can be seen, the behavior of the processors is highly imbalanced when performing synchronizations. The value of the index of dispersion corresponding to loop 5 is equal to 0.30571. Loop 1 is the most imbalanced with respect to the times spent by the processors for performing collective communications, whereas loop 6 is characterized by the largest indices of dispersion in two activities, namely, computation and point-to-point communications.

Table 2. Indices of dispersion ID_{ij} of the activities performed by the loops

loop	computation	point-to-point	collective	synchronization
1	0.03674	-	0.06793	0.12870
2	0.01095	-	0.00318	-
3	0.00672	0.02833	-	-
4	0.01615	0.10742	-	-
5	0.00933	0.08872	0.04907	0.30571
6	0.05017	0.23200	-	0.16163
7	0.00719	-	0.01138	-

To summarize the values of Table 2 by taking into account the relative weights of the wall clock times of the activities and of the loops, we computed the weighted average of the ID_{ij} 's. Tables 3 and 4 present the values of the indices of dispersions ID_{A_j} and ID_{C_i} computed for the activities and the loops, respectively. The tables also present the indices SID_{A_j} and SID_{C_i} scaled with respect to the fraction of the wall clock time accounted by each activity or loop, respectively.

As can be seen from Table 3, the synchronization is the most imbalanced activity. However, as it accounts only for 0.1% of the wall clock time of the program, its impact on the overall performance is negligible. Hence, this activity does not seem a suitable candidate for tuning, as also denoted by the value of the scaled index of dispersion which is equal to 0.00016.

Table 3. Summary of the indices of dispersion of the activity view

activity	<i>ID_A</i>	<i>SID_A</i>
computation	0.01904	0.01132
point-to-point	0.05973	0.00734
collective	0.03781	0.00786
synchronization	0.15559	0.00016

Table 4. Summary of the indices of dispersion of the code region view

loop	<i>ID_C</i>	<i>SID_C</i>
1	0.04809	0.01311
2	0.00750	0.00152
3	0.01798	0.00280
4	0.03790	0.00571
5	0.01655	0.00214
6	0.13734	0.00135
7	0.00760	0.00003

From the analysis of the summaries presented in Table 4, we can conclude that loop 6 is the most imbalanced. The value of its index of dispersion is equal to 0.13734. However, as this loop accounts for a very short wall clock, the value of the corresponding scaled index of dispersion is equal to 0.00135 only. These metrics help the users in deciding which loop is the best candidate for performance tuning. In our study loop 1 is a good candidate as it is the core of the program and it is also characterized by large values of both the index of dispersion and its scaled counterpart.

5 Conclusions

The analysis of performance inefficiencies of parallel programs is a challenging issue. Users do not want to browse too many diagrams or, even worse, to dig into the tracefiles collected during the execution of their programs. They expect from performance tools answers to their performance problems. Thereby, tools should do what expert programmers do when tuning their programs, that is, detect the presence of inefficiencies, localize them and assess their severity.

The identification and localization of the performance inefficiencies of parallel programs are preliminary steps towards an automatic performance analysis. The methodology presented in this paper is aimed at isolating inefficiencies and load imbalance within a program by analyzing performance measurements related to its execution. From the measurements we derive various metrics that guide users in the interpretation of the behavior and of the performance properties of their programs.

As a future work, we plan to define and test new criteria for the identification and localization of performance inefficiencies. Hence, we will analyze measurements collected on different parallel systems for a large variety of scientific programs [3]. Moreover, we plan to integrate our methodology into a performance tool.

References

1. M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessa. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, 1995.
2. L. DeRose, Y. Zhang, and D.A. Reed. SvPablo: A Multi-Language Performance Analysis System. In R. Puigjaner, N. Savino, and B. Serra, editors, *Computer Performance Evaluation - Modelling Techniques and Tools*, volume 1469 of *Lecture Notes in Computer Science*, pages 352–355. Springer, 1998.
3. K. Ferschweiler, S. Harrah, D. Keon, M. Calzarossa, D. Tessa, and C. Pancake. The Tracefile Testbed – A Community Repository for Identifying and Retrieving HPC Performance Data. In *Proc. 2002 International Conference on Parallel Processing*, pages 177–184. IEEE Press, 2002.
4. J.A. Hartigan. *Clustering Algorithms*. Wiley, 1975.
5. M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8:29–39, 1991.
6. B. Helm, A. Malony, and S. Fickas. Capturing and Automating Performance Diagnosis: the Poirot Approach. In *Proceedings of the 1995 International Parallel Processing Symposium*, pages 606–613, 1995.
7. K.L. Karavanic and B.P. Miller. Improving Online Performance Diagnosis by the Use of Historical Performance Data. In *Proc. SC'99*, 1999.
8. A.W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications*. Academic Press, 1979.
9. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K.H. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Measurement Performance Tool. *IEEE Computer*, 28(11):37–46, 1995.
10. P.C. Roth and B.P. Miller. Deep Start: A Hybrid Strategy for Automated Performance Problem Searches. In *Proc. 8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2002.
11. M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society, 1996.
12. W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. In K.M. Decker, editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 333–345. Birkhauser Verlag, 1994.
13. J.C. Yan and S.R. Sarukkai. Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques. *Parallel Computing*, 22(9):1215–1237, 1996.
14. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(2):277–288, 1999.

Software Carry-Save: A Case Study for Instruction-Level Parallelism

David Defour and Florent de Dinechin

ENS Lyon, 46 allée d'Italie, 69364 Lyon, France
{David.Defour, Florent.de.Dinechin}@ens-lyon.fr

Abstract. This paper is a practical study of the performance impact of avoiding data-dependencies at the algorithm level, when targeting recent deeply pipelined, superscalar processors. We are interested in multiple-precision libraries offering the equivalent of quad-double precision. We show that a combination of today's processors, today's compilers, and algorithms written in C using a data representation which exposes parallelism, is able to outperform the reference GMP library which is partially written in assembler. We observe that the gain is related to a better use of the processor's instruction parallelism.

1 Introduction: Modern Superscalar Processors

The increase of performance of recent microprocessors is largely due to the ever-increasing internal parallelism they offer [8]:

- All the workstation processors sold in 2003 possess several functional units which can execute instructions in parallel: between 2 and 4 memory units, usually 2 double-precision floating-point (FP) units, and between 2 and 6 integer units. The capabilities of these units vary widely.
- All these processors are also pipelined, currently with 8 to 20 pipeline stages. More specifically, we focus in the following on the pipeline of integer processing units, characterized by its *latency* and *throughput* as given in Table 1. Pipelines also means parallelism: The table shows for instance that 4 integer multiplications may be running in parallel at a given time in the Pentium-III multiplier.

Integer addition is an ubiquitous operations in typical code, and one-cycle adder units are cheap, so all processors offer several of them. Most processors (Alpha, Pentium III, Athlon, PowerPC) also possess one integer multiplier. However, a recent trend (Pentium IV, UltraSPARC, Itanium) is to make without this integer multiplier, and to delegate the (relatively rare) integer multiplications to an FP multiplier, at the expense of a higher latency due to additional translation costs. As the Itaniums have two identical FP units each capable of multiplication, they are the only architectures in this table on which more than one multiplication can be launched each cycle.

Table 1. Integer unit characteristics. Simple integer means add/subtract, boolean operations, and masks. A latency of l means that the result is available l cycles after the operation has begun. A throughput of n means that a new instruction may be launched every n cycles. This data is extracted from vendor documentation and other vendor-authored papers, and should be taken with caution as many specific architectural restrictions apply. The reader interested in these questions is invited to browse the `mpn` directory of the GMP source code [1], probably the most extensive and up-to-date single source of information on the integer capabilities of processors.

Architecture	concurrent simple integer (Latency/Throughput)	concurrent multiplications (Latency/Throughput)
Pentium III	2 (1/1)	1 (4/1)
UltraSPARC II	2 (1/1)	1 (5-35/5-35)
Alpha EV6/EV7	4 (1/1)	1 (7/1)
AMD Athlon XP	3 (1/1)	1 (4-6/3)
Pentium IV	3 (0.5-1/0.5-1)	1 (15-18/5)
PowerPC G4	3 (1/1)	1 (4/2)
Itanium	4 (1/1)	2 (18/1)
Itanium 2	6 (1/1)	2 (16?/1)

As processors offer ever more parallelism, it becomes increasingly difficult to exploit it. Instruction parallelism is limited by *data-dependencies* of several kinds, and by *structural hazards* [8]. Compilers and/or hardware try to allocate resources and schedule instructions so as to avoid them.

In this paper, we consider several algorithms for multiple-precision, and we show experimentally that on the latest generations of processors, the best algorithm is not the one which executes less operations, but the one which exposes more parallelism.

2 Multiple-Precision as an Algorithmic Benchmark

Most modern computers obey the IEEE-754 standard for floating-point arithmetic, which defines the well-known *single* and *double precision* FP formats. For applications requiring more precision (numerical analysis, cryptography or computational geometry), many general-purpose multiple-precision (MP) libraries have been developed [4,5,6,9,1]. Some offer arbitrary precision with static or dynamic precision control, other simply offer a fixed precision which is higher than IEEE-754 double precision. Here we focus on libraries able to offer *quad-double* precision, *i.e.* 200-210 bits of precision. This is the precision required for computing elementary functions correctly rounded up to the last bit, which is the subject of our main current research.

All libraries code MP numbers as arrays of *machine numbers*, *i.e.* numbers in a native format on which the microprocessor can directly compute: Integer, or IEEE-754 FP numbers. They all also use variations of the same basic multiple-precision algorithms for addition and multiplication, similar to those learnt in

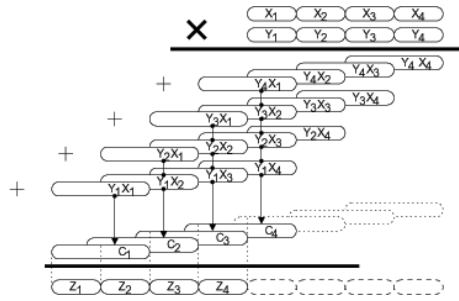


Fig. 1. Multiple-Precision multiplication

elementary school for radix-10 numbers.¹ Figure 1 depicts the algorithm for the multiplication. This figure represents the two input numbers X and Y , decomposed into their n digits x_i and y_i (with $n = 4$ on the figure). Each digit is itself coded in m bits of precision. An array of partial products $x_i y_j$ (each a $2m$ -bit number) is computed, then summed to get the final result.

There is a lot of intrinsic parallelism in this algorithm: The partial products can all be computed in parallel, as can the column sums. However the intermediate sums may require up to than $2m + \log_2 n$ bits, while digits of the result are expected to be m -bit numbers like the inputs. Some conversions of large numbers to smaller ones must therefore take place. For example, in the classical pencil-and-paper algorithm in base 10, this conversion takes the form of a *carry propagation*, with right-to-left data-dependencies that do not appear on Fig. 1. These dependencies are a consequence of the representation of the intermediate results, constrained here to be single digits.

There are many other ways to implement Fig. 1, depending on the data representation of the digits, which entail in turn specific data-dependencies. This explains the variety of MP algorithms.

Dense high-radix representation. The GNU Multiple-Precision (GMP) package uses a direct transposition of the pencil-and-paper sequential algorithm. The difference is that the digits are machine integers (of 32 or 64 bits on current processors). In other words the radix of the representation is 2^{32} or 2^{64} instead of 10. Carry propagation uses processor-specific *add-with-carry* instructions, which are present in all processors but inaccessible from high-level language. This is one reason for which GMP uses assembly code for its inner loops. The other reason is, of course, performance.

However, on pipelined processors, these carry-propagation dependencies entail pipeline stalls, which GMP programmers try to avoid by filling the pipeline bubbles with useful operations like loop handling and memory accesses (see the

¹ Other algorithms exist with a better asymptotic complexity, for example Karatsuba's algorithm [10]. They are relevant for precision much larger than quad-double.

well-commented source [1]). For recent processors this is not enough, and the latest versions of GMP try to compute two lines of Fig. 1 in parallel. All this needs a deep insight in the execution behaviour of increasingly complex processors.

Bailey's MPFUN [3] is a dense high-radix MP package where the digits are FP numbers instead of integers. In this case, there is no carry, but one has to recover and propagate FP rounding errors, using fairly different algorithms. Due to lack of space we do not describe them here.

Software carry-save. Another option is to avoid the previous right-to-left carry propagation altogether, by ensuring that all the intermediate results of Fig. 1 (including intermediate sums, not shown) fit on a machine number. To achieve this, the digits of the inputs and output don't use all the precision available in the machine format: Some of the bits are reserved (set to zero), to be used by the MP algorithms to store intermediate carries. The *carry-save* denomination is borrowed from a similar idea widely used in hardware [11,12].

This idea is first found in Brent's MP library [4] with integer digits. His motivation seems to have been *portability*: Where GMP uses assembler to access the *add-with-carry* instructions, in carry-save MP all the operations are known in advance to be exact, without overflow nor rounding. Therefore algorithms only use basic, and thus portable, arithmetic. The idea has been resurfacing recently: It seems to be used by Ziv [2] with FP digits. Independently, the authors developed the *Software Carry-Save* (SCS) library [7]. Initially we experimented with FP and integer digits, and found that integer was more efficient.

Our motivations for using carry-save MP were again portability (we use the C language), but also *efficiency*: Carry-save MP allows carry-free algorithms which, in addition of being simpler, exposes more intrinsic instruction-level parallelism. Note that there is a tradeoff there: More SCS digits are needed to reach a given precision than in the dense high-radix case, due to the reserved bits. Therefore more elementary operations will be needed.

The actual implementation of SCS uses a mixture of 32-bit and 64-bit arithmetic (well-supported by all processors/compilers and easy to express in the C language in a *de-facto* standard way). For quad-double precision, we use $n = 8$ digits, each digit using $m = 30$ bits of a 32-bit machine word. MP addition uses only 32-bit arithmetic. MP multiplication uses 64-bit arithmetic. As the partial products use 60 bits out of 64, a whole column sum can be computed without overflow. There is only one final carry-propagation in the MP multiplication, although with 36-bit carries. It is written in C using AND masks and shifts.

To sum it up, the SCS representation exposes the whole of the parallelism inherent to the MP multiplication algorithm. The following of the paper shows that the compiler can be trusted to detect and exploit this parallelism.

The library `scslib` is available under the GNU LGPL from

www.ens-lyon.fr/LIP/Arenaire/

3 Experiments and Timings

This section gives experimental measures of the performance of four available MP libraries ensuring about 210 bits of precision, on four recent microprocessors. The libraries are our SCS library, GMP [1] (more precisely its floating representation MPF), and two FP-based libraries, Bailey’s quad-double library [9], and Ziv’s library [2]. The systems considered are the following:

- *Pentium III* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
- *Pentium IV* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
- *PowerPC G4* with MacOS 10.2 and gcc-2.95
- *Itanium* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2

The results are relatively independent on the compiler (we also tested other compilers by Sun and Intel). Each result is obtained by measuring the execution times on 10^3 random values (the same values are used for all the libraries). To leverage the effect of operating system interruptions, the tests are run several times and the minimum timing is reported. Care has also been taken to prefill the instruction caches with the library code before timing (by executing a few untimed operations), to choose a number of random values that fits in all the data-caches, and in general to avoid cache-related irrelevant effects.

We have timed multiplication, addition, and conversions to and from MP format for each library. We have also implemented a test on a “lifelike” application: The evaluation of a correctly rounded double-precision logarithm function. This application converts from double to MP, evaluates a polynomial of degree 20 which makes heavy use of multiplication and addition, then converts back to double. Results are summarized in Fig. 2.

A first glance at these graphs, given in the order of introduction of the respective processors, shows that the performance advantage of SCS over the other libraries seems to increase with each generation of processor. We relate this to the increase of internal parallelism, which favors the more parallel SCS approach. FP-based libraries suffer more, because FP addition is a multicycle, pipelined operation of increasing depth, whereas integer addition remains a one-cycle operation. This is the main reason why we chose integer arithmetic in SCS.

Concerning the timings of the conversions to and from FP, the two integer-based libraries have comparable performance, while the FP-based library have the potential of much simpler conversions. The differences observed reflect the facilities offered by the processors to convert machine integers to/from machine doubles. We didn’t investigate the bad result of the FP-based Ziv library.

Concerning the arithmetic operations, GMP and SCS have a clear lead over the FP-based libraries. In the following, we therefore concentrate on these two libraries. Let us review the effects which may contribute to a performance difference between SCS and GMP:

1. The SCS library (like IBM’s and Bailey’s) provides fixed accuracy selected at compile time, whereas GMP is an arbitrary-precision library. This means that the former use almost only fixed loop (which can be unrolled), whereas the latter must handle arbitrary-length loops.

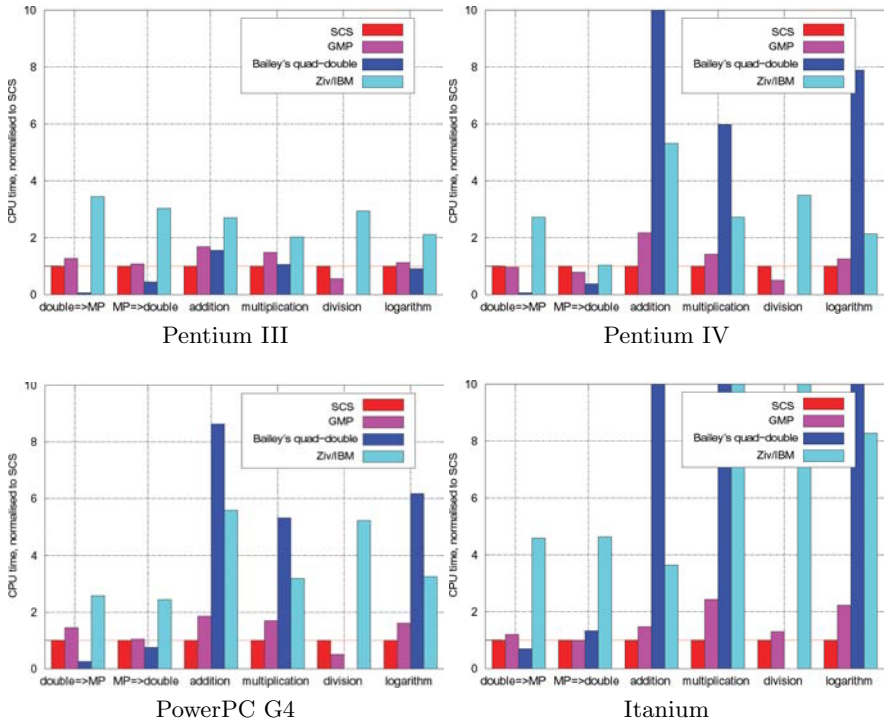


Fig. 2. Compared MP timings on several processors. For the sake of clarity we have normalised results to the SCS timing for each function on each tested architecture: The bars do not represent absolute time. An absent bar means that the corresponding operation showed compilation or runtime errors on this architecture.

2. SCS performs less carry propagations, and therefore less work per digit.
3. GMP uses assembly code, and uses processor-specific machine instructions (the so-called “multimedia extensions”) when they help, for example on the Pentium IV architecture.
4. GMP needs less digits for a given precision.
5. SCS exposes parallelism.

Addition benefits from simplicity. The first effect accounts for the performance difference in the addition. The algorithms for SCS and GMP addition present similar complexity and data-dependencies, and should exhibit similar performance. However, the cost of loop handling (decrement the loop index, compare it to zero, branch, with a possible pipeline hazard) far exceeds the cost of the actual computation (one *add-with-carry*). The only reason why SCS is faster than GMP here is therefore that its loops are static and may be unrolled.

Multiplication benefits from parallelism. On those architectures which can only launch one multiplication each cycle (all but Itanium), the performance ad-

vantage for the multiplication is similar to that of the addition, and for the same reasons. However, on the Itanium architecture, which can launch two pipelined multiplications each cycle, the performance advantage of SCS multiplication over GMP is much higher than that of the addition. This tends to show that GMP fails to exploit this parallelism. To verify that SCS does exploit it, we had a look at the SCS machine code generated by the compiler. The Itanium machine language is interesting in that it explicitly expresses instruction-level parallelism. We could observe that among the 40 fused multiply-and-add involved in the computation of one SCS multiplication, there were 9 places where two multiplications were launched in parallel. An example of this code is given below.

(...)	
;;	
getf.sig r18 = f6	The ;; delimitate <i>bundles</i> of independent expressions that can be launched in parallel.
xma.l f7 = f33, f11, f0	
xma.l f6 = f37, f15, f0	
;;	
add r14 = r18, r14	
xma.l f11 = f13, f11, f9	xma is the integer multiply-and-add instruction.
xma.l f8 = f14, f12, f0	
;;	
(...)	

Only 9 out of 40 is a relatively disappointing result. Should we blame the compiler? Remember that each multiply-and-add instruction needs to be surrounded with two long-latency instructions which transfer the data from the integer datapath to the FP datapath and back (the `getf` instruction above). Initially loading the input digits from memory is also a long-latency operation. These structural hazards probably prevent exploiting the full parallelism of Fig. 1.

Applications: division and logarithm. Concerning division, the algorithms used by SCS and GMP are completely different: SCS division is based on a Newton-Raphson iteration, while GMP uses a digit-recurrence algorithm [11, 12]. These results suggest an obvious improvement to the SCS library.

Finally, the logarithm performance is very close to the multiplication performance: The bulk of the computation time is spent in performing multiplications. We believe that this is a typical application. It clearly justifies the importance of exploiting parallelism in the MP multiplication.

4 Conclusion and Future Work

We have presented and compared measures of performance of several multiple-precision libraries. Our main result is that a MP representation which wastes space and requires more instructions, but exposes parallelism, is a sensible choice on today's deeply pipelined, superscalar processors. Although written in a high-

level language in a portable way, our SCS library is able to outperform GMP, a library partially written in handcrafted assembly code, on a range of processors.

It may be safely expected that future processors will offer even more parallelism. This may take the form of deeper pipeline, although the practical limit is not far from being reached [13]. We also expect that future processors will be able to launch more multiplications each cycle, either in the Itanium fashion (several fully symmetric FP units each capable of multiplication and addition), or through ever more powerful multimedia instructions. The current trend towards hardware multithreading also justifies increasing the number of processing units.

In this case, the SCS approach will prove increasingly relevant, and multiple-precision computing may become another field where assembly programming is no longer needed. Using Brent's variant [4], where carry-save bits impose a carry-propagation every 2^{M-m} bits, these ideas may even find their way into the core of GMP. The pertinence of this approach and the tradeoffs involved remain to be studied.

Acknowledgements. The support of Intel and HP through the donation of an Itanium based machine is gratefully acknowledged. Some experiments were also performed thanks to the HP TestDrive program.

References

1. GMP, the GNU multi-precision library. <http://swox.com/gmp/>.
2. IBM accurate portable math. library. <http://oss.software.ibm.com/mathlib/>.
3. David H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, 1995.
4. Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.
5. K. Briggs. Doubledouble floating point arithmetic. <http://members.lycos.co.uk/keithmbriggs/doubledouble.html>.
6. Marc Daumas. Expansions: lightweight multiple precision arithmetic. In *Architecture and Arithmetic Support for Multimedia*, Dagstuhl, Germany, 1998.
7. D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002–08.
8. John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach (third edition)*. Morgan Kaufmann, 2003.
9. Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, June 2001.
10. Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.
11. I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, 1993.
12. B. Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. Oxford University Press, 2000.
13. Y. Patt, D. Grunwald, and K. Skadron, editors. *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 2002.

A Polymorphic Type System for Bulk Synchronous Parallel ML

Frédéric Gava and Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic – University Paris XII
61, avenue du général de Gaulle – 94010 Créteil cedex – France
{gava,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 data structure named parallel vector, which is given by intention. In order to have an execution that follows the BSP model, and to have a simple cost model, nesting of parallel vectors is not allowed. The novelty of this paper is a type system which prevents such nesting. This system is correct w.r.t. the dynamic semantics which is also presented.

1 Introduction

Bulk Synchronous Parallel ML or BSML is an extension of the ML family of functional programming languages for programming direct-mode parallel Bulk Synchronous Parallel algorithms as functional programs. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [17] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP algorithm is said to be in *direct mode* [2] 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* BS λ calculus [7]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

The BSMLlib library implements the BSML primitives using Objective Caml [13] and MPI [15]. It is efficient [6] and its performance follows curves predicted by the BSP cost model.

Our goal is to provide a certified programming environment for bulk synchronous parallelism. This environment will contain a byte-code compiler for BSML and an extension to the Coq Proof Assistant used to certify BSML programs. A first parallel abstract machine for the execution of BSML programs has been designed and proved correct w.r.t. the BS λ -calculus, using an intermediate semantics [5].

One of the advantages of the Objective Caml language (and more generally of the ML family of languages, for e.g. [9]) is its static polymorphic type inference [10]. In order to have both simple implementation and cost model that follows the

BSP model, nesting of parallel vectors is not allowed. BSMLlib being a library, the programmer is responsible for this absence of nesting. This breaks the safety of our environment.

The novelty of this paper is a type system which prevents such nesting (section 4). This system is correct w.r.t. the dynamic semantics which is presented in section 3. We first present the BSP model, give an informal presentation of BSML (2), and explain in detail why nesting of parallel vectors must be avoided (2.1).

2 Functional Bulk Synchronous Parallelism

Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [17,14] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p , the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h -relation in time gh for any arity h .

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next super-step. The execution time of a super-step s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time: $\text{Time}(s) = \max_{i:\text{processor}} w_i^{(s)} + \max_{i:\text{processor}} h_i^{(s)} * g + l$ where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s . The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is therefore a sum of 3 terms: $W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n , or of more complex parameters like data skew and histogram sizes.

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 that the value of `bsp_p()`

is p , the static number of processes of the parallel machine. This value does not change during execution.

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. Our type system enforces this restriction.

The BSMML 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 (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step).

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 . Neither the implementation of BSMMLib, nor its semantics prescribe a synchronization barrier between two successive uses of `apply`.

Readers familiar with BSPLib will observe that we ignore the distinction between a communication request and its realization at the barrier. 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 .

The full language would also contain a synchronous conditional operation:

`ifat: (bool par) * int * 'a * 'a -> 'a`

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 BSMMLib 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.

2.1 Motivations

In this section, we present why we want to avoid nesting of parallel vectors in our language. Let consider the following BSML program:

```
(* bcast: int->'a par->'a par *)
let bcast n vec =
  let tosend=mkpar(fun i v dst->if i=n then Some v else None) in
  let recv=put(apply tosend vec) in
  apply (replicate noSome) (apply recv (replicate n))
```

This program uses the following functions:

```
(* replicate: 'a -> 'a par *) let replicate x = mkpar(fun pid->x)
(* noSome: 'a option -> 'a *) let noSome (Some x) = x
```

`bcast 2 vec` broadcasts the value of the parallel vector `vec` held at process 2 to all other processes. The BSP cost for a call to this program is:

$$p + (p - 1) \times s \times g + l \quad (1)$$

where s is the size of the value held at process 2. Consider now the expression:

```
let example1 = mkpar(fun pid->bcast pid vec)
```

Its type is $(\tau \text{ par}) \text{ par}$ where τ is the type of the components of the parallel vector `vec`. A first problem is the meaning of this expression. In section 2, we said that `(mkpar f)` evaluates to a parallel vector such that process i holds value `(f i)`. In the case of our example, it means that process 0 should hold the value of `(bcast 0 vec)`. BSML being based on the confluent calculus [7], it is possible to evaluate `(bcast 0 vec)` sequentially. But in this case the execution time will not follow the formula (1). The cost of an expression will then depend on its context. The cost model will no more be compositional.

We could also choose that process 0 broadcasts the expression `(bcast 0 vec)` and that all processes evaluate it. In this case the execution time will follow the formula (1). But the broadcast of the expression will need communications and synchronization. This preliminary broadcast is not needed if `(bcast 0 vec)` is not under a `mkpar`. Thus we have additional costs that make the cost model still non compositional. Furthermore, this solution would imply the use of a scheduler and would make the cost formulas very difficult to write.

To avoid those problems, nesting of parallel vectors is not allowed. The typing ML programs is well-known [10] but is not suited for our language. Moreover, it is not sufficient to detect nesting of abstract type `'a par` such as the previous example. Consider the following program:

```
let example2=mkpar(fun pid->let this=mkpar(fun pid->pid) in pid)
```

Its type is `int par` but its evaluation will lead to the evaluation of the parallel vector `this` inside the outmost parallel vector. Thus we have a nesting of parallel vectors which cannot be seen in the type.

Other problems arise with polymorphic values. The most simple example is a projection: `let fst = fun (a,b) -> a`. Its type is of course `'a * 'b -> 'a`. The problem is that some instantiations are incorrect. We give four cases of the application of `fst` to different kinds of values:

1. two usual values: `fst(1,2)`
2. two parallel values: `fst (mkpar(fun i -> i),mkpar(fun i -> i))`
3. parallel and usual: `fst (mkpar(fun i -> i),1)`
4. usual and parallel: `fst (1, mkpar(fun i -> i))`

The problem arises with the fourth case. Its type given by the Objective Caml system is `int`. But the evaluation of the expression needs the evaluation of a parallel vector. Thus we may be in a situation such as in `example2`. One solution would be to have a syntactic distinction between global and local variables (as in the $\text{BS}\lambda$ -calculus). The type system would be simpler but it would be very inconvenient for the programmer since he would have for example to write three different versions of the `fst` function (the fourth is incorrect).

The nesting can be more difficult to detect:

```
let vec1 = mkpar(fun pid -> pid)
and vec2 = put(mkpar(fun pid -> fun from -> 1+from)) in
  let c1=(vec1,1)    and   c2=(vec2,2) in
    mkpar(fun pid -> if pid < (nproc/2) then snd c1 else snd c2)
```

The evaluation of this expression would imply the evaluation of `vec1` on the first half of the network and `vec2` on the second. But `put` implies a synchronization barrier and not `mkpar` so this will lead to mismatched barriers and the behavior of the program will be unpredictable. The goal of our type system is to reject such expressions. We are first going to equip the language with a *dynamic semantics*, then we will give the inference rules of the *static semantics* and some typing examples.

3 Dynamic Semantics of BSML

Definition of mini-BSML. Reasoning on the complete definition of a functional and parallel language such as BSML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language. It is an attempt to trade between integrating the principal features of functional and BSP language, and being simple. The *expressions* of mini-BSML, written e possibly with a prime or subscript, have the abstract syntax given in Figure 3. In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e , to an argument e' . The form **fun** $x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the first-class function whose parameter is x and whose result is the value of e . Constants c are the integers 1, 2, the booleans and we assume having a unique value $()$ that have the type unit.

$+(n_1, n_2)$	$\xrightarrow[\delta]{\varepsilon} n$ with $n = n_1 + n_2$	(δ_+)
$\mathbf{fst}(\tilde{v}_1, \tilde{v}_2)$	$\xrightarrow[\delta]{\varepsilon} \tilde{v}_1$	(δ_{fst})
$\mathbf{fix}(\mathbf{fun} \ x \rightarrow \tilde{e})$	$\xrightarrow[\delta]{\varepsilon} \tilde{e}[x \leftarrow \mathbf{fix}(\mathbf{fun} \ x \rightarrow \tilde{e})]$	(δ_{fix})
$\mathbf{if} \ \mathbf{true} \ \mathbf{then} \ \tilde{e}_1 \ \mathbf{else} \ \tilde{e}_2$	$\xrightarrow[\delta]{\varepsilon} \tilde{e}_1$	$(\delta_{ifthenelseT})$
$\mathbf{if} \ \mathbf{false} \ \mathbf{then} \ \tilde{e}_1 \ \mathbf{else} \ \tilde{e}_2$	$\xrightarrow[\delta]{\varepsilon} \tilde{e}_2$	$(\delta_{ifthenelseF})$
$\mathbf{isnc}(v)$	$\xrightarrow[\delta]{\varepsilon} \mathbf{false}$ if $v \neq \mathbf{nc}()$	(δ_{isnc})
$\mathbf{isnc}(\mathbf{nc}())$	$\xrightarrow[\delta]{\varepsilon} \mathbf{true}$	(δ_{isnc})

Fig. 1. δ -rules for some primitives operators

$\mathbf{mkpar}(\mathbf{fun} \ x \rightarrow e)$	$\xrightarrow[\delta_g]{\varepsilon} \langle e[x \leftarrow 0], \dots, e[x \leftarrow (p-1)] \rangle$	(δ_{mkpar})
$\mathbf{apply}(\langle \mathbf{fun} \ x \rightarrow e_0, \dots, \mathbf{fun} \ x \rightarrow e_{p-1} \rangle, \langle v_0, \dots, v_{p-1} \rangle)$	$\xrightarrow[\delta_g]{\varepsilon} \langle e_0[x \leftarrow v_0], \dots, e_{p-1}[x \leftarrow v_{p-1}] \rangle$	(δ_{apply})
$\mathbf{if} \ \langle \dots, \overbrace{\mathbf{true}}^n, \dots \rangle \ \mathbf{at} \ v_g \ \mathbf{then} \ e_{g1} \ \mathbf{else} \ e_{g2}$	$\xrightarrow[\delta]{\varepsilon} e_{g1} \quad \text{if } v_g = n$	(δ_{ifatT})
$\mathbf{if} \ \langle \dots, \overbrace{\mathbf{false}}^n, \dots \rangle \ \mathbf{at} \ v_g \ \mathbf{then} \ e_{g1} \ \mathbf{else} \ e_{g2}$	$\xrightarrow[\delta]{\varepsilon} e_{g2} \quad \text{if } v_g = n$	(δ_{ifatF})
$\mathbf{put}(\langle \mathbf{fun} \ dst \rightarrow e_0, \dots, \mathbf{fun} \ dst \rightarrow e_{p-1} \rangle)$	$\xrightarrow[\delta_g]{\varepsilon} \langle e'_0, \dots, e'_{p-1} \rangle$	(δ_{put})

where $\forall i \ e'_j = \mathbf{let} \ v_0^i = e_0[dst \leftarrow i] \ \mathbf{in} \dots \mathbf{let} \ v_{p-1}^i = e_{p-1}[dst \leftarrow i] \ \mathbf{in} \ f_i$
 where $\forall i \forall j \ v_j^i \notin \mathcal{F}(e_j)$
 and where $f_i = \mathbf{fun} \ x \rightarrow \mathbf{if} \ x = 0 \ \mathbf{then} \ v_0^i \ \mathbf{else} \dots \mathbf{if} \ x = (p-1) \ \mathbf{then} \ v_{p-1}^i \ \mathbf{else} \ \mathbf{nc}()$

Fig. 2. δ -rules for some parallel operators

$e ::= x$	variable		c	constant
op	primitive operation		$\mathbf{fun} \ x \rightarrow e$	function abstraction
$(e \ e)$	application		$\mathbf{let} \ x = e \ \mathbf{in} \ e$	local binding
(e, e)	pair		$\mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$	conditional
$\mathbf{if} \ e \ \mathbf{at} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$	global conditional			

Fig. 3. Mini-Bsml syntax

Local values:

$v ::= \mathbf{fun} \ x \rightarrow e$	functional value
c	constant
op	primitive
(v, v)	pair

Global values:

$v_g ::= \mathbf{fun} \ x \rightarrow e_g$	functional value
c	constant
op	primitive
(v_g, v_g)	pair
$\langle v, \dots, v \rangle$	p-wide parallel vector

Fig. 4. Values

The set of primitive operations *op* contains arithmetic operations, fix-point operator **fix**, test function **isnc** of the **nc** constructor (which plays the role of the **None** constructor in Objective Caml) and our parallel operations: **mkpar**, **apply**, **put** and **ifat**.

Before typing these expressions, we present the dynamic semantics of the language, i.e., how the expressions of mini-BXML are computed to *values*. There is one semantics per value of *p*, the number of processes of the parallel machine. In the following, $\forall i$ means $\forall i \in \{0, \dots, p-1\}$. There is two kinds of values: local and global values (Figure 4). e_g are expressions extended with parallel vectors of expressions : $\langle e, \dots, e \rangle$. We noted $\tilde{v}$ (resp. $\tilde{e}$) for a local or global value (resp. expressions or extended expressions).

Small-step semantics. The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use a *small-step semantics*. It consists of a predicate between extended expressions and another extended expression defined by a set of axioms and rules called steps. The small-step semantics describes all the steps of the calculus from an extended expression to a global value and has the following form:

$$\begin{array}{l} e_g \rightarrow e'_g \quad \text{for one step} \\ e_{g_0} \rightarrow e_{g_1} \rightarrow \dots \rightarrow v_g \quad \text{for all the steps of the calculus} \end{array}$$

We note $\xrightarrow{*}$, for the transitive closure of $\rightarrow$ and note $e_{g_0} \xrightarrow{*} v_g$ for $e_{g_0} \rightarrow e_{g_1} \rightarrow \dots \rightarrow v_g$. To define the relation, $\rightarrow$, we begin with some axioms for two relations, $\xrightarrow{\varepsilon}$ and $\xrightarrow{\varepsilon}_{\boxtimes}$, of the head reduction:

$$\begin{array}{ll} (\text{fun } x \rightarrow e) v \xrightarrow{\varepsilon} e[x \leftarrow v] & (\text{let } x = v \text{ in } e) \xrightarrow{\varepsilon} e[x \leftarrow v] \\ (\text{fun } x \rightarrow e_g) v_g \xrightarrow{\varepsilon}_{\boxtimes} e_g[x \leftarrow v_g] & (\text{let } x = v_g \text{ in } e_g) \xrightarrow{\varepsilon}_{\boxtimes} e_g[x \leftarrow v_g] \end{array}$$

We write $e[x \leftarrow v]$ (resp. $e_g[x \leftarrow v_g]$) the expression by substituting all the free occurrences of x in e by v (resp. extended expression). For the primitive operators we have some axioms, the δ -rules, noted $\xrightarrow{\delta}_{\delta}$ (Figure 1 for global and local values) and in the same manner we noted $\xrightarrow{\delta}_{\delta_g}$ for the δ -rules of the parallel operators (Figure 2).

We define two kinds of *head reductions*:

$$\text{Local reduction: } \xrightarrow{\varepsilon}_l = \xrightarrow{\varepsilon} \cup \xrightarrow{\varepsilon}_{\delta} \quad \text{Global reduction: } \xrightarrow{\varepsilon}_g = \xrightarrow{\varepsilon}_{\boxtimes} \cup \xrightarrow{\varepsilon}_{\delta} \cup \xrightarrow{\varepsilon}_{\delta_g}$$

It is easy to see that we cannot always make a head reduction. We have to reduce in depth in the extended sub-expression. To define this deep reduction, we use the following inference rules:

$$\frac{e \xrightarrow{\varepsilon}_l e'}{\Gamma_l(e) \rightarrow \Gamma_l(e')} \quad \frac{e_g \xrightarrow{\varepsilon}_g e'_g}{\Gamma(e_g) \rightarrow \Gamma(e'_g)}$$

In this rule, Γ and Γ_l are *evaluation contexts*, i.e., an expression with a hole and has the abstract syntax given in figure 5. With the evaluation context Γ_l , we can remark that the head reduction is always in a component of a parallel vector (and not for Γ), i.e, a local evaluation. Thus our two kinds of contexts exclude each other by construction.

$\Gamma ::=$	$\square$	head evaluation
	$(\Gamma \tilde{e})$	right application evaluation
	$(\tilde{v} \Gamma)$	left application evaluation
	$(\Gamma, \tilde{e})$	left pair evaluation
	$(\tilde{v}, \Gamma)$	right pair evaluation
	let $x = \Gamma$ in $\tilde{e}$	let evaluation
	if Γ then $\tilde{e}_1$ else $\tilde{e}_2$	conditional
	if Γ at e_{g_1} then e_{g_2} else e_{g_3}	global conditional
	if v_g at Γ then e_{g_2} else e_{g_3}	global conditional
$\Gamma_l ::=$	$(\Gamma_l e_g)$	
	$(v_g \Gamma_l)$	
	(Γ_l, e_g)	
	(v_g, Γ_l)	
	let $x = \Gamma_l$ in e_g	
	if Γ then e_{g_1} else e_{g_2}	
	if Γ at e_{g_1} then e_{g_2} else e_{g_3}	
	if v_g at Γ then e_{g_2} else e_{g_3}	
	$\langle \Gamma, e_1, \dots, e_{p-1} \rangle$	parallel vector, first component
	$\vdots$	
	$\langle e_0, \dots, \overbrace{\Gamma}^i, e_{i+1}, \dots, e_{p-1} \rangle$	i^{th} component
	$\vdots$	
	$\langle e_0, \dots, e_{p-2}, \Gamma \rangle$	last component

Fig. 5. Evaluation contexts

4 A Polymorphic Type System

Type algebra. We begin by defining the term algebra for the basic kinds of semantic objects: the simple types. Simple types are defined by the following grammar:

$\tau ::=$	κ	base type (bool, int, unit etc.)
	α	type variable
	$\tau_1 \rightarrow \tau_2$	type of function from τ_1 to τ_2
	$\tau_1 * \tau_2$	type for pair
	$(\tau \text{ par})$	parallel vector type

We want to distinguish between three subsets of simple types. The set of local types L , which represent usual Objective Caml types, the variable types V for polymorphic types and global types G , for parallel objects. The local types (written $\dot{\tau}$) are:

$$\dot{\tau} ::= \kappa \mid \dot{\tau}_1 \rightarrow \dot{\tau}_2 \mid \check{\tau} \rightarrow \dot{\tau} \mid \dot{\tau}_1 * \dot{\tau}_2$$

the variable types are (written $\check{\tau}$):

$$\check{\tau} ::= \alpha \mid \dot{\tau}_1 \rightarrow \check{\tau}_2 \mid \check{\tau}_1 \rightarrow \check{\tau}_2 \mid \check{\tau}_1 * \check{\tau}_2 \mid \check{\tau}_1 * \dot{\tau}_2 \mid \dot{\tau}_1 * \check{\tau}_2$$

and the global types (written $\bar{\tau}$) are:

$$\begin{aligned} \bar{\tau} ::= & (\check{\tau} \text{ par}) \mid (\dot{\tau} \text{ par}) \mid \check{\tau}_1 \rightarrow \bar{\tau}_2 \mid \dot{\tau}_1 \rightarrow \bar{\tau}_2 \mid \bar{\tau}_1 \rightarrow \bar{\tau}_2 \\ & \mid \bar{\tau}_1 * \bar{\tau}_2 \mid \check{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \check{\tau}_2 \mid \dot{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \dot{\tau}_2 \end{aligned}$$

Of course, we have $L \cap G = \emptyset$ and $V \cap G = \emptyset$. But, it is easy to see that not every instantiations from a variable type are in one of these kinds of types. Take for example, the simple type $(\alpha \text{ par}) \rightarrow \text{int}$ or the simple type $(\alpha \text{ par})$ and the instantiation $\alpha = (\text{int par})$: it leads to a nesting of parallel vectors.

To remedy this problem, we will use *constraints* to say which variables are in L or not. For a polymorphic type system, with this kind of constraints, we introduce a *type scheme* with constraints to generically represent the different types of an expression:

$$\sigma ::= \forall \alpha_1 \dots \alpha_n. [\tau / C]$$

Where τ is a simple type and C is a constraint of classical propositional calculus given by the following grammar:

$$\begin{aligned} C ::= & \mathbf{True} && \text{true constant constraints} \\ & \mid \mathbf{False} && \text{false constant} \\ & \mid \mathcal{L}(\alpha) && \text{locality of variable of type} \\ & \mid C_1 \wedge C_2 && \text{conjunction of two constraints} \\ & \mid C_1 \Rightarrow C_2 && \text{implication of two constraints} \end{aligned}$$

When the set of variables is empty, we simply write $[\tau / C]$ and do not write the constraints when they are equal to **True**. We suppose that we work modulo these following equations that are natural for the $\wedge$ operators: **True** $\wedge$ $C = C$, $C \wedge C = C$ and the commutativity of the $\wedge$ operator.

For a simple type τ , $\mathcal{L}(\tau)$ says that the simple type is in L and we uses the following inductive rules to not have the locality of a type but of its variables:

$$\begin{aligned} \mathcal{L}(\alpha) &= \mathbf{True} && \text{if } \alpha \in \kappa && \mathcal{L}(\tau \text{ par}) = \mathbf{False} \\ \mathcal{L}(\tau_1 \rightarrow \tau_2) &= \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2) && && \mathcal{L}(\tau_1 * \tau_2) = \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2) \end{aligned}$$

In the type system and for the substitution of a type scheme we will use rules to construct constraints from a simple type called *basic constraints*. We note C_τ for the basic constraints from the simple type τ and we use the following inductive rules:

$$\begin{aligned} C_\tau &= \mathbf{True} && \text{if } \tau \text{ atomic} && C_{(\tau_1 \rightarrow \tau_2)} = C_{\tau_1} \wedge C_{\tau_2} \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1) \\ C_{(\tau \text{ par})} &= \mathcal{L}(\tau) \wedge C_\tau && && C_{(\tau_1 * \tau_2)} = C_{\tau_1} \wedge C_{\tau_2} \end{aligned}$$

In our type system, we will use basic constraints and constraints associated to sub-expressions that are needed in cases similar to **example2**.

The set of free variable of a type scheme is defined by:

$$\mathcal{F}(\forall \alpha_1 \dots \alpha_n. [\tau/C]) = (\mathcal{F}(\tau) \cup \mathcal{F}(C)) \setminus \{\alpha_1, \dots, \alpha_n\}$$

where the free variables of the type and the constraints are defined by trivial structural induction. We note Dom for the domain of a *substitution* (i.e. a finite application of variables of type to simple types). With these definitions we can define the substitution on a type scheme:

Definition 1 *The substitution on a type scheme is defined by:*

$$\varphi(\forall \alpha_1 \dots \alpha_n. [\tau/C]) = \forall \alpha_1 \dots \alpha_n. [\varphi(\tau)/\varphi(C)] \bigwedge_{\beta_i \in Dom(\varphi) \cap \mathcal{F}([\tau/C])} C_{\varphi(\beta_i)}$$

if $\alpha_1 \dots \alpha_n$ are out of reach of φ .

We say that a variable α is *out of reach* of a substitution φ if: $\varphi(\alpha) = \alpha$, i.e φ don't modify α (or α is not in the domain of φ) and if α is not free in $[\tau/C]$, then α is not free in $\varphi([\tau/C])$, i.e, φ do not introduce α in its result. The condition that $\alpha_1 \dots \alpha_n$ are out of reach of φ can always be validated by renaming first $\alpha_1 \dots \alpha_n$ with fresh variables (we suppose that we have an infinite set of variables). The substitution on the simple type and of the constraints are defined by trivial structural induction.

Instantiation and Generalization. A type scheme can be seen like the set of types given by instantiation of the quantifier variables. We introduce the notion of instance of a type scheme with constraints.

Definition 2 *We note $[\tau/C] \leq \forall \alpha_1 \dots \alpha_n. [\tau'/C']$ if and only if, there exists a substitution φ of domain $\alpha_1, \dots, \alpha_n$ where $[\tau/C] = \varphi([\tau'/C'])$.*

We write E for an *environment* which associates type schemes to free variables of an expression. It is an application from free variables (identifiers) of expressions to type scheme. We note $Dom(E) = \{x_1, \dots, x_n\}$ for its domain, i.e the set of variables associated. We assume that all the identifiers are distinct. The empty mapping is written $\emptyset$ and $E(x)$ for the type scheme associated with x in E . The substitution φ on E is a point to point substitution on the domain of E . The set of free variables is naturally defined on the free variables on all the type scheme associated in the domain of E .

Finally, we write $E + \{x : \sigma\}$ for the extension of E to the mapping of x to σ . If, before this operation, we have $x \in Dom(E)$, we can replace the range by the new type scheme for x . To continue with the introduction of the type system, we define how to generalize a type scheme. Yet, type schemes have universal quantified variables, but not all the variables of a type scheme can.

$TC(i) = \mathbf{int} \quad i = 0, 1, \dots$	$TC(\mathbf{fix}) = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$
$TC(b) = \mathbf{bool} \quad b = \mathbf{true}, \mathbf{false}$	$TC(\mathbf{fst}) = \forall \alpha \beta. [(\alpha * \beta) \rightarrow \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$
$TC(()) = \mathbf{unit}$	$TC(\mathbf{snd}) = \forall \alpha \beta. [(\alpha * \beta) \rightarrow \beta / \mathcal{L}(\beta) \Rightarrow \mathcal{L}(\alpha)]$
$TC(+) = (\mathbf{int} * \mathbf{int}) \rightarrow \mathbf{int}$	$TC(\mathbf{mkpar}) = \forall \alpha. [(\mathbf{int} \rightarrow \alpha) \rightarrow (\alpha \text{ par}) / \mathcal{L}(\alpha)]$
$TC(\mathbf{nc}) = \forall \alpha. \mathbf{unit} \rightarrow \alpha$	$TC(\mathbf{isnc}) = \forall \alpha. [\alpha \rightarrow \mathbf{bool} / \mathcal{L}(\alpha)]$
$TC(\mathbf{apply}) = \forall \alpha \beta. [((\alpha \rightarrow \beta) \text{ par } * (\alpha \text{ par})) \rightarrow (\beta \text{ par}) / \mathcal{L}(\alpha) \wedge \mathcal{L}(\beta)]$	
$TC(\mathbf{put}) = \forall \alpha. [(\mathbf{int} \rightarrow \alpha) \text{ par} \rightarrow (\mathbf{int} \rightarrow \alpha) \text{ par} / \mathcal{L}(\alpha)]$	

Fig. 6. Definition of TC

Definition 3 Given an environment E , a type scheme $[\tau/C]$ without universal quantification, we define an operator Gen to introduce universal quantification: $Gen([\tau/C], E) = \forall \alpha_1 \dots \alpha_n. [\tau/C]$ where $\{\alpha_1, \dots, \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E)$

With this definition, we have introduced polymorphism. The universal quantification gives the choice for the system to take the good type from a type scheme.

Inductive rules. We note TC (Figure 6) the function which associates a type scheme to the constants and to the primitive operations.

We formulate type inference by a deductive proof system that assigns a type to an expression of the language. The context in which an expression is associated with a type is represented by an environment which maps type scheme to identifiers. Deductions produce conclusions of the form $E \vdash e : [\tau/C]$ which are called *typing judgments*, they could be read as: "in the type environment E , the expression e has the type $[\tau/C]$ ". The static semantics manipulates type schemes by using the mechanism of generalization and instantiation specified in the previous sections. Now the inductive rules of the type system are given in the Figure 7.

In all the inductive rules, if a constraint C is such that $Solve(C) = \mathbf{False}$ then the inductive rule cannot be applied and then the expression is not well typed. To *Solve* the constraints we use the classical boolean reduct rules of propositional calculus and the rules to transform the locality of type to constraints. Our constraints are a sub part of the propositional calculus, so *Solve* is a decidable function.

Like traditional static type systems, the case (Op) , $(Const)$ and (Var) used the definition of an instance of a type scheme. The rule (Fun) , introduce a new type scheme on the environment with the basic constraints from the simple type for carrying of well parameters. In the rules (App) , $(Pair)$ and (Let) we make the conjunction of the constraints to known if the two sub-cases are correct each other. Moreover, in (Let) , we introduce the fact that $\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)$ because an expression like **let** $x = e_1$ **in** e_2 can be seen like $(\mathbf{fun} \ x \rightarrow e_2) \ e_1$. So we have to protect our type system against expression from global values to local values like in **example 2**. The rule $(ifthenelse)$ and $(ifat)$ make also the conjunction of the constraints. The **if then else** construction could return global or usual value. But the **if at** is a synchronous construction which needs global values so

$$\begin{array}{c}
\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]} (\text{Var}) \quad \frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]} (\text{Const}) \quad \frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]} (\text{Op}) \\
\\
\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e : [\tau_2/C_2]}{E \vdash (\text{fun } x \rightarrow e) : [\tau_1 \rightarrow \tau_2 / C_{(\tau_1 \rightarrow \tau_2)} \wedge C_2]} (\text{Fun}) \\
\\
\frac{E \vdash e_1 : [\tau' \rightarrow \tau / C_1] \quad E \vdash e_2 : [\tau' / C_2]}{E \vdash (e_1 \ e_2) : [\tau / C_1 \wedge C_2]} (\text{App}) \\
\\
\frac{E \vdash e_1 : [\tau_1 / C_1] \quad E + \{x : \text{Gen}([\tau_1 / C_1], E)\} \vdash e_2 : [\tau_2 / C_2]}{E \vdash \text{let } x = e_1 \text{ in } e_2 : [\tau_2 / C_1 \wedge C_2 \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)]} (\text{Let}) \\
\\
\frac{E \vdash e_1 : [\tau_1 / C_1] \quad E \vdash e_2 : [\tau_2 / C_2]}{E \vdash (e_1, e_2) : [\tau_1 * \tau_2 / C_1 \wedge C_2]} (\text{Pair}) \\
\\
\frac{E \vdash e_1 : [\text{bool} / C_{e_1}] \quad E \vdash e_2 : [\tau / C_{e_2}] \quad E \vdash e_3 : [\tau / C_{e_3}]}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : [\tau / C_{e_1} \wedge C_{e_2} \wedge C_{e_3}]} (\text{Ifthenelse}) \\
\\
\frac{E \vdash e_1 : [\text{bool } \text{par} / C_{e_1}] \quad E \vdash e_2 : [\text{int} / C_{e_2}] \quad E \vdash e_3 : [\tau / C_{e_3}] \quad E \vdash e_4 : [\tau / C_{e_4}]}{E \vdash \text{if } e_1 \text{ at } e_2 \text{ then } e_3 \text{ else } e_4 : [\tau / C_{e_1} \wedge C_{e_2} \wedge C_{e_3} \wedge C_{e_4} \wedge (\mathcal{L}(\tau) \Rightarrow \text{False})]} (\text{Ifat})
\end{array}$$

Fig. 7. The inductive rules

we add the fact that $(\mathcal{L}(\tau) \Rightarrow \text{False})$ to not allow a return usual value (i.e. τ in L).

The basic constraints are important in our type system but are not suitable. For the following example, a parallel identity:

```
fun x -> if (mkpar (fun i -> true) at 0 then x else x
```

the basic constraints are not sufficient. Indeed, the simple type given by Objective Caml is $\alpha \rightarrow \alpha$ and the basic constraints $(\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\alpha))$ are always solved to **True**. But it is easy to see that the variable x (of type α) could not be a usual value. Our type system, with constraints from the sub-expression (here **ifat**) would give the type scheme: $[\alpha \rightarrow \alpha / \mathcal{L}(\alpha) \Rightarrow \text{False}]$ (i.e. α could not be a usual value and the instantiation are in G).

Afterwards, we need to know when a constraint is *Solved* to **True**, i.e. it is always a valid constraint. It will be important, notably for the correction of the type system:

Definition 4 We write $\varphi \models C$, if the substitution φ on the free variables of C is such that $\mathcal{F}(\varphi(C)) = \emptyset$ and $\text{Solve}(\varphi(C)) = \text{True}$.

We also write $\phi_C = \{\varphi \mid \varphi \models C\}$ for the set of all the substitutions that have these properties.

Safety. To ensure safety, the type system has been proved correct with respect to the small-step semantics. We say that an extended expression e_g is in *normal form* if and only if $e_g \not\rightarrow$, i.e., there is no rule which could be applicative to e_g .

Theorem 1 (Typing safety) *If $\emptyset \vdash e : [\tau/C]$ and $e \xrightarrow{*} e'_g$ and e'_g is in normal form, then e'_g is a value v_g and there exists C' such that $\forall \varphi \in \phi_C$ then $\varphi \models C'$ and $\emptyset \vdash v_g : [\tau/C']$.*

Proof: see [1].

Why C' and not C ? Because with our type system, the constraints of a typing judgment for e contains constraints of the sub-expression of e . After evaluation, some of this sub-expression could be reduced. Example: **let** $f = (\text{fun } a \rightarrow \text{fun } b \rightarrow a)$ **in** 1 have the type $[\text{int}/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$. This expression reduced to 1 has the type int . Thus we have C' is less constrained than C and we do not have problem with compositionality.

Examples. For the **example 2** given at the beginning of this text, the type scheme given for **this** is $(\text{int } \text{par})$ and the type for pid is the usual **int**. So after a *(Let)* rule, the constraints for this let binding construction are $C = \mathcal{L}(\text{int}) \Rightarrow \mathcal{L}(\text{int } \text{par})$ with $\text{Solve}(C) = \text{False}$. So the expression is not well-typed (Figure 8 gives a part of the typing judgment).

$$\frac{\begin{array}{c} \dots \\ \hline \{pid : \text{int}\} \vdash \text{mkpar}(\text{fun } i \rightarrow i) : (\text{int } \text{par}) \end{array} \quad \frac{\text{int} \leq \text{int}}{\{pid : \text{int}\} \vdash pid : \text{int}}}{\{pid : \text{int}\} \vdash \text{let this} = \text{mkpar}(\text{fun } i \rightarrow i) \text{ in } pid) : ?} \\ \emptyset \vdash (\text{fun } pid \rightarrow \text{let this} = \text{mkpar}(\text{fun } i \rightarrow i) \text{ in } pid) : ?$$

Fig. 8. Typing judgment of a part of **example 2**

In the parallel and usual projection (see Figure 9), the expression is well-typed like we want in the previous section. In Figure 10, we present the typing judgment of another example, accepted by the type system of Objective Caml, but not by ours. For the usual and parallel projection, the projection **fst** has the simple type $(\text{int} * (\text{int } \text{par})) \rightarrow \text{int}$. But, with our type scheme substitution, the constraints of this operator are : $C = \mathcal{L}(\text{int}) \Rightarrow \mathcal{L}(\text{int } \text{par})$. Effectively, we have $\text{Solve}(C) = \text{False}$ and the expression is rejected by our type system. In the typing judgments given in the figures, we noted : ? when the type derivation is impossible for our type system.

$$\frac{\begin{array}{c} \dots \\ \hline \emptyset \vdash (\text{fst} : (\text{int} * \text{int } \text{par}) \rightarrow \text{int} : ? \end{array} \quad \frac{\begin{array}{c} \dots \\ \hline \emptyset \vdash (\text{mkpar}(\text{fun } i \rightarrow i)) : \text{int } \text{par} \end{array} \quad \frac{\text{int} \leq \text{int}}{\vdash 1 : \text{int}}}{\emptyset \vdash (\text{mkpar}(\text{fun } i \rightarrow i), 1) : (\text{int } \text{par} * \text{int})}} \\ \emptyset \vdash \text{fst}(\text{mkpar}(\text{fun } i \rightarrow i), 1) : \text{int } \text{par}$$

Fig. 9. Typing judgment of the third projection example

$$\frac{\dots \quad \frac{\text{int} \leq \text{int}}{\vdash 1 : \text{int}} \quad \frac{\dots}{\emptyset \vdash (\text{mkpar} (\text{fun } i \rightarrow i)) : \text{int } \text{par}}}{\frac{\emptyset \vdash \text{fst} : (\text{int} * \text{int } \text{par}) \rightarrow \text{int} : ? \quad \emptyset \vdash (1, \text{mkpar} (\text{fun } i \rightarrow i)) : (\text{int} * \text{int } \text{par})}{\emptyset \vdash \text{fst} (1, \text{mkpar} (\text{fun } i \rightarrow i)) : ?}}$$

Fig. 10. Typing judgment of the fourth projection example

5 Related Works

In previous work on Caml Flight [3], another parallel ML, the global parallel control structure was prevented dynamically from nesting. A static analysis [16] have been designed but for some kinds of nesting only and in Caml Flight, the parallelism is a side effect while it is purely functional in BSML.

The libraries close to our framework based either on the functional language Haskell [8] or on the object-oriented language Python [4] propose flat operations similar to ours. In the latter, the programmer is responsible for the non nesting of parallel vectors. In the former, the nesting is prohibited by the use of monads. But the distinction between global and local expressions is syntactic thus less general than our framework. For example, the programmer need to write three version of `fst`. Furthermore, Haskell is a lazy language: it is less efficient and the cost prevision is difficult [12].

A general framework for type inference with constrained type called $\text{HM}(\mathcal{X})$ [11] also exists and could be used for a type system with only basic constraints. We do not used this system for three reasons: (1) this type system has been proved for λ -calculus (and sequential languages whose types systems need constraints) and not for our theoretical calculus, the $\text{BS}\lambda$ -calculus with its two level structure (local and global); (2) in the logical type system, the constraints depend of sub-expression are not present; (3) in our type system, our abstraction could not be valid and generate constraints (not in $\text{HM}(\mathcal{X})$). Nevertheless, the ideas (but not the framework itself) of $\text{HM}(\mathcal{X})$ could be used for generalized our work for tuple, sum types and imperative features.

6 Conclusions and Future Work

The Bulk Synchronous Parallel ML allows direct mode Bulk Synchronous Parallel (BSP) programming. To preserve a compositional cost model derived from the BSP cost model, the nesting of parallel vectors is forbidden. The type system presented in this paper allows a static avoidance of nesting. Thus the pure functional subset of BSML is safe. We have also designed an algorithm for type inference and implemented it. It can be used in conjunction with the BSMLlib programming library. The extension of the type system to tuples and sum types

have been investigated but not yet proved correct w.r.t. the dynamic semantics nor included in the type inference algorithm.

A further work will concern imperative features. Dynamic semantics of the interaction of imperative features with parallel operations have been designed. To ensure safety, communications may be needed in case of affectation or references may contain additional information used dynamically to insure that dereferencing of references pointing to local value will give the same value on all processes. We are currently working on the typing of effects to avoid this problem statically.

Acknowledgments. This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org).

References

1. Frédéric Gava. A Polymorphic Type System for BSML. Technical Report 2002–12, University of Paris Val-de-Marne, LACL, 2002.
2. A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
3. G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.
4. K. Hinsén. Parallel Programming with BSP in Python. Technical report, Centre de Biophysique Moléculaire, 2000.
5. F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.
6. F. Loulergue. Implementation of a Functional BSP Programming Library. In *14th Iasted PDCS Conference*, pages 452–457. ACTA Press, 2002.
7. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1–3):253–277, 2000.
8. Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.
9. R. Milner and al. *The Definition of Standard ML*. MIT Press, 1990.
10. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
11. M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
12. C. Paraja, R. Pena, F. Rubio, and C. Segura. A functional framework for the implementation of genetic algorithms: Comparing Haskell and Standard ML. In *Trends in Functional Programming*, volume 2. Intellect Books, 2001.
13. D. Rémy. Using, Understanding, and Unravelling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.
14. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.
15. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
16. J. Vachon. Une analyse statique pour le contrôle des effets de bords en Caml-Flight beta. In C. Queinnec et al., editors, *JFLA*, INRIA, Janvier 1995.
17. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

Towards an Efficient Functional Implementation of the NAS Benchmark FT

Clemens Grelck¹ and Sven-Bodo Scholz²

¹ University of Lübeck, Germany

Institute of Software Technology and Programming Languages

`grelck@isp.uni-luebeck.de`

² University of Kiel, Germany

Institute of Computer Science and Applied Mathematics

`sbs@informatik.uni-kiel.de`

Abstract. This paper compares a high-level implementation of the NAS benchmark FT in the functional array language SAC with traditional solutions based on FORTRAN-77 and C. The impact of abstraction on expressiveness, readability, and maintainability of code as well as on clarity of underlying mathematical concepts is discussed. The associated impact on runtime performance is quantified both in a uniprocessor environment as well as in a multiprocessor environment based on automatic parallelization and on OPENMP.

1 Introduction

Low-level sequential base languages, e.g. FORTRAN-77 or C, and message passing libraries, mostly MPI, form the prevailing tools for generating parallel applications, in particular for numerical problems. This choice offers almost literal control over data layout and program execution, including communication and synchronization. Expertised programmers are enabled to adapt their code to hardware characteristics of target machines, e.g. properties of memory hierarchies, and to enhance the runtime performance to whatever a machine is able to deliver.

During the process of performance tuning, numerical code inevitably mutates from a (maybe) human-readable representation of an abstract algorithm to one that almost certainly is suitable for machines only. Ideas and concepts of underlying mathematical algorithms are completely disguised. Even minor changes to underlying algorithms may require a major re-design of the implementation. Moreover, particular demand is made on the qualification of programmers as they have to be experts in computer architecture and programming technique in addition to their specific application domains. As a consequence, development and maintenance of parallel code is prohibitively expensive.

As an alternative approach, functional languages encourage a declarative style of programming that abstracts from many details of program execution. For example, memory management for aggregate data structures like arrays is completely up to compilers and runtime systems. Even arrays are stateless and

may be passed to and from functions following a call-by-value semantics. Focusing on algorithmic rather than on organizational aspects, functional languages significantly reduce the gap between a mathematical idea and an executable specification; their side-effect free semantics facilitates parallelization [1].

Unfortunately, in numerical computing functional languages have shown performance characteristics inferior to well-tuned (serial) imperative codes to an extent which renders parallelization unreasonable [2]. This observation has inspired the design of the functional array language SAC [3]. SAC (for Single Assignment C) aims at combining high-level program specifications characteristic for functional languages with efficient support for array processing in the style of APL including automatic parallelization (for shared memory systems at the time being) [4,5]. Efficiency concerns are addressed by incorporating both well-known and language-specific optimization techniques into the SAC compiler, where their applicability significantly benefits from the side-effect free, functional semantics of the language¹.

This paper investigates the trade-off between programming productivity and runtime performance by means of a single though representative benchmark: the application kernel FT from the NAS benchmark suite [6]. Investigations on this benchmark involving the functional languages ID [7] and HASKELL [8] have contributed to a pessimistic assessment of the suitability of functional languages for numerical computing in general [2]. We show a very concise, almost mathematical SAC specification of NAS-FT, which gets as close as within a factor of 2.8 to the hand-tuned, low-level FORTRAN-77 reference implementation and outperforms that version by implicitly using four processors of a shared memory multiprocessor system.

2 Implementing the NAS Benchmark FT

The NAS benchmark FT implements a solver for a class of partial differential equations by means of repeated 3-dimensional forward and inverse complex fast-Fourier transforms. They are implemented by consecutive collections of 1-dimensional FFTs on vectors along the three dimensions, i.e., an array of shape $[X, Y, Z]$ is consecutively interpreted as a ZY matrix of vectors of length X , as a ZX matrix of vectors of length Y , and as a XY matrix of vectors of length Z .

The outline of this algorithm can be carried over into a SAC specification straightforwardly, as shown in Fig. 1. The function `FFT` on 3-dimensional complex arrays (`complex[...]`) consecutively transposes the argument array `a` three times. After each transposition, the function `Slice` extracts all subvectors along the innermost axis and individually applies 1-dimensional FFTs to them. The additional parameter `rofu` provides a pre-computed vector of complex roots of unity, which is used for 1-dimensional FFTs. The 3-line definition of `Slice` is omitted here for space reasons and because it requires more knowledge of SAC.

The overloaded function `FFT` on vectors of complex numbers (`complex[.]`) almost literally implements the Danielson-Lanczos algorithm [9]. It is based on

¹ More information on SAC is available at <http://www.sac-home.org/>.

```

complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  a_t = transpose( [2,1,0], a);
  b   = Slice( FFT, a_t, rofu);

  b_t = transpose( [0,2,1], b);
  c   = Slice( FFT, b_t, rofu);

  c_t = transpose( [1,2,0], c);
  d   = Slice( FFT, c_t, rofu);

  return( d);
}

complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even      = condense(2, v);
  odd       = condense(2, rotate( [-1], v));
  rofu_even = condense(2, rofu);

  fft_even  = FFT( even, rofu_even);
  fft_odd   = FFT( odd,  rofu_even);

  left      = fft_even + fft_odd * rofu;
  right     = fft_even - fft_odd * rofu;

  return( left ++ right);
}

complex[2] FFT(complex[2] v, complex[1] rofu)
{
  return( [ v[0] + v[1], v[0] - v[1] ]);
}

```

Fig. 1. SAC implementation of NAS-FT.

the recursive decomposition of the argument vector v into elements at even and at odd index positions. The vector `even` can be created by means of the library function `condense(n,v)`, which selects every n -th element of v . The vector `odd` is generated in the same way after first rotating v by one index position to the left. FFT is then recursively applied to even and to odd elements, and the results are combined by a sequence of element-wise arithmetic operations on vectors of complex numbers and a final vector concatenation (`++`). A direct implementation of FFT on 2-element vectors (`complex[2]`) terminates the recursion. Note that unlike FORTRAN neither the data type `complex` nor any of the operations used to define `FFT` are built-in in SAC; they are all imported from the standard library, where they are defined in SAC itself.

In order to help assessing the differences in programming style and abstraction, Fig. 2 shows excerpts from about 150 lines of corresponding FORTRAN-77 code. Three slightly different functions, i.e. `cffts1`, `cffts2`, and `cffts3`, intertwine the three transposition operations with a block-wise realization of a 1-dimensional FFT. The iteration is blocked along the middle dimension to improve cache performance. Extents of arrays are specified indirectly to allow reuse of the same set of buffers for all orientations of the problem. Function `fftz2` is part of the 1-dimensional FFT. It must be noted that this excerpt represents high quality code, which is well organized and well structured. It was written by

<pre> subroutine cffts1 (is,d,x,xout,y) include 'global.h' integer is, d(3), logd(3) double complex x(d(1),d(2),d(3)) double complex xout(d(1),d(2),d(3)) double complex y(fftbblockpad, d(1), 2) integer i, j, k, jj do i = 1, 3 logd(i) = ilog2(d(i)) end do do k = 1, d(3) do jj = 0, d(2)-fftbblock, fftblock do j = 1, fftblock do i = 1, d(1) y(j,i,1) = x(i,j+jj,k) enddo enddo enddo call cfftz (is, logd(1), d(1), y, y(1,1,2)) do j = 1, fftblock do i = 1, d(1) xout(i,j+jj,k) = y(j,i,1) enddo enddo enddo return end </pre>	<pre> subroutine fftz2 (is,l,m,n,ny,ny1,u,x,y) integer is,k,l,m,n,ny,ny1,n1,li,lj integer lk,ku,i,j,i11,i12,i21,i22 double complex u,x,y,u1,x11,x21 dimension u(n), x(ny1,n), y(ny1,n) n1 = n / 2 lk = 2 ** (l - 1) li = 2 ** (m - 1) lj = 2 * lk ku = li + 1 do i = 0, li - 1 i11 = i * lk + 1 i12 = i11 + n1 i21 = i * lj + 1 i22 = i21 + lk if (is .ge. 1) then u1 = u(ku+i) else u1 = dconjg (u(ku+i)) endif do k = 0, lk - 1 do j = 1, ny x11 = x(j,i11+k) x21 = x(j,i12+k) y(j,i21+k) = x11 + x21 y(j,i22+k) = u1 * (x11 - x21) enddo enddo enddo return end </pre>

Fig. 2. Excerpts from the FORTRAN-77 implementation of NAS-FT.

expert programmers in the field and has undergone several revisions. Everyday legacy FORTRAN-77 code is likely to be less “intuitive”.

3 Experimental Evaluation

This section compares the runtime performance achieved by code compiled from the high-level functional SAC specification of NAS-FT, as outlined in the previous section, with that of two low-level solutions: the serial FORTRAN-77 reference implementation² and a C implementation derived from the reference code and extended by OPENMP directives by Real World Computing Partnership (RWCP)³. All experiments were made on a 12-processor SUN Ultra Enterprise 4000 shared memory multiprocessor using SUN Workshop compilers. Investigations covered size classes W and A; as the findings were almost identical, we focus on size class A in the following.

As shown in Fig. 3, SAC is outperformed by the FORTRAN-77 reference implementation by not more than a factor of 2.8 and by the corresponding C code by a factor of 2.4. To a large extent, this can be attributed to dynamic memory management overhead caused by the recursive decomposition of argument vectors when computing 1-dimensional FFTs. In contrast to SAC, both the FORTRAN-77 and the C implementation use a static memory layout.

² The source code is available at <http://www.nas.nasa.gov/Software/NPB/>.

³ The source code is available at <http://phase.etl.go.jp/Omni/>.

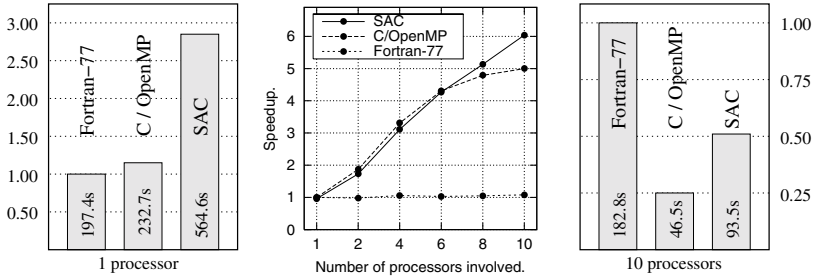


Fig. 3. Runtime performance of NAS-FT: sequential, scalability, ten processors.

Fig. 3 also reports on the scalability of parallelization, i.e. parallel execution times divided by each candidate's best serial runtime. Whereas hardly any performance gain can be observed for automatic parallelization of the FORTRAN-77 code by the SUN Workshop compiler, SAC achieves speedups of up to six. Hence, SAC equalizes FORTRAN-77 with four processors and outperforms it by a factor of about two when using ten processors. SAC even scales slightly better than OPENMP. This is remarkable as the parallelization of SAC code is completely implicit, whereas a total of 25 compiler directives guide parallelization in the case of OPENMP. However, it must also be mentioned that the C/OPENMP solution achieves the shortest absolute 10-processor runtimes due to its superior sequential performance.

4 Related Work and Conclusions

There are various approaches to raise the level of abstraction in array processing from that provided by conventional scalar languages. FORTRAN-90 and ZPL [10] treat arrays as conceptual entities rather than as loose collections of elements. Although they do not at all reach a level of abstraction similar to that of SAC, a considerable price in terms of runtime performance has to be paid [11]. SISAL [12] used to be the most prominent functional array language. However, apart from a side-effect free semantics and implicit memory management the original design provides no support for high-level array processing in the sense of SAC. More recent versions [13] promise improvements, but have not been implemented.

General-purpose functional languages offer a significantly more abstract programming environment. However, investigations involving HASKELL [8] and ID [7] based on the NAS benchmark FT revealed substantial deficiencies both in time and space consumption [2]. Our experiments showed that HASKELL implementations described in [2] are outperformed by the FORTRAN-77 reference implementation by more than two orders of magnitude for size class W. Experiments on size class A failed due to memory exhaustion.

The development of SAC aims at combining high-level functional array programming with competitive runtime performance. The paper evaluates this approach based on the NAS benchmark FT. It is shown how 3-dimensional FFTs can be assembled by about two dozen lines of SAC code as opposed to 150

lines of fine-tuned FORTRAN-77 code in the reference implementation. Moreover, the SAC solution clearly exhibits underlying mathematical ideas, whereas they are completely disguised by performance-related coding tricks in the case of FORTRAN. Nevertheless, the runtime of the SAC implementation is within a factor of 2.8 of the FORTRAN code. Furthermore, the SAC version without any modification outperforms its FORTRAN counterpart on a shared memory multiprocessor as soon as four or more processors are used. In contrast, additional effort and knowledge are required for the imperative solution to effectively utilize the SMP system. Annotation with 25 OPENMP directives succeeded in principle, but did not scale as good as the compiler-parallelized SAC code.

References

1. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer-Verlag (1999)
2. Hammes, J., Sur, S., Böhm, W.: On the Effectiveness of Functional Language Features: NAS Benchmark FT. *Journal of Functional Programming* **7** (1997) 103–123
3. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, accepted for publication
4. Grelck, C.: Shared Memory Multiprocessor Support for SAC. In: Hammond, K., Davie, D., Clack, C. (eds.): Implementation of Functional Languages. Lecture Notes in Computer Science, Vol. 1595. Springer-Verlag (1999) 38–54
5. Grelck, C.: A Multithreaded Compiler Backend for High-Level Array Programming. In: Proc. 21st International Multi-Conference on Applied Informatics (AI'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria, ACTA Press (2003) 478–484
6. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Res. Center (1995)
7. Nikhil, R.: The Parallel Programming Language ID and its Compilation for Parallel Machines. In: Proc. Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, Academic Press (1989)
8. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
9. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press (1993)
10. Chamberlain, B., Choi, S.E., Lewis, C., Snyder, L., Weathersby, W., Lin, C.: The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering* **5** (1998)
11. Frumkin, M., Jin, H., Yan, J.: Implementation of NAS Parallel Benchmarks in High Performance Fortran. In: Proc. 13th International Parallel Processing Symposium/10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99), San Juan, Puerto Rico. (1999)
12. Cann, D.: Retire Fortran? A Debate Rekindled. *Communications of the ACM* **35** (1992) 81–89
13. Feo, J., Miller, P., S.K.Skedzielewski, Denton, S., Solomon, C.: Sisal 90. In: Proc. Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA. (1995) 35–47

Asynchronous Parallel Programming Language Based on the Microsoft .NET Platform

Vadim Guzev¹ and Yury Serdyuk²

¹ Peoples' Friendship University of Russia, Moscow, Russia,
`vadim@joker.botik.ru`

² Program Systems Institute of the Russian Academy of Sciences, Pereslavl-Zalessky,
Russia
`Yury@serdyuk.botik.ru`

Abstract. MC# is a programming language for cluster- and GRID-architectures based on asynchronous parallel programming model accepted in Polyphonic C# language (N.Benton, L.Cardelli, C.Fournet; Microsoft Research, Cambridge, UK). Asynchronous methods of Polyphonic C# play two major roles in MC#: 1) as autonomous methods executed on remote machines, and 2) as methods used for delivering messages. The former are identified in MC# as the "movable methods", and the latter form a special syntactic class with the elements named "channels". Similar to Polyphonic C#, chords are used for defining the channels and as a synchronization mechanism. The MC# channels are generalised naturally to "bidirectional channels", which may be used both for sending and receiving messages in the movable methods. The runtime-system of MC# has as the basic operation a copying operation for the object which is scheduled for execution on remote machine. This copy is "dead" after the movable method has finished its work, and all changes of this remote copy are not transferred to the original object. Arguments of the movable method are copied together with an original object, but the passing of bidirectional channels is realised through transferring the proxies for such channels. By way of experiments in MC#, we have written a series of parallel programs such as a computation of Fibonacci numbers, walking through binary tree, computation of primes by Eratosthenes sieve, calculation of Mandelbrot set, modeling the Conway's game "Life", etc. In all these cases, we got the easy readable and compact code. Also we have an experimental implementation in which the compiler is written in SML.NET, and the execution of movable methods on remote machines is based on the Reflection library of .NET platform.

Keywords: Polyphonic C#, asynchronous parallel programming, movable method, channel, bidirectional channel

1 Introduction

At present time, spread use of computer systems with cluster- and GRID-architectures posed a problem of developing high-level, powerful and flexible

programming languages which allow one to create complex, but at the same time, robust applications that effectively use the possibilities of concurrent computations.

The program interfaces and libraries, which we have now, such as MPI (Message Passing Interface), that are realised for C and Fortran languages, are very low-level and not suited for the modern object-oriented languages, such as C++, C# and Java.

One of the recent seminal achievement in this area is introduction of an asynchronous parallel programming model within the Polyphonic C# programming language in the context of the Microsoft .NET platform [1]. In turn, this model is based on the join-calculus [2] - a process calculus with the high-level message handling mechanism adequately abstracting the low-level mechanism which exists in the current computer systems.

The essence of the new model or, in other words, the key feature of the Polyphonic C# language is the use of so called "asynchronous" methods in addition to conventional synchronous methods of a class. Such asynchronous methods can be declared both autonomously, and in this case they are scheduled for execution in a different thread (either a new one or a working thread from some thread pool), and within a bundle (or a chord, in terminology of Polyphonic C#) of other methods (synchronous and asynchronous). In the latter case, calling an asynchronous method, which was declared in the chord, corresponds to sending a message or posting an event.

Such parallel programming style in Polyphonic C# as before is considered as a programming technique either for a single computer or for many machines interacting through the remote methods calls using .NET Remoting library.

Specific features of the proposed MC# language consist in the transferring of asynchronous parallel programming model of Polyphonic C# to distributed case, where an autonomous asynchronous method can be scheduled for execution in a different machine. With that, the asynchronous methods which are declared by chords and are used to deliver values to synchronous methods, form a special syntactic class with the elements named "channels". Therefore, a parallel program writing in MC# language is reduced to label by the special **movable** keyword the methods which may be transferred for execution to the different processors and arranging their interactions by the channels.

Earlier, an analogous approach in which a programmer has been partitioning all functions in the program into "movable" and "unmovable", used in the T-system [4]. This system is intended for the dynamic scheduling of execution of parallel programs written in an extension of C.

Though the channels in MC# are "one-directional" in their nature (same as in the join-calculus), nevertheless they are generalised naturally to "bidirectional" channels which may be used by movable methods both for sending and receiving messages.

An implementation of the MC# language consists of a compiler for translating from the input language of a system to C#, and a runtime-system to execute a translated program. A compiler replaces the movable methods calls in the source program to queries to manager of computational resources that schedules the execution of parallel fragments of program in computer system.

Having received a query, a manager selects the most suitable node of multiprocessor and copies an object, of which the movable method is scheduled for remote execution, to the selected node together with the arguments of this method. This copy is "dead" after the movable method has finished its work, and all changes that occurred to it are not transferred to the original object. Passing of bidirectional channels as arguments of methods is realised through the transferring the proxies for such channels. Thus, in MC# language, both the channels and the bidirectional channels are the local entities bounded to the place of their declaration. In particular, this means that the programmer is responsible for effective arrangement of communication by the channels.

As an initial stage of our work for the MC# language, we have written in it a series of parallel algorithms such as a computation of Fibonacci numbers, walking through the binary (balanced) tree, computation of primes by Eratosthenes sieve, calculation of Mandelbrot set, modeling the Conway's game "Life", etc. In all these cases, we got the easy readable and compact code for the corresponding problems due to the possibility to write parallel programs in MC# without taking care of their actual distribution over machines during execution. Similarly, there is no need for the manual programming in MC# of object (data) serialization in order to transfer these objects to remote processors (in contrast to MPI, where a special code is needed for a given problem) - the runtime-system of MC# performs an object serialization/deserialization automatically.

The paper is organised as follows. Section 2 gives the detailed explanation of Polyphonic C# asynchronous model and its distributed variant for MC# language. Section 3 gives the examples of using the movable methods and the channels in the typical programs written in MC#. Section 4 describes the MC# implementation, i.e., a compiler and a runtime-system. Finally, in Section 5 we draw conclusions from our work and outline the future plans.

2 Asynchronous Model of Polyphonic C# and Its Distributed Variant

In C#, conventional methods are synchronous: the caller waits until the method called is completed, and then continues its work. In the world of parallel computations, reduction of execution time of a program is achieved by transferring some methods for execution in different processors, after that a program which transferred these methods immediately proceeds to the next instructions.

In Polyphonic C#, methods that commonly are scheduled for execution in the different threads within single computer are called asynchronous and they are declared by using the **async** keyword :

```
async Compute ( int n ) {  
    // method body  
}
```

The specifics of these methods is that their call completes essentially immediately; they never return the result; autonomous asynchronous methods always are scheduled for execution in a different thread (either a new one spawned to

execute this call, or a working thread from some pool). In general case, asynchronous methods are defined using chords. A chord consists of a header and a body, where the header is a set of method declarations separated by the "&" symbol :

```
int Get() & async c (int x ) { return ( x ); }
```

The body of a chord is only executed once all the methods from chord header have been called. The single method calls are queued up until they aren't matched with the header of some chord.

In any chord, at most one method may be synchronous. Just in the thread associated with this method, the body of the chord is executed, and the returned value of it becomes a returned value of synchronous method.

In MC#, autonomous asynchronous methods always are scheduled for execution in a different processor, and they are declared by using the **movable** keyword. The main peculiarity of movable method call for some object consists in that the object itself is only copied (but not moved) to remote processor jointly with the movable method and input data for the latter. As a consequence, all changes of internal variables of the object are performed over variables of the copy and have no influence on the original object.

In MC#, asynchronous methods, which are defined in the chords, are marked by using the **Channel** keyword. And the only synchronous method from the chord plays the role of the method that receives values from the channel:

```
int Get () & Channel c ( int x ) { return ( x ); }
```

By the rules of correct definition, channels may not have a **static** modifier, and so they always are bounded to some object. Thus, we may send a value by `a.c ( 10 )`, where *a* is an object of some class in which the channel *c* is defined. Also, as any object in a program, a channel may be passed as argument to some method. In this case, we must point out the type of the channel as in :

```
movable Compute ( Channel ( int ) c ) {  
    // method body  
}
```

Thus, a **Channel** type plays the role of additional type for the type system of C#.

As in Polyphonic C#, it is also possible to declare a few channels in the single chord with the aim of their synchronization:

```
int Get() & Channel c1(int x) & Channel c2(int y){return ( x + y ); }
```

The calling of *Get* method will return a sum only after receiving both arguments by the channels *c1* and *c2*.

3 Examples of Programming in MC#

Let's consider a simple problem of computing the *n*-th ($n \geq 0$) Fibonacci number. The main computational procedure *Compute* of our program should compute the *n*-th Fibonacci number and return it by the given channel. With the assumption that the above procedure must be executed on a remote processor, we define it as a movable method:

```

class Fib {
    public movable Compute (int n, Channel ( int ) c ) {
        if ( n < 2 ) c ( 1 );
        else {
            new Fib().Compute ( n - 1, c1 );
            new Fib().Compute ( n - 2, c2 );
            c ( Get2 () );
        }
    }
    int Get2() & Channel c1 ( int x )
        & Channel c2 ( int y ) { return ( x + y ); }
}

```

The main program may be the following :

```

class ComputeFib {
    public static void Main(String[] args){
        int n=System.Convert.ToInt32 (args[0]);
        ComputeFib cf = new ComputeFib();
        Fib fib = new Fib();
        fib.Compute ( n, cf.c );
        Console.WriteLine( "n=" + n + "result=" + cf.Get() );
    }
    public int Get() & Channel c ( int x ){
        return ( x );
    }
}

```

The above program has an essential shortcoming - an execution of any call of movable method comprises very few operations. And the effect of parallel execution will be decreased by the overhead charges to transport it in a different processor. A more effective variant for parallel execution is given below:

```

class Fib {
    public movable Compute ( int n, Channel ( int ) c ) {
        if ( n < 20 ) c ( cfib ( n ) );
        else {
            new Fib().Compute ( n - 1, c1 );
            c ( cfib ( n - 2 ) + Get() );
        }
    }
    int Get() & Channel c1 ( int x ) { return ( x ); }
    int cfib ( int n ) {
        if ( n < 2 ) return ( 1 );
        else return ( cfib(n-1) + cfib (n-2) );
    }
}

```


3.1 Bidirectional Channels

If some method got the channel as the argument, then it may send some values by the channel. And then how can we receive messages from this channel - if the corresponding method is "left" in the object where the channel was defined? We may overcome this difficulty as proposed in [3]. A programmer must "wrap up" a chord, in which the channel is defined, by the class with the name *BDChannel* (Bi-Directional Channel) fixed in MC#.

For convenience, public methods for sending and receiving messages for a given channel may be defined in this class. If it is intended to use a few bidirectional channels with the different types, then all of them must be defined in one *BDChannel* class. This is an example of a simple *BDChannel* class:

```
public class BDChannel {
    public BDChannel () {}
    private int Get() & private Channel c ( int x ){
        return ( x );
    }
    public void send (int x) { c ( x ); }
    public int receive () { Get(); }
}
```

Now, having such a class, we can create the corresponding objects and pass them as arguments to other methods, in particular, to movable methods.

Bidirectional channels turn out a convenient feature in the parallel program for constructing primes by the Eratosthenes sieve. Given a natural number *N*, we need to enumerate all primes from 2 to *N*. Main computational procedure *Sieve* have two arguments: input channel *cin* for receiving integers, and output channel *cout* for producing primes extracted from the input stream. The end marker in both streams is -1. A part of the main method for given program is:

```
Main ( String [] args ) {
    int N=System.Convert.ToInt32 (args[0]);
    BDChannel nats = new BDChannel();
    BDChannel primes = new BDChannel();
    Sieve ( nats, primes );
    for (int i=2; i <= N; i++ ) nats.send ( i );
    nats.send ( -1 );
    while ((int p=primes.receive() ) != -1)
        Console.WriteLine ( p );
}
```

The *Sieve* method uses a function *filter* (int *x*, BDChannel *cin*, BDChannel *cout*), that sends integers not divisible by *x*, from *cin* to *cout* :

```

movable Sieve ( BDChannel cin, BDChannel cout ){
    int head = cin.receive();
    if ( head == -1 ) cout.send ( -1 );
    else {
        cout.send ( head );
        BDChannel inter = new BDChannel();
        Sieve ( inter, cout );
        filter ( head, cin, inter );
    }
}

```

It is possible to write a more effective variant for parallel execution, where a function *filter* handles an input stream *cin* not by single prime x , but by each of the primes $x_1, \dots, x_n$ from the package. In this case, bidirectional channels transfer packages of integers, where the package size is regulated by the programmer.

4 Implementation

As usual, for any parallel programming language, an implementation of MC# consists of a compiler and a runtime-system. The main functional parts of the runtime-system are:

1) **Manager** - a process running on the central node and distributing movable methods over the nodes.

2) **WorkNode** - a process running on each working node and controlling execution of movable methods transferred to given node.

3) **Communicator** - a process running on each node and responsible for receiving the channel messages for objects located on given node.

A compiler translates a program from MC# to C#, and its main purpose is to create code realising: 1) execution of movable methods in other processors, 2) transferring the channel messages and 3) synchronization defined in the chords. These functions are provided by the corresponding methods of classes of the runtime-system. Among these classes are:

1) **Session** class - provides for computational session;

2) **TCP** class - provides for sending both queries for movable methods execution and channel messages;

3) **Serialization** class - provides for serialization/ deserialization of objects that are transferred to the remote machines;

4) **Channel** class - contains information about the channel;

5) **LocalHost** class - contains information about the local node.

The main functions of MC#-compiler:

1. Adds the calls to functions *Init()* and *Finalize()* of the class **Session** to the main method of the program. Function *Init()* distributes the executable module to the remote machines, starts a **Manager** process, creates a **LocalNode** object and others. Function *Finalize()* stops the running threads and completes a computational session.

2. Adds an extra parameter *LocalHost* to each constructor of each object; it contains an information needed to create channels defined for a given object.

3. Adds the statements for creation of **Channel** objects for all channels defined in the program.

4. Replaces the calls to movable methods by the queries to **Manager** of computational resources.

5. Replaces the calls to channels by sending corresponding messages by TCP-connection.

Translating of chords, containing channel definitions, is conducted in the same way as in Polyphonic C#.

Passing of bidirectional channels as arguments of movable methods is implemented via creation and passing of proxies for these channels. To send a message from a remote machine, proxy sends this message over the TCP-connection to the node to which the original bidirectional channel is bounded. To receive a message on remote machine, the corresponding query is forwarded to the machine with the original channel, and the thread which issued this command is blocked until a reply message is received. A blocking mechanism is similar to one in Polyphonic C# that is used to handle the thread queues there.

The above implementation is a prototype one, so we use a simple decentralized approach to distribute computational resources amongst movable methods.

5 Conclusion

A distributed variant of an asynchronous parallel programming model of Polyphonic C# was demonstrated in the given work. The key notions of our approach are the movable methods and channels. One-directionality of the channels is overcome by explicit introduction of "bidirectional" channels.

Experiments with the prototype implementation demonstrate the easy readability, compactness and satisfactory effectiveness of program code in MC#.

Further lines of our work are to refine the type system for common and bidirectional channels and to test a decentralized distribution of computational resources in order to increase effectiveness of the whole system.

References

1. N. Benton, L. Cardelli, C. Fournet Modern Concurrency Abstractions for C#, To appear in ACM Transactions on Programming Languages and Systems.
2. C. Fournet, G. Gonthier The reflexive chemical abstract machine and the join-calculus, in : Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages, ACM (2002), 372–385
3. C. Fournet, F. Le Fessant, Jocaml, a Language for Concurrent, Distributed and Mobile Programming, in : Proceedings of the 4th Summer School on Advanced Functional Programming, Oxford (19–24 August 2002)
4. S. Abramov, A. Adamovich, T-system: a programming environment with support of automatic dynamic parallelizing of programs (in Russian), in : Program systems: Theoretical foundations and applications, Ed. A.C.Ailamazyan, Moscow, Nauka (1999), 201–213

A Fast Pipelined Parallel Ray Casting Algorithm Using Advanced Space Leaping Method*

Hyung-Jun Kim, Yong-Je Woo, Yong-Won Kwon,
So-Hyun Ryu, and Chang-Sung Jeong

Department of Electronics Engineering,
Korea University 1-5Ka, Anam-dong, Sungbuk-ku, 136-701, Korea
hjkim@snoopy.korea.ac.kr
csjeong@charlie.korea.ac.kr

Abstract. In this paper we present a very fast pipelined parallel ray casting algorithm for volume rendering. Our algorithm is based on an extended space leaping method which minimizes the traversal of data and image space by using run-length encoding and line drawing algorithms. We propose a more advanced space leaping method which allows the efficient implementation of parallel forward projection by merging the run-lengths for the line drawing. We shall show that the whole algorithm is sharply speed up by reducing the time taken to project the run-lengths onto the image screen, and by exploiting the pipelined parallelism in our space leaping method. Also, we shall show the experimental result of the parallel ray casting algorithm implemented on our Computational Grid portal environment.

Keywords: parallel ray casting, volume rendering, space leaping, Grid, Grid Portal

1 Introduction

A number of scientists and engineers have used volume rendering as a powerful tool to investigate a complex three dimensional structure by extracting 3D shape of objects from volume data. However, due to its high computational cost, it is necessary to develop parallel algorithm for volume rendering. Ray casting algorithm has been known as one of the volume rendering techniques well exploited for parallel processing because of its simple and clear parallel context, and many parallel algorithm have been reported[14,15].

In ray casting, a ray is passed through each pixel of the screen from the view point, and the volume data along the ray are sampled, accumulated to provide the final color and opacity of the corresponding pixel. A variety of acceleration methods has been devised to improve the rendering speed of ray casting

* This work has been supported by KIPA-Information Technology Research Center, University research program by Ministry of Information & Communication, and Brain Korea 21 projects in 2003

algorithm[1,13,3]. However, the previous parallel algorithms have some difficulties and limits in load balancing due to the difference of the computation time taken for each ray traversal. Recently, they[4,5] propose an extended space leaping method which achieves load balancing in each phase of the algorithm while reducing the traversal of data and image space by using run-length encoding and line drawing algorithms. But, their algorithm still have some problem due to the overhead in line drawing algorithm, and the load unbalancing between phases.

In this paper we present a very fast pipelined parallel ray casting algorithm for volume rendering based on the extended space leaping method. We propose a more advanced space leaping method which allows the efficient implementation of parallel forward projection by merging the run-lengths for the line drawing. We shall show that the whole algorithm is sharply speed up by not only reducing the time taken to project the run-lengths onto the image screen, but also by exploiting the pipelined parallelism in our space leaping method. With the advances in high speed network and computing power, Grid, which uses a network of resources as a single unified computing resource, has been used as a large scale high performance parallel computing environments[6]. In this paper, we shall show the experimental result of the parallel ray casting algorithm implemented on our computational Grid portal environment.

The outline of our paper is as follows: In section 2, we examine some existing space leaping methods briefly. In section 3, we describe the basic idea of our advanced space leaping method. In section 4, we describe a pipelined parallel ray casting algorithm using the advanced space leaping method, and in section 5, explain the experimental result of our parallel algorithm. In section 6, we give a conclusion.

2 Previous Work

In this section, we describe previous space leaping methods which accelerates ray casting algorithm by designing various data structures for the efficient traversal of data space or skipping the traversal for the empty data or image spaces.

The well known method for space leaping is to reconstruct original volume data into hierarchical data structure such as octree and pyramid[12,16]. Octree method decomposes the original volume data into eight sub-volumes recursively until all voxels contained in sub-volume satisfy uniform condition. When a ray propagates into volume data, adjusted ray traversal algorithm skips uniform empty space by maneuvering through the hierarchical data structure. When using simple octree, we must perform neighbor search in hierarchical data structures to get the information about empty space whenever ray meets empty data space and this takes a great time consuming over entire octree-based algorithm.

Instead of traversing hierarchical data structure directly, the uniformity information obtained by octree can be stored in additional 3D volume grid. In this type of volume grid, called flat pyramid, each empty voxel is assigned a pointer that indicates information on empty sub-volume to which it belongs. When ray encounters a non-empty voxel, it is handled by usual ray casting algorithm but

when encountering a voxel with a pointer to sub-volume, ray performs a leap forward that will bring it to the first voxel beyond the current empty sub-volume. In flat pyramid, we can drive the information on empty sub-volume directly from the pointer without neighbor search that is the most time consuming operation when using hierarchical data structures.

It is obvious that the empty data space does not have to be resampled to accelerate ray casting. Thus, ray can skip empty data space as fast as possible without doing anything. Vicinity flag[13] method is based on this idea. Vicinity flag algorithm surrounds the non-empty voxels with a one-voxel-deep cloud of adjacent empty voxels. That is, all empty voxels neighboring non-empty voxels are assigned a special “vicinity flags” to represent the boundary of non-empty voxels. Each ray through screen pixel rapidly traverses the empty space until it encounters a voxel with “vicinity flag”. When encountering the first vicinity voxel, we switch to the more accurate traversal algorithm until it encounters the empty voxel that indicates the end of non-empty voxel. Then it rapidly traverses the volume space again until it encounters another vicinity voxel.

When a ray is cast through a pixel in the screen, it may or may not intersect non-empty voxels in data space. If the ray through a pixel intersects any non-empty voxel during the traversal, the pixel contributes to the final image, and is called *active pixel*; otherwise *nonactive pixel*. Nonactive pixels do not contribute to the final image, and we don’t have to cast rays for those nonactive pixels. Extended space leaping method speeds up the ray casting algorithm by casting rays only for the active pixels while skipping empty space for each ray as in the previous space leaping method[4,5]. It also stores, in each pixel, the coordinates of the first and last non-empty voxels encountered by the ray emitted at that pixel. The first and last non-empty voxels are called *nearest and farthest active voxels* and their coordinates *nearest and farthest active depths* respectively. Then, the ray traversal is started for each pixel directly from the nearest active depth and stopped at the farthest active depth instead of traversing the entire propagation path. Therefore, the extended space leaping method reduces the time taken to traverse the empty volume data space as well as the unnecessary image space by calculating active pixels and active depths.

3 Advanced Space Leaping Method

In this section, we describe an advanced space leaping method which shall be exploited in the design of our parallel algorithm. Our method is similar to the extended space leaping method, but differs in that its forward projection technique can be more efficiently implemented in parallel.

The extended space leaping method makes use of the forward projection which maps each voxel in volume data onto the screen in order to find active pixels and active depths. During the traversal of volume data, non-empty voxel is projected onto a pixel in the image screen, and the projected pixel is identified as an active pixel, and the coordinate of the projected voxel is stored to find the nearest and farthest depths of the pixel. Since it may waste a great amount

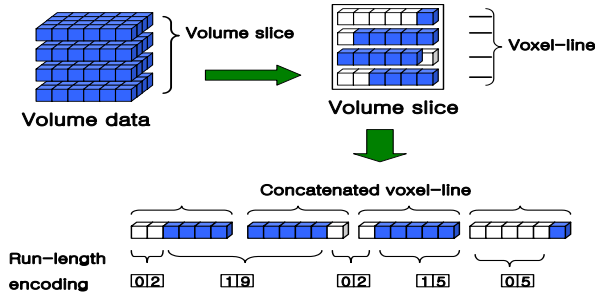


Fig. 1. Run-length encoded volume data: Gray-colored voxels are nonempty ones

of time on traversing empty voxels, run-length encoded volume data and line drawing algorithm is used to further improve the speed of forward projection. By traversing line by line and then slice by slice along volume data, a run-length encoded data is generated which is a series of empty or non-empty voxel runs as in figure 1. By using run-length encoded volume data, we can accelerate the forward projection algorithm by skipping all the empty voxel runs at once. However, it still takes some time to process non-empty voxel run, since we need to traverse each voxel in the non-empty voxel run one by one to project it to the screen. In order to further accelerate the projection process of non-empty voxel run, the line drawing algorithm is used. For each non-empty voxel run during the traversal of the run-length encoding volume data, its first and last voxels are projected onto the screen, and their corresponding two active pixels are found respectively. Then, the active pixels corresponding to the other voxels of the run are calculated by applying line drawing algorithm to those two active pixels as start and ending pixels respectively. Finally, for each active pixel, its depth to the corresponding voxel is obtained by linear interpolation between the depths of the first and last active pixels. Since there generated a lot of voxel runs, a lot of time is spent on distributing them on the nodes for the calculation of active pixels and depths, causing the communication bottleneck.

In our advanced space leaping method, more than one non-empty voxel runs in the same line are combined with the information about internal empty voxel runs, and then projected at once, sharply reducing the time taken to distribute the voxel runs onto the computing nodes. The merged voxel run is called an *active run*. For example, as in figure 2, the voxel runs (1,6) and (1,5) are combined into an active run (1, 13) with the information about empty voxel run (0,2) in between, then projected onto the screen to calculate the active pixels and depths only for the pixels for the non-empty voxel runs, while skipping pixels for the empty voxel run. The number of the merging non-empty voxels is constrained to a fixed threshold in order to manipulate efficient load balancing among computing nodes by preventing the assignment of overlengthed voxel runs.

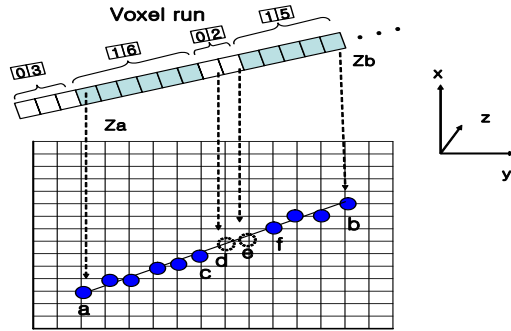


Fig. 2. Line drawing using the active run

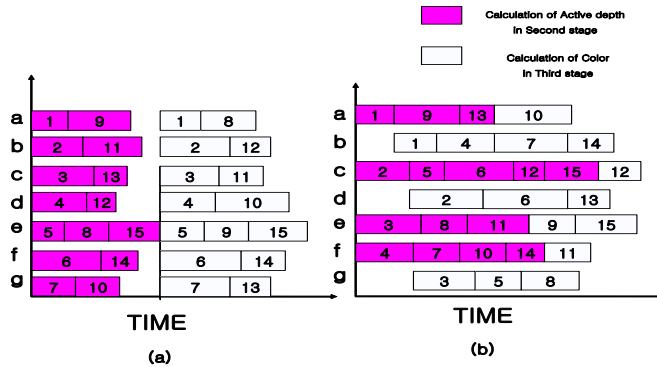


Fig. 3. (a) Execution diagram without a pipeline (b)Execution diagram with a pipeline

4 Pipelined Parallel Ray Casting

In this section, we describe a pipelined parallel ray casting algorithm for volume rendering. Our parallel algorithm is based on master-slave model, and the whole process is configured as a pipeline which consists of three stages as in figure 4. In the first stage, the master process activates the slave processes to calculate the voxel runs, and merge them to produce a series of active runs each of which is projected at once onto the screen by line drawing algorithm in the next stage. In the second stage, on receiving the active runs from the slave processes, the master process distributes them to the available slave processes to calculate active pixels and active depths. Similarly, in the third stage, on receiving the active pixels and active depths from the slave processes, the master process distributes them to the available slave processes, which in turn calculate, for each assigned active pixel, its value by traversing the ray emitted from it. The resulting partial images are returned to the master process to generate a final image. Besides the dynamic distribution of active voxel runs and active pixels in the second and third stages

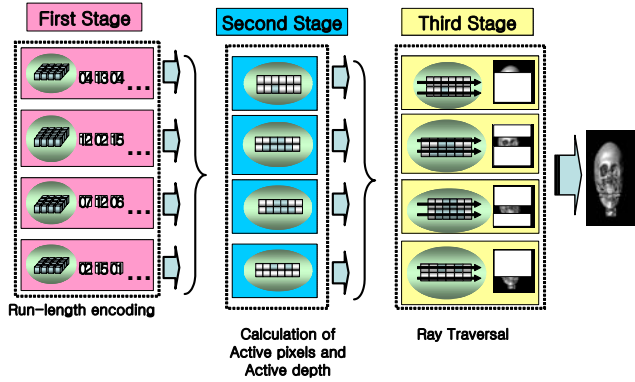


Fig. 4. Three stages of a Pipeline for our parallel ray casting algorithm

respectively, the pipelined scheme which allows immediate processing in the next stage for the data obtained in the previous stage enables drastic speed up of the whole parallel ray casting algorithm. Figure 3 shows speed up by comparing two execution diagrams with and without pipeline respectively.

Our parallel algorithm is implemented on Computational Grid Portal environment developed in Korea university(CGPK). Grid portal has a 3-tier architecture which consists of clients at front end, web application server at the middle, and a network of computing resources at back-end. (See figure 5.) A client at front end delivers HTTP requests from the browser to the web application server at the middle, which in turn executes the processes(master or slave) on remote computing resources using Grid services at back-end, and return the results back to client for display. It provides a easy-to-use user interface for using a parallel programming environment by supporting resource location and authentication, execution, and monitoring and steering. In our parallel algorithm, it provides transparent access to heterogeneous resources by allowing users to allocate the target resources for master and slave processes. The master process plays a key role in the establishment of pipeline as well as the overall control of dynamic data distribution.

5 Experimental Result

Our new parallel ray casting algorithm was implemented on heterogeneous resources of computational Grid by using CGPK. Grid resources consist of 2 Ultrasparc1, one SGI O2, a SGI Octane, 16 Pentium IV PCs running Linux connected by 100 Mbps Ethernet. We experimented with 256 x 256 x 225 human head volume data as data set for 1024 x 1024 screen pixels. The result time for ray casting is measured as average total time for volume images generated by incrementally

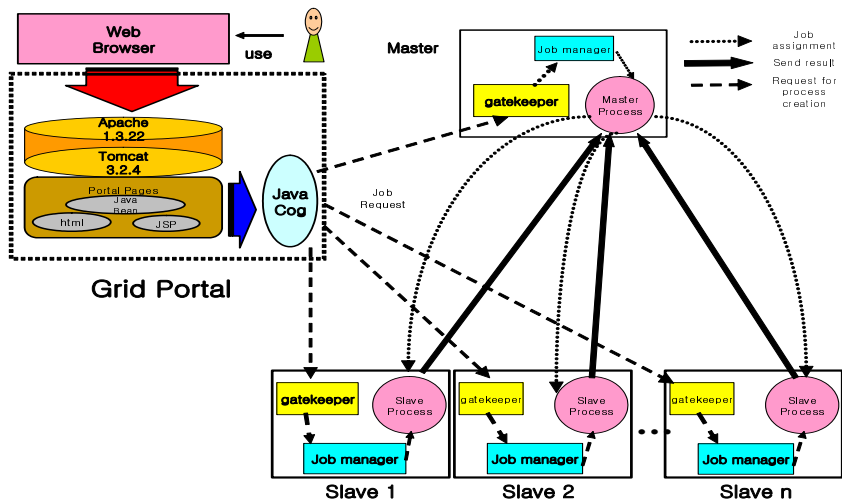


Fig. 5. The Operation of processes through Grid Portal

Table 1. Machine specifications

Machine type	M_1	M_2	M_3	M_4
Model	Pentium IV PC	USparc1	O_2	Octane
CPU	P IV	UltraSPARC	MIPS R10000	
Clock(MHz)	1740	143	150	250
Memory(MBytes)	1024	128	128	512
OS	Linux 2.2	Solaris 2.5	IRIX 6.3	IRIX 6.5

rotating a data by 5 degrees. The details of hardware and software information for each machine are shown in table 1.

For the performance evaluation, we need a reference machine, since a variety of computers with different computing power are used. First, we measured the relative performance with respect to the reference machine, M_1 , by performing identical ray casting algorithm on each machine, and then comparing their execution time. Table 2 shows the relative performance of each machine, and table 3 shows the execution time, speed up and efficiency of ray casting according to the number of machines. The efficiency represents the ratio of speedup with respect to expected speedup. Our algorithm shows good efficiency of more than

Table 2. Measurement of relative performance with respect to M_1 for ray casting

machine i	M_1	M_2	M_3	M_4
OS	Linux	Solaris2.5	IRIX6.3	IRIX6.5
(spec.)	(PIV-1.7G)	(USparc1)	(O_2)	(Octane)
running time	103.01	413.812	205.390	128.690
relative perf.	1.0	0.249	0.502	0.800

Table 3. Performance results of parallel ray casting on GRID

number of machines	$1(M_1)$	$2(M_{1,4})$	$4(M_{1,2,3,4})$	$8(M_{1,\dots,1,2,\dots,4})$	$11(M_{1,\dots,1,2,\dots,4})$	$20(M_{1,\dots,1,2,\dots,4})$
expected speedup	1.0	1.8	2.551	6.351	9.551	17.8
time (sec)	101.05	67.59	47.891	19.763	13.457	7.290
GRID speedup	1.0	1.495	2.110	5.113	7.509	13.86
efficiency (%)	100.0	83.05	82.73	80.51	78.62	77.87

77.87%, and the parallel algorithm achieves relatively good speed up as the number of machines increases due to the pipelined methods which exploits the load balancing dynamically.

6 Conclusion

In this paper we have proposed a new advanced space leaping technique, and based on them, presented a very fast parallel ray casting algorithm by exploiting additional pipelined parallelism. With the advanced space leaping method, the number of active voxel runs participated in the calculation of active pixels is sharply decreased, thus reducing the time taken for the calculation of active pixels and hence decreasing the communication overhead. We have presented a pipelined parallel technique which consists of three stages, and have shown that the whole algorithm is sharply speed up by exploiting dynamic load balancing between the stages of the pipeline as well as in each stage. Our parallel algorithm have been implemented on computational Grid by using Grid portal, called CGPK, and it is shown that our parallel algorithm achieves a very good speed up by incorporating an advanced space leaping method into parallel pipeline technique.

References

1. J. Danskin, P. Hanrahan: Fast algorithms for volume ray tracing, 1992 workshop on Volume Visualization, Boston, MA (1992), 91–98
2. R. Yagel, D. Cohen, A. Kaufman, Q. Zhang,: Volumetric Ray Tracing, TR 91. 01. 09, Computer Science, SUNY at Stony Brook (January 1991)
3. R. Yagel, Z. Shi: Accelerating Volume Animation by Space-Leaping, Visualization '93, (1993), 63–69
4. Sung-Up Jo, Cahng-Sung Jeong: A Parallel Volume Visualization Using Extended Space Leaping Method, LNCS Vol. 1947, pp 0296, (2001)
5. Hyungjun Kim, Sung-Up Jo, et all: Fast Parallel Algorithm for Volume Rendering and its Experiment on Computational Grid, ICCS 2003, (June 2003)
6. I. Foster, C. Kesselman, S. Tuecke: The Anatomy of the Grid: Enabling Scalable Virtual Organizations, International J. Supercomputer Applications, 15(3), (2001)
7. J. Novotny: The Grid Portal Development Kit, Concurrency: Pract. Exper. Vol. 00, (2000), 1–7
8. G. von Laszewski: A Java Commodity grid kit, Concurrency: Pract. Exper. Vol. 13 (2001), 645–662

9. Myproxy, <http://dast.nlanr.net/Projects/MyProxy>
10. P. Steven, P. Michael, L. Yarden, S. Peter-Pike, H. Charles: Interactive Ray Tracing for Volume Visualization, *IEEE Trans. on Visualization and Computer Graphics*, Vol. 5, No. 3, (1999) 238–250
11. MPICH-G2, "http://www.hpclab.niu.edu/mpi/g2_body.html"
12. J. Danskin, R. Bender, and G. T. Herman: Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography, *J. Theoretical Biology*, Vol. 29 (1970), 471–482
13. R. Yagel, D. Cohen, A. Kaufman, and Q. Zhang: Volumetric Ray Tracing, TR 91.01.09, Computer Science, SUNY at Stony Brook, (January 1991)
14. V. Goel and A. Mukherjee: An Optimal Parallel Algorithm for Volume Ray Casting, *Visual Comput*, Vol. 12 (1996), 26–39
15. C. Kose and A. Chalmers: Profiling for efficient parallel volume visualization, *Parallel Computing* Vol. 23, (1997), 943–952
16. M. Levoy: A hybrid ray tracer for rendering polygon and volume data, *IEEE Computer Graphics & Application* Vol.10, No.2 (1990), 33–40.

Formal Modeling for a Real-Time Scheduler and Schedulability Analysis

Sung-Jae Kim and Jin-Young Choi

Department of Computer Science and Engineering,
Korea University
SEOUL, 136-701 KOREA
{sjkim, choi}@formal.korea.ac.kr

Abstract. The reliability of safety-critical embedded real-time system depends partly on that of the system design. Because of this, formal methods have been adopted in the design phase of developing such systems, and various kinds of formal methods have been introduced and used in practice. Many successful results have been published in application systems/software. However, studies on formal specification for embedded kernel, like scheduler, are relatively few due to the complexity of the software. In this paper, we present a formal specification for real-time scheduler based on SyncCharts. We specify a scheduler of which policies are rate monotonic, as well as Priority Ceiling Protocol, and perform schedulability analysis by formal verification. Once requirements of the real-time scheduler and timing properties of given tasks are satisfied, a real code can be automatically generated and, we believe, ported in a real target platform.

1 Introduction

The reliability of safety-critical embedded real-time system depends partly on that of the system design. To improve the reliability of real-time system, therefore, there must be correct specification and verification of the systems in the design phase. In this paper we use formal methods to design and analysis a real-time system. As formal methods are based on mathematics and logics, we can describe a system without ambiguity and prove requirements of the system. Hence, we can reduce the potential disaster and loss of time associated with an incorrect operation of real-time systems by checking the correctness of the systems before the implementation using formal methods. By the reason of above, formal methods have been adopted in the design phase of developing such systems, and various kinds of formal methods have been introduced and used in practice. Many successful results have been published in application systems/software. However, studies on formal specification for embedded kernel, like scheduler, are relatively few due to the complexity of the software.

In this paper, we will present a formal specification and verification for a real-time scheduler and PCP as a part of the research for implementing embedded kernel using formal methods. So, in order to automatically generate embedded

code from formal specification, we will use a reactive system modeling language, SyncCharts. As Bate and Burns [4] stated, in practical systems it is frequently necessary to offset the execution of tasks from one another. Therefore to implement more practical system, we will perform timing analysis for a task set featuring offsets. Once requirements of the real-time scheduler and timing properties of given tasks are satisfied, a ANSI-C code of kernel can be automatically generated and, we believe, ported in a real target platform.

The remainder of this paper is organized as follows. Section 2 introduces SyncCharts, the graphical notation of reactive behavior. Section 3 discusses how to specify tasks and scheduling algorithms. Section 4 describes the schedulability analysis and deadlock analysis of specified model using formal verification. We sum up in section 5.

2 SyncCharts

The practical model we use is SyncCharts [1]. SyncCharts is a graphical notation of reactive behavior on synchronous hypothesis, then the evolution of the system are represented with states and transitions between these states. It offers broadcasting of signals, hierarchy, orthogonality and enhanced preemption capabilities. Syntactically, this model is close to Statecharts and Argos. The SyncCharts semantics is fully synchronous and perfectly fits Esterel [5]'s semantics, and any SyncCharts can be translated into an equivalent Esterel program. For more details on SyncCharts, readers refer [2,3].

3 Formal Specification Using SyncCharts

3.1 Task and Scheduler

We specify real-time system, which is consisted of several tasks that are scheduled by the Rate Monotonic [8]. Each task has its own computation time, period, deadline and offset. Figure 1 describes a scheduler. Under the rules of Rate Monotonic, the scheduler controls tasks with appropriate signals, referencing the state and priority of current task.

3.2 Solution for the Priority Inversion Problem

Figure 2 shows our approach for the solution of the priority inversion problem. This specification illustrates the use of hierarchy. In order to minimize unpredictable blocking time caused by the priority inversion, a blocked task encloses some commands that can control the task which blocks itself. With this technique, a blocks tasks will not be preempted by any other task except a task that has higher priority than blocked task, therefore we can solve the priority inversion problem.

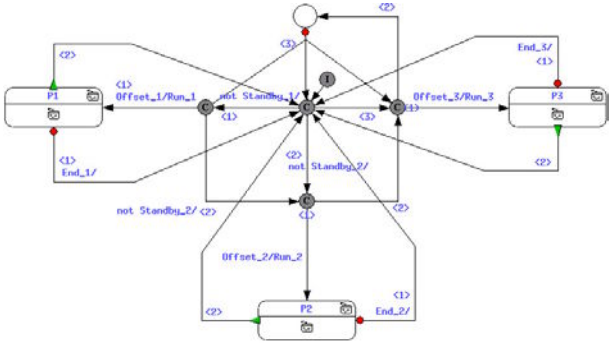


Fig. 1. Rate Monotonic Scheduler

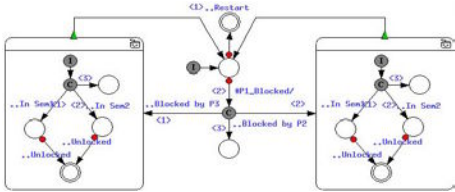


Fig. 2. Solution for the Priority Inversion problem

3.3 PCP (Priority Ceiling Protocol)

Figure 3 presents PCP [10] algorithm and the TIME macro state of the task that uses multiple shared resources. As Figure 3 shows, in order to prevent a deadlock, a task transits through PCP macro state whenever the task accesses shared resource. If any task tries to access a shared resource, PCP becomes aware of potential deadlock, and performs ceiling blocking that blocks itself. This ceiling blocking continues until there is no feasible deadlock.

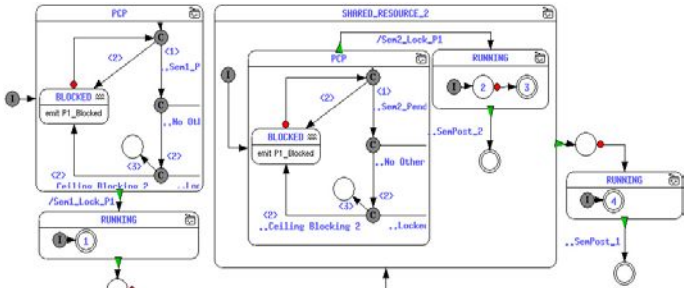


Fig. 3. TIME Macro state using PCP algorithm

4 Analysis of Task Sets Using Formal Verification

4.1 Formal Verification

In this paper, we analyze the schedulability and deadlock of task sets using Esterel Studio. The verification process of Esterel Studio is as follows. First, Esterel Studio compiler translates specified model into a system of a boolean equations with latch that defines a FSM(Finite State Machine) implicitly. Esterel Studio verifier works on these implicitly defined FSMs. Next, the Verifier minimize the FSMs using bi-simulation equivalence and BDD(Binary Decision Diagram). Lastly, check the status of output signals from all reachable states of the FSMs. That is to say, with the emitting of specific signals, we can verify the feasibility of requirements.

4.2 Analysis of Task Set Which Features Shared Resource

In this section, we perform schedulability and deadlock analysis of task set which uses multiple shared resources. We assume that a task set is as Table 1 and scheduled by the Rate Monotonic algorithm. Tasks in Table 1 use shared resources as time units as declared in the parenthesis. For example, task 1 will use the 1st and 4th time unit for the shared resource 1, and the 2nd and 3rd unit for the shared resource 2.

Table 1. Example of Task Set

	Task 1	Task 2	Task 3
Priority	1	2	3
Period	14	18	36
Computation Time	4	6	8
Shared Resource 1	(1,4)		(3,4,5,8)
Shared Resource 2	(2,3)		(6,7)
Offset	Start of Period + 3	Start of Period + 4	Start of Period + 8
Deadline	End of Period - 3	End of Period - 3	End of Period - 6

First, we perform a deadlock analysis for the above task set. As deadlock is an error condition such that processing cannot continue because each of two tasks is trying to access the shared resource occupied by the other task, we can conclude the occurrence of deadlock as the situation which there exists one time units when each of two task is blocked at the same time. Be based on Table 1, only task 1 and task 3 use shared resources, then design an observer which can observe a situation that task 1 and task 3 are blocked at the same time as Figure 4. If there exists a state that task 1 and task 3 are blocked at the same time (emitting P1.Blocked signal and P3.Blocked signal simultaneously), Deadlock signal is emitted. Therefore, with the presence of Deadlock signal, we can judge whether deadlock will happen or not.

Figure 5 shows the result of deadlock analysis with Esterel Studio verifier. In

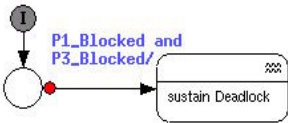


Fig. 4. Deadlock observer

any reachable state of specified model with above task set, Deadlock signal is never emitted.

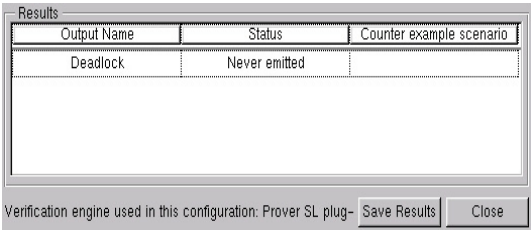


Fig. 5. Deadlock Analysis of the task Set with Verifier

Next, we perform schedulability analysis for the task set of Table 1. If 3 tasks in Table 1 miss their own deadlines, they broadcast Failed_1, Failed_2 and Failed_3 signals respectively. But Figure 6 represents that those Failed signals are never emitted in any reachable state of specified model. Through this result, we can regard the task set in Table 1 as a schedulable task set.

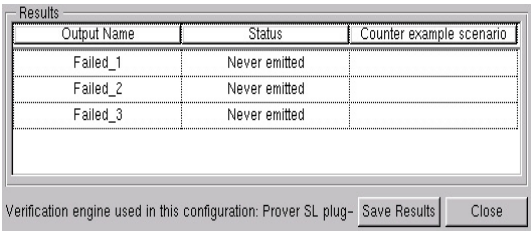


Fig. 6. Schedulability Analysis of the task Set with Verifier

5 Conclusion

To improve the reliability of real-time system, there must be correct specification and correct verification of the systems in the design phase. As the correctness of real-time system can be guaranteed by precisely modeling with formal methods, formal methods have been adopted in the design phase of developing such systems. But, for embedded kernel, there are relatively few studies on formal specification. In this paper, we have proposed an approach for the specification of embedded kernel, like scheduler, and performed timing analysis for the specified model. As the requirements of the real-time scheduler and timing properties of given tasks are satisfied, we can find a possibility for implementing embedded real-time system with ANSI-C code automatically generated from the specified model. An interesting and important direction for future work is to provide a way for the implementation of real time kernel with the generated code and porting it kernel to a real target platform.

References

1. Charles Andre: SyncCharts: A Visual Representation of Reactive Behavior, Technical report RR-95-52, 13S, (1995)
2. Charles Andre: Representation and Analysis of Reactive Behaviors: A Synchronous Approach, CESA'96, (1996)
3. Charles Andre, Marie-Agnes Peraldi-Frati: Behavioral Specification of a Circuit using SyncCharts: a Case Study, In Proc. of the 26th EUROMICRO Conference, (2000)
4. Iain Bate, Alan Burns: Schedulability Analysis of Fixed Priority Real-Time Systems with Offsets, 9th Euromicro Workshop on Real Time Systems, (1997)
5. Gerard Berry, Georges Gonthier: The Esterel Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming, (1992)
6. Amar Bouali: XEVE : an ESTEREL verification environment, Technical Report RT-0214, Inria, (1997)
7. Jin-Young Choi, Insup Lee, Hong-Liang Xie: The Specification and Schedulability Analysis of Real-Time Systems Using ACSR, In Proc. of Real-Time System Symposium, (1995)
8. John Lehoczky, Lui Sha, Ye Ding: The Rate Monotonic Scheduling Algorithm, In Proc. of Real-Time Systems Symposium, (1989)
9. Sung-Mook Lim, Jin-Young Choi: Specification and verification of Real-Time Systems Using ACSR-VP, 4th International Workshop on Real-Time Computing Systems and Applications, (1997)
10. Lui Sha, Rangunathan Rajkumar, John P. Lehoczky: Priority Inheritance Protocol: An Approach to Real-Time Synchronizaion, IEEE Transactions on Computers, (1990)

Disk I/O Performance Forecast Using Basic Prediction Techniques for Grid Computing

DongWoo Lee and R.S. Ramakrishna

Department of Information and Communication,
Kwangju Institute of Science and Technology,
1 Oryong-dong, Buk-gu, Gwangju 500-712 Republic of Korea,
{leeepro,rsr}@kjist.ac.kr

Abstract. From investigations on the impact of Disk I/O load on CPU load, we have found that the immanent Disk I/O load could affect the resource scheduler's decision on assigning an appropriate storage resource to a job in which the Disk I/O operation is dominant. A possible but improper assignment can prolong the execution time of a task due to the contention for Disk I/O when the Disk I/O load in the machine is higher than the CPU load. Because the scheduler uses CPU load only for computing schedules, it does not even know the potential Disk I/O contention that could occur at the assigned resource. To avoid or at least alleviate these effects, we have developed a performance monitoring system and on-line performance forecast functions for providing forecast information to the Grid. In this paper, we examine the impact of Disk I/O workload on the CPU workload using our system, hereinafter referred to as *Storage Weather Service*(SWS). We evaluate several prediction methods in order to get an insight on varying Disk I/O workload.

1 Introduction

Scientific applications need high computational power and intensive analysis of huge data sets spread over large scale computational Grids. This recent trend has encouraged research and development of sophisticated infrastructure for managing large data collections in a distributed fashion. Rapid access of large subsets of data files is a major goal of these efforts. To maximize the efficiency of using Grid resources, resource discovery[17,11,2,9] and resource scheduling[10,18,2,9] have been employed. As large scale grid computing environments begin to be deployed and used, the number of computing resources is expected to grow very fast. Resource discovery and scheduling are, therefore, more important than ever before. Because the status of resources involved in the Grid changes in time, keeping track of it and exploiting it for decision making are very critical for extracting maximum performance out of the Grid. Predicting varying performance and status of a resource is imperative for effective resource discovery/scheduling[16]. In case of a Data Grid(e.g. GriPhyN, DataGrid for HEP and so forth), huge data sets have to be processed by various applications. Data replication and migration are absolutely essential for acknowledging application's purpose and user's needs. Several optimization mechanisms on the storage system have been developed in this regard.

Many researchers have directed their efforts on Disk I/O optimization. In the case of tightly-coupled parallel computing systems(e.g. Intel Paragon, IBM SP2 or Beowulf Clusters having fast network facility such as Myrinet), they have tried to classify the application's I/O access pattern with a view to get its precise profile. Several I/O optimization strategies including collective I/O, prefetching/caching, file layout optimization, efficient I/O interface have been considered. In a loosely-coupled distributed computing environment(i.e. the Grid), different approaches for optimizing I/O operations are available. In the Grid, local machines can exploit the above optimization techniques. But, the Grid is a shared environment consisting of various computing resources spread over a large area. Many computing resources are used by many users at the same time. The resources in the Grid are managed by Grid resource schedulers for optimum efficiency. A scheduler uses dynamic performance data about its resources. Based on the information and with the help of a performance model, it schedules resources for maximum utility. In other words, it minimizes user program execution time while scheduling resources efficiently. For example, to accelerate the performance of a large scale parameter sweep application, priority-based searching algorithm has been used [8]. In this case, a policy to select a resource that minimizes the running time has been proposed. The scheduling policy is to assign a powerful resource at a promising search point of the entire parameter search space. The scheduler uses only the local resource's CPU workload and network traffic information from a network monitor[3]. In this case, if the job to be run at a local host is data-intensive, then we have to consider the Disk I/O workload because an (almost) idle status of the CPU does not necessarily mean an absolutely low Disk I/O workload.

If a process has to compete with other processes for secondary storage resources, its efficiency may suffer due to the increase in time spent by the process waiting for its I/O requests to complete. As a result, there may be significant perturbations in the expected I/O service times and consequently in the job execution durations[4]. We demonstrate this effect, named "*the eye of typhoon*" effect, with simple stress tests in the next section.

The rest of this paper is organized as follows: We suggest the effectiveness of I/O performance forecast as a new performance metric for determining an appropriate resource in Section 2. In Section 3, we present several prediction methods. In Section 4, we show experimental results using real Disk I/O workload(trace) to find out which prediction method is effective for Disk I/O workload. In Section 5, we mention related work. The paper ends with conclusions in Section 6.

2 Motivation: *The Eye of Typhoon Effect*

CPU workload, memory availability and end-to-end network performance have been used as resource scheduling metrics. Most resource schedulers currently in use employ them for determining appropriate schedules in which available resources are assigned to a job. From stress tests described below, we found out a situation that could occur in a shared computing environment. When the CPU workload status is "*almost idle*", it does not always mean low or idle Disk I/O

workload. This is very similar to *the eye of a typhoon* in that the eye of a typhoon is very calm, but its surrounds are very rough. We performed some simple I/O stress tests as a starting point to understand this "eye of the typhoon" effect.

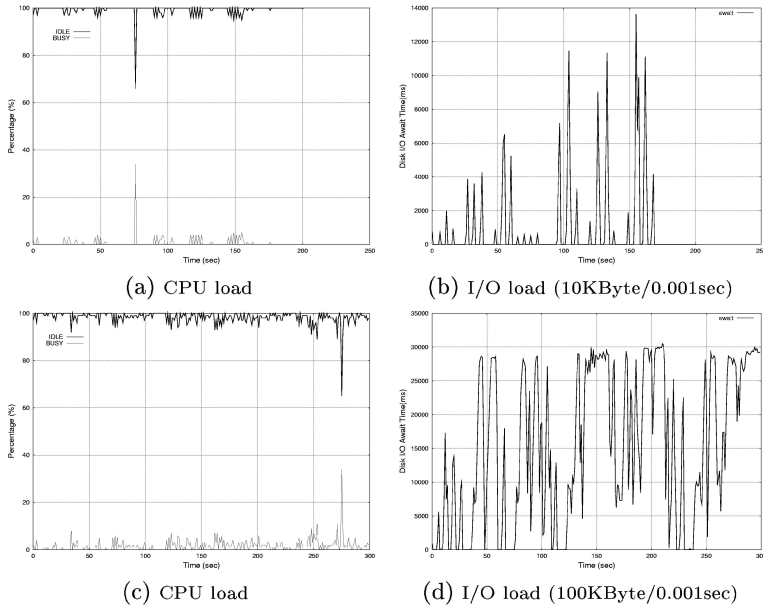


Fig. 1. Results of Disk I/O Stress Test: (a) CPU load, (b) I/O load of 10KByte/0.001sec and (c) CPU load, (d) I/O load 100KByte/0.001sec TBO Disk I/O stress (The CPU load is in percentage(%) and The I/O workload is await time of I/O completion in *ms*)

2.1 Stress Test

To investigate the effect of I/O workload on CPU workload, we performed a series of tests. For this stress test, we used Pentium III 933MHz RedHat 7.2 Linux box and 10GB IDE Hard disk drive. Experimental setting for stress test was as follows:

- Normal CPU workload (there are no active processes in the host except the stressing applets)
- Increasing I/O workload by overlapping I/O intensive applets (from 1 to 6) with 10 sec launching interval.
- Data Size to be written: 1KByte¹, 10KByte, 100KByte/0.001 sec TBO (Time between I/O Operation)
- Monitoring measurements: CPU workload² and I/O workload³

¹ The load graphs of this size is not shown in this paper due to the lack of space. It also shows the similar effect.

² This workload was traced using our SWS in which the workload consists of IDLE, USER, SYS and WAIT. These measurements are taken by Linux *iostat* utility.

³ This workload was traced using our SWS in which the workload consists of I/O completion await time, read/write bytes per second, and so forth.

2.2 Analysis of Stress Test

We looked at the CPU workload(idle %) performance changes with the I/O workload changes over time. Fig.1 shows the results of a stress test. Fig.1 (a)/(b) and (c)/(d) show the CPU load and the I/O load plotted against stress. We confirmed "the eye of a typhoon" effect through the test: On performing I/O operations through concurrent I/O intensive applets(writing data) to disk in a shared user environment, the fluctuations of CPU idle % did not change appreciably(Fig.1 (a),(c)). We can figure out the growing number of applets in Fig.1 (b) through the peaks. As the number of applets competing to complete their I/O operations increase, the CPU idle percentage decreases slightly. This is a natural phenomenon resulting from multiple applications. As a result, I/O operations do not affect the CPU workload so much when the applications do not have computation intensive codes. If a resource scheduler uses the CPU workload alone for assigning a job to the resource, it can lead to competitive I/O operations. An inevitable consequence will be increased job execution times. Even though the resource has low CPU workload (the eye of a typhoon), the job's execution time is prolonged due to the background applications performing I/O operations (outside the eye of the typhoon) unknown to the scheduler. Earlier resource schedulers such as AppLeS[11], NetSolve[1], Ninf[6], FAST[12] used CPU workload and network performance such as bandwidth and latency between computational resources to assign a job to an appropriate resource. The use of I/O workload information can be quite effective in scheduling data-intensive applications. The usefulness of performance forecasting to resource schedulers in grid computing is a well recognized issue[3,13,9,10].

From Fig.1, we can figure out the effect of Disk I/O workload that lurks behind CPU workload. Thus, a resource scheduler can create a situation that results in I/O contentions when only the CPU workload is used by the scheduler. The schedule that does not consider Disk I/O can lead to undesirable results.

3 Disk I/O Performance Forecast

3.1 Prediction Methods

With a view to utilize the (more meaningful) Disk I/O workload as a new performance metric, we consider performance forecast in our SWS. Because SWS works as an on-line storage performance forecaster, we consider simple and low-overhead prediction methods only at this time. It executes a set of forecasting methods that it can invoke dynamically. To find out appropriate prediction methods for Disk I/O workload, we evaluate several simple prediction methods. Most of the methods are very commonly used by network performance forecasters, such as NWS(*Network Weather Service*)[3]. We selected some prediction methods only from NWS. Because NWS is focused on end-to-end network traffic, we omitted several prediction methods used in it for our Disk I/O workload. The omitted methods, (e.g. stochastic gradient error predictor) perform well for network traffic, but they are not suitable for Disk I/O workload due to the behavior of I/O workload and, in addition it needs high overhead to compute. In

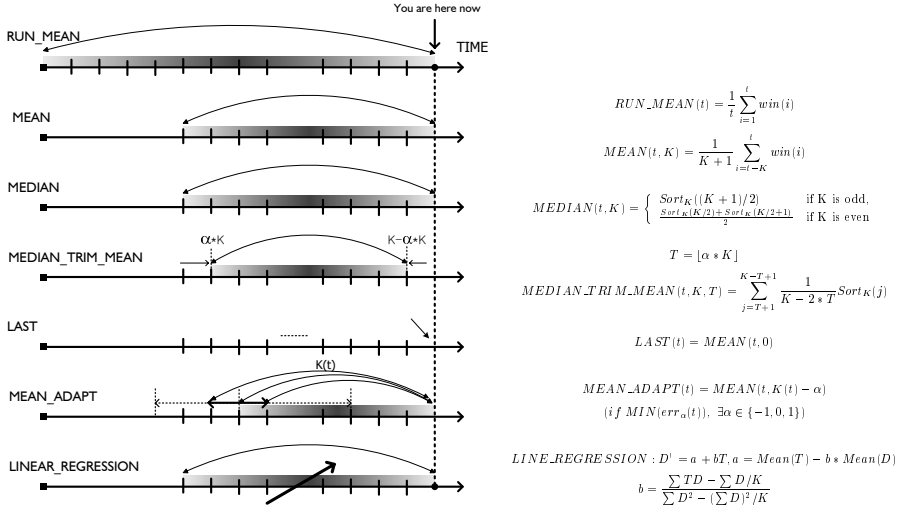


Fig. 2. Prediction Methods evaluated in SWS: K is the length of a sliding window. $win(i)$ is a measurement of the sliding window with index i .

case of network traffic, the performance measure is always positive. But, the measure of Disk I/O workload is non-negative. Our study shows that the Disk I/O workload has short-term sporadic spurts between zero workloads. We intend to evaluate more complex prediction methods including autoregressive methods in the future.

Fig.2 shows the prediction methods used with Disk I/O workload. In this section, we describe the methods for predicting near-future performance. After a new measurement is taken, it is passed to all the methods, and a new forecast is generated.

We evaluated RUN_MEAN, MEAN, MEDIAN, MEDIAN_TRIM_MEAN, LAST, MEAN_ADAPT, LINEAR_REGRESSION and RANDOM. RANDOM just picks a measurement within the sliding window randomly. Except RUN_MEAN, the other methods work within a limited width of the sliding window and maintains a short history of SWS. The notation for each method is in conformity with NWS[3]. The values provided by the monitoring sensor are treated as a time series by the forecasting methods, and each method maintains a history of previous activity and accuracy information.

RUN_MEAN uses arithmetic averaging as an estimate of the mean value over some portion of the measurement history to predict the value of the next measurement. If the most recent values better predict the next measurement, then an average taken over a fixed-length history will be a better predictor. For this, MEAN method is a sliding window version of RUN_MEAN. LAST takes the last measurement. MEDIAN can also provide a useful predictor, particularly if the measurement sequence contains randomly-occurring, asymmetric outliers. MEDIAN_TRIM_MEAN

works well in case of impulses. To deal with jitter, it uses an α -trimmed mean filter. MEAN_ADPT uses time-varying window size that adapts to unpredictable errors. LINEAR_REGRESSION uses the two variables(time and Disk I/O workload).

3.2 Dynamic Predictor Selection

An appropriate predictor for Disk I/O cannot be specified in advance. As time goes by, an unexpected access from a user application may changes the shape of the I/O workload. Rather than using a static prediction method, selective usage of prediction methods having minimum MSE(Mean Square Error) is preferable[3]. The method exhibiting the best overall predictive performance at any time t is used to generate the forecast of the measurement at time $t+1$. Whenever a new measurement is taken by the monitor(disk sensor), each prediction method is evaluated. That is, at time t , the output of the method yielding the minimum MSE is used as a forecast of the next measurement.

3.3 Adapting Grid Middleware

We define the LDAP schema for Globus MDS[2]. SWS pushes forecast information to LDAP server. Any Grid middleware can use SWS's information through the MDS directory service. Also, SWS server provides SOAP(Simple Object Access Protocol) interface that is used in OGSA[14], which was proposed as a next generation Grid architecture using web technologies including SOAP, WDSL(Web Service Specification Language) and UDDI(Universal Discovery and Directory Interface).

4 Experiment Results

With a precaptured real trace⁴ of Disk I/O workload, we evaluated the above prediction methods. The trace was gathered from an UltraSun Workstation hosting a network file server. The tracing duration is about 12 hours, its size is approximately 120 MBytes. To get a relationship among different sliding window sizes, sampling rates of the workload and prediction methods, we vary the experimental parameters. We made individual trace files with different sampling rates(1s,5s,10s,20s,30s,40s,50s,60s,70s,80s,90s,100s,110s,120s). Also, we used different sizes of sliding window(ranging from 10 to 100). There are 980 results (7 methods * 14 sample rates * 10 window size). In Fig.3 through Fig.6, the prediction methods are numbered from 0 to 6(0: RUN_MEAN, 1: MEAN, 2: MEDIAN, 3: MEDIAN_TRIM_MEAN, 4: LAST, 5: MEAN_ADAPT, 6: LINEAR_REGRESSION). We do not include RANDOM method because of its poor predictive power. This is related to the inherent behavior of Disk I/O workload. We figure out that some smoothed measurement is always preferable.

In this section, we present experimental results on the prediction method selection frequency with different window sizes and sampling rates⁵. The selection

⁴ You can get the trace at http://sws.kjist.ac.kr/trace/exp_trace120.data

⁵ In this paper, the sampling rate means the sampling interval.

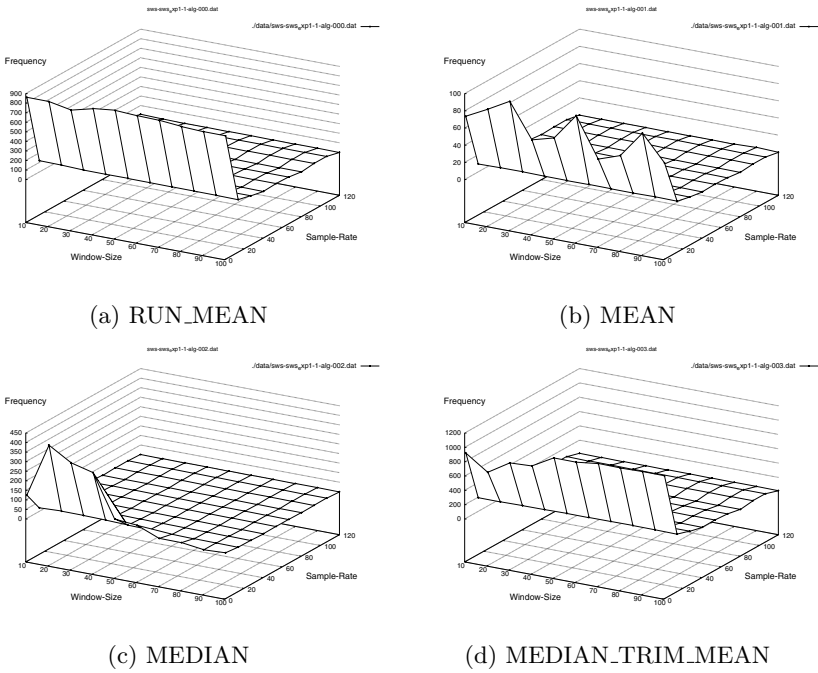


Fig. 3. Selection Frequency of Each Prediction Methods

frequency is incremented by 1 when a prediction method returns a least MSE. Fig.3 and Fig.4 plot the selection rate against different window sizes and sampling rates for each prediction method. Most of the methods work well within a sample rate of 20s. Because of the sporadic nature of the Disk I/O, a sample rate below 10 seconds is valid for predictions. From Fig.4 (a) and (c) it is seen that **LAST** and **LINEAR_REGRESSION** return a higher frequency than others. Also, **LAST** outperforms **LINEAR_REGRESSION** if a large window size(e.g. 100) is used. Further, **LINEAR_REGRESSION** outperforms **LAST** with a small window(e.g. 10). With a large window, **LINEAR_REGRESSION** finds it difficult to fit a line to measurements because the variability of the Disk I/O is too high(high fluctuation). So, with a smaller window, the regression is valid; that is, it provides a short term trend(increasing and decreasing). Fig.5 graphs frequency versus window size. As already mentioned, we can figure out the relationship between **LAST** and **LINEAR_REGRESSION**. Fig.6 presents experimental results with regard to selection frequency under varying with window size and sample rate. With smaller sample rates the methods return smaller MSEs. Within the 20s sample rate, we can get a valid short term Disk I/O workload trend. Details can be found in [19].

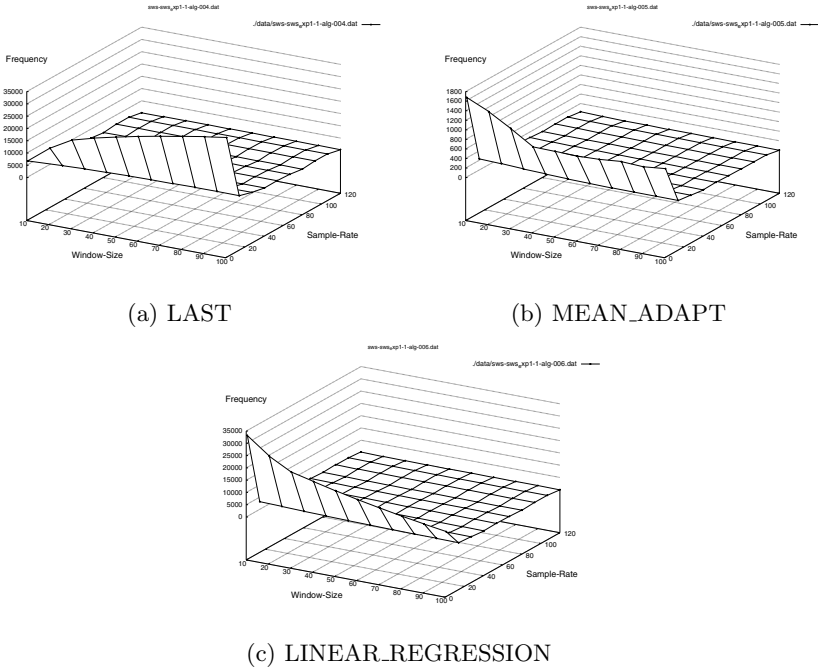


Fig. 4. Selection Frequency of Each Prediction Methods

5 Related Work

A distributed multi-storage resource architecture is well known[7]. This distributed multiple storage system has been used in Astro3D application. Using application's I/O access operation model, it builds a performance predictor database and predicts the remote storage access time with the database. The prediction is quite close to the actual I/O time. But, the data intensive application's access pattern including data size, data access frequency and so forth must be prepared for a correct prediction. Our system is independent of any specific application. It is a fully automated storage forecasting system. Host load prediction study[5] focuses on on-line host load(CPU workload) prediction using ARIMA models. This study is closely related to ours. The behavioral feature is somewhat different. We plan to evaluate these models for Disk I/O workload in the future. NWS[3] was the starting point of our system. It proposes several prediction methods for network traffic. As we mentioned earlier, the type of workload is different. We have worked with several methods of NWS with different parameters. Disk throughput data has been used to predict data transportation time[15]. It helps to reduce MSE. The work used coarse sampling rate(about 2 minutes) for measuring disk throughput data. Also, it does not employ different sampling rates and window sizes. From our study, it is possible to get more precise insight into Disk I/O workload. The sampling rate should be within the

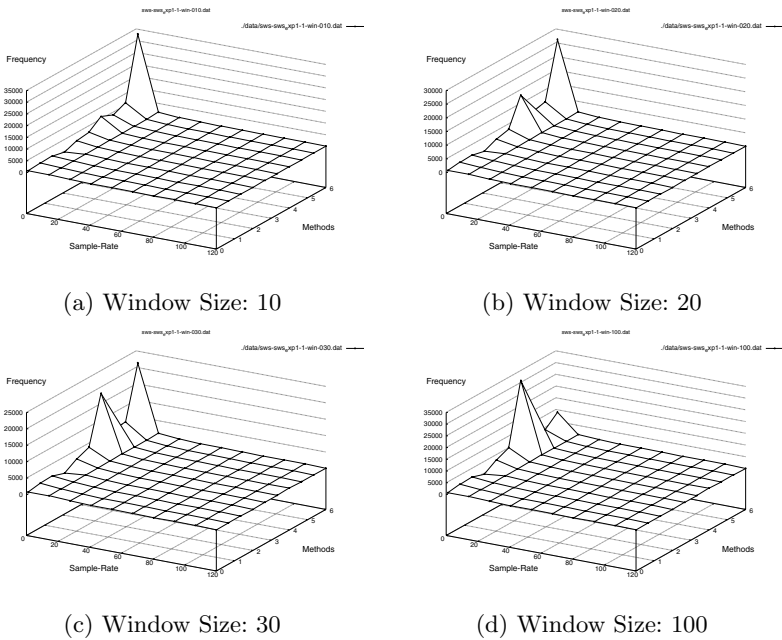


Fig. 5. Method Selection Frequency with various Window Size

sample rate of 20 seconds from disk software sensor due to the bursty behavior of Disk I/O workload.

6 Conclusion and Future Work

We studied the "*the eye of typhoon*" effect with stress tests to emphasize the need for considering Disk I/O workload as a new performance metric for Grid resource scheduling. To supply more meaningful performance data to a resource scheduler, we built the forecasting system SWS(*Storage Weather Service*) using a disk monitor and a forecast engine involving several prediction methods. We presented simple prediction methods for on-line performance forecasts. The latter were shown to be quite useful. This forecast data gathered by SWS can be accessed through LDAP directory service and SOAP interface for OGSA. In future, we intend to work on time series analysis algorithms that have low-overhead and acceptable forecasting power. The time series analysis algorithms should account for the fact that data points taken over time may have internal structure(such as autocorrelation, trend or seasonal variation) that should be accounted for.

Acknowledgments. This work was supported by the Brain Korea 21 Project in 2003 at K-JIST, South Korea.

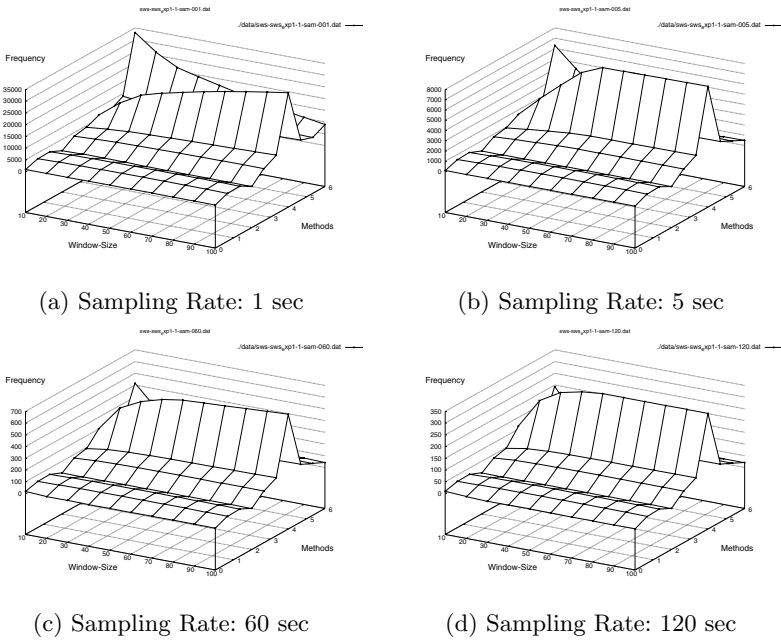


Fig. 6. Method Selection Frequency with various Sampling Rate

References

1. Arnold, D. and Agrawal, S. and Blackford, S. and Dongarra, J. and Miller, M. and Seymour, K. and Sagi, K. and Shi, Z. and Vadhiyar, S.: Users' Guide to NetSolve V1.4.1, Innovative Computing Dept. University of Tennessee, Technical Report, ICL-UT-02-05 (June, 2002)
2. I. Foster and C. Kesselman: Globus: A Metacomputing Infrastructure Toolkit, The International Journal of Supercomputer Applications and High Performance Computing (1997), 115–128
3. Rich Wolski and Neil T. Spring and Jim Hayes: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, Future Generation Computer Systems, Vol. 15 (1999), 757–768
4. Rosti, E. and Serazzi, G. and Smirni, E. and S. Squillante, M.: The Impact of I/O on Program Behavior and Parallel Scheduling, Proceeding of IOPADS (1998)
5. Peter A. Dinda and David R. O'Hallaron: Host Load Prediction Using Linear Models, No. 4, Volume 3, Cluster Computing (2000)
6. Hidemoto Nakada, Mitsuhiro Sato, Satoshi Sekiguchi: Design and Implementations of Ninf: towards a Global Computing Infrastructure, Future Generation Computing Systems, Metacomputing Issue, Volume 15 (1999), 649
7. Xiaohui Shen and Alok Choudhary: A Distributed Multi-Storage Resource Architecture and I/O Performance Prediction for Scientific Computing, Ninth IEEE International Symposium on High-Performance Distributed Computing (HPDC'00) (2000)

8. Faerman, M. and Birnbaum, A. and Casanova, H. and Berman, F.: Resource Allocation for Steerable Parallel Parameter Searches, IEEE Proceedings of the 3rd International Workshop on Grid Computing (November, 2002)
9. Francine Berman, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon and Rich Wolski: The GrADS Project: Software Support for High-Level Grid Application Development, Intl Journal of High Performance Computing Applications (2001)
10. Henri Casanova, Thomas Bartol, Francine Berman, Adam Birnbaum, Jack Dongarra, Mark Ellisman, Marcio Faerman, Erhan Gockay, Michelle Miller, Graziano Obertelli, Stuart Pomerantz, Terry Sejnowski, Joel Stiles, and Rich Wolski: The Virtual Instrument: Support for Grid-enabled Scientific Simulations, submitted to Journal of Parallel and Distributed Computing (2002)
11. Casanova, H. and Obertelli, G. and Berman, F. and Wolski, R.: The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid, SC'00 (2000)
12. Desprez, F. and Quinson, M. and Suter, F.: Dynamic Performance Forecasting for Network Enabled Servers in an Heterogeneous Environment, Int'l Conference on PDPTA'01 (2001)
13. K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima.: Performance Evaluation Model for Scheduling in a Global Computing System, The International Journal of High-Performance Computing Applications, Vol. 14, No. 3. (2000), 268–279
14. I. Foster and C. Kesselman and 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)
15. Vazhkusai, S. and Schopf M., J.: Using Disk Throughput Data in Predictions of End-to-End Grid Data Transfers, GRID '02 Workshop in conjunction with SC2002 (2002)
16. Smith, W. and Foster, I. and Taylor, V.: Predicting Application Run Times Using Historical Information, Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing (1998)
17. Iamnitchi, A. and Foster, I.: On Fully Decentralized Resource Discovery in Grid Environments, GRID2001, LNCS 2242, (2001), 51–62
18. Weissman, J. B. and Srinivasan, P.: Ensemble Scheduling: Resource Co-allocation on the Computational Grid, GRID2001, LNCS 2242 (2001), 87–98
19. SWS: Storage Weather Service, <http://sws.kjist.ac.kr>

Glosim: Global System Image for Cluster Computing*

Hai Jin, Guo Li, and Zongfen Han

Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

Abstract. This paper presents a novel *single system image* (SSI) architecture for cluster system, called Glosim. It is implemented in the kernel layer of operating system, and modifies system invokes relative to IPC objects and process signals. This system not only supports global IPC objects including message queue, semaphore and shared memory, but also a new concept of global working process. Combined with Linux Virtual Server, single IO space, it completely constructs a high performance cluster network server with SSI.

1 Introduction

Single system image (SSI) [12] is the property of a system that hides the heterogeneous and distributed nature of the available resources and presents them to users and applications as a single unified computing resource. SSI can be enabled in numerous ways, ranging from those provided by extended hardware to various software mechanisms. SSI means that users have a global view of the resources available to them irrespective of the node to which they are physically associated.

The design goals of SSI for cluster systems are mainly focused on complete transparency of resource management, scalable performance, and system availability in supporting user applications. In this paper, we present a novel single system image architecture for cluster system, called *Global System Image* (Glosim). Our purpose is to provide system level solution needed for a single system image on high performance cluster with high scalability, high efficiency and high usability. Glosim aims to solve these issues by providing a mechanism to support global *Inter-Process Communication* (IPC) objects including message queue, semaphore and shared memory and by making all the working process visible in the global process table on each node of a cluster. It completely meets the need of people of visiting a cluster as a single server, smoothly migrates Unix programs from traditional single machine environment to cluster environment almost without modification.

The paper is organized as follows: section 2 provides the background and related works on single system image in cluster. Section 3 describes the architecture design of Glosim. Section 4 describes Glosim software architecture and introduces the global IPC and global working process. Section 5 presents performance analysis of Glosim. Finally, a conclusion is made in section 6 with more future works.

* This paper is supported by National High-Tech 863 Project under grant No. 2002AA1Z2102.

2 Related Works

Research of OS kernel-supporting SSI in cluster system has been pursued in a number of other projects, such as SCO UnixWare 3, Mosix 4, Beowulf Project 5, Sun Solaris-MC 6 and GLUnix 7.

UnixWare NonStop cluster is a high availability software. It is an extension to the UnixWare operating system in which all applications run better and more reliably inside a SSI environment. The UnixWare kernel has been modified via a series of modular extensions and hooks to provide single cluster-wide file system view, transparent cluster-wide device access, transparent swap-space sharing, transparent cluster-wide IPC, high performance internode communications, transparent cluster-wide process migration, node down cleanup and resource fail-over, transparent cluster-wide parallel TCP/IP networking, application availability, cluster-wide membership and cluster time sync, cluster system administration, and load leveling.

Mosix provides the transparent migration of processes between nodes in the cluster to achieve a balanced load across the cluster. The system is implemented as a set of adaptive resources sharing algorithms, which can be loaded into the Linux kernel using kernel modules. The algorithms attempt to improve the overall performance of the cluster by dynamically distributing and redistributing the workload and resources among the nodes of a cluster of any size.

Beowulf cluster refers to a general class of clusters built for speed, not reliability, built from commodity off the shelf hardware. Each node runs a free operating system like Linux or Free-BSD. The cluster may run a modified kernel allowing channel bonding, global PID space or DIPC 8. A global PID space lets users see all the processes running on the cluster with *ps* command. DIPC make it possible to use shared memory, semaphores and wait-queues across the cluster. The clusters generally run software using parallel programming languages like MPI or PVM.

Solaris MC is a distributed operating system prototype, extending the Solaris UNIX operating system using object-oriented techniques. To achieve these design goals each kernel subsystem was modified to be aware of the same subsystems on other nodes and to provide its services in a locally transparent manner. For example, a global file system called the proxy file system, or PXFS, a distributed pseudo /proc file-system and distributed process management were all implemented.

GLUnix is an OS layer designed to provide support for transparent remote execution, interactive parallel and sequential jobs, load balancing, and backward compatibility for existing application binaries. GLUnix is a multi-user system implementation built as a protected user-level library using the native system services as a building block.

GLUnix aims to provide cluster-wide name space and uses network PIDs (NPIDs) and virtual node numbers (VNNs). NPIDs are globally unique process identifiers for both sequential and parallel programs throughout the system. VNNs are used to facilitate communications among processes of a parallel program. A suite of user tools for interacting and manipulating NPIDs and VNNs is supported. The main features provided by GLUnix include co-scheduling of parallel programs; idle resource detection, process migration, load balancing; fast user level communication; remote paging; and availability support.

3 Glosim System Overview

Figure 1 is the infrastructure of Glosim. Cluster based on Glosim consists of three major parts with three logical layers to provide high performance network services.

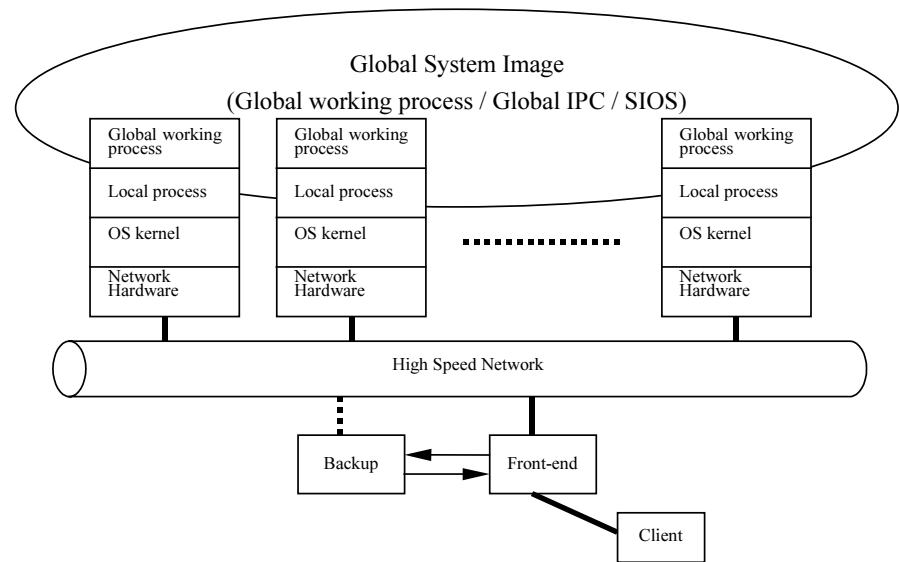


Fig. 1. Glosim Software Architecture

The first layer includes front-end and backup running Linux Virtual Server 9 to schedule and backup each other to improve system reliability. They take care of receiving client requests and distributing these requests to the processing nodes of the cluster according to certain strategy. In front-end and backup, hot standby daemon is running for fault tolerance. In case of failure in front-end, backup can take over its functions by IP faking and recover front-end kernel data to achieve high reliability.

The second layer are local OS kernel and local processes of the processing nodes inside cluster, which are in charge of system booting, providing basic system call and application software support. Logically, they are outside the boundary of SSI.

The third layer is Glosim layer, as middleware of cluster system, which includes global IPC, global working process. Glosim is implemented in OS kernel and transparently provides SSI service to user applications by modifying system invokes relative to IPC objects and process signals and by mutual cooperation and cooperative scheduling between nodes inside the cluster.

4 Glosim Software Architecture

Glosim software architecture includes two major components: global IPC and global working process. Global *inter-process communication* (IPC) mechanism is the globalization of IPC objects. It provides the whole cluster with global IPC, including consistency of shared memory, message queue and semaphore.

In global IPC, a strict consistency protocol with multiple-read/single-write is used, meaning that a read will return the most recently written value. Global IPC replicates the contents of the shared variable in each node with reader processes, but there is only one node with processes that write to a shared variable.

Global IPC can be configured to provide a segment-based or a page-based DSM. In the first case, Global IPC transfers the whole contents of the shared memory from node to node. In the page-based mode, 4KB pages are transferred as needed. This makes multiple parallel writes to different pages possible. In global IPC, each node is allowed to access the shared memory for at least a certain time period. This lessens the chances of the shared memory being transferred frequently over the network.

Global inter-process communication inside cluster system provides the basis for data transfer between processes within the cluster. The programmer can use standard inter-process communication functions and interface to conveniently write application programs suitable to cluster system. Former programs can be migrated to Glosim without too much payout. Glosim enabled applications only require the programmer to do some special transactions to global variables.

Considering a practically running cluster system, it is necessarily for a server providing single or multiple services, called working process. What we should do is to include working processes to the boundary of SSI and provide them SSI service from system level. In this way, working processes in different nodes can communicate transparently and become global working processes.

By introducing the concept of working process, we may solve the inconsistency of OS system processes between each node. OS process of each node is relatively independent, while global working process is provided by the system with single system image service. The independence of working process is convenient for the cluster system to extend to OS heterogeneous environment.

Single process space has the following basic features. Each process holds one unique pid inside a cluster. A process of any node can communicate with other processes in any remote node by signals or IPC objects. The cluster supports global process management and manages of all working processes just like in local host.

In order to uniquely tag the working process in the whole cluster, we provide a mechanism of global working process space. The purpose is to make working process pid unique within the whole cluster. As for each node, local process pid is decided by `last_pid` of OS kernel itself. Each node is independent on each other but every local pid is smaller than the lower limit of global process space fixed in advance. That is, local process pid is independent on each other while global working process pid is unique in the whole cluster.

Global proc file system is to map status messages of all processes in every node to the proc file system of each node within the cluster. It includes local processes with the process number smaller than `max_local_pid` and global processes with the process number larger than `max_local_pid`.

With the actualizing of global proc file system, we can use `ps -ax` command to examine local processes running in current node and all global working processes in the whole Glosim system. We can employ `kill` command to send signals to any process and even use `kill -9` command to end any process we want with no regard to which node it actually runs.

5 Performance Analysis

This section presents some basic experimental results on the performance of the Glosim cluster. The benchmark program sets up a shared memory, a message queue, and a semaphore set. The tested system calls are: `semctl()`, `msgctl()`, `shmctl()` with the `IPC_SET` command, `semop()` and `kill()` operations. Besides, `msgsnd()` and `msgrcv()` with different message sizes are also evaluated.

Figure 2 is an experiment result of the execution speed for some Glosim system calls. It shows that Glosim has a normal and acceptable response speed to atomic operation of IPC objects and global signal processing mechanism.

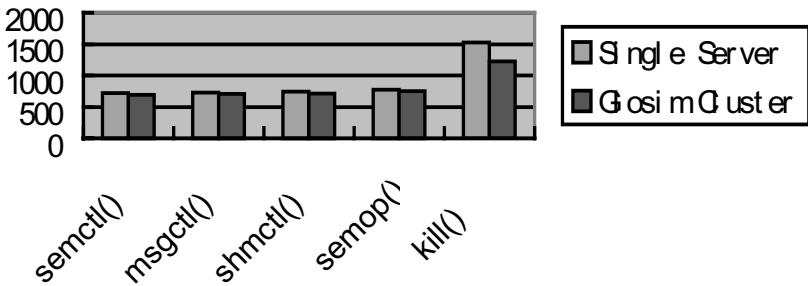


Fig. 2. Speed of executing some Glosim system calls

Figure 3 shows some experiment results to measure the speed of send and receive message in the system. Fig.3(a) shows the message transfer bandwidth with various message sizes in single server. Fig.3(b) shows the system message transfer bandwidth with various message sizes in a Glosim server with 8 nodes. It shows that in case of small data load (4KB below), message send/receive bandwidth in Glosim server is a little less than that in single server. The results appear to be quite normal and acceptable because Glosim server involves network transfer and the overload of data copy of kernel and user space. Thus, from these simple experimental results, we can see that Glosim also has excellent performance support for practical system.

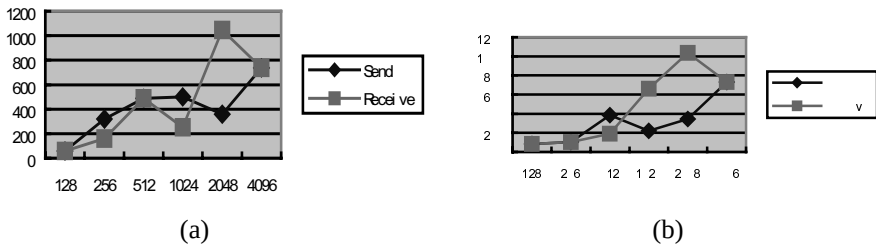


Fig. 3. Speed of send and receive messages for (a) single server (b) Glosim server with 8 nodes

6 Conclusions and Future Works

In this paper, we present a novel architecture of a single system image built on cluster system, named Glosim. This system not only supports global IPC objects including message queue, semaphore and shared memory, but also presents a new concept of global working process and provides it SSI support smoothly and transparently. Combined with Linux Virtual Server and SIOS 10, it completely constructs a high performance SSI cluster network server.

Based on Glosim, processes can exchange IPC data through numerical key value. Global process pid is used to identify each global working process, which means it is unnecessary to know where the responding IPC structure physically exists and in which node a global working process physically runs. Logical addressing of resources and global pid addressing make processes independent of physical network and transparently provide single system image to upper application programs.

With all its advantages mentioned, however, Glosim is utilized currently with the disadvantages of relatively high system overhead, fault tolerance to be improved and only indirect support to application program global variables. These are the issues we need to solve in our future plan.

References

1. Buyya, R., Cortes, T., and Jin, H.: Single System Image, The International Journal of High Performance Computing Applications, Vol.15, No.2, Summer 2001, pp.124–135
2. Hwang, K. and Xu, Z.: Scalable Parallel Computing: Technology, Architecture, and Programming, McGraw-Hill, New York, 1998
3. Walker, B. and Steel, D.: Implementing a full single system image UnixWare cluster: Middleware vs. underware, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), 1999
4. Barak, A. and La'adan, O.: The MOSIX multicomputer operating system for high performance cluster computing, Future Generation Computer Systems, Vol.13, No.4–5, pp.361–372, March 1998
5. Hendriks, E.: BProc: the Beowulf distributed process space, Proceedings of the 16th International Conference on Supercomputing, 2002, pp.129–136
6. Khalidi, Y. A., Bernabéu Aubán, J. M., Matena, V., Shirriff, K., and Thadani, M.: Solaris MC: A Multi-Computer OS, Proceedings of 1996 USENIX Annual Technical Conference, January 1996, pp.191–204
7. Petrou, D., Rodrigues, S. H., Vahdat, A., and Anderson, T. E.: GLUnix: A global layer Unix for a network of workstations, Software Practice and Experience, Vol.28, No.9, pp.929–961, 1998
8. Karimi, K. and Sharifi, M.: DIPIC: A System Software Solution for Distributed Programming, Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, USA, 1997
9. Zhang, W.: Linux virtual servers for scalable network services, Proceedings of Ottawa Linux Symposium 2000, Canada
10. Hwang, K., Jin, H., Chow, E., Wang, C. L., and Xu, Z.: Designing SSI clusters with hierarchical checkpointing and single I/O space, IEEE Concurrency, Vol.7, No.1, pp.60–69, 1999

Exploiting Locality in Program Graphs^{*}

Joford T. Lim, Ali R. Hurson, and Larry D. Pritchett

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
hurson@cse.psu.edu

Abstract. The success of multithreaded systems depends on how quickly context switching between threads can be achieved. A fast context switch is only possible if threads are resident in fast but small memories. This limits, however, the number of active threads and thus the amount of latency that can be tolerated. The generality of dataflow makes it difficult to fetch and execute a sequence of logically related threads through the processor pipeline, thereby removing any opportunity to use registers across thread boundaries. Relegating the responsibilities of scheduling and storage management to the compiler alleviates this problem to some extent. In conventional architectures, the reduction in memory latency is achieved by providing (explicit) programmable registers and (implicit) high-speed caches. Amalgamating the idea of caches or register-caches within the dataflow framework can result in a higher exploitation of parallelism and hardware utilization. This paper investigates the suitability of cache memory in a dataflow paradigm. We present two heuristic schemes that allow the detection, exploitation, and enhancement of temporal and spatial localities in a dataflow graph (dataflow program). Dataflow graphs are partitioned into sub-graphs while preserving localities, and sub-graphs are distributed among the processors in order to reduce cache misses and communication cost. Simulation results showing the performance of the partitioning algorithms are presented and analyzed.

1 Introduction

The traditional approaches to concurrent processing are based on the control-flow model of computation, where a program counter is used to schedule instructions of a task. In a multiprocessor organization, the basic control-flow model is extended to allow more than one thread of control to be active at any instant, by introducing special control operators for activating and synchronizing these threads.

The dataflow model presents an alternative to the traditional control-flow model of computation, where the execution of an instruction is based on the availability of its operands. Consequently, instructions in the dataflow model do not impose any sequencing constraints except for the data dependencies of the program. In a dataflow machine, maximal concurrency can be exploited, constrained only by the availability of hardware resources. Dataflow machines may be regarded as extreme examples of multithreaded machines where each instruction constitutes an

^{*} This work was supported in part by NSF under Grant MIP-9622836.

independent thread and all enabled threads (instructions) are scheduled for execution. Synchronization is enforced at the instruction level, as every instruction waits for its operands to be produced before execution. The self-scheduling of threads (instructions) can tolerate arbitrary memory latencies. The fine-grained nature of dataflow threads, however, fails to exploit spatial and temporal localities that architectural features such as cache memories are designed to capitalize on. The current trend in the design of dataflow processors suggests a hybrid of non-strict fine-grained instruction execution and strict coarse-grained thread execution to exploit locality. It has been shown that a coarse-grained execution model outperforms a fine-grained execution model in dataflow execution of numeric codes [13]. This suggests that the effect of increased thread granularity on latency tolerance is minimal, and in large part is offset by performance gains from intra-thread locality. This observation led us to believe that many control-flow features, such as register files, cache memories, and instruction prefetching, should be studied in a dataflow context.

Several researchers [1], [5], [8], [15], [16], [19] have studied the application of caches in a dataflow environment. Our detailed investigation of cache design showed that instruction cache issues were very similar to those in control-flow architectures [8]. To truly exploit cache memories, it is necessary to investigate optimization techniques for enhancing localities in dataflow programs. Compile time analysis (partitioning of programs into threads, proper scheduling of threads to exploit localities, effective placement and prefetching techniques) within the context of dataflow is needed. In control-flow environments, the reuse of instructions within a loop in successive iterations enhances temporal locality. It may also be possible to achieve similar results in dataflow programs. Straight-line code may provide opportunities for exploiting spatial localities — a set of instructions representing a path of activity determined by data dependencies constitute a locality if they are grouped together in the virtual address space.

The Vertically Layered (VL) allocation scheme proposed in [11] addresses the issue of partitioning and allocation of a program graph in a multiprocessor system. In this scheme, the nodes of a program graph are arranged into vertical layers (partitions) such that each vertical layer can be allocated to a processor. Spatial localities are thus exploited by clustering nodes connected serially in a vertical layer. Collapsing some of the vertical layers together and allocating them to the same processor can further minimize inter-processor communication cost (IPC).

This work expands the domain of our previous research in the application of caches in the dataflow environment, and scheduling and allocation of the program graphs in a multiprocessing environment. The scope of the VL algorithm is enhanced by exploiting temporal localities in a program graph. In addition, a simple heuristic is used to properly allocate and distribute temporal localities among processors. Section 2 introduces issues pertaining to the cache memories in a dataflow context. The vertically layered (VL) allocation scheme is presented In Section 3. A new locality-enhancing scheme (VL-Cache) is described in Section 4. Section 5 discusses the simulation of the VL-cache and analyzes the simulation results. Finally, section 6 concludes the paper and addresses some future research directions.

2 Cache in Dataflow Environment

The design of cache memories is subject to more constraints and trade-offs than the design of main memories. Issues such as the placement/replacement policy, fetch/update policy, homogeneity, the addressing scheme, block size, and bandwidth are among those that should be taken into consideration. Optimizing the design of a cache memory is concerned with four major aspects:

1. Maximizing the probability of finding a memory reference in the cache,
2. Minimizing the time to access information that is residing in the cache,
3. Minimizing the delay time due to a cache miss (miss-penalty), and
4. Minimizing the overhead of maintaining multi-cache consistency.

2.1 Locality in Program Graph

If we consider the body of a loop to comprise a locality pattern, then the complete execution of the loop appears as a number of repetitions of that pattern. These repetitions may be partially distinct (e.g., DOACROSS) or they may overlap (e.g., DOALL). In a sequential environment, the instructions of a loop are reused in successive iterations. If instructions are similarly reused in a dataflow environment, temporal locality can result. Straight-line code may also produce spatial locality in a dataflow environment. In fact, any section of the code may produce several exploitable spatial localities. An exploitable spatial locality is a set of instructions representing a path of activity determined by data dependencies if they are grouped together in the virtual address space.

2.2 Limits of Dataflow Multiprocessing

While locality-of-reference is enhanced by coarse-grained threads, the success of multithreaded dataflow depends on how quickly context switching can be achieved. Fast context switch is possible if threads are resident in fast memories, such as cache. Caches are relatively small, and hence, the number of active threads that can be resident in caches is limited. Since latency tolerance is fundamental to the performance of multithreading [3], a large degree of parallelism is needed to achieve greater latency tolerance. On the other hand, it has been shown that in dataflow multithreaded systems, the best performance is obtained when the number of enabled threads (i.e., degree of parallelism) is equal to the maximum number of thread contexts that can be contained in the cache — increasing the number of active threads beyond this maximum actually degrades the performance [16]. Thus it is necessary to carefully manage cache memories and the amount of parallelism. In [8], [19], the degree of parallelism was controlled by using limiting the number of enabled threads considered for scheduling, minimizing cache misses. Alternatively, cache prefetching and replacement policies can be utilized to ensure that enabled threads have their data and instructions already in cache – minimizing long latencies.

TAM [4] employs a storage directed scheduling scheme to minimize latencies. A TAM program consists of a collection of code-blocks, roughly corresponding to

functions in the source code. Each code-block in turn consists of a number of threads. When a code-block is invoked, an activation frame is allocated to act as local storage for the code-block. The scheduling of threads in TAM is closely tied to the storage model; all active threads in an activation frame are allowed to complete before switching to another activation frame. This approach has the potential to improve cache performance since storage for related threads can be co-located and prefetched.

In the Multi-Threaded Architecture (MTA), a Register-Use Cache (RU-cache) which corresponds to register sets assigned to threads [6] was used. This approach requires multiple register sets and a large register file. A register file with n register sets will have an RU-cache of n entries. Each entry corresponds to a register set and contains the function pointer (FP) of the function instance to which a register set is assigned. Once a thread is enabled, the RU-cache is associatively searched for a frame pointer, which matches the frame pointer value of the ready thread. A match indicates that the thread should be prioritized. Hence, there is a high probability that once a thread is executed, its data will be resident in a register set.

Similar to conventional control-flow computers, cache blocks in dataflow systems can be prefetched by defining working sets associated with threads and prefetching the cache blocks that have a high probability of future references [8]. It has been shown that the reference streams of programs executed in TAM can be characterized by a working set function that is similar to those associated with uni-processor, single threaded programs [12]. Alternatively, an enabled thread can be added to a ready queue only when its data and instructions are in the cache.

Cache replacement policies may also play a role in the performance. For example, when multiple activation frames are associated with a code-block (i.e., multiple loop iterations are active), the instruction cache blocks associated with the code-block are poor candidates for replacement [8]. Proper replacement policies for data caches can produce substantial performance gains, particularly for loop iterations, which contain temporal locality. Information predicting future references based on compile time analysis can also be utilized to improve replacement strategies. As can be seen from the discussion thus far, it is important to carefully balance thread scheduling and data placement to permit appropriate prefetching and replacement techniques to effectively utilize cache memories.

2.3 Cache Memory Designs with ETS

Issues related to operand and instruction caches within the Explicit Token Store (ETS) dataflow model were explored in [8]. In ETS, a program consists of a collection of code-blocks (disjoint sub-graphs) — a code-block usually represents a loop or a function. When a code-block is invoked, a block of memory known as an activation frame is allocated for storing and matching operands belonging to the instructions in the code-block. There can be several activation frames associated with a code-block, representing the invocation of multiple loop iterations in parallel. As with other dynamic dataflow models, ETS tokens carry a tag, consisting of an instruction pointer (IP), which refers to the instruction within a code-block and (activation) frame pointer (FP) which points to the base address of an activation frame — direct matching. Each instruction (identified by the IP) contains an offset (r) within

the activation frame where the match will take place, and one or more displacements that define the destination instructions receiving the result token(s), along with input port (left/right) indicators to specify appropriate input arc for destinations. FP+r is the memory location where the tokens for the instruction are matched [14].

2.3.1 Instruction Cache in ETS

The structure of the instruction cache is very similar to a conventional set-associative cache. The low order bits of the instruction address (IP) are used to map instruction blocks into N sets. Within each set the search for a block is done associatively using the higher order bits. Each block in the cache contains the following information:

- **Tag:** Usual information needed for locating an address.
- **Valid-bit:** Usual bit for detecting invalid data.
- **Process count:** Represents the number of active threads (or frames) that refer to instructions in the cache block. This information is used for cache replacement — an instruction block with the smallest process count is a good candidate for replacement.

ETS instructions within a code-block can be reordered to increase locality. The reordering can be based on the techniques described earlier to exploit localities. In the study reported in [18], instructions are reordered based on the expected time of availability of operands — E-level reordering. The instruction memory is then partitioned into blocks and working sets. Blocking is defined to achieve compatibility with the memory bandwidth. Working set defines the average number of instructions that are data independent — the instructions in a working set are prefetched. While the optimum working set depends on the program, it was found that working sets of 4 to 8 instructions yield significant performance improvement [8], [18].

It was observed that the ETS instruction cache behaves similarly to conventional instruction cache memories in terms of the performance due to variations in the total cache size, set associativity, and cache block size. Using process count as a cache replacement policy, compared to a random replacement strategy, reduced the number of cache misses by 20% to 70%.

2.3.2 Operand Cache in ETS

The design of the operand cache in ETS-like architectures is more complex. A two-level set-associative design for the operand cache was employed. The two levels of associativity result from the need to maintain the association between operands and instructions within a context, and to maintain multiple invocations of the same code-block. At the first level of associativity, the operand cache is organized as a set of super-blocks. Each active context (activation frame associated with a code-block) occupies a super-block. The second level of associativity is used for accessing individual locations within a frame. A super-block consists of the following information:

- **Cold bit:** Used to indicate if the super-block is occupied or not. This information is used to eliminate misses due to cold starts. In the dataflow model, since the first operand to arrive will be stored (written), there is no need to fetch an empty location from memory. The cold bit with a super-block is used to allocate an entire frame (or context), and is set when the first operand is written to the frame.

- **Tag:** Serves to identify the context (or frame) that occupies the super-block. This is based on the FP address obtained from a token.
- **Working set identifiers:** The memory locations within an activation frame (used for token matching) are divided into blocks and working sets, paralleling the blocks and working sets of the instructions in the code block. Thus, a super-block contains more than one working set, and these are accessed associatively (the second level of set associativity). Each working set of a super-block also contains a cold start bit. This bit is used to eliminate unnecessary fetches from memory when the operands are being stored in the activation frame.

A few replacement algorithms for replacing working sets within a super-block and super-blocks themselves were explored [8]. For working set replacement, a used words policy was employed. This policy replaces working sets containing memory locations already used for matching operands (hence will not be used in this activation). For super-block replacement, the dead context replacement policy that replaces a super-block representing a completed context (or frame) was used.

The operand cache must accommodate several contexts corresponding to different loop iterations, as well as contexts belonging to other code-blocks. In order to minimize the possibility of thrashing, the number of active contexts must be carefully managed (process control). The number of active contexts will depend on the cache size and the size of an activation frame. By reusing locations within a frame, the size of an activation frame can be reduced, accommodating more active threads in cache.

The effect on cache miss ratio was explored by varying the number of active processes. It was observed that for an operand cache with k -way super-block associativity and N sets, the optimal number of processes is $N*k$. It was also observed that the use of the dead context replacement policy for replacement of super-blocks produced as much as 70% improvement over random replacement strategies. In addition, used words replacement policy in replacing working sets within a super-block produced between 3% and 9% improvement over random replacement strategies. Finally, the use of cold start bits with operand locations that are yet to be defined eliminated between 40% and 100% of cache misses.

3 Vertically Layered Allocation Scheme

The Vertically Layered (VL) allocation scheme [11] was developed to compromise computation and communication costs. VL performs both thread partitioning and allocation. The input to VL is a DAG representation of a program $G \equiv G(N, A)$, where N represents the set of instructions and A represents the partial ordering $<\bullet$ between the instructions. A directed path from node n_i to node n_j implies that n_i precedes n_j (i.e., $n_i <\bullet n_j$). An expected execution time t_i is associated with every node $n_i \in N$ and a communication cost c_{ij} is considered for every arc $a(n_i, n_j) \in A$.

The VL allocation scheme consists of two separate phases: a separation phase and an optimization phase. In the separation phase, a program graph is partitioned into vertical layers based only on the execution times t_i , where each vertical layer consists of one or more serially connected set of nodes (threads) that are considered for

assignment to a processing element (PE). To determine the appropriate vertical layers, approximate methods are used to estimate the execution times of the conditional nodes and loops. Once the expected execution times are assigned, iteratively, the critical path and the longest directed paths (for identifying vertical layers) of the program graph are computed. By assigning the nodes that lie on the critical path (or longest path) to a single vertical layer, the communication overhead associated with nodes in a thread is minimized.

In the optimization phase, the communication to execution time ratio (CTR) heuristic is used to further optimize the allocation by considering the inter-PE communication costs. This is done by considering whether the inter-PE communication overhead offsets the advantage gained by overlapping the execution of two threads in separate processing elements. This process is repeated in an iterative manner until no improvement in performance is obtained by combining two threads allocated to different processors.

4 Proposed Scheme – VL-Cache

After allocation of the threads to the processors, the horizontal levels (E-level) of nodes in each processor should be rearranged to take the communication cost into account. The activation of a node is delayed if its inputs come from a remote processor. These delays could alter the actual formation of horizontal levels for execution. The earliest start-time of each node (operation) including communication cost can be used to identify the horizontal layers of execution (or E-level partitioning). The E-levels will be used for interleaving threads and scheduling them on the processor pipeline.

In section III, serially connected nodes (threads) were referred to as either a critical path or longest directed path (LDP). In this section, we will simply refer to them as threads. An example of threads that are allocated to vertical layers/processor with their corresponding horizontal layering (h) is shown in Figure 1(a). The dependencies between threads were omitted in order to improve clarity. Nodes 1 to 8, 9 to 12, 13 to 16, and 17 to 22, respectively, represent threads A, B, C, and D. Each thread is partitioned into groups of x (*block-interleaving*) operations and these groups are interleaved and stored in memory — each block partition is a multiple of the cache line size. In other words, starting from $h = 1$ groups of x consecutive instructions from all threads that have unassigned nodes in horizontal layer h are block-interleaved and assigned to memory. After exhausting all nodes in level h , the assignment is repeated for unassigned nodes of level $h + 1$. This process completes when all nodes in the graph are assigned to memory. For example, referring to figure 1 and with $x = 4$, the first four nodes of thread A (nodes 1, 2, 3, and 4) are assigned to the first four locations of memory. No other thread has a node in $h = 1$, so h becomes 2. The next thread that has unassigned node in $h = 2$ is thread B. Therefore, nodes 9, 10, 11, and 12 are assigned to the next four locations in memory. The first four nodes of threads C (nodes 13, 14, 15, and 16) and D (nodes 17, 18, 19, and 20) are then allocated to the memory. At this point all nodes at $h = 2$ are exhausted and h is incremented by 1. The next value of h with unassigned nodes is $h = 5$. Nodes 5, 6, 7, and 8 from thread A are allocated to the next memory locations. At $h = 6$ only thread D has unassigned

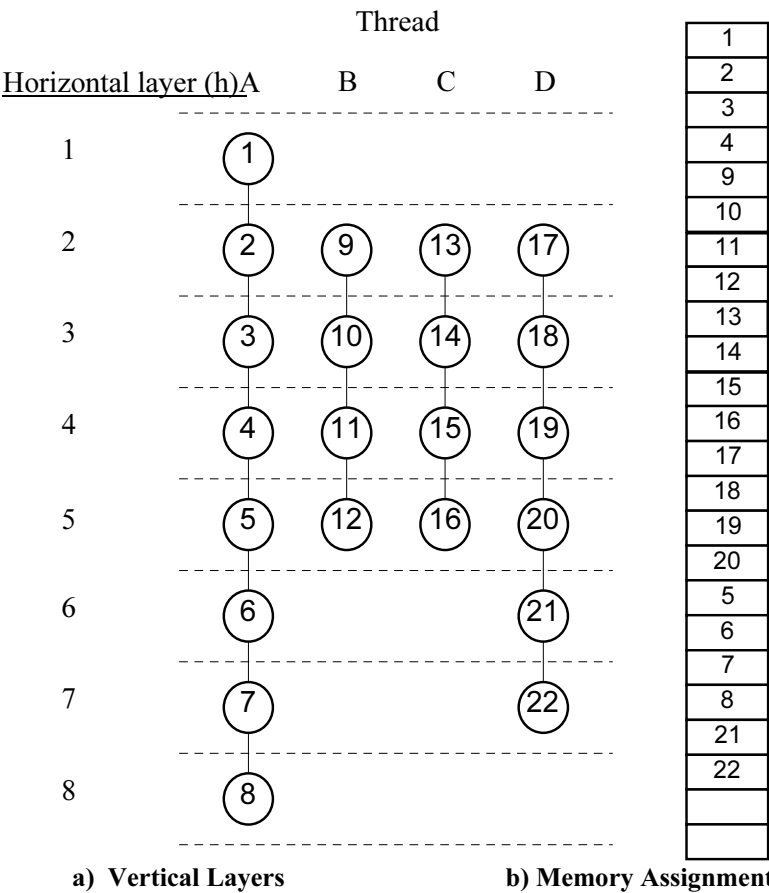


Fig. 1. An example of threads assigned to a vertical layer

nodes (nodes 21 & 22) that are finally assigned to memory. The resulting node arrangement in memory is shown in Figure 1(b). The ordering policy maintains locality within threads and accommodates a high degree of parallelism. Because of the block-interleaved assignment of memory, instructions from different threads can be interleaved during execution time without causing unnecessary cache misses.

The best value of x is dependent on the computational model (e.g., data-driven, blocking-threads or sequential control-flow) and the thread scheduling policy used (i.e, interleaving, priority, preemptive). A small value of x increases the probability that when a node executes, the matching memory locations for its inputs and the operand locations for its destinations are resident in cache, but still offers a greater parallelism by accommodating more active threads. A large value of x achieves greater locality within threads at the cost of limited instruction level parallelism.

In E-level ordering, the nodes are also arranged based on horizontal layering, but the value of $x = 1$. This leads to smaller probability of having destination operands

resident in the operand cache than that for VL-Cache scheduling policy with $x > 1$. In addition, the VL-allocation of threads reduces overall execution times. Our scheduling policy would also improve cache utilization since it allows for prefetching of cache blocks. “Used word” replacement policy is also simple to implement with our technique. For the data blocks, the size of the working set is a crucial factor to guarantee the availability of the resultant destinations in the cache.

5 Performance of VL-Cache Policy

A simulator was developed to measure the feasibility of the proposed locality enhancing policy (VL-Cache). The simulator was used to compare the VL-Cache algorithm with E-level ordering in terms of cache misses for both instruction and operand caches. We used IF-1 graphs from a Sisal compiler [2] for our experiments. The Fast Fourier Transform (FFT), Simple, and SLoop 74 are used as our test bed. Simple is a hydrodynamics and heat conduction code widely used as an application benchmark and SLoop 74 is a loop from Simple. Table 1 lists the characteristics of the programs used in our current experiment.

Table 1. Program Characteristics

Program	No. of Instruction References	No. of Operand References
FFT	173135	132522
Simple	227219	143919
Sloop 74	133210	90347

Figure 2 shows the performance of the instruction cache for SLoop 74, using VL-Cache algorithm for different x -values. In Figure 2, VL represents a policy in which threads assigned to a processor are not interleaved — the x -value for VL is equal to the length of each thread. The differences between the miss rates for different values of x are insignificant. We recorded a cache miss rate of 0.00015 for 128-byte cache and 0.00013 for cache sizes in the range of 256-2048 bytes. This emphasizes the conclusions of [8] that even small instruction caches offer substantial performance gains for dataflow processing.

The operand cache, on the other hand, was very sensitive to the value of x . Figure 3 shows that the best performance is attained for $x = 2$. In Section IV, we stated that the value of x is dependent on the execution paradigm and scheduling policy. As observed in Section IV, the data driven paradigm underlying dataflow architecture favors smaller x values — a small value of x increases the probability that when a node executes, the matching memory locations for its inputs and the operand locations for its destinations are resident in cache, but still offers a greater parallelism by accommodating more active threads. The VL-scheme (where x is equal to the length of each thread) is not very well suited for dataflow, since a large value x achieves greater locality within threads at the cost of limited instruction level parallelism and

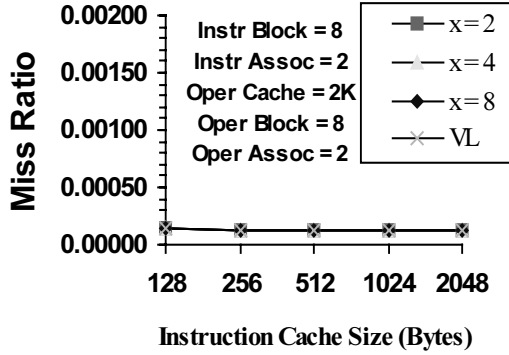


Fig. 2. Effect of different values of x on instruction cache for Sloop74

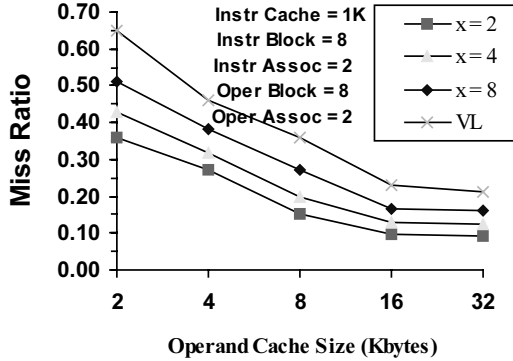


Fig. 3. Effect of different values of x on operand cache for SLoop74

opportunities for interleaving threads or prefetching to support interleaving. In the remaining experiments we will use $x = 2$.

Figure 4 compares the performance of VL-Cache with E-level (i.e., $x = 1$) for SLoop74. Because of very small cache miss rates, the results show only negligible differences in instruction cache misses. In the case of operand cache, however, as depicted in Figure 4(b), the VL-Cache policy shows improvements over E-level ordering. This improvement is due to the fact that the VL-Cache policy utilizes the “non-far reaching” effect of dataflow execution model — VL-Cache policy schedules the destination operands very close to the current instructions, hence, increases the probability of destination operands being resident in the operand cache. E-level policy does not account for intra-thread locality spanning across horizontal layers.

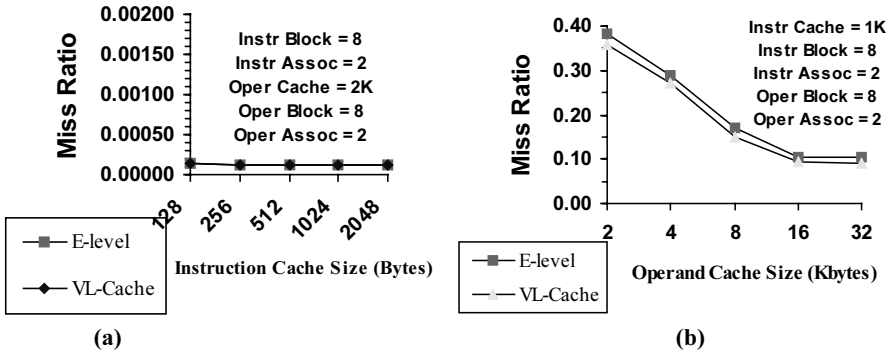


Fig. 4. VL-Cache vs. E-level ordering (SLoop 74)

We also compared the behavior of VL-Cache policy against E-level ordering for various cache block sizes. Instruction cache behavior for the two polices are

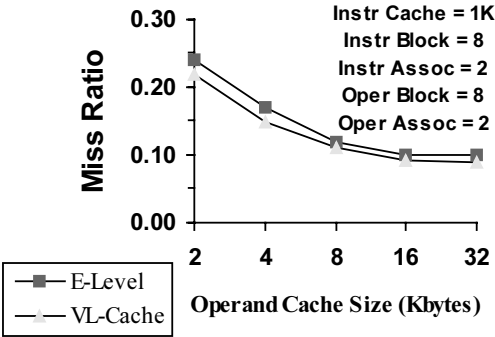


Fig. 5. VL-cache vs. E-level ordering (FFT)

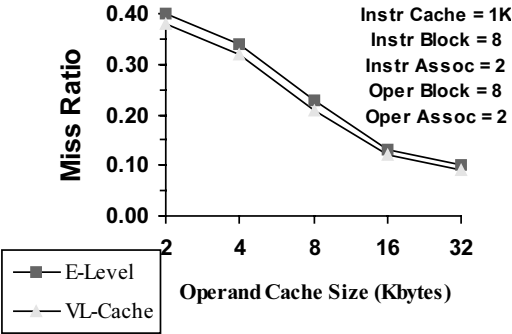


Fig. 6. VL-Cache vs. E-level ordering (Simple)

indistinguishable. VL-Cache policy consistently performed better than the E-level policy for the operand-caches (Figures 5 and 6).

Finally, Figure 7 depicts the percentage of performance improvement attained by using VL-Cache over E-level ordering for FFT and Simple. In our experiments, we

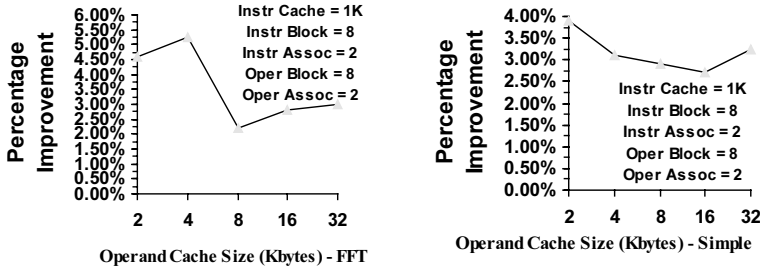


Fig. 7. Improvement of VL-Cache over E-level ordering

assumed that memory accesses required 6 cycles while cache accesses required 2 cycles. The largest improvement for FFT was 5.26% with 4K byte operand caches. When the experiment was repeated for Simple, the best improvement was 3.9% with 2K byte operand caches. For larger caches, the performance differences decrease, since the overall cache misses become small. This feature is attractive for cases when the cache size of a processor is fixed or limited.

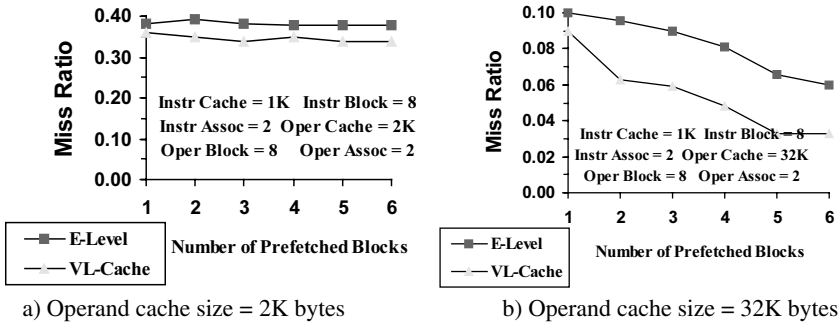


Fig. 8. Performance of operand cache with prefetching for SLoop74

To further determine the effectiveness of the VL-Cache scheme, our simulator was extended to allow prefetching. A simple prefetching scheme, wherein the processor fetches a fixed number of blocks adjacent to the fetched block was adopted. We varied the number of prefetched blocks from 1 to 6. Figure 8 shows the prefetching effect on the performance of the operand cache for both the VL-Cache and E-level ordering for various cache sizes. From figure 8a, it can be concluded that for the 2K-byte operand cache, VL-Cache ordering offers some improvement over E-level ordering. Here we see an almost constant gap between the miss ratio of VL-cache and E-level ordering. The miss ratio of VL-cache decreases somewhat when the number of prefetched blocks was increased from 1 to 3, and then starts to level off. The lowest obtained miss ratio dropped by only 0.02 or 5% with prefetching

compared to no prefetching. This demonstrates that prefetching provides minimal improvement when the cache size is small for this type of application program. Similar to our earlier observations (Figure 3), in organizing cache blocks the VL algorithm does not take non-far reaching effect of the dataflow model into consideration — a cache block could be swapped back and forth between cache and main memory several times during the corresponding activation frame’s lifetime. VL-Cache showed an improvement of 10% over the best performance of E-level ordering, compared to only 6% without prefetching (Figure 4).

For the 8K-byte operand cache, the result is very similar, except that the amount of improvement has increased. The lowest miss ratio obtained, dropped by 0.025 or 17% with prefetching compared to no prefetching. A 23% improvement for VL-cache over E-level was obtained, compared to 12% without prefetching (Figure 4). Finally, for the 32K-byte operand cache, an even greater improvement was obtained. The lowest obtained miss ratio obtained dropped by 0.057 or 64% with prefetching compared to no prefetching. A 45% improvement for VL-cache over E-level was obtained, compared to 10% without prefetching (Figure 4).

The performance of prefetching for Simple and FFT are shown in Figures 9 and 10, respectively. For large applications like Simple, prefetching offers a significant improvement. For Simple (Figure 9), the miss ratio for the 2K-byte cache, with 4

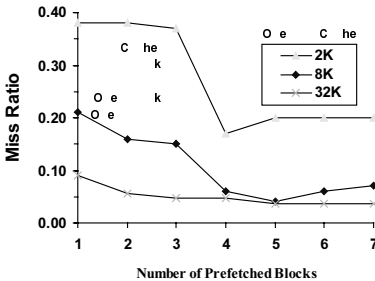


Fig. 9. Performance of operand cache with prefetching, using VL-Cache for Simple

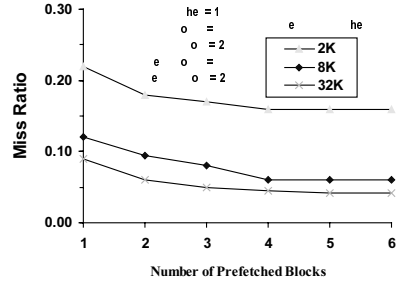


Fig. 10. Performance of operand cache with prefetching, using VL-Cache for FFT

prefetched blocks is less than the miss ratio for the 8K-byte cache with no prefetching. In fact, this miss ratio is close to the miss ratio for 2 prefetched blocks for the 8K case. This means that prefetching allows similar performance for smaller cache sizes. The same results can be found for 8K-byte and 32K-byte cache sizes. The miss ratio of the 8K cache with 4 prefetched blocks performs better than the 32K cache with no prefetching, and with 5 prefetched blocks, the 8K case obtains the same result as the best result obtained for the 32K case. Again this shows that by using prefetching with VL-Cache, for some applications, we can obtain the same performance using smaller caches than we would have with larger caches.

For FFT (Figure 10), results similar to Simple were obtained for operand caches of size 8-32K-byte. The miss ratio for the 8K cache with 3 prefetched blocks is lower than the miss ratio of the 32K cache with no prefetching. Also, the best performance for the 8K cache is very close to the best performance of the 32K cache.

6 Conclusions and Future Research

A new locality enhancing policy called VL-Cache that utilizes the threads produced by the Vertically Layered allocation scheme has been introduced in this paper. This new scheme interleaves thread instructions, at the block level, based on both horizontal and vertical layering. The effectiveness of the VL-Cache policy relative to E-level ordering was presented. VL-Cache attains better performance on operand caches than E-level ordering. In addition, VL-cache performs better when the operand cache size is small. Further observations show that VL-Cache improves its performance even further when prefetching is performed. The performance of a smaller cache with prefetching is comparable to the performance of a much larger cache without prefetching. This shows the effectiveness of instruction reordering in improving the performance of cache. We feel that the proposed VL-Cache policy is general enough to accommodate a variety of architectures, including architectures that exhibit behavior similar to multithreaded dataflow architectures, such as multithreading (switch on event, SoEMT or simultaneous, SMT) and out-of-order execution. The reordering strategy (x -value) can be tailored to the type of processing paradigm. For non-blocking, dataflow like scheduling, small values of x are better, while for blocking thread models or priority based thread scheduling systems, a larger value of x may result in better cache performance. We plan to further explore this issue in the near future.

We are currently enhancing the ETS cache simulator with smart prefetching and replacement policies to further reduce cache misses. A smart replacement policy would improve cache utilization as well, since it would replace the blocks that have been executed (and are no longer needed). Prefetching can also reduce memory latency, since prefetching can overlap execution and bring needed blocks into cache before they are actually required. Reusing matching locations for more than one instruction within a code-block [8] within the context of VL-Caches and prefetching will also be investigated. Operand reuse not only increases the number of active threads that can be accommodated in a cache, it also reduces the number of cache blocks that must be fetched.

References

1. Ang, B. S., Arvind, and Chiou, D.: StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. Proceedings of the 19th International Symposium on Computer Architecture, Dataflow Workshop. (1992).
2. Cann, D. C.: The Optimizing SISAL Compiler Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA (1992).
3. Culler, D., Schauser, K. E., Eicken, T.: Two Fundamental Limits on Dataflow Multiprocessing. Proceedings of the IFIP WG 10.3 Working Conference on Architecture and Compilation Techniques for Fine and Medium Grain Parallelism. (1993).
4. Culler, D. E., Goldstein, S. C., Schauser, K. E., and Eicken, T.: TAM ó A Compiler-Controlled Threaded Abstract Machine. Journal of Parallel and Distributed Computing, Vol. 18 (1993) 347–370.
5. Hum, H. H. J. and Gao, G. R.: A High-Speed Memory Organization for Hybrid/von Neumann Computing. Future Generation Computer Systems. Vol. 8, No. 4 (1992) 287–301.

6. Hum, H. H. J., Theobald, K. B., and Gao, G. R.: Building Multithreaded Architectures with Off-the-Shelf Microprocessors. *Proceedings 8th International Parallel Processing Symposium*. (1994) 288–294.
7. Hurson, A.R., Kavi, K., Lee, B., and Shirazi, B.: Cache Memories in Dataflow Architectures: A Survey. *IEEE Parallel and Distributed Technology*, Vol. 4, No. 4 (1996) 50–64.
8. Kavi, K. M., Hurson, A. R., Patadia, P., Abraham, E., and Shanmugam, P.: Design of Cache Memories for Multithreaded Dataflow Architecture. *Proceedings of the 22nd International Symposium on Computer Architecture*. (1995) 253–264.
9. Kwak H., Lee B., Hurson A.R., Yoon S-H., and Han W-J.: Effect of Multithreading on Cache Performance. *IEEE Transactions on Computers*, Special issue on Cache Memory. Vol. 48, No. 2 (1999) 177–184.
10. Lee, B. and Hurson, A. R.: Dataflow Architectures and Multithreading. *Computers*. (1994) 27–38.
11. Lee, B., Hurson, A. R., and Feng, T. Y.: A Vertically Layered Allocation Scheme for Dataflow Systems. *Journal of Parallel and Distributed Computing*. Vol. 11 (1991) 175–187.
12. Motomura, M. and Papadopoulos, G. M.: Local Memory Reference Behavior of Fine-Grain Multithreaded Execution. *Computation Structures Group Memo 346*, Massachusetts Institute of Technology Laboratory for Computer Science. (1993)
13. Najjar, A. W., Bohm, A. P. W., and Miller, M.: A Quantitative Analysis of Dataflow Program Execution ó Preliminaries to a Hybrid Design. *Journal of Parallel and Distributed Computing*, Vol. 18. (1993) 314–326.
14. Papadopoulos, G. M.: *Implementation of a General-Purpose Dataflow Multiprocessor*. Cambridge, Massachusetts, MIT Press. (1991)
15. Patadia, P., Karani, V., Kavi, K., Shanmugam, P., Shirazi, B., and Hurson, A. R.: Improvements to the ETS Dynamic Dataflow Architecture. *Proceedings of the 27th Hawaii Conference on System Sciences*. (1994), 378–387
16. Shanmugam, P., Andhare, S., Kavi, K. M., Shirazi, B., and Hurson, A. R.: Cache Design for an Explicit Token Store Dataflow Architecture. *Proceedings 5th Symposium on Parallel and Distributed Processing*. (1993) 45–50
17. Skedzielewski, S. and Glauert, J.: IF1 - An Intermediate Form for Applicative Languages. *Lawrence Livermore Laboratory Manual M-170*. (1985)
18. Sunahara H., and Tokoro, M.: On the Working Set Concept for Data-Flow Machines: Policies and Their Evaluation. *Fifth Generation Computer Architectures*, J. V. Woods (editor), Elsevier Science Publishers B. V. (1986) 147–160
19. Takesau, M.: Cache Memories for Dataflow Machines. *IEEE Transactions on Computers*, Vol. 41, No. 6. (1988) 677–687

Asynchronous Timed Multimedia Environments Based on the Coordination Paradigm

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, CY-1678, P.O. Box 537, CYPRUS
george@cs.ucy.ac.cy

Abstract. This paper combines work done in the areas of Artificial Intelligence, Multimedia Systems and Coordination Programming to derive a framework for Distributed Multimedia Systems based on asynchronous timed computations expressed in a certain coordination formalism. More to the point, we propose the development of multimedia programming frameworks based on the declarative logic programming setting and in particular the framework of object-oriented timed concurrent constraint programming (OO-TCCP). The real-time extensions that have been proposed for the concurrent constraint programming framework are coupled with the object-oriented and inheritance mechanisms that have been developed for logic programs yielding an integrated declarative environment for multimedia objects modelling, composition and synchronisation. Furthermore, we show how the framework can be implemented in the general purpose coordination language Manifold, without the need for using special architectures or real-time languages.

Keywords: Multimedia; Timed Concurrent Constraint Programming; Timed Asynchronous Languages; Coordination; Distributed Computing.

1 Introduction

The development of distributed multimedia frameworks is a quite common phenomenon in our days. Furthermore, any distributed programming environment can be viewed as being comprised by two separate components: a computational part consisting of a number of concurrently executing processes and responsible for performing the actual work, and a communication/coordination part which is responsible for inter-process communication and overall coordination of the executing activities. This has led to the development of the so called *family of coordination models and languages* ([3, 12]) which can be used to support the coordinated distributed execution of a number of concurrently executing agents.

The purpose of this paper is to present a framework for coordinating the distributed execution of multimedia applications exhibiting real-time behaviour. However, unlike most of the other approaches that are primarily based on using special purpose real-time languages and platforms ([2, 6, 7, 8, 13]), our model is

based on declarative programming and, in particular, that of concurrent constraint programming. More to the point, we show how the *timed* version of concurrent constraint programming ([15]), combined with already existing techniques supporting object-oriented programming ([5]), can be used to produce a framework for multimedia programming which we name OO-TCCP. We then show how a general purpose coordination formalism, namely Manifold ([1]), can be used to support the run-time environment that satisfies the real-time execution requirements of OO-TCCP agents, thus effectively presenting an implementation of OO-TCCP in Manifold, or in other words, a coordination formalism for distributed multimedia applications.

The rest of the paper is organised as follows: The next section presents OO-TCCP and shows how it can be used as the basis for multimedia programming. The following section describes briefly the coordination language Manifold and shows how OO-TCCP can be implemented in it. The last section concludes the paper with a discussion of current and future work.

2 A Declarative Object-Oriented Real-Time Multimedia Programming Framework

Timed concurrent constraint programming (TCCP), developed by Saraswat *et al.* ([15]), is an extension of concurrent constraint programming, itself being a combination of constraint logic programming and concurrent logic programming, with temporal capabilities along the lines of state-of-the-art real-time languages such as ESTEREL, LUSTRE and SIGNAL ([2, 8]), offering temporal constructs and interrupts, and suitable for modelling real-time systems. In TCCP variables play the role of *signals* whose values from one time instance to another can be different. At any given instance in time the system is able to detect the presence of any signals; however, the absence of some signal can be detected only at the end of the time interval and any reaction of the system will take place at the *next* time interval. Thus, the behaviour of a process is influenced by the set of positive information input up to and including some time interval t and the set of negative information input up to but not including t . This has been called the *timed asynchrony hypothesis* ([15]) and contrasts the perfect synchrony hypothesis, usually advocated by real-time languages ([6]). These time intervals t at the end of which no more positive information can be detected are termed the *quiescent* points of the computation.

Thus, the fundamental differences between the timed and the untimed version of concurrent constraint programming are that in the timed version: (i) recursion (and iteration for that matter) are eliminated, and (ii) no information is carried over (by means of variables) from one time instance to the next one. These restrictions guarantee bounded time response and hence a real-time behaviour. Note that the basic ideas characterising TCCP are not unique to concurrent constraint programming and in fact could be introduced into any asynchronous model of computation. It is precisely this property that we exploit in the next section to derive an implementation of the model in terms of a general purpose coordination formalism.

If c is a constraint and A and B are agents, the fundamental temporal construct in TCCP is the following combination:

```
now c then A else B
```

whose interpretation is as follows: if there is enough positive information to entail the constraint *c* then the process reduces immediately (in the current time interval) to *A* and the operations further performed by *A* are also observable immediately; otherwise, if at the end of the current time interval the store cannot entail *c* (i.e. negative information, or in other words, the absence of some signal has been detected), the process reduces to *B* *at the next* time interval (the work performed by *B* will not be observable in the current time instance). As implied by the syntax for the agents above, either of the *then* or *else* parts can be omitted. By „guarding“ recursion within an *else* (or *next*) part it can be guaranteed that computation within a time interval is bounded. In fact, reachable states of the computation in a TCCP program can be identified at compile time leading to the generation of a finite state automaton in the same way that this is possible for state-of-the-art real-time languages ([2]). Note that when moving from one time interval to another all the positive information accumulated within the current time interval are discarded. Thus, the value of a program's „variable“ varies at different time intervals and any data must either be kept as arguments to the relative predicate or be posted as signals at every time interval.

To recapitulate, at any moment in time a number of agents are executed concurrently exchanging information by means of posting signals to a, possibly only notionally, common store. Each agent is allowed to either suspend waiting for some signal to be posted from some other concurrently running agent, or post itself signal(s) and/or spawn other agents. Any (mutually) recursive call will have to wait until the next time instance. Thus, each (loop-free) agent performs only a bounded amount of work and eventually the whole system quiescences. The store is discarded and computation moves on to the next time instance where only those agents present in the *else* and *next* constructs are executed (any agent still remaining suspended in the current time instance is also discarded).

As shown in [15], the above construct can be used to implement a number of temporal constructs that are usually found in real-time languages such as ESTEREL, LUSTRE and SIGNAL. In the sequel we show only the basic ones. The construct

```
whenever c do A = now c then A else (whenever c do A)
```

suspends until the constraint *c* can be entailed and then reduces the executing process to *A*, thus modelling a temporal *wait* construct. Alternatively, the construct

```
always A = A, next (always A)
```

defines a process that behaves like *A* at every time instance.

Timeouts and interrupts in TCCP can be handled by a *do...watching* construct similar to that found in languages like ESTEREL but with a slightly different semantics. In particular,

```
do A watching c timeout B
```

executes A and if c becomes true before A completes execution, the process will reduce to B at the *next* time instance. Since (agent) A can be a number of things, the above construct is actually defined by a set of rules rather than a single one, the most important of which are the following ones.

The OO-TCCP framework can be used as the basis for developing multimedia programming environments based on the timed asynchronous paradigm, i.e. frameworks that essentially exhibit soft real-time behaviour; such a framework is reported in [9]. Here, we show how we can model time-based media in OO-TCCP.

```
time_media_object(Object)+(QualityFactor,Duration,Encoding,Rate,ScFactor=1,...) :-
    whenever Object:access
        do (now Object:setScFactor:X then
            (ScFactor'=X, next self),
            now Object:setDuration:Length then
            (Duration'=Length, next self),
            etc. for the rest of the set function
primitives
        now Object:getRate then
        ({Object:rate:Rate}, next self),
        now Object:getEncoding then
        ({Object:encoding:Encoding}, next self),
        etc. for the rest of the get function
primitives
    ).
```

The above code defines a time-based media object comprising a name, which plays effectively the role of a communication channel, and a set of attributes such as quality factor (eg. VHS or CD depending on whether it is video, sound, etc.), duration (in, say, seconds) and rate of presentation (in frames for video or samples for audio). Note that all the attributes are defined as implicit arguments; note also that the scaling factor has a default value of 1. The main part of the code defines its interface where we note that the object remains suspended until it receives an initial message `Object:access`; upon receiving such a message `time_media_object` expects the presence of an accompanying message which can belong to either of two categories: i) it can be an updating type of message (implementing effectively the `set` type of primitive functions) in which case it updates the relevant parameter (and calls itself recursively at the *next* time instance), or ii) it can be a request type of message (implementing effectively the `get` type of primitive functions) in which case it posts a signal with the value(s) of the requested parameter(s). Note that the accompanying message may be a parameterised one carrying complicated information that should be passed on by the object to some other agent (e.g. some device driver) for processing. We do not explore this scenario any further here.

We can use the above object class to define a video and an audio object subclass as follows.

```
video_object(Video)+(QualityFactor=„VHS“,Duration,Encoding,Rate,ScFactor,Colour,...) :-
    + time_media_object;
```

```

        whenever Video:access
            do (now Video:setColour:C then (Colour'=C,
next self),
            etc. for the rest of the set function
primitives particular to this object
            now Video:getColour then
({Video:colour:Colour}, next self),
            etc. for the rest of the get function
primitives particular to this object
            now Video:play
            then (video_device_PLAY(Video,...), next
self),
            now Video:stop
            then (video_device_STOP(Video), next
self)
        ).

audio_object(Audio)+(QualityFactor=„CD“,Duration,Encodi
ng,Rate,ScFactor,Volume,...) :-
    + time_media_object;
    whenever Audio:access
        do (now Audio:setVolume:V then (Volume'=V,
next self),
        etc. for the rest of the set function
primitives particular to this object
        now Audio:getVolume then
({Audio:volume:Volume}, next self),
        etc. for the rest of the get function
primitives particular to this object
        now Audio:play then
(audio_device_PLAY(Audio,...), next self),
        now Audio:stop then
(audio_device_STOP(Audio), next self)
        possibly other control signals
particular to this object
    ).

```

Note that both objects inherit the methods handling the common signals of their superclass. Note also that there is a third category of messages, that of control messages (such as START or STOP) in which case the appropriate device is accessed.

3 The Coordination Language Manifold

Manifold ([1]) is a control-driven coordination language. In Manifold there are two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is

completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. A Manifold process features *ports*, *states* and *events*. All these notions are directly derived from the equivalent constructs of Manifold's underlying coordination model, namely IWIM, which is briefly described in the relevant section of another paper by the author in this proceedings volume.

Figure 1 below shows diagrammatically the infrastructure of a Manifold process. The process *p* has two input ports (*in1*, *in2*) and an output one (*out*). Two input streams (*s1*, *s2*) are connected to *in1* and another one (*s3*) to *in2* delivering input data to *p*. Furthermore, *p* itself produces data which via the *out* port are replicated to all outgoing streams (*s4*, *s5*). Finally, *p* observes the occurrence of the events *e1* and *e2* while it can itself raise the events *e3* and *e4*. Note that *p* need not know anything else about the environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

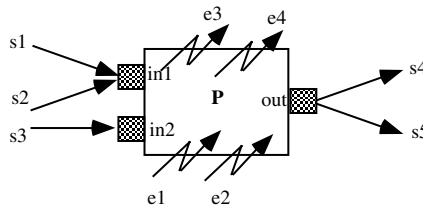


Fig. 1. The basic infrastructure of a Manifold process

The following is a Manifold program computing the Fibonacci series.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event) port in x. port in y. import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
  begin:(v0->sigma.x, v1->sigma.y,v1->v0,
        sigma->v1,sigma->print).
  overflow.sigma:halt.
}
```

The above code defines *sigma* as an instance of some predefined process *sum* with two input ports (*x*,*y*) and a default output one. The main part of the program sets up the network where the initial values (0,1) are fed into the network by means of two „variables“ (*v0*,*v1*). The continuous generation of the series is realised by

feeding the output of `sigma` back to itself via `v0` and `v1`. Note that in *Manifold* there are no variables (or constants for that matter) as such. A *Manifold* variable is a rather simple process that forwards whatever input it receives via its input port to all streams connected to its output port. A variable „assignment“ is realised by feeding the contents of an output port into its input. Note also that computation will end when the event `overflow` is raised by `sigma`. `Main` will then get preempted from its `begin` state and make a transition to the `overflow` state and subsequently terminate by executing `halt`. Preemption of `Main` from its `begin` state causes the breaking of the stream connections; the processes involved in the network will then detect the breaking of their incoming streams and will also terminate.

Manifold contrasts with the „Linda-like“ family of *data-driven* coordination models and languages, where computational components intermix with coordination ones, and coordinator agents see and examine the data involved in some computation. In *Manifold* all agents are treated as black boxes and there is no concern as to what they actually compute, or indeed whether they are software processes or hardware devices. Thus, this formalism is ideal for coordinating distributed Multimedia frameworks. However, the basic *Manifold* system does not support real-time behaviour. We show below how the OO-TCCP abstract machine can be implemented in *Manifold*.

4 Implementing the OO-TCCP Abstract Machine in *Manifold*

Although *Manifold*'s features were designed with other purposes in mind, we have found them to be suitable in implementing the run-time environment required by OO-TCCP. In particular, a *Manifold* configuration exhibiting real-time behaviour in the OO-TCCP sense consists of the following components:

- A *Manifold* coordinator process (the clock) responsible for monitoring the status of the coordinated processes, detecting the end of the current time instance, and triggering the next one. The coordinator process is also responsible for detecting the end of the computation.
- A set of *Manifold* coordinated processes, each one monitoring the execution of some group of atomic processes. Each such coordinated process performs a bounded amount of work between the ticks as dictated by the coordinator process (thus any loops in such a process „spread over“ the next one or more ticks).
- A set of groups of atomic processes (i.e., processes written in some language other than *Manifold*), each group being monitored by a coordinated process. In order for the whole configuration to exhibit asynchronous real-time behaviour, these atomic processes must also produce results in bounded time. There are two approaches possible here: (i) enforce the constraint that there are no loops within these processes and instead, put these loops in their respective coordinated processes, or (ii) treat them as asynchronous parallel components that take an unbounded amount of time.

The overall configuration is a hierarchical one with the *Manifold* coordinator process on the top, monitoring a number of *Manifold* coordinated processes, themselves possibly monitoring groups of atomic (non *Manifold*) processes. One can

regard Manifold as being the „host language“ for writing the control structures of reactive systems, while most of the actual computation (data handling, interfaces with any embedded systems) are done in other more conventional languages, typically C. This fits nicely into the spirit of real-time coordination models as we perceive them and separates the real-time coordination requirements from the rest of the performed activities.

An application featuring timed asynchronous behaviour takes the general form:

```
<application> ::= <coordinator>
                  <coordinated>+ <atomic>+
```

The general behaviour of a coordinator (clock) process is shown, as a first approximation, below (note that the construct $(A_1, \dots, A_n)$ denotes a block where all n activities will be executed concurrently; there is also a ‘;’ separator imposing sequentiality).

```
manifold Clock()
port in term_in, next_in. port out term_out, next_out.
{
  event tick, next_phase, end_comp.
  begin: (<set up network of initial processes>;
         terminated(void)).
  next_phase: (raise(tick), terminated(void)).
  end_comp: (<perform clean up>; post(end)).
}
```

Clock first sets up the initial network of coordinated processes. It then suspends waiting for either of the following two cases to become true (one way to achieve suspension in Manifold is by waiting for the termination of the special process `void` which actually never terminates):

- The coordinated processes have completed execution within the current time instance and are waiting for the next clock tick. `Clock` posts the appropriate event (or signal) and suspends again.
- The computation has terminated in which case `Clock` terminates, possibly after performing some clean up.

Detecting the completion of both the current phase and the end of the computation is done in a distributed fashion, provided some constraints regarding the organisation and communication protocols between the participating coordinated processes are imposed. We elaborate further on the exact nature of the work done by `Clock` once we describe the activities performed by a coordinated process.

The general behaviour of a coordinated process is as follows:

```
manifold Process()
port in term_in, next_in.
port out term_out, next_out.
{
  begin: (<raise event>;
         <wait until input event received>)
```

```

input_event: (<perform data transfer p1->p2>,
             <generate new processes>;
             terminated(void)).

tick.Clock: (<perform further actions>;
            post(end)).
}

```

A typical behaviour of a timed asynchronous coordinated process, as understood in the Manifold world, is to post some events, possibly wait until the presence of some event in the current time instance is detected and then react by producing some data transfer between a group of atomic processes that it itself coordinates (say from p_1 to p_2), post more events and/or generate further processes. Upon termination of its activities within the current time instance, the process suspends waiting for the next `tick` event from the coordinator (`Clock`) process, in which case it performs more activities of similar nature or simply terminates within the current time instance.

We now present in more detail the way detection of the end of the current phase, as well as the whole computation, is achieved. Due to space limitations only the most essential parts of the Manifold code are shown below. The techniques we are using are reminiscent of the ones usually encountered within the concurrent constraint programming community based on short circuits. We recall that the coordinator and each one of the coordinated processes have (among others) two pairs of ports: the `term_in/term_out` pair is used to detect termination of the whole computation whereas, the `next_in/next_out` pair is used to detect termination of the current clock phase. Upon commencing the computation, the `Clock` process sets up a configuration like the one shown in figure 2 below. This is achieved by means of the following Manifold constructs:

```

(C.next_out->P1.next_in,...,P3.next_out->C.next_in)
(C.term_out->P1.term_in,...,P3.term_out->C.term_in)

```

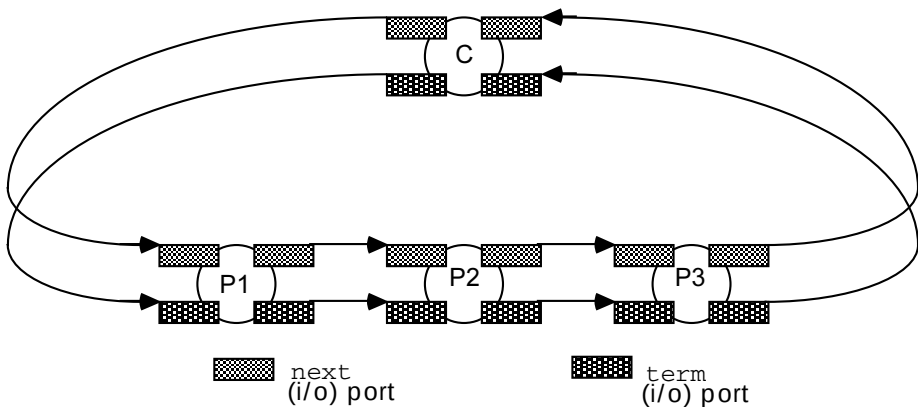


Fig. 2. A short circuit of inter-connecting processes

Any process wishing to further generate other processes is also responsible for setting up the appropriate port connections between these newly created processes. Detecting termination of the whole computation is done as follows: a process *P* wishing to terminate, first redirects the stream connections of its input and output *term* ports so that its left process actually bypasses *P*. It also sends a message down the *term.in* port of its right process. If *P*'s right process is another coordinated process the message is ignored; however, if it happens to be the *Clock* controller, the latter sends another message down its *term.out* port to its left process. It then suspends waiting for either the message to reappear on its *term.in* port (in which case no other coordinated process is active and computation has terminated) or a notification from its left coordinated process (which signifies that there are still active coordinated processes in the network). The basic Manifold code realising the above scenario for the benefit of the *Clock* controller is shown below.

```
begin:    guard(term_in,transport,check_term).
check_term: ("token" -> term_out, post(begin)).
got_token: post(begin).
check_term: post(end).
```

A guard process is set up to monitor activity in the *term.in* port. Upon receiving some input in this port, guard posts the event *check_term*, thus activating *Clock* which then sends *token* down its *term.out* port waiting to get either a *got_token* message from some coordinated process or have *token* reappear again. The related code for a coordinated process is as follows:

```
begin:    guard(term_in,transport,check_term).
check_term: (<term_in->void, if data in port is "token"
             raise(got_token)>).
```

Detecting the end of the current time instance is a bit more complicated. Essentially, quiescence, as opposed to termination, is a state where there are still some processes suspended waiting for events that cannot be generated within the current time instance. We have developed two methods that can detect quiescent points in the computation. In the first scheme, all coordinated processes are connected to a *Clock* process by means of reconnectable streams between designated ports. A process that has terminated its activities within the current time instance breaks the stream connection with *Clock* whereas a process wishing to suspend waiting for an event *e* first raises the complementary event *i_want_e*. Provided that processes wishing to suspend but also able to raise any events for the benefit of other processes, do so before suspending, quiescence is the point where the set of processes still connected to *Clock* is the same as the set of processes that have raised *i_want_e* events. The advantage of this scheme is that processes can raise events arbitrarily without any concern about them being received by some other process. The disadvantage however is that it is essentially a centralised scheme, also needing a good deal of run-time work in order to keep track of the posted events.

An alternative approach requiring less work that is also distributable is a modification of the protocol used to detect termination of the computation: a process wishing to suspend waiting for an event performs the same activities as if it were

about to terminate (i.e. have itself bypassed in the port connections chain) but this time using the `next` input/output ports. A process wishing to raise an event before suspending (or terminating for that matter) does so, but waits for a confirmation that the event has been received before proceeding to suspend (or terminate). A process being activated because of the arrival of an event, adds itself back into the next ports chain. Quiescence now is the point where the `Clock` detects, as before, that its `next.out` port is effectively connected to its own `next.in` port, signifying that no event producer processes are active within the current time instance. Note that unlike the case for detecting termination, here the short circuit chain can shrink and expand arbitrarily. Nevertheless, it will eventually shrink completely provided that the following constraints on raising events are imposed:

- Every raised event must be received within the current time instance so that no events remain in transit. An event multicast to more than one process must be acknowledged by all receiver processes whose number must be known to the process raising the event; this latter process will then wait for a confirmation from all the receiver processes before proceeding any further.
- A process must perform its activities (where applicable) in the following order: 1) raise any events, 2) spawn any new processes and set up the next and term port connections appropriately, 3) suspend waiting for confirmation of raised events, 4) repeat the procedure.

The code for the `Clock` controller is very similar to the one managing the `term` ports, with the major difference that upon detecting the end of the current phase `Clock` raises the event `tick`, thus reactivating those coordinated processes waiting to start the activities of the next time instance.

```
begin:    guard(next_in,transport,check_term).
check_term: ("token" -> next_out, post(begin)).
got_token: post(begin).
check_term: (raise(tick), post(begin)).
```

The code for a coordinated process is as follows:

```
some_state: { begin: (raise(e),
                    <possibly spawn other processes>;
                    terminated(void)).
              i_got_e:
            }
<continue>
```

The framework presented above can be used to implement the OO-TCCP primitives and, thus, provide a Manifold-based implementation for OO-TCCP. We show below the implementation of three very often used such primitives:

```
manifold Whenever_Do(event e, process p)
{
  begin:    terminated(void).
  e:        activate(p).
  tick.Clock: { ignore *.
```

```

        begin: post(begin) .
    } .
}

manifold Always(process p)
{
    begin: (activate(p), terminated(void)) .
    tick.Clock: post(begin) .
}
manifold Do_Watching(process p, event e)
{
    begin: (activate(p), terminated(void)) .
    e: {      begin: terminated(void) .
           tick.Clock: raise(abort) . } .
           tick.Clock: terminated(void) .
}

```

Note that `ignore *` clears the event memory of the manifold executing this command. By using `ignore a` „recursive“ manifold can go to the next time instance without carrying with it events raised in the previous time instance.

5 Conclusions; Related and Further Work

We have presented an alternative (declarative) approach to the issue of developing multimedia programming frameworks, that of using object-oriented timed concurrent constraint programming. The advantages for using OO-TCCP in the field of multimedia development are, among others, the use of a declarative style of programming, exploitation of programming and implementation techniques that have developed over the years, and possible use of suitable constraint solvers that will assist the programmer in defining inter and intra spatio-temporal object relations. Furthermore, we have shown how this framework can be implemented in a general purpose coordination language such as Manifold in ways that do not require the use of specialised architectures or real-time languages.

Our approach contrasts with the cases where specialised software and/or hardware platforms are used for developing multimedia frameworks ([2, 7, 13]), and it is similar in nature to the philosophy of real-time coordination as it is presented, for instance, in [4, 14]. We believe our model is sufficient for *soft* real-time Multimedia systems where the *Quality of Service requirements* impose only soft real-time deadlines.

References

1. F. Arbab, I Herman and P. Spilling: An Overview of Manifold and its Implementation, Concurrency: Practice and Experience, Vol. 5, No. 1 (1993), 23–70
2. G. Berry: Real-Time Programming: General Purpose or Special Purpose Languages, Information Processing '89, G. Ritter (ed.), Elsevier Science Publishers, North Holland (1989), 11–17

3. N. Carriero and D. Gelernter: Coordination Languages and their Significance, *Communications of the ACM* **35**(2) (Feb. 1992), 97–107
4. S. Frolund and G. A. Agha: A Language Framework for Multi-Object Coordination, ECOOP'93, Kaiserslautern, Germany, LNCS 707, Springer Verlag, (July 1993), 346–360
5. Y. Goldberg, W. Silverman and E. Y. Shapiro: Logic Programs with Inheritance, FGCS'92, Tokyo, Japan, Vol. 2 (June 1-5 1992), 951–960
6. N. Halbwachs: Synchronous Programming of Reactive Systems, Kluwer (1993)
7. F. Horn, J. B. Stefani: On Programming and Supporting Multimedia Object Synchronization, *The Computer Journal*, Vol. 36, No 1. (1993), 4–18
8. IEEE Inc. Another Look at Real-Time Programming, Special Section of the *Proceedings of the IEEE* **79**(9) (September 1991)
9. G. A. Papadopoulos: A Multimedia Programming Model Based On Timed Concurrent Constraint Programming, *International Journal of Computer Systems Science and Engineering*, CRL Publs., Vol. 13 (4) (1998), 125–133
10. G. A. Papadopoulos: Distributed and Parallel Systems Engineering in Manifold, Parallel Computing, Elsevier Science, special issue on Coordination, Vol. 24 (7) (1998), 1107–1135
11. G. A. Papadopoulos, F. Arbab: Coordination of Systems With Real-Time Properties in Manifold, Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96), Seoul, Korea, 19–23 August, IEEE Press (1996), 50–55
12. G. A. Papadopoulos, F. Arbab: Coordination Models and Languages, *Advances in Computers*, Academic Press, Vol. 46 (August 1998), 329–400.
13. M. Papathomas, G. S. Blair, G. Coulson: A Model for Active Object Coordination and its Use for Distributed Multimedia Applications, Object-Based Models and Languages for Concurrent Systems, Bologna, Italy, LNCS 924, Springer Verlag (July 5, 1994), 162–175
14. S. Ren, G. A. Agha: RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, La Jolla, California (June 21–22 1995)
15. V. A. Saraswat, R. Jagadeesan, V. Gupta: Programming in Timed Concurrent Constraint Languages, *Constraint Programming*, B. Mayoh, E. Tyugu and J. Penjam (eds.), NATO Advanced Science Institute Series, Series F: Computer and System Sciences, LNCS, Springer Verlag (1994)

Component-Based Development of Dynamic Workflow Systems Using the Coordination Paradigm

George A. Papadopoulos and George Fakas

Department of Computer Science
University of Cyprus
75 Kallipoleos Street, P.O. Box 20537, CY-1678, Nicosia, CYPRUS
{george, fakas}@cs.ucy.ac.cy

Abstract. We argue for the need to use control-based, event-driven and state-defined coordination models and associated languages in modelling and automating business processes (workflows). We propose a two-level architecture of a hierarchical workflow management system modelled and developed in such a state-of-the-art coordination language. The main advantage of a hierarchical, coordination-based architecture is that individual workflow entities can be easily replaced with others, without disrupting the overall workflow process. Each individual workflow entity exhibits a certain degree of flexibility and autonomy. This makes possible the construction of workflow systems that bring further improvements to process automation and dynamic management, such as dynamic (re-) allocation of activities to actors, reusability of coordination (collaboration) patterns, etc. A case study is presented to demonstrate the use of our approach.

Keywords: Component-Based Systems; Coordination Models and Languages; Workflow Systems; Dynamic (Re-) Configurable Systems; Collaborative Environments.

1 Introduction

Workflow management is concerned with the coordination of the work undertaken by a number of parties. It is usually applied in situations where processes are carried out by many people, possibly distributed over different locations. A workflow application automates the sequence of actions and activities used to run the processes. Such an ensemble of cooperative distributed business processes requires coordination among a set of heterogeneous, asynchronous, and distributed activities according to given specifications.

Therefore, it is not surprising that a number of researchers have proposed workflow models, where the notion of coordination plays a central role in the functionality of their frameworks. Typical examples are DCWPL ([7]), a coordination language for collaborative applications, ML-DEWS ([8]), a modelling language to support dynamic evolution within workflow systems, Endeavors ([10]), a workflow support system for exceptions and dynamic evolution, OPENflow ([20]), a CORBA-based workflow environment, and the framework proposed in [11]. A notable

common denominator in all these proposals is the fact that they take seriously issues of dynamic evolution and reconfiguration. Interestingly, another notable common denominator is the fact that the line of research they pursue seems to be quite independent from similar research pursued in Component-Based Software Engineering (CBSE), particularly within the subfield of coordination. It is precisely this relationship between coordination in CBSE and workflow systems that we explore in this paper.

More to the point, we have seen a proliferation of the so-called *coordination models* and associated programming languages ([17]). Coordination programming provides a new perspective in constructing software programs. Instead of developing a software program from scratch, the coordination model allows the gluing together of existing components. Whereas in ordinary programming languages a programmer describes individual computing components, in a coordination language the programmer describes interrelationships between collaborating but otherwise independent components. These components may even be written in different programming languages or run on heterogeneous architectures.

Coordination as a science of its own whose role goes beyond software composition, has also been proposed ([11, 12]). However, using the notion of coordination models and languages in modelling workflows, the so-called *coordination language-based approach to groupware construction* ([6]), is a rather recent area of research. Using such a coordination model and language has some clear advantages, i.e. work can be decomposed into smaller steps which can be assigned to and performed by various people and tools, execution of steps can be coordinated (e.g. in time), and coordination patterns that have proved successful for some specific scenario can be reused in other similar situations. Furthermore, this approach offers inherent support for reuse, encapsulation and openness, distribution and heterogeneous execution. Finally, the coordination model offers a concrete modelling framework coupled with a real language in which we can effectively compose *executable specifications* of our coordination patterns.

The rest of the paper is organised as follows. In the next section we present a specific coordination model and associated language, namely IWIM and Manifold. This is followed by the presentation of a hierarchical workflow coordination architecture, where we show how this can be used as the main paradigm for modelling workflow activities. We then validate the proposed architecture by using a case study. We end with some conclusions and description of related and further work.

2 The Coordination Model IWIM and the Manifold Language

In this section we describe a framework for modelling workflows in the coordination language Manifold (and its underlying coordination model IWIM). As will be explained in the next section, Manifold plays the role of the execution environment for the workflow model presented there. The IWIM model ([3]) belongs to the class of the so-called control-oriented or event-driven coordination models. It features a hierarchy of processes, playing the role of either computational processes or coordinator processes, the former group performing collectively some computational

activity in a manner prescribed by the latter group. Both types of processes are treated by the model as black boxes, without any knowledge as to the constituent parts of each process or what precisely it does. Processes communicate by means of well-defined input-output interfaces connected together by means of streams.

Manifold is a direct realisation of IWIM. In Manifold there exist two different types of entities: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manifold possess the following characteristics:

- *Processes*. A process is a *black box* with well defined *ports* of connection through which it exchanges *units* of information with the rest of the world.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives.
- *Streams*. These are the means by which interconnections between the ports of processes are realised.
- *Events*. Events are broadcast by their sources in the environment, yielding *event occurrences*.

Activity in a Manifold configuration is *event driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. It then remains in that state until it observes the occurrence of some other event which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities.

More information on IWIM and Manifold can be found in [3, 5, 15, 16, 17] and another paper by the first author in this proceedings volume.

3 A Hierarchical Workflow Coordination Architecture

The motivation behind our approach lies in the observation made in [10] that „traditional approaches to handling [problems related to the dynamic evolution of workflow systems] have fallen short, providing little support for change, particularly once the process has begun execution“. Intelligent process management is a key requirement for workflow tools. This is catered for in our approach as agents of the underlying coordination model are able to manage themselves. In particular, workflow processes are modelled and developed in a number of predefined interrelated entities which together form a meta-model i.e. process, activity, role, and actor. We propose a hierarchical architecture where the individual workflow entities can be easily replaced with others, without disrupting the overall workflow process.

Each individual workflow entity exhibits a certain degree of flexibility and autonomy. This makes possible the construction of workflow systems that bring further improvements in process automation and dynamic management, for example dynamic (re-) allocation of activities to actors. In that respect, we advocate the approach proposed in [8] which involves a two-level hierarchy: the upper level is the specification environment which serves to define procedures and activities, whereas the lower level is the execution environment which assists in coordinating and performing those procedures and activities. In this section we describe the top level (itself consisting of a number of sublayers), whereas in section 4 we show how it can be mapped to the lower (execution) level, realized by the coordination language Manifold. Figure 1 below visualises the layered co-ordination workflow architecture. Agents of each layer utilise (trigger) agents from the layer below. The hierarchical nature of the architecture allows flexible workflow systems to be designed in a modular way.

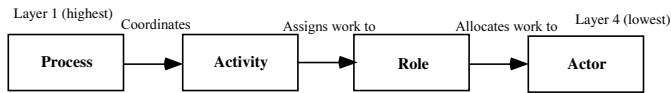


Fig. 1. A Hierarchical Workflow Management Architecture

3.1 Process

A process is a collection of coordinated activities that have explicit and/or implicit relationships among themselves in support of a specific process objective. A process is responsible for coordinating the execution of activities. Its main functionality therefore is to manage, assist, monitor and route the workflow. Process objects are able to manage the execution of the workflow:

- **Via alerting using deadlines.** A deadline is assigned for every activity. If the activity is not completed before the deadline, the process is responsible to send an alert message to the activity.
- **By prioritising.** Every activity is characterised by a priority level relative to other activities. This knowledge is used by the Process object for more efficient task allocation and scheduling.
- **By real-time monitoring.** The process keeps track of parameters related to its execution such as Total Running Time, Current Activity and its status (Waiting Time, Deadline, Role and Actor Selected), etc. This information is useful to trace any bottlenecks in the process.
- **By estimating the time and resources required for execution.** The process is capable of estimating the total duration of the execution and the resources required. It achieves this by interrogating the activity objects, which in turn may query role objects and so on.

The following table summarizes the events that trigger a process and its states.

Process	
Event	State
Start process	Triggers the process activities. Process is responsible for coordinating activities and the sequence and rules of activities execution.
Process administrator examines process status	Process reports current state; i.e. Total Running Time, Current Activity and its status (Waiting Time, Deadline, Role and Actor Selected), etc.

3.2 Activity

An activity is a single step within a process definition that contributes to the achievement of the objective. It represents the smallest grain of abstracted work that can be defined within the workflow management process. Every activity is related to a role (which is going to perform the work) and to in/out data. An activity instance monitors the execution of the work over time by maintaining information about the activity such as: deadline, priority, estimated waiting time or execution time. The following table summarizes the events that trigger the activities and their states.

Activity	
Event	State
Process triggers activity	Receives in in-tray activity input and then assigns the work to the relevant role; then waits until activity deadline expires or executed.
Actor executes activity	Finished, put output in out-tray.
Activity deadline expires	Every activity is associated with a deadline; when this expires the activity asks the corresponding role to examine the actors workload and take the appropriate actions.

3.3 Role

It is important to define roles independently of the actors who carry out the activities, as this enhances the flexibility of the system. Roles assign activities to actors. If an actor is unavailable (e.g. an employee is ill) then somebody else is chosen to carry out the activity. Role objects have the following features and responsibilities:

Allocation of activities to actors. It is the role’s responsibility to allocate activities to actors. Its aim is to make an optimized allocation of work which is dynamic by taking into account parameters such as:

- **The actor’s level of experience.** Actors have different levels of experience (novice, expert or guru) in performing an activity. Typically, an activity will be allocated to actors with the highest level of expertise available.

- **The actor's workload.** Actors with a heavy workload are less preferable when activities are allocated by roles.
- **Allocation by role-base reference.** In the case of process loops, roles can allocate iterated activities either to the same actor or to a different one.

Report Actors Overload. The role examines the actors' workload and if none of the actors are able to execute the activity before its deadline because they are overloaded, then the role notifies the activity.

If the role discovers an actor that will not be able to execute any of the activities allocated to it before their deadlines then the role might try to *reallocate* the work. For reallocation of work, the same criteria are used (i.e. taking into account the actor's level of experience, workload, use of role-based references, etc.).

The following table summarizes the events that trigger the roles and their states.

Role	
Event	State
Activity assigns work or deadline expires	Role checks its actors' workloads. If none of the actors is able to execute the current activity before its deadline because they are overloaded then the role deals with overload.
Role assigns work to actor	Receives in in-tray activity input and then assigns the work (and associated input) to an actor according to some criteria: actor's level of experience, actor's workload and role-based reference, and then waits until work is executed or reassigned to another actor.
Deadline expires and actors are not overloaded	The role is checking up whether it is preferable to reassign the activity to a different actor less busy to perform it or just alert the user responsible for it.
Role reassigns work to a different actor	The role reallocates those activities to other actors. Reallocation of work considers the same criteria as initial allocation of work does. When finished, put output in out-tray.
Role alerts actor	The role alerts the actor responsible for performing the activity.
Actors are overloaded	Deal with actors' overload by either extending the activity's deadline, allocating more actors to the process, or changing the activities' priorities

3.4 Actor

An actor can be either a person or piece of machinery (software application, etc.). Actors can perform and are responsible for activities. Actor workflow objects have the capability to schedule their activities. Activity scheduling is done using policies such as the earliest due job is done first, the shortest job is done first, etc.

The following table summarizes the events that trigger the actors and their states.

Actor	
Event	State
Role assigns a work	Receives work in in-tray
Actor schedules his work	The way the actor schedules his work i.e.: FIFO, Shorter First and etc.
Executes work	Executes work and puts output in out-tray
Reports overload	The actor can manually report overload and then the corresponding role will try to solve it

4 A Case Study

The expenses claim process has been used to validate our approach. It is a very common administrative process where an employee is claiming his/her expenses back from the company. The employee fills in a claim form and then sends it to an authorized person for approval. An authorized person could be the head of the department's secretary. In case where the amount claimed is over 1,000 pounds, it must be approved by the head of the department. If the authorized person does not approve the employee's claim, then (s)he sends a rejection message back to the employee; otherwise (s)he sends a message to the company's cashier office to issue a cheque. Finally, the cashier issues and sends a cheque to the employee.

The following table shows how the above scenario is modelled in IWIM. The Expenses Claim Process is a manager entity and the rest are worker ones.

Expenses Claim Workflow Process		
Activity	Role	Actor
Claim (employee)	Employee	Actor AP1
Approve (Authorized Person)	Authorized Person	Actor HP1
Approve (Head of Dept)	Head of Dept	Actor C1
Pay (Cashiers)	Cashiers	Actor C2

The following coding shows the process logic that contains the process activities and is activated when the process starts. We use a user-friendly pseudo-Manifold coding which is more readable and dispenses us with the need to provide a detailed description of how we program in this language, something we would rather avoid due to lack of space. This pseudo-code however is directly translatable to the language used by the Manifold compiler. Every time a user wishes to start a claim process, an instance of the process and its activities are constructed. When the user finishes with the `putClaim()` activity then the next activity will be called. Assuming that the claim is less than 1000 pounds then the `approve()` activity by the authorized person is called. Then `authorisedPerson` role is assigning the work to an actor. The role, before assigning the work, examines all actors workload (i.e. checks whether any actor can perform the activity before its deadline). If all the actors of a role are overloaded and are not able to perform extra work, then the role has to deal with the actors overload (`DealWithActorsOverload` state) and solve

the overload problem either by extending the activity's deadline or by allocating more workers to the process; otherwise, the role assigns the work to an actor. The activity is in a waiting state until either the actor assigned the work performs it or the activity's deadline expires. If the activity deadline expires before the actor performs it, then the role examines again whether to reassign the work to a different actor or just send an alert message.

Eventually, when the activity is executed, the process proceeds to the next activity, i.e. cashier issues `Cheque()` (if authorised person approves payment). Again, all these activity actions are taken dynamically to manage the process execution.

Manifold Process (port in, port out) .

Manifold Activity (port in, port in, port out, port out) .

Manifold Role (port in, port in, port out, port out) .

Manifold Actors (port in, port out) .

Manifold ClaimForm, ApproveForm, PaySlip.

Manifold main

```
{
  event processMonitoring, assignActivityToRole,
  deadlineExpires.
  auto process ClaimExpenses is Process.
  auto process startClaim, ApproveAuthPer,
    ApproveHeadDep, Pay is Activity.
  auto process Employee, AuthorisedPerson,
    HeadOfDept, Cashiers is Role.
  auto process ActorAP1, ActorHP1, ActorC1, ActorC2 is
  Actor.

  begin: (ClaimExpenses -> ApproveHeadDep ->
  AuthorisedPerson
    -> ActorAP1, ClaimExpenses -> Pay ->
    Role -> (-> ActorC1, ->
  ActorC2).
    deadlineExpires.ActorC1: ClaimExpenses ->
    ApproveHeadDep -> AuthorisedPerson ->
    Pay -> Role -> (-> ActorC1, -> ActorC2).
}
```

Manifold Process (port in empty_form, port out completed_form)

```
{
  begin: //contains the process definition
  (raise startClaim.AssignActivityToRole.
    IF ClaimForm.ClaimAmount < 1000
      raise Approve.AssignActivityToRole
    ELSE raise Approve.AssignActivityToRole.
    IF ApprovalForm.Approved==YES raise
```

```

Pay.AssignActivityToRole.
)
ProcessMonitoring ().
}
Manifold Activity (port in empty_form, competed_form
                  port out empty_form, competed_form)
{
  AssignActivityToRole:
    (raise role.AssignActivityToActor).
  deadlineExpires:
    (IF AreActorsOverloaded()=YES raise
role.DealWithActorsOverload
    ELSE IF ReassignYN==TRUE raise
role.ReassignActivityToActor
    ELSE raise AlertActor.
)
  ActivityExecuted: // activity finished
  { out in outtray the output form }
}
Manifold Role (port in empty_form, competed_form
               port out empty_form, competed_form)
{
  assignActivityToActor:
    (raise ExamineActorWorkload
    IF NOT Overloaded raise AssignActivityToActor
    ELSE raise DealWithActorOverload
    waits.
)
  ReassignActivityToActor:
    (raise reAssignActivityToAcotr
    waits.
)
  ExamineRoleActorsWorkload: ()
  DealWithActorsOverload:
    (extend deadline
    allocate more workers
    change activity priorities
    )
  AlertActor: ()
  ShallIReassign: ()
}

```

We end this section by visualizing the framework in Visifold ([5]), Manifold's visual interface. Figure 2 below shows how the `Process` coordinates the allocation of activities to actors through `Roles` and how dynamic reallocation of work occurs when `ActorC1` is not able to perform allocated work on time.

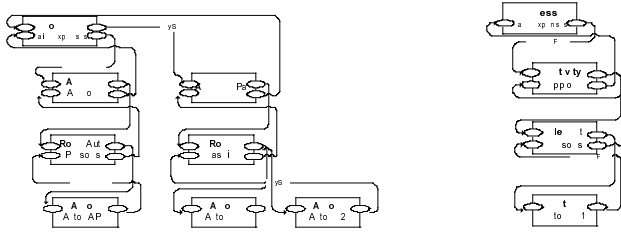


Fig. 2. Hierarchical Allocation and Reallocation of work to actors

5 Discussion; Related and Further Work; Conclusions

In a recent paper, Andrade and Fiadeiro ([2]) argue that Coordination Technologies, as these are understood in the field of Component-Based Software Engineering and Parallel/Distributed Programming, have a contribution to make, in terms of concepts and techniques, to the development of agile Information Systems. The first author of this paper has also argued along the same lines in [15]. Here we elaborate further on the model described in [16] by presenting a two-level hierarchical workflow coordination model. In the process, we have argued for the need to use control-based, event-driven and state-defined coordination programming to model and develop dynamic workflow management systems. We have explained its benefits compared with other approaches that have been used so far, and illustrated its capabilities by means of a specific, if rather simple scenario.

In the short space of a conference paper it would be impossible to describe in detail all the characteristics of our model or compare them in detail with other related approaches. For instance, we have said nothing about examining types of values transmitted via streams (sometimes it may be desirable to know of the data's structure, if not content), etc. This and other issues can be adequately addressed by our model. In particular, our model supports almost all of the functionalities that an adaptive workflow system must exhibit, as those are defined in [10]. More to the point, it supports run-time dynamism, dynamic (re-) configuration, logical decomposition, reusability, and event monitoring.

Over the past few years a number of coordination models and languages have been developed such as Linear Objects (LO), TAO, Gamma and the Chemical Abstract Machine ([17]). However, the first such model, which still remains the most popular one, is Linda ([1]). Although Linda is indeed a successful coordination model, when it is evaluated from the point of view of acting as a framework for modelling human and other activities in information systems, it has some potentially serious deficiencies. The most important deficiency is that it is data-driven i.e. the state of some agent is defined in terms of what kind of data it posts to or retrieves from the Tuple Space. However, there are many cases where we are not interested in the data itself that is being handled; indeed, for security reasons we may not want to allow the examination of data but only coordinate the workflow processes. The issue of security is also relevant in that the medium of communication between processes (the Tuple Space) is an open forum where anyone can post or retrieve tuples. Thus, there is the possibility

of a process either accidentally or deliberately forging, intercepting or stealing information. This has led to the development of models that provide the required security, at the unavoidable cost of increasing the complexity of the model ([14]).

Other related to Linda models are Sonia ([4]) which features the notion of *Agora* ([13]), LAURA ([18]) where the shared space (referred to as *service-space*) is used by agents to post to or retrieve from *forms*, and finally Ariadne ([9]) where the shared workspace is used to hold tree-shaped data and access to them is performed by means of *record templates*.

Our model on the other hand has some clear advantages over the traditional Linda approach and related models:

- Every worker agent is only concerned with getting workload from its input port(s), performing the required work for which it is responsible, and putting the outcome to its out port(s). Such a worker has no need (or way!) to know the environment in which operates and can therefore be substituted with another one without affecting the operation of the rest of co-workers involved.
- Every manager agent is only concerned with making sure that the output produced by some worker agents are sent to some other worker agents that require it. The manager has no need (or way!) of knowing the exact data being transmitted between the worker processes. Thus, security is preserved.
- All entities comprising an activity are treated *homogeneously*. This makes the model very flexible; for instance, new agents can come and go dynamically, some processes may be devices while others may be software programs or humans, etc. The workflow apparatus of our model is not concerned with the nature of the processes being coordinated, only with their input-output inter-dependencies.

We are currently developing a full version of the model as described in this paper, particularly suited to modelling and coordinating activities in distributed information systems, using both a visual ([5]) and textual representation.

References

1. S. Ahuja, N. Carriero, D. Gelernter: Linda and Friends, IEEE Computer 19 (8) (Aug. 1986), 26–34
2. L. F. Andrade, J. L. Fiadeiro: Coordination Technologies for Managing Information System Evolution, CAiSE 2001, Interlaken, Switzerland, LNCS, Vol. 2068. Springer Verlag (4–8 June 2001), 374–387
3. F. Arbab: The IWIM Model for Coordination of Concurrent Activities, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, LNCS Vol. 1061. Springer Verlag (15–17 April 1996), 34–56
4. M. Banville: Sonia: an Adaptation of Linda for Coordination of Activities in Organizations, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, LNCS, Vol. 1061, Springer Verlag (15–17 April, 1996), 57–74
5. P. Bouvry, F. Arbab: Visifold: A Visual Environment for a Coordination Language, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, LNCS Vol. 1061. Springer Verlag (15–17 April, 1996), 403–406

6. N. Carriero, D. Gelernter, S. Hupfer: Collaborative Applications Experience with the Bauhaus Coordination Language, 30th Hawaii International Conference on Systems Sciences (HICSS-30), Mauni, Hawaii, IEEE Press (7–10 Jan., 1997), 310–319
7. M. Cortes: A Coordination Language for Building Collaborative Applications, Computer Supported Cooperative Work, Kluwer Academic Publishers **9** (2000), 5–31
8. C. Ellis, K. KeddarA: ML-DEWS: Modeling Language to Support Dynamic Evolution Within Workflow Systems, Computer Supported Cooperative Work, Kluwer Academic Publishers **9** (2000), 293–333
9. G. Florijn, T. Besamusca, D. Greefhorst: Ariadne and HOPLa: Flexible Coordination of Collaborative Processes, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, LNCS Vol. 1061, Springer Verlag (15–17 April, 1996), 197–214
10. P. T. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi and M. Bergman: Techniques for Supporting Dynamic and Adaptive Workflow, Computer Supported Cooperative Work, Kluwer Academic Publishers **9** (2000), 269–292
11. M. Klein: Challenges and Directions for Coordination Science, Second International Conference on the Design of Cooperative Systems, Juan-les-Pins, France (12–14 June 1996), 705–722
12. T. W. Malone, K. Crowston: The Interdisciplinary Study of Coordination, ACM Computing Surveys **26** (1994), 87–119
13. M. Marchini, M. Melgarejo: Agora: Groupware Metaphors in OO Concurrent Programming, Object-Based Models and Languages for Concurrent Systems, Bologna, Italy, LNCS Vol. 924. Springer Verlag (5 July, 1994)
14. N. H. Minsky, J. Leichter: Law-Governed Linda as a Coordination Model, Object-Based Models and Languages for Concurrent Systems, Bologna, Italy, LNCS Vol. 924. Springer Verlag (5 July, 1994), 125–145
15. G. A. Papadopoulos, F. Arbab: Control-Based Coordination of Human and Other Activities in Cooperative Information Systems, Second International Conference on Coordination Models and Languages, Berlin, Germany, LNCS Vol. 1282. Springer Verlag (1–3 Sept., 1997), 422–425
16. G. A. Papadopoulos, F. Arbab: Modelling Activities in Information Systems Using the Coordination Language MANIFOLD, Thirteenth ACM Symposium on Applied Computing (SAC'98), Atlanta, Georgia, U.S.A., ACM Press (27 Feb.–1 March, 1998), 185–193
17. G. A. Papadopoulos, F. Arbab: Coordination Models and Languages, Advances in Computers, Vol. 46. Marvin V. Zelkowitz (ed.), Academic Press (August 1998), 329–400
18. R. Tolksdorf: Coordinating Services in Open Distributed Systems With LAURA, First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, LNCS Vol. 1061. Springer Verlag (15–17 April, 1996), 386–402
19. B. C. Warboys, R. M. Greenwood, P. Kawalek: Case for an Explicit Coordination Layer in Modern Business Information Systems Architectures, IEE Proceedings Software, Vol. 146 (3) (June 1999), 160–166
20. S. M. Wheeler, S. K. Shrivastava, F. Ranno: OPENflow: A CORBA Based Transactional Workflow System, Advances in Distributed Systems, LNCS Vol. 1752. Springer Verlag (2000), 354–374

A Multi-threaded Asynchronous Language

Hervé Paulino¹, Pedro Marques², Luís Lopes²,
Vasco Vasconcelos³, and Fernando Silva²

¹ Department of Informatics, New University of Lisbon, Portugal
`herve@di.fct.unl.pt`

² Department of Computer Science, University of Oporto, Portugal
`pmarques@med.up.pt`, `{lblopes, fds}@ncc.up.pt`

³ Department of Informatics, University of Lisbon, Portugal
`vv@di.fc.ul.pt`

Abstract. We describe a reference implementation of a multi-threaded run-time system for a core programming language based on a process calculus. The core language features processes running in parallel and communicating through asynchronous messages as the fundamental abstractions. The programming style is fully declarative, focusing on the interaction patterns between processes. The parallelism, implicit in the syntax of the programs, is effectively extracted by the language compiler and explored by the run-time system.

1 Introduction

Dataflow architectures represent an alternative to the mainstream von Neumann architectures. In von Neumann architectures, the order in which the instructions in a program are executed is established at compile time, when the executable is produced. A special purpose register, the program counter, keeps track of the flow of execution. In a dataflow architecture, in contrast, instructions are executed as soon as their arguments are available (the so called *firing rule*) and regardless of any pre-established order. This makes the model totally asynchronous and the instructions self scheduling.

Multi-threaded architectures attempt to improve the performance of classic von Neumann architectures by introducing some features from dataflow architectures such as out-of-order execution and fine grained context switching, usually supported by the microprocessor hardware. These additions aim to provide high processor utilization in the presence of large memory or interprocessor communication latency.

The current generation of superscalar microprocessors requires great amounts of fine grained parallelism to fully explore their aggressive dynamic dispatch capabilities, multiple functional units and, in some cases, rather long pipelines. In this context, support for multi-threading at the hardware level may help to avoid pipeline hazards in current von Neumann implementations, thus eliminating the need for complex forwarding and branch prediction logic.

Despite these interesting possibilities, single-thread performance in multi-threaded architectures is typically low, and this has a negative impact on the

performance of individual applications. The ideal situation would call for applications themselves to be partitioned into several fine grained threads by a compiler. A multi-threaded microprocessor would then overlap the multiple threads from that single application, improving performance. In particular, languages that allow efficient compilation from high-level constructs into low-level, fine grained, code blocks, easily mapped into threads at run-time, may potentially profit from multi-threaded hardware.

Programming languages with compiler support for parallel execution have been extensively researched in the past, namely for dataflow architectures [1,2,5] However, the recent introduction of process calculi [3,11] as the reference models for parallel computations provides an interesting alternative. In fact, process calculi are, in a way, a natural choice since they model systems with processes running in parallel and communicating through message passing. Their compact formal definition and well understood semantics may potentially diminish the usual gap between the semantics of a programming language and that of the corresponding implementations.

In this paper, we describe a multi-threaded run-time system for a programming language based on the TyCO (Typed Concurrent Objects) process calculus [6]. The run-time system is based on a specification previously proposed by the authors and formally demonstrated to be sound relative to the base process calculus [4].

The remainder of the paper is organized as follows. Section 2 presents the core programming language. In section 3 we present the specification and describe the implementation of the language's multi-threaded run-time system. Finally, in section 4, we discuss some issues for future research.

2 The TyCO Programming Language

Our source programming language is called TyCO [6]. The language is based on a process calculus in the lines of the asynchronous π -calculus. The main abstractions are communication channels, objects (collections of methods that wait for incoming messages at channels) and asynchronous messages (method invocations targeted at channels). It is also possible to define process templates, parameterized on a series of variables, that may be instantiated anywhere in the program (this allows for unbounded behavior). The syntax for the language kernel is as follows:

$P ::=$	0	terminated process
	$P \mid P$	concurrent composition
	new x P	new local variable
	$x!l[\tilde{e}]$	asynchronous message
	$x?\{l_1(\tilde{x}_1) = P_1, \dots, l_n(\tilde{x}_n) = P_n\}$	object
	def $X_1(\tilde{x}_1) = P_1 \dots X_n(\tilde{x}_n) = P_n$ in P	definition
	$X[\tilde{e}]$	instantiation
	if e then P else Q	conditional

where x represents a variable, e an expression over integers, booleans, strings or channels [7], X an identifier for a process template, and l a method name.

From an operational point of view, TyCO computations evolve for two reasons: object-message reduction (i.e., the execution of a method in an object in response to the reception of a message) and, template instantiation. These actions can be described more precisely as follows (where v is the result of the evaluation of an expression e , either an integer, a boolean, a string or a channel):

$$x? \{ \dots, l(\tilde{x}) = P, \dots \} \mid x!l[\tilde{v}] \rightarrow \{ \tilde{v}/\tilde{x} \} P$$

The message $x!l[\tilde{v}]$ targeted to channel x , invokes the method l in an object $x? \{ \dots, l(\tilde{x}) = P, \dots \}$ at channel x . The result is the body of the method P running with the parameters $\tilde{x}$ replaced by the arguments $\tilde{v}$. For instantiations we have something similar:

$$\mathbf{def} \dots X(\tilde{x}) = P \dots \mathbf{in} X[\tilde{v}] \mid Q \rightarrow \mathbf{def} \dots X(\tilde{x}) = P \dots \mathbf{in} \{ \tilde{v}/\tilde{x} \} P \mid Q$$

A new instance $X[\tilde{v}]$ of the template process bound to X is created. The result is a new process with the same body as the definition but with the parameters $\tilde{x}$ replaced by the arguments $\tilde{v}$ given in the instantiation.

This kernel language constitutes a kind of assembly language upon which higher level programming abstractions can be implemented as derived constructs. In the example below, we use two such constructs for sequential execution of processes ($;$) and for synchronous method calls (**let/in**) as defined in [8]. The programming example illustrates the use of these primitives and derived constructs. We begin by defining a simple template for a bank account, and creating an account with 100 euro.

```

def Account(self, balance) =
  self ? {
    deposit(amount,replyto) =
      replyto![] | Account[self, balance+amount]
    balance(replyto) =
      replyto![balance] | Account[self, balance]
    withdraw(amount, replyto) =
      if amount >= balance then
        replyto!overdraft[] | Account[self, balance]
      else
        replyto!dispense[] | Account[self, balance-amount]
  }
in new myAccount Account[myAccount, 100]

```

To deposit a further 100 euro and then get our account balance place the following processes running in parallel with the above code:

```

myAccount!deposit[100] ;
let x = myAccount!balance[] in
  io!puts["your account balance is:"] ; io!printi[x]

```

The **let/in** construct calls the method **balance** at channel **myAccount** and waits for a reply value. On arrival, the reply triggers the execution of the process after the **in** keyword and prints the current value of the **balance** attribute.

3 The Virtual Machine

The TyCO source code is compiled into a small and compact language, the TyCO Intermediate Language (TyCOIL) [9], that is given as input to the TyCO's runtime system. This language features simple and fast instructions, thus sharing the advantages of RISC machines.

3.1 The TyCO Intermediate Language

The TyCO virtual machine uses a variable number of general purpose registers ($r_0, \dots, r_{n-1}$), where n , the number of registers, is specified by the TyCOIL directive `registers`. A thread's execution begins by placing its activation record in register r_0 . Registers may contain integers (primitive types integer and boolean) or heap references (for strings and channels).

TyCOIL Program

```

registers 9

string string#1 "your account balance is:"

code main_code {
  - code for def Account
  - code for new myAccount
  - code for Account[myAccount, 100]
  - code for myAccount!deposit[100]
  - code for first semicolon
}
code Account_code {
  - code for self ? {
    - deposit ...
    - balance ...
    - withdraw ...
  }
}
data self_table {
  self_deposit_code
  self_balance_code
  self_withdraw_code
}

code self_deposit_code {
  - code for replyto![]
  - code for Account[self, balance + amount]
}
code self_balance_code {
  - code for replyto![balance]
  - code for Account[self, balance]
}
code self_withdraw_code {
  - code for if amount >= balance ...
}
code continuation_for_let {
  - code for io!puts[string#1]
  - code for second semicolon
}
code continuation_for_first_semicolon {
  - code for let x = ...
}
code continuation_for_second_semicolon {
  - code for io!printi[x]
}

```

Fig. 1. A sketch of a TyCOIL program

TyCOIL programs are composed of three kinds of labeled fragments: `code`, `data`, and `string`. `code` is a sequence of instructions terminated by `schedule`, also referred to as a thread. `data` fragments describe initialized data that may be used,

for example, for method tables. **string** fragments are used to hold string constants. Figure 1 illustrates the structure of the TyCOIL program corresponding to the TyCO example in section 2.

Code for replyto![balance]	Code for self ? {deposit ... balance ... withdraw ... }
– channel replyto in register r3 – balance in register r4 r3.lock() r1 := r3.getStatus() if r1 <= 0 jump enqueue r1 := r3.dequeue() r1[2] := r4 – Fill arguments launch r1 jump done enqueue: – Message frame at r1 r1 := malloc 3 r1[2] := r4 – Fill arguments – End of message frame r3.enqueueMsg(r1) done: r3.unlock()	– channel self in register r4 r4.lock() r5 := r4.getStatus () if r5 >= 0 jump enqueue r5 := r4.dequeue () r6 := r5 [1] – Get offset of the method r6 := self_table [r6] r5 [1] := r6 – Fill address of thread code r5 [2] := r3 – Fill closure launch r5 jump done enqueue: – Object frame at r5 r5 := malloc 4 r5 [1] := self_table – Fill code r5 [2] := r3 – Fill closure – End of object frame r4.enqueueObj (r5) done: r4.unlock()

Fig. 2. TyCOIL code example

The TyCOIL instructions can be divided into six categories: memory allocation, channel (communication queue) manipulation, thread manipulation, external service execution, arithmetic, and program flow. Since the last two are common to most of the languages, we will focus our description on the other instructions.

The memory allocation instruction **malloc** allocates a new, uninitialized, frame in the heap. Frames that contain a thread's execution environment (activation records) are of a special kind: they must start with a slot (used by the machine to enqueue the frame), followed by the address of the code to be executed.

Another category of instructions manipulate channels: **newChannel** allocates a frame from the heap to serve as a communication channel. **enqueueObj** and **enqueueMsg** place a frame at the end of the channel's queue, updating its status accordingly. **dequeue** retrieves and removes a frame from the front of the channel's queue, updating the channel's status. **getStatus** retrieves the channel's current

status: zero for the empty queue, a negative number, $-n$, for a queue containing n messages, a positive number, n , for a queue containing n objects.

The thread manipulation instructions operate on the virtual machine's run-queue: **launch** places a new task in the run-queue; and **schedule** frees the processor, allowing the machine to dequeue and execute a thread from the run-queue.

TyCO allows the definition of services external to the machine's core features. These are invoked through the **external** instruction that executes the service synchronously. Examples of external services are input/output and string operations.

Figure 2 shows a small example of the TyCOIL code corresponding to a message and an object taken from the example in section 2.

The code for the message starts by querying the channel on **r3** for its status. If there are no objects on the queue, it jumps to **enqueue** to insert a newly created frame, representing the message, in the channel's queue. When the channel has objects the code retrieves a frame from the queue, fills slots with message's argument, and launches the frame as a thread ready for execution, in the run-queue.

The object's code is symmetric, the main difference lies in the frame inserted in the queue, that contains the code to execute if a reduction occurs, **code-for-io!prints[x]**, rather than the argument, as was the case with the code for the message.

3.2 The Multi-threaded Virtual Machine

The virtual machine's implementation consists on a **org.tyco.vm** Java package, that includes the following subpackages:

- org.tyco.vm.core** - the core virtual machine implementation;
- org.tyco.vm.values** - values assignable to a general purpose register;
- org.tyco.vm.assemble** - construction of a fragments table containing object representations for each fragment in the TyCOIL program;
- org.tyco.vm.externals** - supplies the **org.tyco.vm.externals.External** Java class, whose extension is required to implement services external to the machine.

The virtual machine is initialized with the program's table of fragments, the number of registers required to run the program, and an external services' table.

The multi-threaded architecture, illustrated in figure 3, contains several concurrent threads. Each thread has its own set of general purpose registers, and a program counter that points to the code fragment it is executing. The remainder of the data structures are shared by all threads, namely: the tables for fragments and external services; the heap, that stores all the frames and channels in current use; and the run-queue holding the tasks ready to execute.

The heap is currently managed by the Java runtime itself, while the interactions with the run-queue are performed through the **org.tyco.vm.core.RunQueue** class, more explicitly, through its **enqueue** and **dequeue** methods.

The TyCO virtual machine starts by creating the run-queue and spawning a designated number of threads. Once thread execution is triggered, each

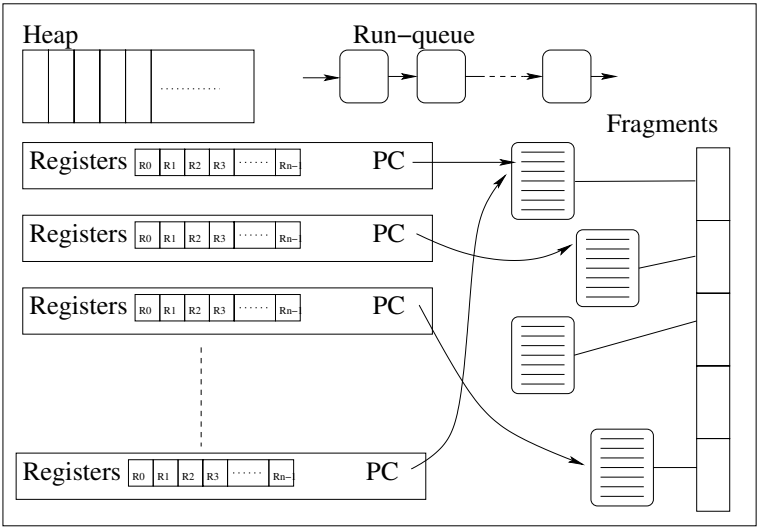


Fig. 3. The multi-thread virtual machine's architecture

of them tries, concurrently, to retrieve a new task from the run-queue. This concurrent behavior implies taking run-queue access control measures to insure data consistency and to prevent race conditions. The code blocks of the `org.tyco.vm.core.Runqueue`'s `enqueue` and `dequeue` methods are critical sections as they can change the shared run-queue status. Access to these methods is made mutually exclusive by declaring the methods synchronized.

Every time a thread executes a `schedule` instruction it tries to retrieve another task from the run-queue. In order to avoid a polling loop, continuously checking for new work when the run-queue is empty, a wait/notify mechanism is used to control access to the run-queue. Thus, if the run-queue is empty, any eager work-seeking thread is put on hold.

After a new task is added to the run-queue, waiting threads are notified that work is available. The machine halts when the run-queue is empty and all threads reach a waiting status.

As seen in the example in figure 2 a sequence of instructions is required to retrieve or add a frame to a channel's queue. This action consists of getting or adding a new frame to the channel's queue and setting the channel's status accordingly. On the multi-threaded virtual machine channels can be used by any running thread. The need for exclusive channel access during this operations is imperative. This is achieved by using the TyCOIL instructions `lock` and `unlock` (bold in figure 2) that explicitly tell the virtual machine the limits of a critical region.

4 Future Work

Work in the TyCO language and runtime system is ongoing. At the virtual machine level we plan to experiment with hardware platforms supporting multi-threading (e.g., Intel's Pentium IV Hyper-threading feature) to evaluate the system's performance.

TyCO is also being used as the building block for the development of a language for programming distributed systems with support for code mobility [10]. The development of the multi-threaded virtual machine is of great importance as it provides the run-time for the nodes of such a system.

Acknowledgement. This work was supported by projects Mikado (IST2001-32222) and MIMO (POSI/CHS/39789/2001), and the CITI research center.

References

1. Blumofe R. D. and Joerg C. F. et.al. Cilk: an efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, August 1995.
2. McGraw J. and Skedzielewski S. et.al. *The SISAL Language Reference Manual – Version 1.2*, March 1985.
3. Honda K. and Tokoro M. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.
4. Lopes L., Vasconcelos V., and Silva F. Fine grained multithreading with process calculi. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 217–226. IEEE Computer Society Press, October 2000.
5. Nikhil R. The Parallel Programming Language Id and its Compilation for Parallel Machines. *International Journal of High Speed Computing*, 5:171–223, 1993.
6. Vasconcelos V. Typed Concurrent Objects. In *European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.
7. Vasconcelos V. Core-TyCO, appendix to the language definition, yielding version 0.2. DI/FCUL TR 01–5, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
8. Vasconcelos V. TyCO Gently. DI/FCUL TR 01–4, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
9. Vasconcelos V. and Lopes L. The TyCO Intermediate Language. To appear.
10. Vasconcelos V., Lopes L., and Silva F. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.
11. Vasconcelos V. and Tokoro M. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

An Efficient Marshaling Framework for Distributed Systems

Konstantin Popov¹, Vladimir Vlassov², Per Brand¹, and Seif Haridi²

¹ Swedish Institute of Computer Science (SICS), Kista, Sweden. <http://www.sics.se>

² Department of Microelectronics and Information Technology, Royal Institute of Technology, Kista, Sweden. <http://www.imit.kth.se>

Abstract. An efficient (un)marshaling framework is presented. It is designed for distributed applications implemented in languages such as C++. A marshaler/unmarshaler pair converts arbitrary *structured* data between its host and network representations. This technology can also be used for persistent storage. Our framework simplifies the design of efficient and flexible marshalers. The network latency is reduced by concurrent execution of (un)marshaling and network operations. The framework is actually used in Mozart, a distributed programming system that implements Oz, a multi-paradigm concurrent language. Mozart, including the implementation of the framework, is available at www.mozart-oz.org.

1 Introduction

This paper presents an efficient *marshaling/unmarshaling* framework for distributed systems. When a message (a data structure) is sent from one host to another (see Figure 1), it is copied from the memory of the source host to the network, and then from the network to the memory of the destination host. This process is rather trivial if the data to be sent has a regular structure such as an array of bytes. In this case, the bytes are sequentially copied to the network, forming a *serial (network) representation* of the array's *memory (host) representation*. Our work targets arbitrary *structured* data which marshaling is less trivial, as illustrated in the Figure 1. Furthermore, the memory representation of a transferred data on the receiving host does not have to be the same as on the sending host, in particular, on the hosts with different architectures.

We address the issues of run-time performance and compactness of the serial representation for data structures with large number of elements, even at the expense of portability between systems, e.g. in the sense of XML.

The way marshaling interfaces the rest of the software of a host in a distributed system can affect the network *latency* and *throughput*. In order to explain this, consider the architecture of a node in a distributed system shown in Figure 2. Here, a message is first constructed by the host application software. Then a reference to that data is passed to the marshaler, which constructs the serial representation of the message in the *marshaling buffer*, or just *buffer* thereafter. The buffer is copied to the network by the *network layer*. Observe that

messages are shared between the application and the marshaller, whereas the buffer is shared between the marshaller and the network layer. In order to fully utilize the network, the network layer has to be invoked sufficiently frequently with a sufficient amount of data on hand. Lost bandwidth cannot be “made up” later by calling the network layer more frequently, or with bigger chunks of data.

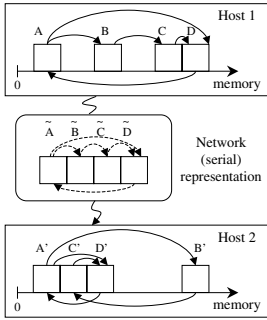


Fig. 1. (Un)Marshaling.

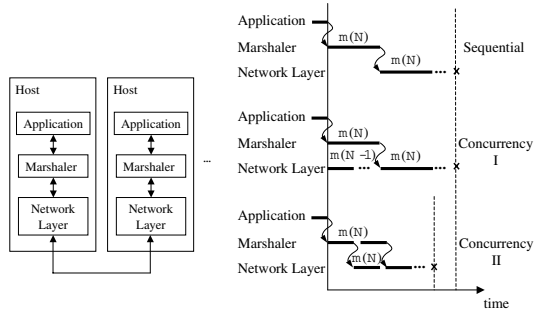


Fig. 2. Layers and Concurrency in a Distributed System.

These three layers (the application, the marshaller and the network layer) can run sequentially as shown in Figure 2 (case **Sequential**). In this case, the network layer waits until marshaling completes causing some network bandwidth to be lost. Alternatively, the network layer can run concurrently with the application layer and with marshaling of the *next* message(s) as shown in Figure 2 (case **Concurrency I**). This approach does not affect the latency, but does affect the throughput of the whole system because of the better utilization of the network bandwidth. We take the third approach depicted in Figure 2 as case **Concurrency II**. In our approach, the marshaller and the network layer run concurrently handling *the same message*, thus reducing the latency since the network layer can start sending a message *before* its marshaling completes. This requires the marshaller to be preempted and resumed. It also allows to use a fixed-size buffer for the serial representation, as well as enables to limit the time spent in the marshaller at every invocation.

Concurrently running the marshaller and the network layer synchronize on the shared buffer. Synchronization between the concurrently running application and the marshaller is avoided by *copying* messages if the application can alter the data in the message. Our framework also supports copying. Furthermore, if the application’s data model distinguishes between *immutable* and *mutable* data, copying of only *mutable* data can be arranged.

A marshaller in our framework consists of two parts: (1) a set of methods that marshal structural elements of data, that we call nodes, and (2) a *traverser* that applies the former methods to nodes according to some *traversal strategy*. There is a marshaling method for each node type. A serial representation consists of a series of tokens that correspond to data elements. An unmarshaller, in turn, is a procedure that reads the serial representation token by token, constructs data

elements from tokens, and passes those nodes to the *builder* that assembles them into the structural data. There are explicit fixed interfaces between the traverser and the marshaling methods, as well as between the unmarshaling procedure and the builder. The presence of that interfaces simplifies design and code maintenance. In addition to marshaling of messages, the same traverser/builder pair can be used for persistent storage, inspecting data, even copying data in e.g. a real-time garbage collector, etc.

Large nodes, which we call *binary areas*, are treated specially in our framework. Since their sizes are unlimited, marshaling of binary areas can be preempted. When it happens, the status of marshaling needs to be saved. Furthermore, binary areas may have irregular structure that requires parsing the content to retrieve references. Also, large and complex structures tend to change as the software is further developed or maintained, therefore it appears to be unwise to encode the knowledge about a binary area's structure directly into the traverser. Finally, marshaling such an area may require information from its parent node, for example, in the case of a "procedure" node and its "byte code" binary area node. We developed the concept of *binary area processors* to meet these requirements. A processor is an abstract data type that provides the "marshal" procedure, and encapsulates the state of marshaling. A marshaler gives a processor to the traverser, and the traverser runs it. References in a binary area are passed back to the traverser, and marshaled in the usual way. A corresponding approach is used also for unmarshaling.

Our framework suits distributed systems that are implemented in languages like C++ and run on conventional single-processor hardware under an OS like Unix or Windows. To our best knowledge, this is the first paper that addresses concurrent marshaling of structured data in this context. Serialization is known for programming languages such as Java, Python, Erlang [1], but we have not seen any publications about the serialization in their implementations. Efficient serialization is known for e.g. RPC/XDR [5,6], CORBA [3], and for parallel programming environments such as MPI [7], but these implementations address non-structured data. Work on portable serialization is being carried on (e.g. WDDX [4] or XML-RPC [8]), but not with the efficiency as the first priority.

2 Data Model

The marshaling framework addresses structural data. A data structure consists of *nodes*, each containing *values* and *references* to other nodes. Nodes that contain references are called *compound*; other nodes are *primitive*. References are directional; nodes that are pointed by references are called *direct descendants*, or just *descendants* whenever no confusion arises. References are labeled according to the semantic roles of the descendants, and labels are ordered. There is a root node. There can be cycles in a data structure. In a C++ application, for example, this data model naturally maps to memory objects such as records, and their addresses. We also assume that data structure can be built top-down, i.e. nodes can be constructed without their descendants.

The serial representation of a data structure consists of a sequence of *tokens* that represent nodes. A token contains representations of the corresponding node's values.

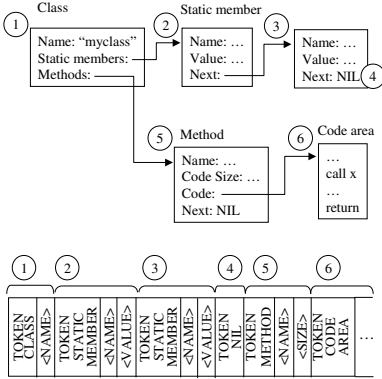


Fig. 3. Data Model.

is built according to the depth-first, left-to-right traversal strategy. The class has two references: a list of static class members and a list of class methods. The serial representations of these descendants follow the class node representation.

3 Marshaling

In our framework, marshaling a data structure is a process of marshaling nodes that come along traversing the data. A straightforward implementation of traversing is recursive: whenever a descendant node is to be marshaled, its marshaling procedure is invoked. This approach is widely used, e.g. in Java, Erlang, and was used in Mozart. Our traverser (see Figure 4) is iterative: it processes elements to be marshaled one by one. In this paper, we stick to the depth-first traversal strategy, which corresponds to a stack as a repository of nodes to be marshaled. Observe that the stack explicitly represents the frontier between traversed and not yet traversed nodes; initially it contains the root node.

Cycles are resolved by means of a “node table” that records compound nodes. If an already visited node is reached again, a special “reference” token is generated. Preemption of marshaling merely implies breaking the loop and saving the stack until resumption, for which the flag **running** is reset. Resumption proceeds by calling **traverser** again. In comparison, preempting/resuming a recursive marshaller in a single-threaded application requires to save/restore the *execution stack*, which is expensive, in particular for large data structures.

A typical “marshal” method for the “method” node shown in Figure 3 is presented in Figure 4. Two node's values are marshaled: the method's name and the code size. The 'Method's code area is marshaled as a binary area. **CodeAreaProc** is a binary area processor. **traverseBinary** pushes the processor (**proc**) onto the stack. Traverser eventually reaches the code node and calls the processor (the code in Figure 4 is extended to handle such stack entries). Note that the formal

A simplest representation of a value is its memory representation, but in practice other representations are used because of cross-platform portability, space efficiency, network security and other reasons. Note that tokens do not contain any representation of references. Instead, references are represented implicitly through the order in which data is traversed, and nodes are marshaled to the buffer. Figure 3 illustrates the issue: here, we consider a class in an object-oriented language and present its possible memory representation and a corresponding serial representation. The serial representation

```

Bool Traverser::run; Stack* Traverser::stack;
HashTable* Traverser::nodeTab;
void Traverser::traverse() {
    while (Node *n = stack->getNode() && run)
        if (n->isCompound())
            if (int index = nodeTab->isKnown(n)) {
                marshalReference(this, index);
            } else {
                int index = hashTable->record(n);
                n->marshal(this, index);
                stack->push(n->descendants());
            }
        } else
            n->marshal(this);
}
}

Bool CodeAreaProc::marshal(Traverser *tr) {
    while (addr < end && tr->run) {
        // marshal an instruction at 'addr';
        ... tr->traverseNode(node); ...
    }
    return (addr ≥ code->getEnd());
}

void Method::marshal(Traverser *tr) {
    marshalStr(name); marshalNum(codeSize);
    CodeAreaProc *proc =
        new CodeAreaProc(start, codeSize);
    tr->traverseBinary(proc);
}

void Traverser::traverseBinary(Proc *p) {
    stack->push(p);
}

```

Fig. 4. Marshaling.

```

Data* unmarshaler(Builder *b) {
    while(1) {
        Tag *tag = getTag();
        switch (tag) {
            case TOKEN_EOF:
                return (b->finish());
            case TOKEN_CLASS:
                Class *class = new Class();
                class->name = unmarshalStr();
                ...
                b->buildCLASS(class);
                break;
            ...
            case TOKEN_METHOD:
                Method *m = new Method();
                ...
                CodeDesc *desc = new
                    CodeDesc(m->start, m->codeSize);
                b->buildBinary((BinaryDesc *) desc);
                b->buildMETHOD(m);
                break;
            case TOKEN_CODE:
                CodeDesc *desc = (CodeDesc *)
                    b->fillBinary(opaque);
                if (unmarshalCode(b, desc))
                    b->finishFillBinary(opaque);
                else
                    b->suspendFillBinary(opaque);
                break;
        }
    }
}

```

Fig. 5. Unmarshaling.

argument of `traverseBinary` has the type `Proc`, which is an abstract superclass of `CodeAreaProc`. This allows the traverser to handle different processors in the uniform way. The code area can contain references to other nodes declared by means of the `traverseNode` traverser's method which pushes a node onto the stack. A binary processor returns `true` when the area is finished; otherwise marshaling is preempted and the processor's stack entry is preserved.

To avoid synchronization between the marshaler and the application, we use the following method of message copying that works best in a system with automatic memory management and distinction between mutable and immutable data. A set of 'marshal' methods is defined that record all mutable values and references of a data structure to a special structure which we call a *snapshot*. When marshaling is resumed, actual values from the data structure are exchanged with the saved ones, and restored back when marshaling is finished or preempted.

4 Unmarshaling

In our framework, unmarshaling a serial representation of a data structure is a process of unmarshaling nodes and “putting them together” to form a data structure. Again, a straightforward implementation is recursive: node’s descendants are unmarshaled by application of the same “unmarshal” procedure. Our unmarshaller is iterative (see Figure 5): it constructs and initializes nodes *except* references using tokens from the buffer, and passes nodes to the builder that assembles nodes into structured data. In other words, the builder’s job is to create references. In the example, unmarshaling a “class” token includes constructing a class node, unmarshaling its values, and passing the node over to the builder.

Builder contains a stack of entries representing references to be created. Specifically, a stack entry contains a memory address where a reference, i.e. a memory address of a node should be stored. A reference is created and stack entry dropped when the unmarshaller passes a node to the builder, which happens by means of `build*()` builder’s methods. Note that the stack represents the boundary between unmarshaled (i.e. constructed) and not yet unmarshaled nodes, as it is in the traverser. The top entry marks the spot in a value being constructed by the unmarshaller where the next node will be attached. Unmarshaling starts with a memory address where the reference to the root node should finally appear. This memory address points to the cell in the builder, which is retrieved by means of the `finish()` builder’s method.

In our example, `buildCLASS` does two things. First, it stores the reference to the ‘class’ node as dictated by the top stack entry, after which the stack entry is discarded. Since this node is the root, there will be the cell in the builder that is returned by `finish()`. Second, it pushes two more entries onto the stack, representing static members and methods, respectively.

Note that the builder and the traverser constitute a matching pair: the strategy of the traverser must correspond to the order in which the builder expects the entries. Specifically, since our traverser works in the left-to-right depth-first order, the builder must build values depth-first, i.e. use the stack, and also push the entry for the left-most descendant last, so it is retrieved and used first.

Like the marshaler, the unmarshaller treats a binary area as a node. When a node being unmarshaled has a binary area descendant, the builder is informed, and the binary area is represented in the builder’s stack by means of the `buildBinary` traverser’s method (see Figure 5). Such a stack entry corresponds to a forthcoming binary area token in the serial representation. When that binary area token is reached and its unmarshaling is in progress, the corresponding stack entry is at the top of stack, and can be accessed by the unmarshaller through the `fillBinary` method. In this way, the builder stack keeps information necessary for unmarshaling of the binary area, which is supplied by the parent node. Finally, if unmarshaling of the area is preempted, the stack entry is preserved using the `suspendFillBinary` method. Note that the builder handles different kinds of binary area in the uniform way, similarly to the traverser.

5 Marshaling in Mozart

We have used our framework in the programming system Mozart [2],[9]. Mozart is an implementation of Oz, a multi-paradigm concurrent programming language. Oz offers symbolic computation with a variety of builtin data types. Oz distinguishes between mutable (e.g. dataflow, single-assignment variables) and immutable (e.g. records) data. Concurrent computation is organized in *threads*.

Mozart provides a *network-transparent* distribution: an application can run on a *network of computers* as on a single computer. Oz entities can be shared between processes. Stateless data that becomes shared is replicated between sites. The consistency of distributed stateful Oz entities, such as a dataflow variable, is guaranteed by entity type-specific distributed protocols. Whenever an operation on a distributed stateful Oz entity is performed, the *distribution subsystems* of the involved Mozart processes exchange message(s) according to the protocol of the entity. Protocol messages can contain Oz values. The operation on the entity is blocked until the protocol completes.

The core of Mozart is a *run-time system* that interprets the intermediate code that an Oz program is compiled to. The run-time system, including the distribution subsystem, is implemented in C++ and runs as a single-threaded application (in terms of native threads as provided by some OS such as Solaris).

The current Mozart's incarnation of the framework is more refined since it also allows that for certain types of nodes, a node can be constructed only after some of its descendants. For example, an Oz record node can be constructed only when the names of its fields are known. Furthermore, marshaling synchronizes also with the garbage collector. The C++ code of the marshaler is optimized on its own, e.g. static resolution of C++ methods is used whenever possible.

The performance of marshaling in Mozart compares favourably with the performance of serialization in Sun Java j2re 1.4.1. We have implemented a linked list holding integers in Oz and in Java. Our comparison actually favours Java, because list elements in Mozart are always polymorphic, therefore the virtual machine has to determine the element types at run time. In the Java code the type of elements are known at compile time. Figure 6 refers to lists of 5000 elements (Java failed to serialize larger structures). We used a 1GHz AMD Athlon.

The plot on the left-hand side of Figure 7 shows that preemption of marshaling induces very little overhead because it requires very little state to be saved and restored. The plot also demonstrates good scalability of the marshaler. The plot on the right-hand side of Figure 7 illustrates the impact of concurrency between the marshaling and network layers: larger buffers correspond to more coarse-grained concurrency, while run-time requirements still remain essentially constant. In the example, the serial representation of the list of integers takes approximately 5Mb, so with a 10Mb buffer there is effectively no concurrency between these two layers. These tests were executed on a cluster of AMD Athlon 1900+ computers connected by 100Mbit switched Ethernet.

Activity	Java	Mozart
marshaling	246ms	1.1ms
unmarshaling	223ms	0.6ms

Fig. 6. Marshaling performance.

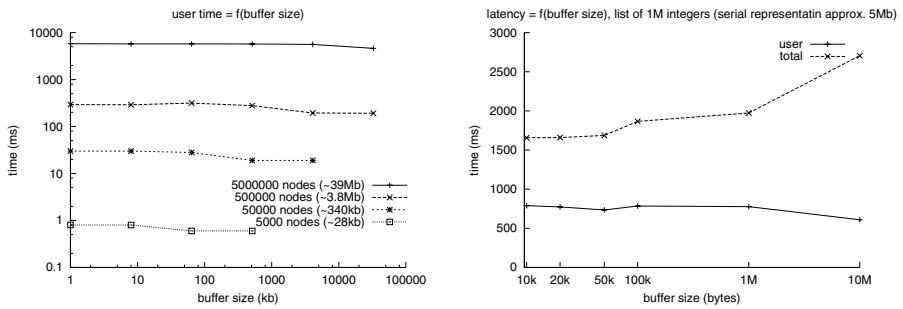


Fig. 7. Overhead due Preemption, and Latency Reduction due Concurrent Marshaling.

6 Conclusions

We have presented an efficient (un)marshaling framework for distributed systems. It allows to minimize the latency of communication between computers in the system through concurrency between the application, marshaling and operations on the network. Concurrency is achieved by preemption of (un)marshaling, which, however, imposes very little overhead. Furthermore, preemption of marshaling enables the fixed-size marshaling buffers, as well as to limit the time per marshaler invocation. From the software engineering point of view, a particular marshaler can be designed more easily due to separation of traversing the structure and marshaling of particular nodes, as well as due to special support for marshaling of nodes with irregular structure. We have developed, evaluated, and actually use in practice our framework within the Mozart programming system. Evaluation of our work against other approaches and systems is carried on.

References

1. Armstrong, J., Virding, R., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1993)
2. Mozart Consortium: The Mozart Programming System. <http://www.mozart-or.oz/> (1998–2003)
3. Object Management Group (OMG): Common Object Request Broker Architecture (CORBA). <http://www.omg.org/> (1997–2003)
4. Open WDDX: The Web Distributed Data Exchange (WDDX). <http://www.openwddx.org/> (1998–2003)
5. Srinivasan, R.: RPC: Remote Procedure Call Protocol Specification. Version 2. Network Working Group Request for Comments (RFC) 1831 (1995)
6. Srinivasan, R.: XDR: External Data Representation Standard. Network Working Group Request for Comments (RFC) 1832 (1995)
7. The MPI Forum: MPI: A Message Passing Interface. Proceedings of Supercomputing '93 (1993) 878–883.
8. UserLand Software, Inc.: XML-RPC. <http://www.xmlrpc.com/> (1998–2003)
9. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004 (to appear)

Deciding Optimal Information Dispersal for Parallel Computing with Failures

Sung-Keun Song, Hee-Yong Youn, and Jong-Koo Park

School of Information and Communications Engineering, Sungkyunkwan University

Suwon, Korea

songsks@mail.skku.ac.kr, youn@ece.skku.ac.kr,

pjk@yurim.skku.ac.kr

Abstract. Supporting availability, integrity, and confidentiality of data is crucial for parallel computer systems. The parallel computer systems require to encode and distribute data over multiple storage nodes to survive failures and malicious attacks. Information Dispersal Scheme(IDS) is one of the most efficient schemes allowing high availability and security with reasonable overhead. In this paper we propose an algorithm determining an optimal IDS in terms of availability.

1 Introduction

As the modern society increasingly relies on digitally stored and accessed data, supporting availability, integrity, and confidentiality of data is crucial for parallel computer systems. Here, we need a mechanism with which users can securely store critical information, ensuring that data are continuously accessible, cannot be destroyed, and are kept confidential.

The survivable storage systems [5, 6] require to encode and distribute data over multiple storage nodes to survive failures and malicious attacks. For that, the systems use some data distribution schemes that can allow a desired level of availability and security. There exist many data distribution schemes [7] including Replication, Splitting, Information Dispersal [3], Secret Sharing [1], and Ramp scheme [2]. Among them, Information Dispersal Scheme (IDS) is one of the most efficient schemes allowing high availability and security with reasonable overhead. In this paper, thus, we research the properties of IDS. As a result, we propose an algorithm determining an optimal IDS in terms of availability, given a set of IDS's.

The rest of the paper is organized as follows. Section 2 discusses the properties of IDS. Section 3 proposes an algorithm determining an optimal IDS in terms of availability. Finally, we conclude the paper in Section 4.

This work was supported by Korea Research Foundation Grant(KRF-2002-041-D00421) and BK21. Corresponding author: Hee Yong Youn

2 The Properties of IDS

This section discusses the property of IDS. First, the notations used are introduced, and then one problem of the classic availability formula is identified. Based on this, a new one is proposed.

n	total pieces of data stored at the storage nodes by replication
m	the number of pieces which can reconstruct the original data
k	n/m ; information expansion ratio(IER); $k \geq 1$
P_s	node survivability ; $0 < P_s < 1$
$P_d(m, n)$	availability of (m, n) -IDS
Class _{i}	all IDS's whose k is i

The IDS is a data distribution scheme which stores n pieces of data, where the original data are partitioned into m pieces. Thus m pieces suffice to reconstruct the original data. Here m and n have the following relationship.

$$n = km \quad (k \geq 1, \quad 1 \leq m \leq n) \quad (1)$$

If k is smaller than 1, original data is lost. Therefore, k must be larger than 1. If k is an integral number, all the m pieces are replicated k times, respectively. If k is a fractional number, a portion of m pieces are replicated $\lfloor k \rfloor - 1$ times while the others are replicated $\lfloor k \rfloor$ times. Note that the maximum number of node failures that an (m, n) -IDS can tolerate is $\lfloor k \rfloor - 1$. The (m, n) -IDS is able to tolerate up to $n - m$ node failures[8]. However, the combination of failed nodes needs to satisfy a condition to reconstruct the data. If k is an integral number, the (m, n) -IDS replicate each one of the m pieces k times. Therefore, in order to reconstruct the data, at least one of the same k pieces for every one of the m pieces must survive. Consequently, the maximum number of node failures that the (m, n) -IDS can tolerate is $k - 1$. If k is a fractional number, the number of node failures that the (m, n) -IDS can tolerate is $\lfloor k \rfloor - 1$.

The previous schemes proposed for IDS estimate the availability using the following equation.

$$P(m, n) = \sum_{i=m}^n \binom{n}{i} P_s^i (1 - P_s)^{n-i} \quad (2)$$

Note here, however, that the equation above is not correct. For example, for $(3, 8)$ -IDS, Piece-1, 2 are replicated 3 times while Piece-3 is replicated twice. If the two of Piece-3 fail, the original data cannot be constructed. However, the classic availability formula above includes such cases. Therefore, we propose a new availability formula:

$$P(m, n) = \left(\sum_{i=1}^k \binom{k}{i} P_s^i (1 - P_s)^{k-i} \right)^m, \quad k = \frac{n}{m} \text{ (Here } k \text{ is assumed to be an integer)} \quad (3)$$

The availability of (i, i) -IDS $i \geq 1$ for a fixed P_s decreases as i increases. Observe that availability cannot be improved as the degree of information dispersal increases

under 'IER=1' [4]. The availability of (2, 8)-IDS of Class₄ and (3, 15)-IDS of Class₅ have the following relationships:

$$\begin{aligned} P_d(2, 8) &> P_d(3, 15) \text{ if } P_s < 0.3 \\ P_d(2, 8) &= P_d(3, 15) \text{ if } P_s = 0.3 \\ P_d(2, 8) &< P_d(3, 15) \text{ if } P_s > 0.3 \end{aligned} \quad (4)$$

As we see from this example, the IDS allowing highest availability becomes different for different P_s values. Also, we find that an IDS with a higher IER may often have lower availability than that with a lower IER. Figure 1 shows the availability of different IDS's. Critical survivability is defined as the P_s allowing $P_d(m, n) = P_d(i, h)$. Some important properties of IDS are as follows.

1. $P_d(m, n) < P_d(m, n+mi)$ for $i > 1$
2. $P_d(m, n) > P_d(m+i, n)$, for $n-m \geq i > 0$ and $\frac{n}{m+i}$ is an integer
3. $P_d(m, n) > P_d(m+i, n+j)$ if $k = \frac{n}{m} = \frac{n+j}{m+i}$, $i, j \geq 1$
4. $P_d(i, j) < P_d(m, n)$ if $i > m$ and $j < n$
5. $P_d(i, j) > P_d(m, n)$ if $i < m, j < n$, and $\frac{j}{i} \geq \frac{n}{m}$

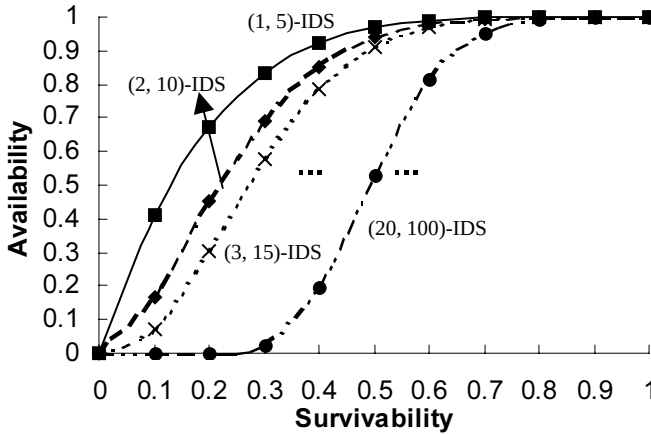


Fig. 1. Example of Case 3 above.

3 Deciding Optimal IDS

A system designer may want to decide an optimal IDS in a set of IDS's. The following algorithm reduces the size of the set of IDS's, and eventually decides an optimal IDS.

Algorithm 1 : Search the possible optimal information dispersal scheme

- Input S : S is a set of IDS's which the designer can choose
- Output S' : S' is a reduced set of IDS

Step 1. Let $S' = S$.

Step 2. Classify S' into classes of different values of k .

Step 3. Select (m, n) -IDS which has the smallest m and n in the class.

Step 4. Repeat Step 3 for all classes.

Step 5. If m of the (m, n) -IDS of a class which has the biggest IER is the same or smaller than all a 's of the (a, b) -IDS of other classes, select the (m, n) -IDS.

Step 6. If the IDS's do not satisfy Step 5, select (m, n) -IDS of a class which has the biggest IER and all (a, b) -IDS which have the same critical survivability with the (m, n) -IDS.

Step 7. Output S' .

4 Conclusion

In this paper we have studied the properties of information dispersal schemes which can be used for survivable storage of parallel computer systems. According to the properties found, we developed an algorithm by which an optimal IDS is decided. An important property pertaining to information dispersal scheme is that a designer needs to determine the range of P_s first, and then choose a right IDS. In this paper, we made some assumptions in considering the properties of IDS. We will develop a more vigorous model without such assumptions which allows the best IDS in real environment.

References

1. Shamir, A.: How to Share a Secret: Comm. ACM. Vol. 22. (1979) 612–613
2. Blakley, G.R., Meadows, C.: Security of Ramp Schemes. Advances in Cryptology – CRYPTO. (1985) 242–268
3. Rabin, M.O.: Efficient Dispersal of Information for Security. Load Balancing and Fault Tolerance. ACM. (1989) 335–348
4. Hung-Min Sun., Shiuh-Pyng Shieh.: Optimal Information Dispersal for Increasing the Reliability of a Distributed Service. IEEE Trans. Vol. 46. (1997) 462–472
5. Wylie, J.J., Bigrigg, M.W., Strunk, J.D., Ganger, G.R., Kiliccote, H., Khosla, P.K.: Survivable information storage systems. IEEE Computer. (2000) 61–68
6. Wylie, J.J., Bakkaloglu, M., Pandurangan, V., Bigrigg, M.W., Oguz, S., Tew, K., Williams, C., Ganger, G.R., Khosla, P.K.: Selecting the Right Data Distribution Scheme for a Survivable Storage System. Technical Report CMU-CS-01-120 Carnegie Mellon University. (2001)
7. Agrawal D., Malpini A.: Efficient dissemination of information in computer networks. The computer journal. (1991) 534–541
8. Agrawal G., Jalote P.: Coding-based replication schemes for distributed systems. IEEE Transactions on Parallel and Distributed Systems. (1995) 240–250.

Parallel Unsupervised k -Windows: An Efficient Parallel Clustering Algorithm

Dimitris K. Tasoulis^{1,2} Panagiotis D. Alevizos^{1,2}, Basilis Boutsinas^{2,3}, and Michael N. Vrahatis^{1,2}

¹ Department of Mathematics, University of Patras, GR-26500 Patras, Greece
`{dtas, alevizos, vrahatis}@math.upatras.gr`

² University of Patras Artificial Intelligence Research Center (UPAIRC), University of Patras, GR-26500 Patras, Greece

³ Department of Business Administration, University of Patras, GR-26500 Patras, Greece
`vutsinas@bma.upatras.gr`

Abstract. Clustering can be defined as the process of partitioning a set of patterns into disjoint and homogeneous meaningful groups (clusters). There is a growing need for parallel algorithms in this field since databases of huge size are common nowadays. This paper presents a parallel version of a recently proposed algorithm that has the ability to scale very well in parallel environments.

1 Introduction

Clustering, that is the partitioning a set of patterns into disjoint and homogeneous meaningful groups (clusters), is a fundamental process in the practice of science. In particular, clustering is fundamental in knowledge acquisition. It is applied in various fields including data mining [6], statistical data analysis [1], compression and vector quantization [15]. Clustering is, also, widely applied in most social sciences.

The task of extracting knowledge from large databases, in the form of clustering rules, has attracted considerable attention. Due to the growing size of the databases there is also an increasing interest in the development of parallel implementations of data clustering algorithms. Parallel approaches to clustering can be found in [9,10,12,14,16].

Recent software advances [7,11], have provided the ability to collections of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The vast number of individual Personal Computers available in most scientific laboratories suffices to provide the necessary hardware. These pools of computational power exploit network interfaces to link individual computers. Since network infrastructure is currently immature to support high speed data transfer interfaces, it comprises a bottleneck to the entire system. So applications that have the ability to exploit specific strengths of individual machines on a network, while minimizing the required data transfer rate are best suited for these environments.

The results reported in the present paper indicate that the recently proposed k -windows algorithm [17] has the ability to scale very well in such environments.

A fundamental issue in cluster analysis, independent of the particular technique applied, is the determination of the number of clusters that are present in the results of a clustering study. This remains an unsolved problem in cluster analysis. The k -windows algorithm is equipped with the ability to automatically determine the number of clusters.

The rest of the paper is organized as follows. Section 2 is devoted to a brief description of the workings of the k -windows algorithm. In Section 3 the parallel implementation of the algorithm is exposed, while Section 4, is devoted to the discussion of the experimental results. The paper ends with concluding remarks and a short discussion about further research directions.

2 The k -Windows Algorithm

The key idea behind this algorithm is the use of windows to determine clusters. A window is defined as an orthogonal range in d -dimensional Euclidean space, where d is the number of numerical attributes. Therefore each window is a d -range of initial fixed area a . Intuitively, the algorithm tries to fill the mean space between two patterns with non overlapping windows. Every pattern that lies within a window is considered to belong to the corresponding cluster. Iteratively the algorithm moves each window in the Euclidean space by centering them on the mean of the patterns included. This iterative process continues until no further movement results in an increase in the number of patterns that lie within each window (see solid line squares in Fig. 1). Subsequently, the algorithm enlarges every window in order to contain as many patterns as possible from the corresponding cluster.

In more detail, at first, k means are selected (possibly in a random way). Initial d -ranges (windows) have as centers those initial means and each one is of area a . Then, the patterns that lie within each d -range are found, using the Orthogonal Range Search technique of Computational Geometry [2,4,5,8, 13]. The latter has been shown to be effective in numerous applications and a considerable amount of work has been devoted to this problem [13]. The main idea is to construct a tree-like data structure with the properties that give the ability to perform a fast search of the set of the patterns.

An orthogonal range search is based on this pre-process phase where the *tree* is constructed. Thus patterns that lie within a d -range can be found traversing the tree. The *orthogonal range search* problem can be stated as follows:

– **Input:**

- a) $V = \{p_1, \dots, p_n\}$ is a set of n points in $\mathbb{R}^d$ the d -dimensional Euclidean space with coordinate axes $(Ox_1, \dots, Ox_d)$,
- b) a query d -range $Q = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ is specified by two points $(a_1, a_2, \dots, a_d)$ and $(b_1, b_2, \dots, b_d)$, with $a_j \leq b_j$.

– **Output:**

- report all points of V that lie within the d -range Q .

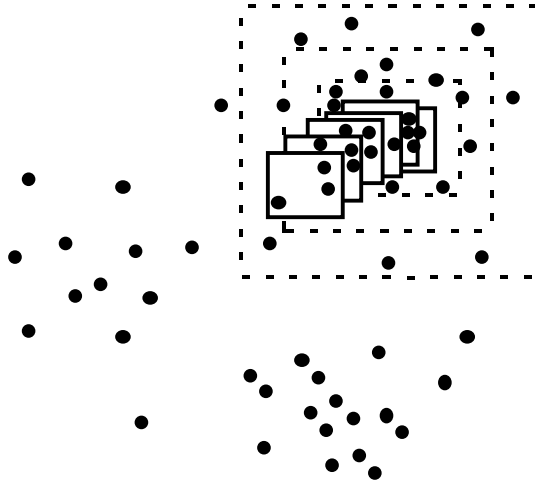


Fig. 1. Movements and enlargements of a window.

Then, the mean of the patterns that lie within each range, is calculated. Each such mean defines a new d -range, which is considered as a movement of the previous one. The last two steps are executed repeatedly, until there is no d -range that includes a significant increment of patterns after a movement.

In a second phase, the quality of the partition is calculated. At first, the d -ranges are enlarged in order to include as many patterns as possible from the cluster. Then, the relative frequency of patterns assigned to a d -range in the whole set of patterns, is calculated. If the relative frequency is small, then it is possible that a missing cluster (or clusters) exists. Thus, the whole process is repeated.

The windowing technique of the k -windows algorithm allows for a large number of initial windows to be examined, without any significant overhead in time complexity. Then, any two overlapping windows are merged. Thus the number of clusters can be automatically determined by initializing a sufficiently large number of windows. The remaining windows, define the final set of clusters.

3 Parallel Implementation

When trying to parallelize the k -windows algorithm, it is obvious that the step that requires the most computational effort is the range search. For this task we propose a parallel algorithmic scheme that uses the Multi-Dimensional Binary Tree for a range search.

Let us consider a set $V = \{p_1, p_2, \dots, p_n\}$ of n points in d -dimensional space $\mathbb{R}^d$ with coordinate axes $(Ox_1, Ox_2, \dots, Ox_d)$. Let $p_i = (x_1^i, x_2^i, \dots, x_d^i)$ be the representation of any point p_i of V .

Definition: Let V_s be a subset of the set V . The *middle point* p_h of V_s with respect to the coordinate x_i ($1 \leq i \leq d$) is defined as the point which divides the set $V_s - \{p_h\}$ into two subsets V_{s_1} and V_{s_2} , such that:

- i) $\forall p_g \in V_{s_1}$ and $\forall p_r \in V_{s_2}$, $x_i^g \leq x_i^h \leq x_i^r$.
- ii) V_{s_1} and V_{s_2} have approximately equal numbers of elements: If $|V_s| = t$ then $|V_{s_1}| = \lceil \frac{t-1}{2} \rceil$ and $|V_{s_2}| = \lfloor \frac{t-1}{2} \rfloor$.

The **multidimensional binary tree** T which stores the points of the set V is constructed as follows.

1. Let p_r be the *middle point* of the given set V , with respect to the first coordinate x_1 . Let V_1 and V_2 be the corresponding partition of the set $V - \{p_r\}$. The point p_r is stored in the root of T .
2. Each node p_i of T , obtains a left child $left[p_i]$ and a right child $right[p_i]$ as follows: $MBT(p_r, V_1, V_2, 1)$

procedure $MBT(p, L, R, k)$

begin

if $k = d + 1$ **then** $k \leftarrow 1$

if $L \neq \emptyset$ **then**

begin

let u be the middle point of the set L with respect to the coordinate x_k and

let L_1 and L_2 be the corresponding partition of the set $L - \{u\}$.

$left[p] \leftarrow u$

$MBT(u, L_1, L_2, k + 1)$

end

if $R \neq \emptyset$ **then**

begin

let w be the middle point of the set M with respect to the coordinate x_k and
let R_1 and R_2 be the corresponding partition of the set $R - \{w\}$.

$right[p] \leftarrow w$

$MBT(w, R_1, R_2, k + 1)$

end

end

Let us consider a query d -range $Q = [a_1, b_1] \times [a_2, b_2] \times \dots \times [a_d, b_d]$ specified by two points $(a_1, a_2, \dots, a_d)$ and $(b_1, b_2, \dots, b_d)$, with $a_j \leq b_j$. The search in the tree T is effected by the following algorithm which accumulates the retrieved points in a set $\mathcal{A}$, initialized as empty:

The orthogonal range search algorithm

1) Let p_r be the root of T

2) $\mathcal{A} \leftarrow \text{SEARCH}(p_r, Q, 1)$

3) return $\mathcal{A}$

procedure $\text{SEARCH}(p_t, Q, i)$

begin

initialize $\mathcal{A} \leftarrow$ **if** $i = d + 1$ **then** $i \leftarrow 1$

```

if  $p_t \in Q$ 
then  $\mathcal{A} \leftarrow \mathcal{A} \cup \{p_t\}$ 
if  $p_t \neq leaf$  then
begin
if  $a_i \leq x_i^t$  then  $\mathcal{A} \leftarrow \mathcal{A} \cup \text{SEARCH}(left[p_t], Q, i + 1)$ 
if  $x_i^t \leq b_i$  then  $\mathcal{A} \leftarrow \mathcal{A} \cup \text{SEARCH}(right[p_t], Q, i + 2)$ 
end
return  $\mathcal{A}$ 
end

```

The **orthogonal range search algorithm** has a complexity of $O(dn^{1-\frac{1}{d}} + k)$ [13], while the preprocessing step for the tree construction has $\theta(dn \log n)$.

In the present paper we propose a parallel implementation of the previous range search algorithm. The algorithmic scheme we propose uses a Server-Slave model. More specifically, the server executes the algorithm normally but when a range search query is to be made, it spawns a sub-search task at an idle node. Then it receives any new sub-search messages from that node, if any, and spawns them to different nodes. As soon as a node finishes with the execution of its task it sends its results to the server and it is assigned a new sub search if one exists. At the slave level during the search if both branches of the current node have to be followed and the current depth is smaller than a preset number (user parameter) then one of them is followed and the other is sent as new sub-search message to the server. For example Fig. 2, illustrates how the spawning process works, when both children of a tree node have to be followed then one of them is assigned to a new node.

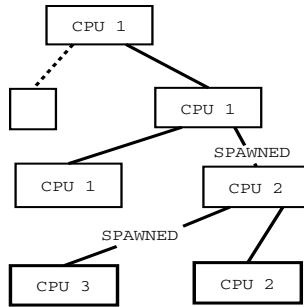


Fig. 2. The spawning process.

Let us assume that N computer nodes are available. In the proposed implementation the necessary communication between the master and the slaves is a “Start a new sub-search” message at node p_i for the d -range Q . The size of this message depends only on the dimension of the problem and subsequently of the range Q . On the other hand the slave-to-master communication, has two different types. The first type is the result of a sub-search. This, as it is shown in the

code below, is a set of points that belong to the analogous range. In a practical setting it is not obligatory to return all the points but only their number and their median, since only these two quantities are necessary for the k -windows algorithm. The other type of slave-to-master communication is for the slave to inform the master that a new sub-search is necessary. This message only needs to contain the node for which the new sub-search should be spawned since all the other data are already known to the master.

The parallel orthogonal range search algorithm

```

1) Let  $p_r$  be the root of  $T$ 
2)  $\mathcal{A} \leftarrow \text{P\_SEARCH}(p_r, \mathcal{Q}, 1)$ 
3) return  $\mathcal{A}$  procedure P_SEARCH ( $p_t, \mathcal{Q}, \mathcal{A}, i$ )

begin
Init a queue  $NQ$  of  $N$  nodes.
Init a queue of tasks  $TQ$  containing only task  $(p_t, i)$ 
set  $NTASKS \leftarrow 0$ 
set  $FINISHED \leftarrow 0$ 
do
  begin
  if  $NQ$  not empty and  $TQ$  not empty then
    begin
      pop an item  $N_i$  from  $NQ$ 
      pop an item  $(p_i, i)$  from  $TQ$ 
      spawn the task SLAVE_SEARCH( $p_i, \mathcal{Q}, i$ ) at node  $N_i$ 
      set  $NTASKS \leftarrow NTASKS + 1$ 
    end
  if received End-Message  $\mathcal{A}_i$  from node  $N_i$  then
    begin
      add  $N_i$  to  $NQ$ 
      set  $\mathcal{A} = \mathcal{A} \cup \mathcal{A}_i$ 
      set  $FINISHED \leftarrow FINISHED + 1$ 
    end
  if received Sub-Search message  $(p_i, i)$  from node  $N_i$  then
    begin
      add  $(p_i, i)$  to  $TQ$ 
      set  $NTASKS \leftarrow NTASKS + 1$ 
    end
  while  $NTASKS \neq FINISHED$ 
end

procedure SLAVE_SEARCH( $p_t, \mathcal{Q}, i$ )
begin
initialize  $\mathcal{A} \leftarrow$  if  $i = d + 1$  then  $i \leftarrow 1$ 
if  $p_t \in \mathcal{Q}$ 
  then  $\mathcal{A} \leftarrow \mathcal{A} \cup \{p_t\}$ 
if  $p_t \neq \text{leaf}$  then
```

```

begin
if  $b_i < x_i^t$  then SLAVE_SEARCH( $left[p_t], Q, i + 1$ )
if  $x_i^t < a_i$  then SLAVE_SEARCH( $right[p_t], Q, i + 1$ )
if  $a_i < x_i^t$  AND  $x_i^t < b_i$  then
  begin
    SLAVE_SEARCH( $left[p_t], Q, A, i + 1$ )
    if ( $i \leq \text{PREDEFINED\_VALUE}$ ) then
      send Sub-Search message ( $right[p_t], Q, i + 1$ ) to server
    else SLAVE_SEARCH( $right[p_t], Q, A, i + 1$ )
  end
end
send End-Message  $A$  to server
end

```

It should also be noted that using this approach all the different nodes of the parallel machine must have the entire data structure of the tree stored in a local medium. This way there is no data parallelism, in order to minimize running the time of the algorithm.

4 Results

The k -Windows clustering algorithm was developed under the Linux operating system using the C++ programming language. Its parallel implementation was based on the PVM parallel programming interface. PVM was selected, among its competitors because any algorithmic implementation is quite simple, since it does not require any special knowledge apart from the usage of functions and setting up a PVM daemon to all personal computers, which is trivial.

The hardware used for our proposes was composed of 16 Pentium III personal computers with 32MB of RAM and 4GB of hard disk availability. A Pentium 4 personal computer with 256MB of RAM and 20GB of hard disk availability was used as the server for the algorithm, as it is exhibited in Fig. 3.

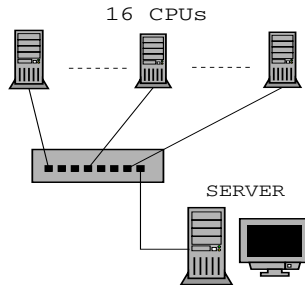


Fig. 3. The hardware used.

To evaluate the efficiency of the algorithm a large enough dataset had to be used. For this purpose we constructed a random dataset using a mixture of Gaussian random distributions. The dataset contained 40000 points with 5 numerical attributes. The points were organized in 4 clusters (small values at the covariance matrix) with 2000 points as noise (large values at the covariance matrix).

The value of the user parameter appears to be critical for the algorithm. If it is too small then no sub-searches are spawned. If it is too large the time to perform the search at some computers might be smaller than the time that the spawning process needs so an overhead to the whole process is created that delays the algorithm. From our experiments the value of 9, for a dataset of this size, appears to work very well.

As it is exhibited in Fig. 4 while there is no actual speedup for 2 nodes, as the number of nodes increases the speed up increases analogously.

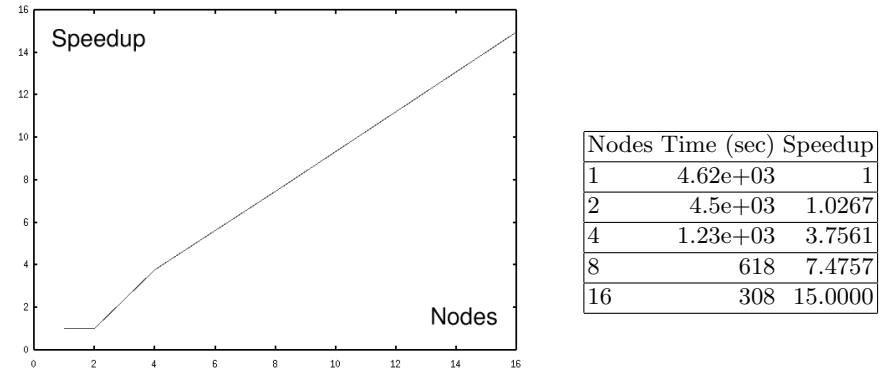


Fig. 4. Times and Speedup for the different number of CPUs

5 Conclusions

Clustering is a fundamental process in the practice of science. Due to the growing size of current databases, constructing efficient parallel clustering algorithms has been attracting considerable attention. The present study presented the parallel version of a recently proposed algorithm, namely the k -windows. The specific algorithm is also characterized by the highly desirable property that the number of clusters is not user defined, but rather endogenously determined during the clustering process. The numerical experiments performed indicated that the algorithm has the ability to scale very well in parallel environments. More specifically, it appears that its running time decreases linearly with the number of computer nodes participating in the PVM.

Future research will focus on reducing the space complexity of the algorithm by distributing the dataset to all computer nodes.

References

1. M. S. Aldenderfer and R. K. Blashfield, *Cluster Analysis*, in Series: Quantitative Applications in the Social Sciences, SAGE Publications, London, 1984.
2. P. Alevizos, *An Algorithm for Orthogonal Range Search in $d \geq 3$ dimensions*, Proceedings of the 14th European Workshop on Computational Geometry, Barcelona, 1998.
3. P. Alevizos, B. Boutsinas, D. Tasoulis, M.N. Vrahatis, *Improving the Orthogonal Range Search k -windows clustering algorithm*, Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washington D.C. 2002 pp.239–245.
4. J.L. Bentley and H.A. Maurer, *Efficient Worst-Case Data Structures for Range Searching*, Acta Informatica, 13, 1980, pp.1551–68.
5. B. Chazelle, *Filtering Search: A new approach to query-answering*, SIAM J. Comput., 15, 3, 1986, pp.703–724.
6. U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth, *Advances in Knowledge Discovery and Data Mining*, MIT Press, 1996.
7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, 1994.
8. B. Chazelle and L. J. Guibas, *Fractional Cascading: II. Applications*, Algorithmica, 1, 1986, pp.163–191.
9. D. Judd, P. McKinley, and A. Jain, *Large-Scale Parallel Data Clustering*, Proceedings of the Int. Conf. on Pattern Recognition, 1996.
10. D. Judd, P. McKinley, A. Jain, *Performance Evaluation on Large-Scale Parallel Clustering in NOW Environments*, Proceedings of the Eight SIAM Conf. on Parallel Processing for Scientific Computing, Minneapolis, March 1997.
11. *MPI The Message Passing Interface standard*, <http://www-unix.mcs.anl.gov/mpi/>.
12. C.F. Olson, *Parallel Algorithms for Hierarchical Clustering*, Parallel Computing, 21:1313–1325, 1995.
13. F. Preparata and M. Shamos, *Computational Geometry*, Springer Verlag, 1985.
14. J.T. Potts, *Seeking Parallelism in Discovery Programs*, Master Thesis, University of Texas at Arlington, 1996.
15. V. Ramasubramanian and K. Paliwal, *Fast k -dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding*, IEEE Transactions on Signal Processing, 40(3), pp.518–531, 1992.
16. K. Stoffel and A. Belkoniene, *Parallel K-Means Clustering for Large Data Sets*, Proceedings Euro-Par '99, LNCS 1685, pp. 1451–1454, 1999.
17. M. N. Vrahatis, B. Boutsinas, P. Alevizos and G. Pavlides, *The New k -windows Algorithm for Improving the k -means Clustering Algorithm*, Journal of Complexity, 18, 2002 pp. 375–391.

Analysis of Architecture and Design of Linear Algebra Kernels for Superscalar Processors

Oleg Bessonov¹, Dominique Fougère², and Bernard Roux²

¹ Institute for Problems in Mechanics of Russian Academy of Sciences,
101, Vernadsky ave., 119526 Moscow, Russia

² Laboratoire de Modélisation en Mécanique à Marseille, L3M-IMT, La Jetée,
Technopôle de Château-Gombert, 13451 Marseille Cedex 20, France
bess@ipmnet.ru, {fougere, broux}@l3m.univ-mrs.fr

Abstract. In this paper we present methods for developing high performance computational kernels and dense linear algebra routines. First, the microarchitecture of AMD Athlon processors is analyzed, with the goal to achieve peak computational rates. These processors are widely used for building inexpensive PC clusters. Then, different approaches for implementing matrix multiplication algorithms are analyzed for hierarchical memory computers, taking into account their architectural properties and limitations. Block versions of matrix multiplication and LU-decomposition algorithms are considered. Finally, the obtained performance results for AMD Athlon/Duron processors are discussed in comparison with other approaches.

Keywords: instruction level parallelism, microarchitecture, out-of-order processors, cache memories, linear algebra kernels, performance measurements, LINPACK benchmark.

1 Introduction

The main problem with CPU performance on real compute-intensive applications is that the achieved floating-point computational speed is as low as only 15 to 25 percent of the "theoretical speed" due to limitations in computer memories. For dense linear algebra algorithms [1], fortunately the much higher levels can be achieved (50 to 95 percent, depending on CPU and/or system architecture). Therefore, development of this kind of algorithms can be considered as approaching "the speed of sound" of particular architecture and demonstrating its computational potential. To achieve this level of performance, investigation of CPU and system architecture is necessary in order to exploit fully the intrinsic instruction level parallelism (ILP), with the following development of efficient and robust computational algorithms.

The main idea for the efficient implementation of linear algebra software is based on the fact that such algorithms perform $O(n^3)$ arithmetic operations on $O(n^2)$ data elements. Therefore, different blocking strategies can be employed, with the use of fast processor caches. In the previous paper [2] we proposed a

new approach based on multiplication of block vector by matrix, as opposed to vector–matrix (BLAS 2) and matrix–matrix (BLAS 3) ones. This approach (to be called "BLAS 2 $\frac{1}{2}$ ") combines the efficiency of BLAS 3 with the flexibility and scalability of BLAS 2, because it depends less on the shape and size of submatrices to be multiplied. Linear equation solvers based on this new algorithm have demonstrated the record level of LINPACK benchmark [3] performance for some RISC microprocessors (Intel i860, SGI R8000/R10000).

Exploiting the ILP of RISC processors is possible due to their regular structure with uniform and orthogonal instruction sets, large number of addressable registers and deterministic execution times. This is not true for modern superscalar CISC microprocessors of x86 architecture (Intel Pentium-4 and AMD Athlon), that are developing rapidly and have become a very attractive opportunity for building inexpensive computer systems and PC clusters. However, the architectural irregularities of these processors can be partly compensated by their highly asynchronous microarchitecture, with deep out-of-order execution and register renaming. To achieve a good performance on CISC processors, computational cores must be based on building long independent chains of instructions, rather than on cycle-by-cycle instruction scheduling (as in the case of in-order RISC). Therefore, deep understanding of a processor's microarchitecture is required, along with knowledges about the structure and limitations of memory levels.

The goal of this work is to analyze the architecture of AMD Athlon family of processors and to employ the previous experience for building efficient computational cores and dense linear algebra kernels. In the paper we describe the new methods (as applied to AMD processors) and compare them to the "Cache-contained matrix multiply" approach (ATLAS Project [4]). In comparisons, our implementations demonstrate superiority for a range of matrix sizes (small to medium). The record results of LINPACK-1000 benchmark are achieved for AMD Athlon and Duron processors and registered appropriately. The achieved performance is comparable to the performance of fastest RISC microprocessors.

2 Microarchitecture of an Out-of-Order Processor

AMD Athlon/Duron are very fast CISC-architecture (Complex Instruction Set Computer) processors with deep out-of-order execution, multiple functional units and two levels of cache memory [5]. Some of these properties prevent from achieving peak performance (e.g. legacy x86 instruction set), others compensate and overcome different limitations (e.g. memory prefetch to increase throughput, or asynchronous execution to reduce negative effect of instruction dependencies). Appropriate coding is required in order to avoid bottlenecks and tolerate limitations, imposed by several stages of processor pipeline. Main characteristics of AMD processors are:

- 3-way superscalar architecture with multiple functional units, deep out-of-order execution and register renaming;

- pipelined x87 Floating Point Unit (FPU) that can execute up to two 64-bit arithmetic operations per cycle (Mult + Add , 3.06 GFLOPS peak performance for 1.53 GHz CPU frequency);
- pipelined load/store unit, with issue rate up to 2 loads per cycle;
- 8 floating point registers (80-bit) organized as a stack;
- two large physical register files (88 entries for FPU and 24 entries for integer arithmetic) used for renaming (remapping) architectural registers in asynchronous execution;
- 64 KByte level 1 data cache (2-way associative) and 64 KByte level 1 instruction cache;
- 256 KByte exclusive level 2 cache (16-way) with limited throughput;
- relatively slow main memory with prefetch, and more limited throughput.

3 Development of Computational Cores for AMD Processors

The core of most linear algebra routines is the matrix multiplication algorithm. Different forms of this algorithms were investigated in [2]. The scalar product form (Fig. 1) is chosen for AMD processor architecture as the most efficient.

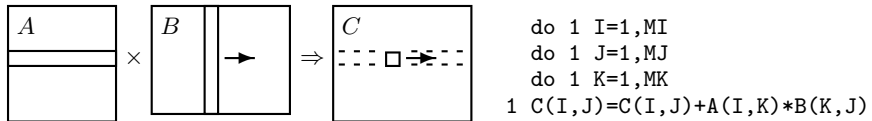


Fig. 1. Scalar product core of the matrix multiplication algorithm

In order to achieve 80 to 90 % of the peak computational speed (for 64-bit floating point arithmetic), the following method is used:

- Multiply a block row in A by the matrix B, using the L1 cache as a pool of vector registers (to contain a block row);
- Rely on the asynchronous (out-of-order) execution in order to hide instruction and cache/memory access latencies, because the static instruction scheduling and loop skewing are not applicable for x87 stack architecture;
- Pack every 3 instructions into the aligned 8-byte bundle to guarantee 3 instructions/clock decoding rate;
- Set the width of a block row in A to 4 (or to 6 when applicable), because this is limited by the depth of x87 register stack;
- Group every 4 block rows in A (of the width 4 or 6) into a "wide" block row (of the width 16 or 24) to tolerate the limited memory throughput (Fig. 2);
- Employ data prefetch to the next column of the matrix B to hide memory latencies.

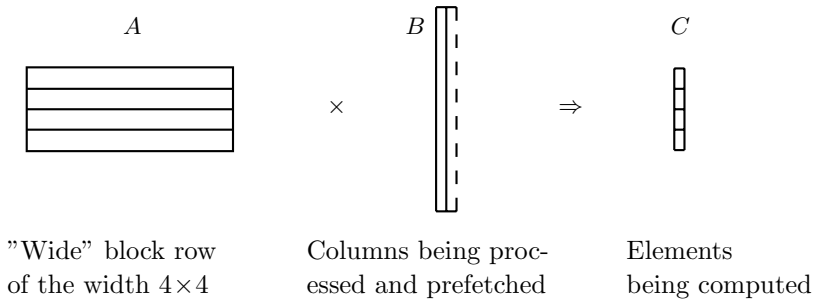


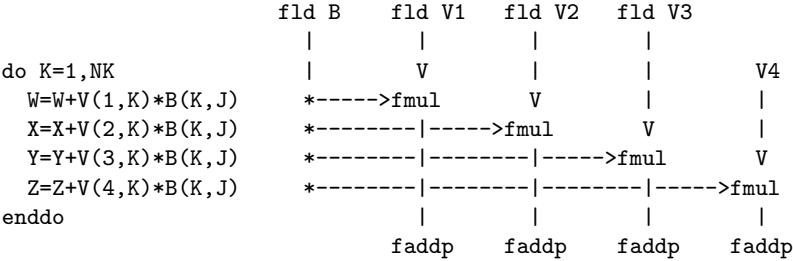
Fig. 2. Grouping block rows in a computational core

A group of 4 block rows in A, used for the multiplication by the matrix B, is copied preliminarily into a dense work array. Owing to the cache replacement algorithm, this array becomes effectively "locked" in the L1 cache after the first iteration of the calculation loop. The typical length of this group (work array) is 256 (for the block width 4), that corresponds to the array size 32 KB, i.e. half of the L1 cache. All other memory accesses are implicitly served through another half (cache line) of the L1 cache and therefore don't influence the core loop.

The scalar product results are accumulated in the x87 register stack. Four block rows are multiplied in sequence by the same column of the matrix B simultaneously with prefetching the next column in B into the cache. A basic block of the inner loop of the algorithm written in assembler looks as follows:

fldl (%edi,%eax)		Basic block:
fldl (%edx,%eax,4)		
fmul %st(1),%st # 1		12 instructions
faddp %st,%st(5)		
fldl 8(%edx,%eax,4)		8 FP operations
fmul %st(1),%st # 2		
faddp %st,%st(4)		4 aligned
fldl 16(%edx,%eax,4)		8-byte bundles
fmul %st(1),%st # 3		
faddp %st,%st(3)		executes (ideally)
fmull 24(%edx,%eax,4)		in 4 clock cycles
faddp %st,%st(1) # 4		

It corresponds to the following FORTRAN notation and the dependency graph:



The dependency graph consists of 4 instruction chains independent on each other, each ~ 12 clock cycles long, with deep overlap due to asynchronous (out-of-order) execution and register renaming. Execution speed of this inner loop (accounting indexing, loop control and prefetch instructions) achieves 90% of the peak floating point computational rate, i.e. 1.8 FP instructions per clock cycle. This corresponds to performance about 2750 MFLOPS for the processor frequency 1530 MHz.

4 Matrix Multiplication and Solving Linear Systems

Generally, two-level blocking strategy is used for matrix multiplication algorithms [2] (Fig. 3, top):

- strip-mining (vertical splitting of the matrix A) to fit a part of block-row in A to L1 cache;
- tiling (vertical splitting of the matrix B) to fit a rectangular block to L2 cache.

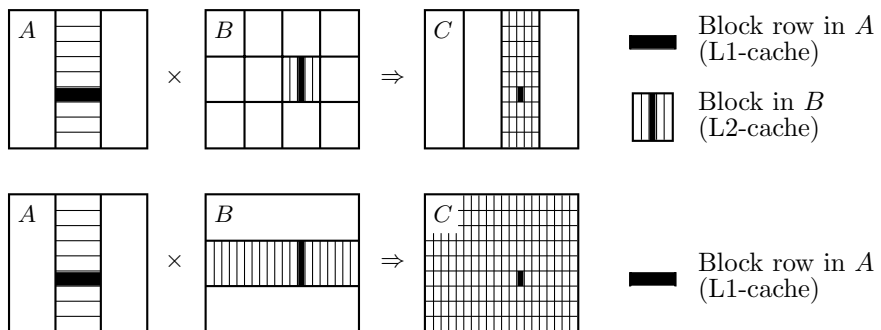


Fig. 3. Blocking strategies for matrix multiplication algorithm, with tiling (top) and without it (bottom)

For AMD processors, tiling is no more necessary because the new algorithm with "wide" block-rows tolerates the limited memory throughput. The measured memory throughput for different configurations is as follows:

- for dual Athlon MP1800+ (1530 MHz) processor with AMD762 chipset: up to 1300 MB/s (for 1 CPU), 1800 MB/s (for 2 CPUs);
- for Duron (900 MHz) processor with KT133A chipset: up to 1000 MB/s.

Therefore, only strip-mining can be applied, that makes the algorithm more simple and straightforward (Fig. 3, bottom). The width of strips is determined by the size of the L1 cache (more exactly, by half size of the cache) and is equal to 256.

For implementing a linear equation solver, the top-looking variant employing partial pivoting is used, with strip-mining and without further blocking [6,2] (Fig. 4).

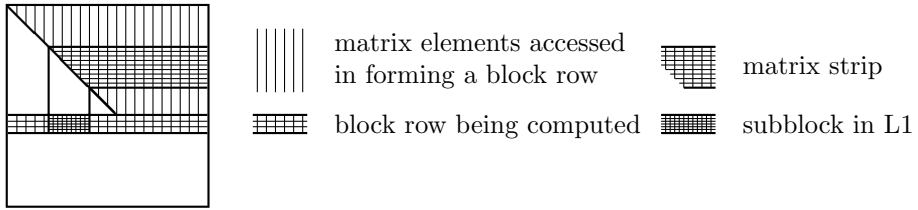


Fig. 4. Top-looking variant of LU-decomposition for solving linear systems

Basic steps of this LU-decomposition algorithm are as follows:

- Processing a trapezoidal strip (this is similar to the multiplication of a "wide" block row in L1 cache by a matrix strip);
- Solving a subsystem within a "wide" block with the following pivoting.

The first step is performed using the matrix multiplication algorithm as above. The second step, as well as the solution of triangular systems (using the computed matrix factors U and L) are performed separately. They are not time-consuming in comparison with the first step. Nevertheless, the resulting performance of the linear solver is less than that of the matrix multiplication.

5 Results and Comparisons with Other Approaches

Performance results of the new algorithms are presented on Fig 5 for Athlon MP1800+ processor (1530 MHz). Performance of the matrix multiplication algorithm reaches about 80 % of the peak computational speed (3060 MFLOPS) for small matrices that fit into the L2 cache (256 KB), and reduces to about 72 % for bigger matrices. For solving linear systems, performance is more monotonic and increases with the matrix size.

Performance results for Duron processor (900 MHz) are presented on Fig 6. The AMD Duron is the cheaper processor of the same architecture as the AMD Athlon, with the smaller L2 cache (64 KB), simpler memory subsystem and lower frequency. Its performance behavior is similar to that of the Athlon.

These results are compared on Fig 5 and Fig 6 with the results of the Athlon-optimized ATLAS software implementation [4]. The idea of the ATLAS approach is "Cache-contained matrix multiply" when matrices are split into square submatrices of such a size that several submatrices can fit into L1 cache simultaneously. Due to this, the amount of memory accesses can be reduced by a factor of the order of submatrix size. For the Athlon-optimized implementation, this size is equal to 30, that correspond to about 7 KB of memory (for 64-bit data elements).

For the comparison, the results for Intel MKL library are also presented. Here, the version of MKL library optimized for Pentium-III processor was used.

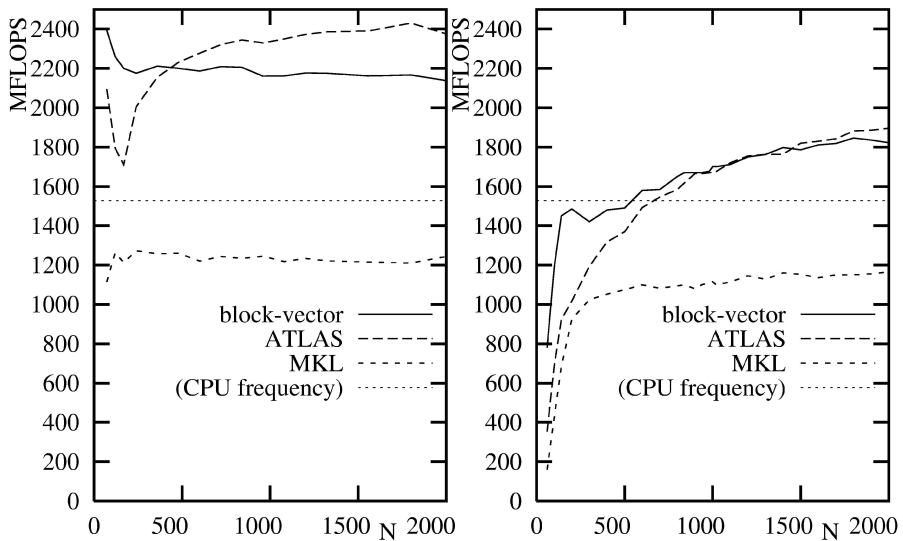


Fig. 5. Performance of different algorithms for matrix multiplication (left) and solving linear systems (right) on Athlon processor (1530 MHz)

Generally, P-III optimizations are quite applicable for AMD processors. However, the results of linear algebra routines from MKL are rather disappointing.

It can be seen that the new algorithm demonstrates competitive performance and wins on small and medium-size matrices. In particular, it outperforms the Athlon-optimized ATLAS for the LINPACK-1000 benchmark configuration (solving a linear system of the size 1000). For Athlon MP1800+ (1530 MHz) and Duron (900 MHz) processors the LINPACK-1000 results obtained with the new algorithm are 1705 MFLOPS and 977 MFLOPS respectively. These results are registered as record values (for the mentioned processors) in the database [3].

The best LINPACK-1000 results for Athlon MP1800+ (1530 MHz) using ATLAS are 1667 MFLOPS (obtained by the authors) and 1623 MFLOPS (obtained elsewhere), compared to 1705 MFLOPS for the new algorithm.

These LINPACK-1000 performance results achieved for Athlon processors are comparable to the performance results of some modern RISC microprocessors. For example, the results for the fastest Alpha processors 21264 (1250 MHz) and 21364 (1150 MHz) equipped with large caches and configured into expensive computers are 1945 MFLOPS and 1879 MFLOPS, respectively. This means that the modern inexpensive commodity microprocessors (like AMD Athlon) have become a very attractive alternative for performing scientific computations and building low-cost computer systems and clusters.

Performance of the new algorithm will be even higher on the last Athlon processors with frequencies up to 2250 MHz, as well as on the new AMD64 processors (Opteron and Athlon64) [8]. These new processors have the extended x86-64 architecture that includes some performance enhancements like SSE2

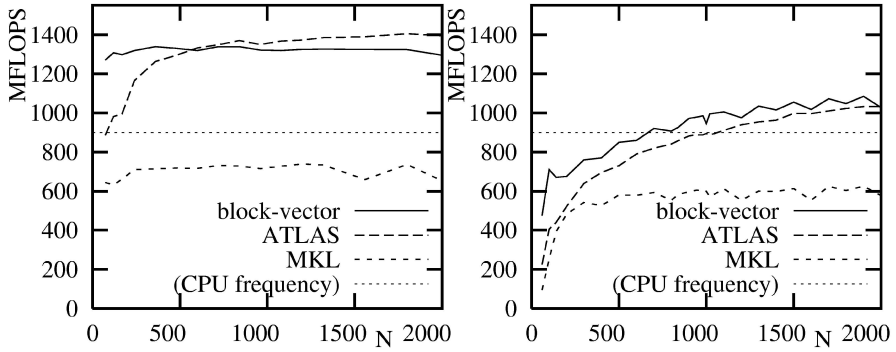


Fig. 6. Performance of different algorithms for matrix multiplication (left) and solving linear systems (right) on Duron processor (900 MHz)

Floating Point instructions and extended register sets. Due to this, more efficient implementations of the developed algorithms will become possible in a future.

Summary of the general properties of both approaches:

1. Cache-contained matrix multiply (ATLAS):

- minimizes the required memory access rate as much as possible;
- behaves better for very large matrices;
- is more complicated in implementation (e.g. in clean-up section for arbitrary matrix sizes);
- is convenient for incorporating into a "Self-adaptive software" [7].

2. Multiplication of block-vector by matrix (BLAS 2 $\frac{1}{2}$):

- minimizes the required memory access rate below some reasonable level;
- behaves better for small and medium-size matrices;
- is more simple in implementation;
- is more flexible for application to complicated matrix shapes (e.g. "triangular matrix multiply" for LU-decomposition).

The further optimization of new algorithms will be performed in a future, based on the results of this comparison. The combination of these two competing approaches ("cache-contained" and "block-vector") seems to be very attractive for implementing adaptive linear algebra kernels, to be used in parallel linear algebra software for clusters and MPPs.

6 Conclusion

The new developed methods and algorithms, described in this paper, combine the efficiency of large block approach with the flexibility and scalability of matrix-vector operations. They demonstrate the record level of performance for several

processor architectures, can be easily adapted to new architectures and incorporated into other serial and parallel libraries.

Implementation for x86/x87 CISC CPUs (AMD Athlon/Duron) tolerates low memory bandwidth and does not depend much on blocking strategies and outer cache levels. The record LINPACK-1000 results are obtained with this algorithm and registered in [3].

In comparison with the Athlon-optimized ATLAS implementation, our algorithm shows competitive performance and wins on some matrix sizes. It is more flexible for processing narrow matrices (block-vectors), that is convenient for efficient solution of linear systems. To achieve better results, these two approaches of building computational cores (block vector-matrix, and L1 cache-contained) may be combined.

As a result, ability to achieve multi-gigaflops performance on a single inexpensive commodity microprocessor increases attractiveness of x86 CISC architectures for building high-performance computing systems and clusters.

Acknowledgements. This work was partially supported by the program "Réseau de coopération universitaire et scientifique Franco-Germano-Russe" of the French Ministry of National Education, and by the Russian Foundation for Basic Research (grants RFBR-01-01-00745 and RFBR-02-01-00210). Measurements of AMD processors were performed with the help of Linagora SA (France), integrator of the L3M cluster, and with the support of AMD-HPC Europe.

References

1. Dongarra, J., Walker D.: The Design of Linear Algebra Libraries for High Performance Computers. LAPACK Working Note 58. University of Tennessee, Knoxville, TN (1993)
2. Bessonov, O., Fougère, D., Dang Quoc, K., Roux, B.: Methods for Achieving Peak Computational Rates for Linear Algebra Operations on Superscalar RISC Processors. In: Malyshkin, V. (ed): Proceedings / PaCT-99. Lecture Notes in Computer Science, Vol. 1662. Springer-Verlag, Berlin Heidelberg New York (1999) 180–185
3. Dongarra, J.: Performance of Various Computers Using Standard Linear Equations Software. Report CS-89-85. University of Tennessee, Knoxville, and ORNL, Oak Ridge, TN (2003)
4. Whaley, R.C., Petitet, A., Dongarra, J.: Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing* **27** (1–2) (2001) 3–35
5. AMD AthlonTM Processor x86 Code Optimization Guide. Advanced Micro Devices, Publication No. 22007 (February 2002)
6. Ortega J.M.: Introduction to Parallel and Vector Solution of Linear Systems. Plenum Press, New York (1988)
7. Chen, Z., Dongarra, J., Luszczek, P., Roche, K.: Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. LAPACK Working Note 160. University of Tennessee, Knoxville, TN (2003)
8. Software Optimization Guide for AMD AthlonTM 64 and AMD OpteronTM Processors. Advanced Micro Devices, Publication No. 25112 (April 2003)

Numerical Simulation of Self-Organisation in Gravitationally Unstable Media on Supercomputers^{*}

Elvira A. Kuksheva¹, Viktor E. Malyshkin², Serguei A. Nikitin³,
Alexei V. Snytnikov², Valery N. Snytnikov¹, and Vitalii A. Vshivkov⁴

¹ BIC SB RAS, Novosibirsk, Russia

`kuksheva@fir.catalysis.nsk.su`, `snyt@catalysis.nsk.su`,

² ICMMG SB RAS, Novosibirsk, Russia

`{malysh, snytav}@ssd.sccc.ru`,

³ BINP SB RAS, Novosibirsk, Russia `S.A.Nikitin@inp.nsk.su`,

⁴ ICT SB RAS, Novosibirsk, Russia `vsh@ict.nsc.ru`

Abstract. A numerical 3D-model for investigation of non-stationary processes in a gravitating system with gas is created. The model is based on the solution of the Poisson equation for gravitational field, the Vlasov-Liouville equation for solids and equations of gas dynamics. For solution of the Poisson equation at each timestep an efficient iterational solver is created with extrapolation of the evolutionary processes under study. It provides fast convergence at high precision.

Discussed in detail are parallelisation technique and load balancing strategy. Given are the parameters of test computations that meet the requirements of the problem of protoplanetary disc simulation.

1 Introduction

The level of knowledge about the surrounding world, which has been achieved by the present moment has intensified the interest to the problem of the origin of Life and, in particular, to the synthesis of prebiotic organic compounds. An intensive search is going on for an answer to the question of organic matter genesis on the Earth surface. What did the prebiotic world look like? What were the first stages of evolution?

Two hypotheses of prebiotic organic compounds genesis on Earth surface have been in use in scientific research up to the present moment. The first one is the Oparin-Holdein "primary bouillon" [3]. According to this hypothesis the abiogenous synthesis of prebiotic organic compounds occurred on the Earth surface. The primary atmosphere played an important role. According to the Arrhenius hypothesis [3] the Life appeared in the space outside the solar

^{*} The present work was particularly supported by SB RAS integration project (grants 2 and 148), INCO-COPERNICUS program (grant 97-7120), RFBR (grants 99-07-90422 and 02-01-00864) and Federal Programme "Integration" (contract Б - 0072/836)

System or exists together with the Universe and the first organic compounds or microorganisms migrated to the Earth surface in meteorite-comet rains or via interstellar dust.

According to the hypothesis proposed in [14], the abiogenous synthesis of primary organic compounds took place in the circumstellar disc during the accretion of the matter to the protostar before the planet formation.

In our research it was necessary to solve two first-order problems:

1. Find the mechanism of self-organisation of circumstellar disc that condenses the matter, destroys the disc and forms planets.
2. Determine the physical conditions: pressure, temperature and other parameters which could take place in circumstellar disc during its self-organisation. After determination of these conditions the further research becomes possible of the complex of chemical reactions of abiogenous synthesis of chemical compounds.

The solution of these problems is in the development of numerical model of gravitationally unstable dynamics and large-scale numerical experiments to study the mechanism of self-organisation of circumstellar disc. Self-organisation here means the transition from one quasi-stationary state to another through the development of instability including its non-linear stages and saturation. The stage sequence of self-organisation defines the development of the system. It is clear that the transition to the new state may involve only a part of the whole system.

The mechanism that we have proposed consists of two parts. Firstly, the mass of condensed phase (κ -phase) in circumstellar propoplanetary disc increases. It is due to catalytic synthesis of high-molecular organic compounds from the elements of highest prevalence. Secondly, the strong centrifugal effect enriches the disc with κ -phase relative to H_2 and He. When the κ -phase reaches its critical value the gravitational instability starts again, but now in two-phase medium with rotation in central field of a star. To confirm this idea it is necessary to solve the complete problem of gravitational instability in medium with two phases represented by gas and solid bodies, the medium residing in both central and self-consistent gravitational fields. The problem should be solved up to non-linear stages of the instability development. It is an essentially non-stationary problem with gas dynamics for H_2 and He, Vlasov-Liouville equation for κ -phase (the velocity distribution function depends on 7 variables), Poisson equation for self-consistent gravitational field with free mobile boundaries.

The only way to solve this 7D problem is the numerical simulation. For the integration of the Vlasov-Liouville equation in this high dimensionality the only known method is the method of large particles [7]. Because the development of a physical instability is under study the traditional methods of numerical mathematics as well as the fundamental Lax theorem for this problem do not work. The numerical method should be conceptually unstable in linear approximation. The escape from the situation was thought to be in following the fundamental conservation laws for mass impulse, angular momentum, energy, phase volume and CPT - theorem. But in the 1960s the developers of large particles method showed that it either saves angular momentum and breaks the energy conserva-

tion law or the opposite. Only one of its variants was in use in practice depending on the problem [5]. In our problem it is necessary to keep all the conservation laws, they are of the same importance. In [12] the ways to solve the problem were found. Moreover, the computers appeared in recent time enable us to conduct numerical experiments in reasonable time.

As usual this complex research is done by the mixed team of researchers. Physical and chemical aspects of the problem are developed by V.N.Snytnikov and S.A.Nikitin. Numerical model and experiments are done by A.V.Snytnikov and V.A.Vshivkov. Assembly technology of problem parallelisation and its parallel implementation was developed by V.E.Malyshkin. Visualisation and animation was done by E.A.Kusheva.

1.1 Model Size Estimate

Let us consider one of the ultimate cases of particle dynamics in infinite layer described by the Vlasov-Liouville equation. Linear analysis of this equation together with the Poisson equation leads us to the following dispersion relation for density waves in infinite layer

$$1 - \frac{2\pi G \Sigma_0}{kT} \left[1 + i\sqrt{\pi} \frac{\omega}{kv_T} W\left(\frac{\omega}{kv_T}\right) \right] = 0$$

where k is the wavenumber, ω is the wave frequency, T is the temperature, Σ_0 is the surface density, G is the gravitational constant, $W(z)$ is the Krump function, $v_T = \sqrt{2T}$ is the average thermal velocity of particles.

This equation was solved numerically by the tangent method in MathCAD2000. The dispersion function $Im\omega(k)$, that is, the dependence of imaginary part of frequency on wavenumber is displayed on fig. 1 as a solid line. Here $Im\omega(k) > 0$ means that the wave is unstable.

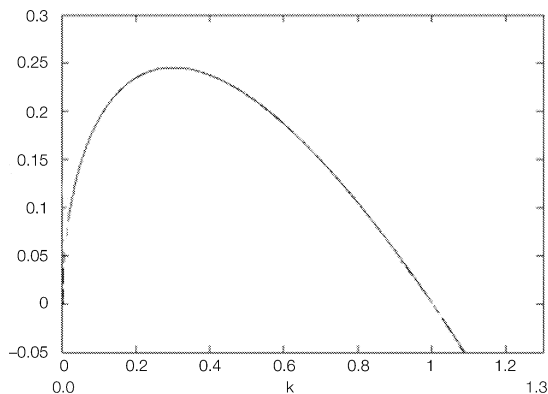


Fig. 1. Dispersion curve for infinite layer

Let us take into account the wavelength range that corresponds to the linear instability mode. The obtained solution shows that in order to solve the dynamics simulation problem in infinite layer with essentially unstable modes it is necessary to have also a wide range of physically stable collapsing modes.

The question is: what should the size of computational grid be to solve the simulation problem correctly?

First, we know from observations that the ratio of maximal unstable wavelength to the minimal one is about thirty. At least four nodes are needed to represent a harmonic function. It means that the minimal size of grid for unstable wavelength range is 120. Second, we can derive from the very simple case displayed on fig. 1 that the total computational domain size should be significantly larger than the unstable wavelength range. We know from experience that it should be at least two times larger. Thus we obtain the minimal grid size of 240 nodes in one direction. The exact necessary number of nodes could be defined only in numerical experiments. The grid with 240^3 nodes is at the limit of the workstation capacities and is likely insufficient to solve the problem.

The conclusion that could be derived is the following: to solve the problem of gravitational dynamics correctly, it is necessary to employ supercomputers.

1.2 Gravitational Solvers Review

A number of papers are known that are devoted to numerical solution of gravitational dynamics problems on supercomputers. They use different numerical techniques and parallelisation methods. The typical ones will be characterized in this section.

The first is the direct evaluation of the particles' interaction force, the so called "particle-particle" method or P^2 , [7]. This method is the most precise, but also the most time-consuming, its complexity is $O(N^2)$, N being the number of particles. Evaluation of a particle movement by this method requires positions of all the rest of particles. Thus a simulation employing more than 10^5 particles is impossible even with a supercomputer.

A variant of P^2 method is the treecode, the method in which closely situated particles are united in groups in order to treat them as one particle of corresponding mass in gravitational force evaluation. If such force approximation is inaccurate, the group is divided into smaller subgroups. Decomposition of particles into groups has the shape of an octal tree, which gives the name to the method. This method is much faster than P^2 , its complexity being $O(N \log N)$, but less accurate. Distribution of the processor workload is done during the tree setting [10].

According to the "particle-mesh" method, or PM [7] gravitational potential is computed via Poisson equation. The force acting at a particle is evaluated by interpolation of the grid values into the particle position. PM method is the fastest one, its complexity is only $O(NM)$, M being the number of grid nodes. For parallelisation of the method the main difficulty is in the Poisson equation solver. If the main time is consumed by computation of the movement of particles it is necessary to use dynamic load balancing [8].

P³M method is a combination of P² and PM methods: the interaction between closely situated particles is evaluated directly, whereas the force acting from distant particles is computed via Poisson equation. During parallelisation of the P³M method in [9] two different schemes of decomposition are involved: one to compute the interaction of particles (P² part) and the other to solve the Poisson equation (PM part). The complexity of this method varies from $O(NM)$ when density distribution is uniform to $O(N^2)$ when the matter is strongly clumped and evaluation of particle couple interaction takes the main time.

A combination of all mentioned methods is the novel method called TPM ("Treecode-Particle-Mesh") [11]. This algorithm is based on the fact that in cosmological case the density field could be broken into isolated dense subdomains. The trees corresponding each subdomain are distributed between the processors to make their workload equal.

Table 1. Comparison of gravitational solvers

Method	Grid size	Number of processors	Efficiency, ξ	Number of particles	Paper
P ²	-	112	-	3×10^3	[6]
PM	256^3	64	85 %	16.7×10^6	[4]
Treecode	-	16	93 %	10^5	[10]
P ³ M	1024^3	500	75 %	10^9	[9]
TPM	512^3	128	90 %	1.34×10^8	[11]

A comparison of implementation parameters for aforementioned methods is given in the table 1. Here the parallelisation efficiency is the ratio of computation speedup to increase of the number of processors. Let T_1 be the worktime on N_1 processors and T_2 the worktime on N_2 processors. Then parallelisation efficiency ξ is evaluated this way

$$\xi = \frac{T_1}{T_2} \frac{N_1}{N_2} \times 100\%$$

2 Source Equations

Vlasov—Liouville kinetic equation in the collisionless approximation of an averaged self-consistent field is written in the following form:

$$\frac{\partial f}{\partial t} + \mathbf{u} \frac{\partial f}{\partial \mathbf{r}} + \mathbf{a} \frac{\partial f}{\partial \mathbf{u}} = 0,$$

where $f(t, \mathbf{r}, \mathbf{u})$ is the depending on time (t) one-particle distribution function in coordinates ($\mathbf{r}$) and velocities ($\mathbf{u}$); $\mathbf{a} = -\nabla\Phi$ is the acceleration of a unit

mass particle. Gravitational potential Φ , where the movement takes place could be divided into two parts:

$$\Phi = \Phi_1 + \Phi_2,$$

where Φ_1 represents either the potential of rigid central mass (galactic black hole, protostar) or the potential of rigid matter system that is situated outside the disc plane (star, galactic halo, molecular cloud) depending on the simulated conditions. The second part of potential Φ_2 is determined by the common distribution of the moving particles and satisfies the Poisson equation $\Delta\Phi_2 = 4\pi G\rho$, which will be written in cylindrical coordinates chosen for the solution as

$$\frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial \Phi_2}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 \Phi_2}{\partial \varphi^2} + \frac{\partial^2 \Phi_2}{\partial z^2} = 4\pi G\rho.$$

The part of the model representing gas dynamics describes the dynamics of hydrogen-helium mixture. It consists of the following equations

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \text{div} \rho \mathbf{v} &= 0 \\ \rho \frac{\partial \mathbf{v}}{\partial t} + \rho (\mathbf{v} \nabla) \mathbf{v} &= -\nabla p - \rho \nabla \Phi + \mathbf{F}_{fr} \\ \rho \frac{dh}{dt} &= -\rho \nabla \Phi \mathbf{v} + \frac{\partial p}{\partial t} + Q_r + q \end{aligned}$$

These are the equations for alterations of mass, impulse and energy in a space point. Here $\mathbf{F}_{fr}$ is the volume friction force due to the solid component, h is the volume unit enthalpy, Q_r is the radiation absorption, q is the heat flux. The equations were simplified by outcasting the local chemical reactions, magnetic, electrodynamical and plasma effects, coagulation processes and radiation transport. Further simplification that could be made at the stage of model investigation is associated with the presumptions of isothermal and infinitesimally thin disc.

2.1 Infinitesimally Thin Isothermal Disc Model

In the case of an infinitesimally thin disc the volume density of the mobile medium ρ in all the volume is equal to zero ($\rho = 0$). At the disc itself a normal derivative shear of the potential occurs giving the boundary condition for potential determination Φ_2 :

$$\frac{\partial \Phi_2}{\partial z} = 2\pi G\sigma,$$

here σ is the surface density.

Initial distribution of the particle density is set according to the model of solid body rotation:

$$\sigma(r, \varphi) = \begin{cases} \sigma_c \sqrt{1 - \left(\frac{r}{r_0}\right)^2}, & r < r_0, \\ 0, & r \geq r_0, \end{cases}$$

where r_0 is the radius of the corresponding disc. The coefficient σ_c is chosen for the total mass to be equal to the given (M).

3 Numerical Methods

3.1 Vlasov-Liouville Equation Solution

To solve the Vlasov-Liouville kinetic equation the Particle-in-Cell (PIC) method is employed. At the initial moment the model particles of unique mass are placed in simulation domain for their number in each cell to be proportional to the density and the size of the cell. The velocity of particles is equal to the matter velocity at the corresponding point. The solution of particles movement equations is conducted by the well-known Boris scheme [2]. This scheme succeeds in satisfying the conservation laws of energy and angular momentum for an individual particle moving in the given central gravitational field.

3.2 Gas Dynamics Equations Solution

To solve the equations the gas dynamics the Fluid-in-Cell method ([7] — [13]) is employed. This method is the most concordant with the PIC method for the Vlasov-Liouville kinetic equation solution. The method enables to trace the gas-vacuum boundary and gives automatical satisfaction of mass and angular momentum conservation laws. The implemented variant of the scheme has the first precision order in spatial variables and time. At the scheme choice it was taken into account that the first order scheme from the given class possesses intrinsic viscosity which suppresses the computational dispersion. The long-wave disturbances are more unstable than the short-wave ones. Thus the occurrence of short-wave numerical fluctuations from the computational dispersion, their wavelength being similar to cell size, is thought to be worse than some dissipation of the function peaks. The absence of computational density fluctuations is necessary for evaluation of physical instabilities of gravitational type. Moreover, the schemes of higher order imply hard difficulties in being implemented at boundaries with vacuum.

3.3 Poisson Equation Solution

Poisson equation is solved on a grid in cylindrical coordinate system in order to take disc symmetry into account and rule out the non-physical structures appearing in Cartesian coordinates. It has the following form

$$\begin{aligned} & \frac{1}{h_r^2 r_{i-1/2}} \left[r_i (\Phi_{i+1/2, k-1/2, l} - \Phi_{i-1/2, k-1/2, l}) - \right. \\ & \quad \left. - r_{i-1} (\Phi_{i-1/2, k-1/2, l} - \Phi_{i-3/2, k-1/2, l}) \right] + \\ & + \frac{1}{h_\varphi^2 r_{i-1/2}^2} (\Phi_{i-1/2, k+1/2, l} - 2\Phi_{i-1/2, k-1/2, l} + \Phi_{i-1/2, k-3/2, l}) + \\ & + \frac{1}{h_z^2} (\Phi_{i-1/2, k-1/2, l+1} - 2\Phi_{i-1/2, k-1/2, l} + \Phi_{i-1/2, k-1/2, l-1}) = 0, \\ & i = 1, \dots, I_{max}, \quad k = 1, \dots, K_{max}, \quad l = 1, \dots, L_{max} - 1. \end{aligned}$$

where $I_{max}, K_{max}, L_{max}$ are the numbers of nodes in radial, angular and vertical coordinates correspondingly; i, k and l are the numbers of the grid node under consideration in these coordinates; $r_{i-1/2}$ is the radial coordinate of the i -th node; h_r, h_φ and h_z are the grid steps.

It is known that the system of linear algebraic equations obtained at the approximation of Poisson equation is ill-conditioned. Moreover, the conditionality gets worse when the step h decreases.

Due to this reason the direct methods (Gauss exclusion method, Fourier transform method) may accumulate a large uncontrolled error in the course of computation, that is critical for non-stationary problems solution. On the other hand, the iterational methods may require huge amounts of iterations. In order to avoid both troubles and obtain a robust Poisson equation solver a combined procedure is offered which incorporates both direct and iterational methods.

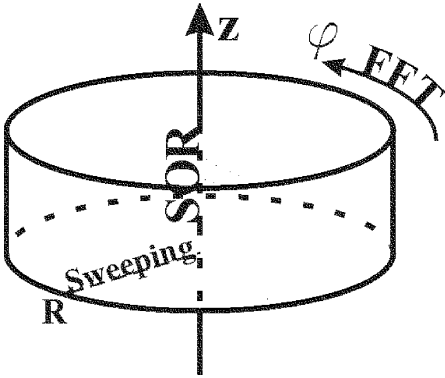


Fig. 2. Structure of Poisson equation solver

Fig. 2 shows the general scheme of the procedure. The first stage is the Fast Fourier Transform in the angular coordinate resulting in a system of linear algebraic equations. Each equation describes only one harmonic of the potential:

$$\begin{aligned} & \frac{1}{h_r^2 r_{i-1/2}} [r_{i-1} H_{i-3/2, l-1/2}(m) + r_i H_{i+1/2, l-1/2}(m)] + \\ & \frac{1}{h_z^2} [H_{i-1/2, l-3/2}(m) + H_{i-1/2, l+1/2}(m)] \\ & - \frac{2}{h_\varphi^2 r_{i-1/2}^2} \left[1 + 2 \sin^2 \frac{\pi m}{K_{max}} \right] H_{i-1/2, l-1/2}(m) \\ & = 4\pi R_{i-1/2, l-1/2}(m) \cos \frac{2\pi}{K_{max}} km \\ & m = 1, \dots, K_{max} \end{aligned}$$

where m is the number of harmonic or angular wavenumber, all the other symbols have the same meaning as in the previous equation. These equations are completely independent from each other, which is the most important fact.

During the second stage the two-dimensional equations are solved via the Successive Over-Relaxation method. Along the radial coordinate the sweeping procedure is applied to decrease the number of iterations. When convergence is reached, the inverse Fourier Transform is done on potential harmonics at the disc surface.

4 Parallelisation and Parallel Implementation of the Model

Assembly technology (AT) [8,15] was applied for problem parallelisation and its parallel implementation. AT is based on two key ideas. Firstly, the whole program is assembled out of the ready-made fragments of computations (procedures, for example), that should be small enough (i.e. consume few resources). Secondly, the program's fragmentation is kept in the course of computation. At any moment of time any fragment can be extracted from the running program and assigned for execution on another processor element (PE) of multicomputer. This results that executing program is represented by the set of asynchronous interacting processes. Communications define the neighbourhood relation on the set of processes. For execution processes are assigned on PEs of multicomputer keeping the neighbourhood relation, i.e. communicating processes are assigned on the same PE or on different PEs connected by the links. This provides implementation of interprocess communications with good performance. Additionally equal workload of every PE should be provided. It might be done because the fragments are small enough. If one of PEs is being overloaded, then some of processes should leave the overloaded PE and fly to neighbour underloaded PEs (diffusive dynamic load balancing). Many algorithms of dynamic load balancing are now known [8].

One of the challenges of parallelisation is to minimize data exchange between processors. The considered Poisson equation solver succeeds to avoid completely data exchange during the iteration stage. That is because equations for potential harmonics do not depend on each other. After iteration stage the potential should be gathered from all the processors for further computation. Therefore it is possible to divide the computation domain into completely independent subdomains along angular wavenumbers as shown in fig. 3. Computations inside a subdomain constitute a separate program's fragment. Domain decomposition is initially uniform, each PE gains equal number of harmonics.

Particles are also uniformly distributed between the PEs with no dependency on their spatial location. Since a particle might fly to any point of the disc in the course of simulation every PE should possess the potential values for the whole disc surface.

Interprocessor communications in the program were implemented with collective functions of the MPI library. At each timestep data exchange is performed

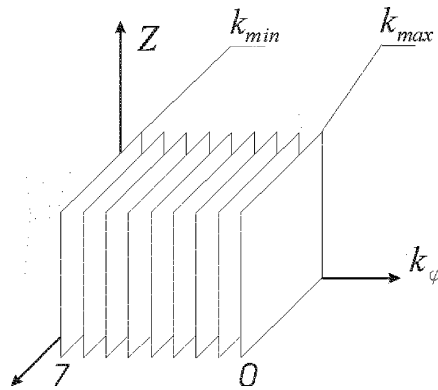


Fig. 3. Domain decomposition

twice. First, after the convergence is reached potential harmonics in the disc plain are gathered for inverse Fourier transformation. Then the partial density fields, computed in each PE, are added up.

The 2D equation systems for potential harmonics require different number of iterations for convergence. Here the number of iterations depends on the conditionality of the 2D equation system matrix. It means that the PEs would have different workload when provided with the same number of harmonics. Thus, initially equal workload can not be provided for all the PEs.

Initially equal number of harmonics are assigned to each PE. It follows from physical considerations that there could be the only PE, whose workload greatly exceeds the workload of the other PEs. After completion of the iterations at a timestep the average workload is calculated.

A PE is considered overloaded if its overload exceeds a threshold. Some harmonics should leave overloaded PE. Contrary to pure diffusive load balancing algorithm the harmonic that requires the minimal number of iterations is transferred to the most underloaded PE. The transfer process lasts while this PE is overloaded, or until only one harmonic is assigned to it. Physical peculiarity of the problem provides high quality of this dynamic load balancing algorithm.

Fig. 4 shows the speedup for several adjacent timesteps of a simulation (from 300-th to 312-th timestep) with dynamic load balancing and with static load setting.

5 Visualisation and Analysis

Important parts of numerical simulation are the development of visualisation techniques and software for simulation results analysis. Animation is a good solution of visualisation problem of non-stationary problems simulation. The developed software GalA solves a part of aforementioned problems. The source data for GalA is the collection of arrays produced by parallel programs and sequential program running on Athlon 1600. The preliminary data for the program

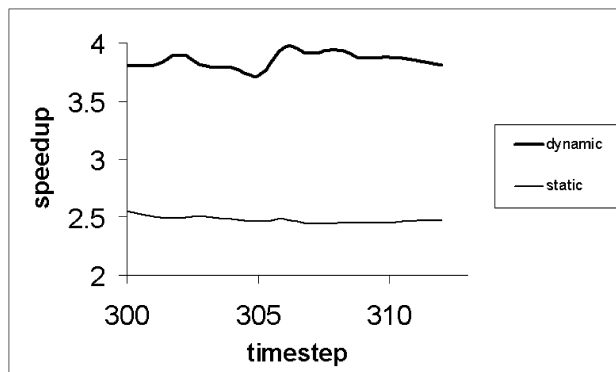


Fig. 4. Change of speedup due to dynamic load balancing

debug and verification were taken from the results of parallel program run on various supercomputers (see section 6).

At present GalA includes the opportunities for creation of 2D isolines, colour maps, vector fields, and also 3D-surfaces and isosurfaces. There is an opportunity of visualisation of the movement of the matter general mass on the plane. The parameters of visualisation such as colours, visualisation domain, size of the visualisation frame (the film) could be set on user choice. There is an opportunity to rotate and zoom the 3D images. Several output formats could be used, for example the well-known formats PNG, GPEQ, BMP, XPM, etc. GIF animation format is used for films. GNU C++ and Qt were employed for development of GalA. The software was developed for both Unix/X server and Windows. At present it is implemented under Unix/X server.

6 Numerical Experiments

Debug of the program was performed on a Linux workstation with two PentiumIII processors. The experiments were conducted on MVS-1000M supercomputer based on Alpha21264 processor at Siberian Supercomputer Centre, Novosibirsk (16 PE), at Joint Supercomputer Centre, Moscow (768 PE). Also the cluster of the Boreskov Institute of Catalysis SB RAS with 12 nodes of 2xPentiumIII was employed. Parameters of some computations like the number of processor elements (PE), grid size ($N_R \times N_\varphi \times N_Z$), number of particles and average worktime for one timestep are given in table 2. Also the computer is pointed out on which the experiment was performed.

Let us note that the cluster built on general purpose hardware (PentiumIII CPU and ScaLi communication network) like the present BIC cluster works much slower than MVS-1000M with Alpha21264 CPU and MIRINet communication network.

The simulation of protoplanetary disc evolution is performed on MVS-1000M at both Siberian Supercomputer Centre, Novosibirsk and Joint Supercomputer

Table 2. Parameters of numerical experiments

PE	Machine	Grid size			Number of particles	Worktime for one timestep
		N_R	N_φ	N_Z		
2	Cluster	120	128	100	10^6	29.7
2	MVS	400	512	200	2×10^7	19
4	MVS	400	512	200	2×10^7	11.1
8	MVS	400	512	200	2×10^7	7.2
8	MVS	400	512	200	10^8	28
8	Cluster	500	512	500	5×10^7	173.8
64	MVS	1000	1024	1000	10^9	141
128	MVS	500	1024	400	10^8	204
64	MVS	1000	2048	800	4×10^8	229.8
128	MVS	1000	2048	800	4×10^8	180
256	MVS	1000	2048	800	4×10^8	177.6

Centre, Moscow with the following typical parameters. The size of grid is 128.5×10^6 nodes ($N_R \times N_\varphi \times N_Z = 500 \times 512 \times 500$); the number of particles is 5×10^7 ; eight processors are employed.

The wall-clock time of a typical simulation is approximately 100 hours. The maximal time of continous computation at MVS-1000M is only 24 hours, so the simulation is interrupted several times. On an interrupt the program saves all the data and waits in queue for continuation of the simulation. Thus the real duration of a simulation is about two weeks.

Fig. 5 shows the speedup for the test grid having $400 \times 512 \times 200$ nodes with 2×10^7 particles.

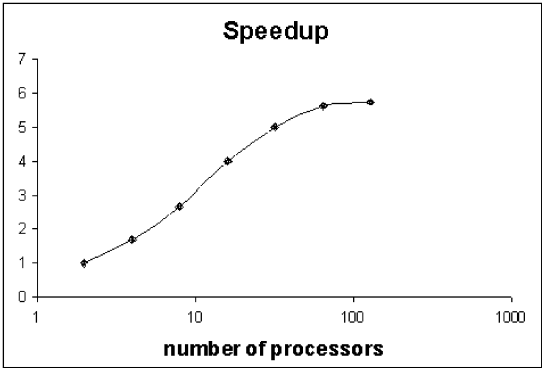


Fig. 5. Speedup for the $400 \times 512 \times 200$ nodes grid

Thus for small number of processors the efficiency of parallelisation was 85 % (fig. 5). The speedup becomes smaller as the number of processors increases

and for more than 128 processors almost disappears. Such a phenomenon is easily explained because the workloads of the processors are not equal in fact. The assignment of harmonics to processors is uniform, but different harmonics require different number of iterations as it is illustrated by the fig 6. Here the 10th timestep is shown, when the disc is near to axisymmetric form: few long harmonics dominate in computation time. Density distribution function initially has the axial symmetry. Therefore after FFT only 0-th harmonic is not equal to zero. Convergence for all the rest harmonics is reached in one iteration. It is clear that on the first timestep only one processor is working. In the course of simulation the symmetry is being lost and some time is required to compute all the harmonics. Still the long harmonics require more time than short ones.

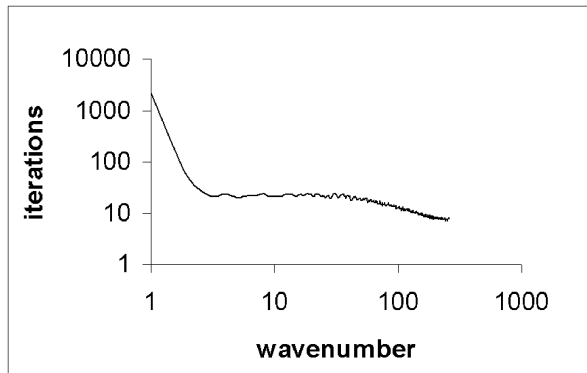


Fig. 6. Number of iterations depending on wavenumber

It should be noticed that speedup is different for different grids. On the plot above the difference in worktime between 64 and 128 is only 2 %. When the grid gets larger the computation time grows faster than communication time. The same difference for a larger grid with $1000 \times 2048 \times 800$ nodes is 10 % (see table 2).

7 Results and Discussions

One of the results of numerical simulation on MVS-1000M is shown in Fig 7. The density of particles is displayed in the equatorial plane. The scale is logarithmic, dark colour differs from the light one by four orders. The higher density is white. Everything below dark colour is displayed as black.

The particles rotate around the protostar with solid body distribution of velocities. Their initial density is set according to equilibrium, but it is yet supercritical. After a number of rotation cycles the accumulated perturbations are seen as macroscopic alterations of density along the rotation angle. Then the clumps of density are formed that move with low velocities. These clumps are

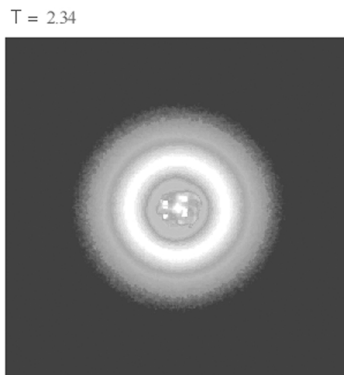


Fig. 7. Particle density in the equatorial plane

the density waves that can stand, move in the direction of rotation and in the opposite direction, towards the protostar and from it.

The particles fly into these waves with velocities that correspond to the rotation around the protostar and fly out of them. The gas temperature in density waves in circumstellar disc cannot slightly differs from the background temperature due to high heat conductivity of the prevalent hydrogen and helium.

The parameters of the simulation are listed in section 6. The pure computation time was 107.8 hours on MVS-1000M. It should be marked that the performed experiment is of intermediate-type character. The non-stationary process under study requires numerous experiments with large number of nodes and particles as well as with various boundary and initial conditions. Thus the real amount of computations within the present problem requires further development of tools for parallel numerical experiments on supercomputers.

8 Conclusions

The physico-chemical investigations were conducted and novel results in the cross-disciplinary research were produced.

Numerical model was created that enables the supercomputer simulation on investigation of self-organisation processes in circumstellar protoplanetary disc. Also the tools for visualisation of the produced data were developed.

Conducted were the numerical experiments with parameters meeting the requirements of the problem of protoplanetary disc simulation. These experiments make it possible to state that the developed program is competitive with world analogues. Parallelisation efficiency is 75 % up to 32 processors which is due to non-uniform processor workload. Whereas the dynamic load balancing increases the speedup close to the ideal value, and thus greatly improves efficiency.

References

1. Benz W.: Formation of planets: Problems and prospects. *Geochimica et Cosmochimica Acta*, Goldschmidt Conference, Davos, Switherland, August 18–23. (2002) A69.
2. Boris J.P.: Relativistic plasma simulation – optimization of a hybrid code. *Proc. Fourth Conf. Num. Sim. Plasmas*, Washington. (1970). 3–67.
3. Edited by Andre Brack. *The Molecular Origins of Life*. Cambridge University Press. (1998) 417.
4. Caretti E., Messina A.: Dynamic Work Distribution for PM Algorithm. <http://xxx.lanl.gov/astro-ph/0005512>, (2000).
5. Grigoryev Yu.N., Vshivkov V.A., Fedoruk M.P.: Numerical "Particle-in-Cell" Methods. Theory and Applications. VSP, Utrecht-Boston. (2002).
6. Griv E., Gedalin M., Liverts E., Eichler D., Kimhi Ye., Chi Yuan: Particle modeling of disk-shaped galaxies of stars on nowadays concurrent supercomputers. *NATO ASI "The Restless Universe"*, <http://xxx.lanl.gov/astro-ph/0011445>. (2000).
7. Hockney, R.W. and Eastwood, J.W.: *Computer Simulation Using Particles*, IOP Publishing, Bristol. (1988).
8. Kraeva M.A., Malyshev V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-multicomputers. *Future Generation Computer Systems*. Vol 17, number 6. (2001). 755–765.
9. MacFarland T., Couchman H.M.P., Pearce F.R., Pilchmeier J.: A New Parallel P3M Code for Very Large-Scale Cosmological Simulations. *New Astronomy*, <http://xxx.lanl.gov/astro-ph/9805096>. (1998).
10. Miocchi P., Capuzzo-Dolcetta R.: An efficient parallel tree-code for the simulation of self-gravitating systems. *A&A*, <http://xxx.lanl.gov/astro-ph/0104152>, (2001).
11. Bode P., Ostriker J., Xu G.: The Tree-Particle-Mesh N-body Gravity Solver. *ApJS*, <http://xxx.lanl.gov/astro-ph/9912541>. (1999).
12. Snytnikov V.N., Vshivkov V.A.: Correct Particles Method for Solving the Kinetic Vlasov Equation. *J. of Computational Mathematics and Mathematical Physics*. Vol.38, No.11. (1998). 1877–1883.
13. Snytnikov V.N., Vshivkov V.A., Dudnikova G.I., Nikitin S.A., Parmon V.N., Snytnikov A.V.: Numerical Simulation of N-body Gravitational Systems with Gas. *Vychislitel'nye Tehnologii* number 3, volume 7. (2002). 72–84.
14. Snytnikov V.N., Vshivkov V.A., Parmon V.N.: Solar Nebula as a Global Reactor for synthesis of Prebiotic Molecules. 11th Int. Conf. on the Origin of Life, Orleans, France, July 5–12. (1996)b 65.
15. Valkovski V.A., Malyshev V.E.: Parallel Program Synthesis on the Basis of Computational Models. *Nauka*, Novosibirsk, (1988) – in Russian.

Communication-Efficient Parallel Gaussian Elimination^{*}

Alexander Tiskin

Department of Computer Science
University of Warwick, Coventry CV4 7AL, UK
`tiskin@dcs.warwick.ac.uk`

Abstract. The model of bulk-synchronous parallel (BSP) computation is an emerging paradigm of general-purpose parallel computing. In this paper, we consider the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We define a common framework that unifies both problems, and present a new communication-efficient BSP algorithm for their solution. Apart from being a useful addition to the growing collection of efficient BSP algorithms, our result can be viewed as a refinement of the classical “parallelism-communication tradeoff”.

1 Introduction

The model of *bulk-synchronous parallel (BSP) computation* (see [19,10,12]) provides a simple and practical framework for general-purpose parallel computing. Its main goal is to support the creation of architecture-independent and scalable parallel software. Key features of BSP are its treatment of the communication medium as an abstract fully connected network, and strict separation of all interaction between processors into point-to-point asynchronous data communication and barrier synchronisation. This separation allows an explicit and independent cost analysis of local computation, communication and synchronisation.

In [18], we began the study of the BSP complexity of Gaussian elimination. In this paper, we make a further step in that direction, by considering the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We define a common framework that unifies both problems, and present a new communication-efficient BSP algorithm for their solution.

2 The BSP Model

A *BSP computer*, introduced in [19], consists of p *processors* connected by a communication network. Each processor has a fast *local memory*.

^{*} Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A BSP computation consists of S supersteps, with costs $w_s + h_s \cdot g + l$, $1 \leq s \leq S$, where g, l are parameters of the computer. The total cost of such computation is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=1}^S w_s$ is the local computation cost, $H = \sum_{s=1}^S h_s$ is the communication cost, and S is the synchronisation cost. The values of W , H and S typically depend on the number of processors p and on the problem size.

Papers [10,12] present the McColl–Valiant BSP algorithm for standard (non-Strassen) matrix multiplication, based on an idea from [1]. The BSP cost of this algorithm for $n \times n$ matrices is

$$W = O(n^3/p) \quad H = O(n^2/p^{2/3}) \quad S = O(1)$$

Paper [11] extends this result to fast (Strassen-type) matrix multiplication. The local computation, communication and synchronisation costs of the extended algorithm are $W = O(n^\omega/p)$, $H = O(n^2/p^{2/\omega})$, $S = O(1)$, where ω is the exponent of fast matrix multiplication (currently 2.376 by [5]).

Another important BSP algorithm, presented in [10], concerns the computation of *the cube dag*, which is a three-dimensional grid of nodes with sequential dependence between the nodes in each dimension. The BSP cost of this algorithm for cube dag of size n is

$$W = O(n^3/p) \quad H = O(n^2/p^{1/2}) \quad S = O(p^{1/2})$$

A communication-efficient BSP algorithm for Gaussian elimination without pivoting was presented in [18]; we review this algorithm in Section 3. In Section 4 we introduce a new communication-efficient BSP algorithm for Gaussian elimination with pairwise pivoting.

For the sake of simplicity, throughout the paper we ignore small irregularities that arise from imperfect matching of integer parameters. For example, when we say that an array of size n is divided equally across p processors, the value n may not be an exact multiple of p , and therefore the shares may differ in size by ± 1 . We use square bracket notation for matrices, referring to an element of an $n \times n$ matrix A as $A[i, j]$, $1 \leq i, j \leq n$.

3 Gaussian Elimination without Pivoting

This section and the next describe a BSP approach to Gaussian elimination, the method primarily used for direct solution of linear systems of equations. More generally, Gaussian elimination and its variations are applied to a broad spectrum of numerical, symbolic and combinatorial problems.

In this section we consider the simplest form of Gaussian elimination, which does not involve the search for pivots. This basic form of elimination is not guaranteed to produce the correct result, or to terminate at all, when performed on arbitrary matrices. However, it works well for matrices over some particular domains, such as closed semirings, or for matrices of some particular types, such as symmetric positive definite matrices over real numbers.

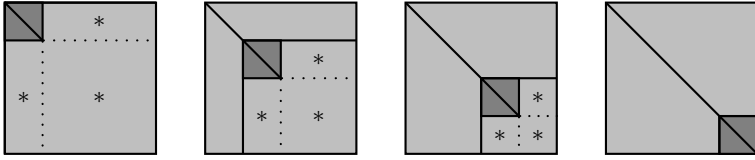


Fig. 1. Iterative block Gaussian elimination

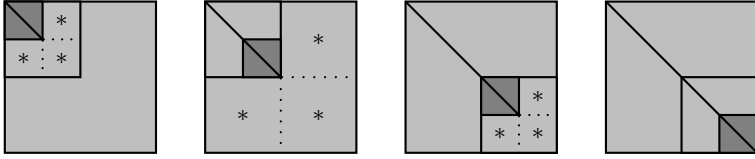


Fig. 2. Recursive block Gaussian elimination

Gaussian elimination can be represented in many ways. In this section we consider it in the form of LU decomposition. Let A be an $n \times n$ real diagonally dominant or symmetric positive definite matrix. The LU decomposition of A is $A = L \cdot U$, where L is an $n \times n$ unit lower triangular, and R is an $n \times n$ upper triangular matrix. This decomposition can be computed in sequential time $O(n^3)$ by plain Gaussian elimination, or in time $O(n^\omega)$ by block Gaussian elimination, using fast matrix multiplication.

The parallel complexity of Gaussian elimination has been extensively studied in various models of parallel computation. In [10] it is shown how to reduce the problem to the computation of a cube dag. The BSP cost of the resulting computation is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$ (see Section 2). The cube dag method is straightforward for LU decomposition, and can be easily adapted to other forms of Gaussian elimination, such as QR decomposition by Givens rotations.

Alternatively, we can apply a more general form of Gaussian elimination, replacing the elimination of single elements by the elimination of square blocks. The resulting method is known as *block Gaussian elimination*. Elimination within a block can be done by the standard pivoting-free algorithm, or take another form, e.g. employing column or full pivoting.

Figure 1 shows the process of block Gaussian elimination. Active blocks are denoted by small squares with darker shading. After block elimination, the resulting transformations must be applied to update the remaining matrix. The updated parts of the matrix are denoted in Figure 1 by asterisks.

In block Gaussian elimination, the block size must be chosen small enough, so that each iteration can be performed in $O(1)$ supersteps. When the size of the diagonal blocks is $n/p^{1/2}$, the BSP cost of the resulting algorithm is asymptotically equal to the cost of the cube dag method (see Section 2).

A lower communication cost for LU decomposition can be achieved by applying the block algorithm recursively (see e.g. [8,7]). This standard method was

suggested as a means of reducing the communication cost in [1] (for the transitive closure problem), and subsequently applied e.g. in [9]. The BSP cost of block Gauss–Jordan elimination was analysed in [18]; we summarise the results here for completeness.

Given a nonsingular matrix A , the algorithm produces the LU decomposition $A = L \cdot U$, together with the inverse matrices L^{-1} and U^{-1} . Matrices A , L , U are partitioned into regular square blocks of size $n/2$,

$$A = L \cdot U : \quad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \cdot \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \end{pmatrix} \quad (1)$$

where the dot $\cdot$ indicates a zero block. The algorithm proceeds as follows:

1. Apply the algorithm recursively to obtain the LU decomposition of the block $A_{11} = L_{11} \cdot U_{11}$, along with the inverse blocks L_{11}^{-1} and U_{11}^{-1} .
2. Apply these inverse blocks to obtain L_{21} , U_{12} :

$$L_{21} \leftarrow A_{21} \cdot U_{11}^{-1} \quad U_{12} \leftarrow L_{11}^{-1} \cdot A_{12} \quad (2)$$

3. Apply the algorithm recursively to obtain the LU decomposition $A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$, along with the inverse blocks L_{22}^{-1} , U_{22}^{-1} .
4. Obtain the inverse matrices L^{-1} , U^{-1} by

$$L^{-1} = \begin{pmatrix} L_{11}^{-1} & \cdot \\ -L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1} & L_{22}^{-1} \end{pmatrix} \quad U^{-1} = \begin{pmatrix} U_{11}^{-1} & -U_{11}^{-1} \cdot U_{12} \cdot U_{22}^{-1} \\ \cdot & U_{22}^{-1} \end{pmatrix} \quad (3)$$

Figure 2 shows the process of recursive block Gaussian elimination. As before, active blocks are denoted by small squares with darker shading, and the updating by asterisks.

We now describe the allocation of block processing and block multiplication tasks in (2)–(3) to the BSP processors. In each level of recursion, every block multiplication in (2)–(3) is performed in parallel by all p processors. Each block LU decomposition is assigned to all p processors, if the block size is large enough. When blocks become sufficiently small, block LU decomposition is computed sequentially by an arbitrarily chosen processor.

The depth at which the algorithm switches from p -processor to single-processor computation can be varied, allowing us to trade off the costs of communication and synchronisation in a certain range. In order to account for this tradeoff, we introduce a real parameter α , controlling the depth of recursion. The algorithm is as follows.

Algorithm 1 (Gaussian elimination without pivoting).

Parameters: integer $n \geq p$; real number α , $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.

Input: $n \times n$ real matrix A ; we assume that A is diagonally dominant or symmetric positive definite.

Output: decomposition $A = L \cdot U$, where L is an $n \times n$ unit lower triangular matrix, and U is an $n \times n$ upper triangular matrix.

Description. The computation is defined by recursion on the size of the matrix. Denote the size of the block at the current level of recursion by m , keeping n for the original matrix size. Let $n_0 = n/p^\alpha$. Value $m = n_0$ is the threshold at which the algorithm switches to sequential computation. The recursion proceeds as described earlier in this section.

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(m^3/p)$	$O(n_0^3)$
$H(m) =$	$2 \cdot H(m/2) + O(m^2/p^{2/3})$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \quad H = O(n^2/p^\alpha) \quad S = O(p^\alpha) \quad \blacksquare$$

For $\alpha = \alpha_{\min} = 1/2$, the cost of Algorithm 1 is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. This is asymptotically equal to the BSP cost of the cube dag method from [10]. For $\alpha = \alpha_{\max} = 2/3$, the cost of Algorithm 1 is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(p^{2/3})$. In this case, the communication cost is as low as in matrix multiplication. This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. For large n , the communication cost of Algorithm 1 dominates the synchronisation cost, and therefore the communication improvement should outweigh the loss of synchronisation efficiency. This justifies the use of Algorithm 1 with $\alpha = \alpha_{\max} = 2/3$. Smaller values of α , or the cube dag algorithm, should be considered when the problem is moderately sized.

Fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The resulting algorithm has BSP cost $W = O(n^\omega/p)$, $H = O(n^2/p^\alpha)$, $S = O(p^\alpha)$, where $\alpha_{\min} = 1/(\omega - 1) \leq \alpha \leq 2/\omega = \alpha_{\max}$. As ω approaches the value of 2, the range of parameter α becomes tighter. If an $O(n^2)$ matrix multiplication algorithm is eventually discovered, the tradeoff between H and S will disappear.

4 Pairwise Pivoting and Givens Decomposition

In the previous section, we described an efficient parallelisation of the basic, pivoting-free variant of Gaussian elimination. This method works well in situations where pivoting is not necessary. However, in most cases some form of pivoting is essential. This is the case e.g. for numerical matrices without special structure, or for matrices over a finite field. In this section, we extend our Gaussian elimination algorithm with a simple, but useful, version of pivoting applicable in such cases.

Consider, for instance, the case of a matrix A over a finite field. We assume that field operations, including inversion of a nonzero element, can be performed in constant time. Plain Gaussian elimination without pivoting may fail on matrix

A , since it requires the inversion of diagonal elements, some of which may be zero initially, or can become zero during elimination. Block Gaussian elimination without pivoting may fail for a similar reason, if some diagonal blocks that need to be inverted are singular or become singular in the course of the elimination.

A standard answer to such concerns in the case of numerical matrices is using column or full pivoting to achieve numerical stability. However, since the computation in our case is performed over a finite field, any elimination method would be suitable as long as it only inverts elements known to be nonzero. We can use this additional freedom of choosing the elimination pattern, in order to save on the BSP communication and synchronisation costs.

As the initial example, consider a 2×1 matrix $A = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$. To eliminate its bottom element a_2 , we pre-multiply matrix A by a 2×2 transformation matrix:

$$\begin{pmatrix} 1 & \cdot \\ -a_2/a_1 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ \cdot \end{pmatrix} \text{ if } a_1 \neq 0$$

$$\begin{pmatrix} \cdot & 1 \\ 1 & \cdot \end{pmatrix} \begin{pmatrix} 0 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_2 \\ \cdot \end{pmatrix} \text{ if } a_1 = 0$$

In the more general case of a $2 \times n$ matrix A , the above transformation will create a zero in the bottom-left corner, either by modifying the bottom row (if the top-left element is nonzero), or by swapping the rows (if the top-left element is zero). For a general $m \times n$ matrix, several such transformations on blocks of size $2 \times n$ can be carried out simultaneously. This technique is known as *pairwise pivoting* (see e.g. [17,8]).

A similar pattern can be applied to perform elimination on numerical matrices. In this case, we have to be more careful in choosing the transformation, for reasons of numerical stability. A standard, numerically stable transformation known as *Givens rotation* is defined as

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \cdot \end{pmatrix}$$

where $c = a_1/(a_1^2 + a_2^2)^{1/2}$, $s = a_2/(a_1^2 + a_2^2)^{1/2}$, $b_1 = (a_1^2 + a_2^2)^{1/2}$. Again, for a general $m \times n$ matrix, several such elementary transformations can be performed in parallel on blocks of size $2 \times n$. Givens reduction for parallel solution of linear systems was proposed in [15] (see also [6]). Paper [4] describes a BSP algorithm for Givens reduction; however, its communication, and especially synchronisation efficiency is relatively low.

Since the above two examples share the same elimination pattern, we will refer to both by the term “pairwise pivoting”. For our purposes it is convenient to consider, without loss of generality, the elimination process on a $2n \times n$ matrix $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$, where A_1 is an arbitrary $n \times n$ matrix, and A_2 an upper triangular $n \times n$ matrix. Matrices A_1 , A_2 may not have full rank. The problem consists in finding a full-rank $2n \times 2n$ transformation matrix D , and an $n \times n$ upper triangular matrix B_1 , such that

$$D \cdot \begin{array}{|c|} \hline A_1 \\ \hline A_2 \\ \hline \end{array} = \begin{array}{|c|} \hline B_1 \\ \hline \\ \hline \end{array} \quad (4)$$

This problem is closely related to the problem of transforming a matrix to row echelon form (see [3]).

Similarly to the pivoting-free case, the problem can be solved in the BSP model by the cube dag method, using the standard elimination scheme (see e.g. [15,13,14]). Alternatively, we can apply a more general form of pairwise pivoting, replacing the elimination of 2×1 blocks by the elimination of $2k \times k$ blocks. We call the resulting method *block-pairwise elimination*. Elimination within a block can either be done by the standard algorithm with pairwise pivoting, or take another form, e.g. employing column or full pivoting.

Figure 3 shows the process of block-pairwise elimination on a $2n \times n$ matrix. Active blocks are denoted by small rectangles, with the eliminated part marked by darker shading. Blocks that are active simultaneously form *the wavefront*, which is, when viewed at high level, a straight sloping line $2j - i = \text{const}$. After block elimination, the resulting transformations must be applied to update the remaining matrix. The updated parts of the matrix are denoted in Figure 3 by asterisks.

Block-pairwise elimination can be efficiently implemented in the BSP model as follows. We set the block size to $2n/p^{1/2} \times n/p^{1/2}$. The computation proceeds in $2p^{1/2} - 1$ stages. In each stage, at most $p^{1/2}$ blocks are eliminated in one superstep. The updating process can be partitioned into at most p independent multiplications of square matrices of size $2n/p^{1/2}$. Therefore, the BSP cost of a stage is $W = O(n^3/p^{3/2})$, $H = O(n^2/p)$, $S = O(1)$. The overall BSP cost of the block-pairwise elimination algorithm is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. It is asymptotically equal to the cost of the cube dag algorithm (see Section 2) and of iterative pivoting-free block Gaussian elimination (see Section 3).

Similarly to the recursive pivoting-free algorithm described in Section 3, block-pairwise elimination can also be applied recursively (this was first proposed in [16]; see also [3, section 16.5]). The difference from the recursive pivoting-free algorithm is that now we cannot, in general, compute block inverses. After partitioning matrix $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ into regular square blocks of size $n/2$, the algorithm proceeds as follows:

1. Apply the algorithm recursively to the $n \times n/2$ matrix $\begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix}$. Apply the resulting transformation matrix to update $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix}$:

$$E \cdot \begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline A_{31} & A_{32} \\ \hline & A_{42} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A'_{21} & A'_{22} \\ \hline & A'_{32} \\ \hline & A_{42} \\ \hline \end{array} \quad (5)$$

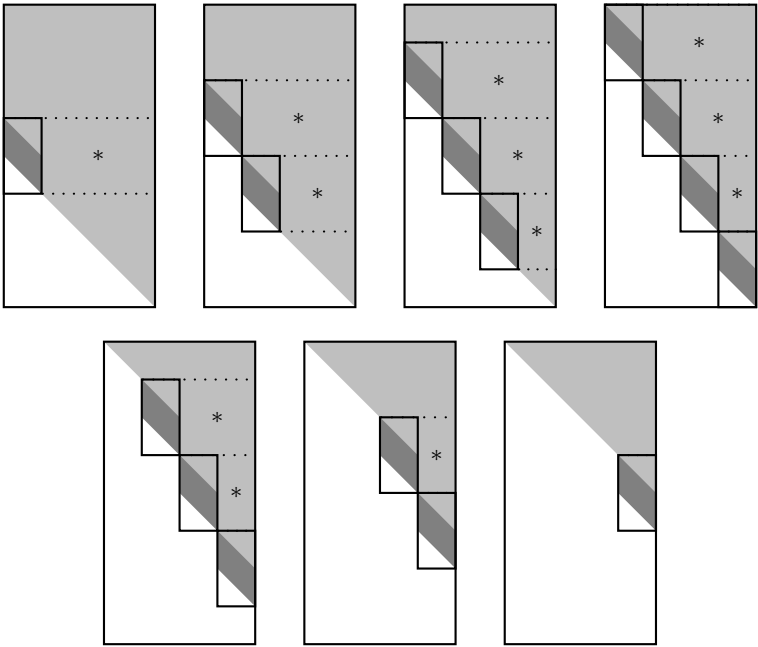


Fig. 3. Iterative block-pairwise elimination

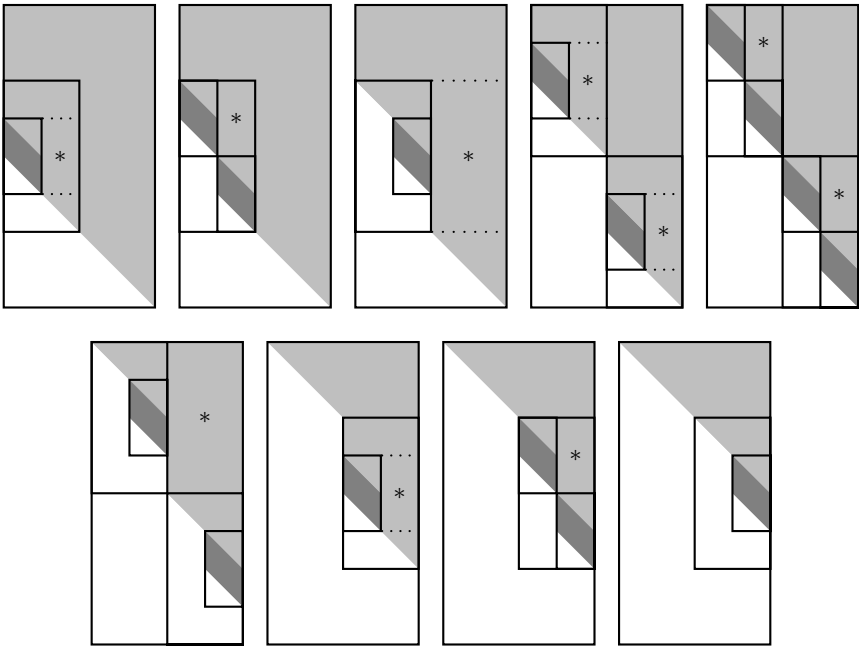


Fig. 4. Recursive block-pairwise elimination

2. Apply the algorithm recursively to matrices $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ and $\begin{pmatrix} A'_{32} \\ A_{42} \end{pmatrix}$. Apply the first resulting transformation matrix to update $\begin{pmatrix} A_{12} \\ A'_{22} \end{pmatrix}$:

$$E' \cdot \begin{pmatrix} A_{11} & A_{12} \\ A'_{21} & A'_{22} \\ & A'_{32} \\ & A_{42} \end{pmatrix} = \begin{pmatrix} A''_{11} & A''_{12} \\ & A''_{22} \\ & A''_{32} \end{pmatrix} \quad (6)$$

3. Apply the algorithm recursively to matrix $\begin{pmatrix} A''_{22} \\ A''_{32} \end{pmatrix}$:

$$E'' \cdot \begin{pmatrix} A''_{11} & A''_{12} \\ & A''_{22} \\ & A''_{32} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ & B_{22} \end{pmatrix} \quad (7)$$

4. Obtain the transformation matrix D in (4) as the product of the three transformation matrices from (5)–(7): $D = E'' \cdot E' \cdot E$.

Figure 4 shows the process of recursive block-pairwise elimination, using the same conventions as in Figure 3. Note that, in contrast with Figure 3, there is no straight-line wavefront.

Overall, the above method resembles recursive pivoting-free Gaussian elimination (Section 3, Algorithm 1); however, it requires four, rather than two, recursive calls on half-sized matrices. Consequently, the communication cost is higher than in the pivoting-free case. Moreover, the recursion has to be significantly deeper to achieve work-optimality, which leads to an increase in the synchronisation cost. We leave it as an exercise for the reader to check that the best synchronisation cost achievable under the work-optimality condition $W = O(n^3/p)$ is $S = O(p^{\log 3}) \approx O(p^{1.585})$, and to work out the corresponding communication cost H .

The inefficiency of the BSP algorithm based on the above approach motivates us to look for an alternative method, that would play for pairwise elimination the role that Algorithm 1 plays for pivoting-free elimination. We present such a method in the rest of this section.

It is clear that, although the recursive blocking approach based directly on [16] fails to provide an efficient BSP algorithm, any alternative method would need to use some form of blocking. One of the first blocked algorithms for Gaussian elimination with pivoting was proposed in [2]. Unfortunately, the quite complicated pivoting scheme of [2] does not lead to any improvement in BSP cost over the iterative block-pairwise elimination. Other existing blocking methods operate on whole columns (or, symmetrically, whole rows) of the matrix (see e.g.

[8]). Such methods are suitable for row or column pivoting, but this comes at the price of a significantly higher communication cost.

In [18], we proposed a BSP algorithm that improves on the iterative block-pairwise elimination; however, its cost is still a polylogarithmic factor higher than the asymptotic cost of the pivoting-free algorithm (Algorithm 1). In the current work, we close this gap, and present an algorithm that achieves the cost asymptotically equal to that of the pivoting-free algorithm. In particular, our algorithm is also work-optimal, and also exhibits a tradeoff between communication and synchronisation.

From our discussion of the standard pairwise, the iterative block-pairwise, and the recursive block-pairwise elimination schemes, it is apparent that an efficient algorithm should maintain a relatively straight wavefront. To achieve that, our main idea is as follows: instead of operating on rectangular blocks, we proceed recursively on strips delimited by sloping lines $2j - i = \text{const}$. We call every such line a *pseudo-diagonal*.

For the presentation of our algorithm, it is convenient to consider elimination with pairwise pivoting on a matrix whose elements are zero below a given pseudo-diagonal $2j - i = r$. We call such a matrix *pseudo-triangular*. For simplicity of exposition, we describe our algorithm in terms of semi-infinite strips, unbounded in the positive direction of i, j . Actual computation will only be required in the region delimited by matrix boundaries. Let $A\langle k : l \rangle_b$ denote a semi-infinite strip in matrix A , that consists of elements $A[i, j]$ with $i \geq b$ and $k \leq 2j - i < l$. The value $l - k$ will be called *the width* of the strip.

The input of our algorithm is a strip $A\langle r : r + 2d \rangle_r$ for given r, d . The algorithm eliminates the lower half-strip $A\langle r : r + d \rangle_r$, apart from its upper triangle, which cannot be eliminated:

$$D \cdot \begin{array}{c} \text{[Diagram: A parallelogram representing a strip with a shaded lower half and an unshaded upper half, bounded by two parallel diagonal lines.]} \end{array} = \begin{array}{c} \text{[Diagram: A parallelogram representing a strip with a shaded lower half and an unshaded upper half, bounded by two parallel diagonal lines, similar to the first but with different shading.]} \end{array} \quad (8)$$

In the course of its execution, the algorithm also modifies (but does not eliminate) the upper half-strip $A\langle r + d : r + 2d \rangle_r$. The resulting transformation matrix D is banded of bandwidth $4d$. The part of the input matrix above the strip is left intact. For simplicity of exposition, we sacrifice some constant-factor efficiency by discarding the modifications made by the algorithm to the strip, and considering only the matrix D as the output of the algorithm. The matrix in the right-hand side of (8) can then be obtained by taking the product in the left-hand side, which recomputes the discarded modifications, and also modifies the previously untouched parts of matrix A . This updating pattern is applied in every level of recursion. In practice, recomputation is not necessary, but avoiding it requires a more complicated control structure than the one presented here.

Gaussian elimination on a square matrix can be reduced to the above problem by embedding an $n \times n$ matrix into a strip $A\langle 1 : 1 + 4n \rangle_1$, so that the top-left corner of the embedded matrix is at $A[n, n]$, and all elements outside the embedded matrix are set to zero.

The algorithm proceeds on the strip $A\langle r : r + 2d \rangle_r$ as follows:

1. Save the initial state of the strip $A\langle r : r + 2d \rangle_r$.
2. Apply the algorithm recursively to the lower half-strip $A\langle r : r + d \rangle_r$, reducing the quarter-strip $A\langle r : r + d/2 \rangle_r$ to upper-triangular form, and modifying the quarter-strip $A\langle r + d/2 : r + d/2 \rangle_r$. Just before returning from the recursive call, the algorithm discards the modifications and reverts the half-strip $A\langle r : r + d \rangle_r$ to its state previous to stage 2. The result of the recursive call is the transformation matrix E of bandwidth $4d$.
3. Apply the transformation matrix E to the current strip $A\langle r : r + 2d \rangle_r$, registering the result only in the middle half-strip $A\langle r + d/2 : r + 3d/2 \rangle_{r+d/2}$. The correct updating of the whole strip would require access to matrix entries outside the current strip; we have chosen to update only the middle half-strip so that such a situation can be avoided.
4. Apply the algorithm recursively to the middle half-strip $A\langle r + d/2 : r + 3d/2 \rangle_{r+d/2}$, reducing the quarter-strip $A\langle r + d/2 : r + d \rangle_{r+d/2}$ to upper-triangular form, and modifying the quarter-strip $A\langle r + d : r + 3d/2 \rangle_{r+d/2}$. Just before returning from the recursive call, the algorithm discards the modifications and reverts the half-strip $A\langle r + d/2 : r + 3d/2 \rangle_{r+d/2}$ to its state previous to stage 4. The result of the recursive call is the transformation matrix E' of bandwidth $4d$.
5. Revert the current strip $A\langle r : r + 2d \rangle_r$ to the state of stage 1.
6. Obtain the resulting transformation matrix as the product of the above two transformation matrices: $D = E' \cdot E$. Matrix D is of bandwidth $8d$.

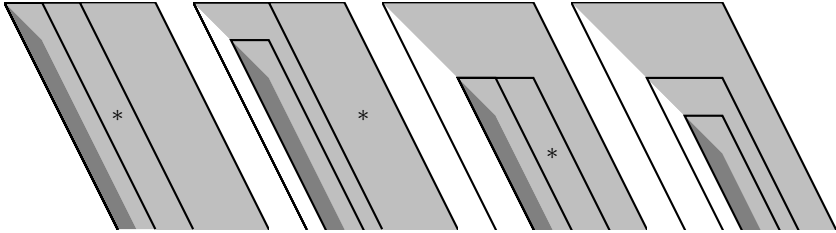


Fig. 5. New algorithm

Figure 5 shows the process of recursive block-pairwise elimination by our algorithm. Due to the recursive nature of the algorithm, the strips in each panel of Figure 5 are overlapped; for example, the first panel shows the overlapping strips $A\langle r : r + 2d \rangle_r$, $A\langle r : r + d \rangle_r$, $A\langle r : r + d/2 \rangle_r$, and the last panel the strips $A\langle r : r + 2d \rangle_r$, $A\langle r + d/2 : r + 3d/2 \rangle_{r+d/2}$, $A\langle r + 3d/4 : r + 5d/4 \rangle_{r+3d/4}$. The eliminated part of the matrix is marked, as usual, by darker shading. The updated parts of the matrix are denoted by asterisks. More precisely, an asterisk indicates the current strip in stage 3 of the algorithm (for convenience of presentation, each asterisk is shifted to the upper half of its strip). The result of the update is registered in the middle half-strip of the current strip. This middle half-strip becomes the current strip in the subsequent recursive call.

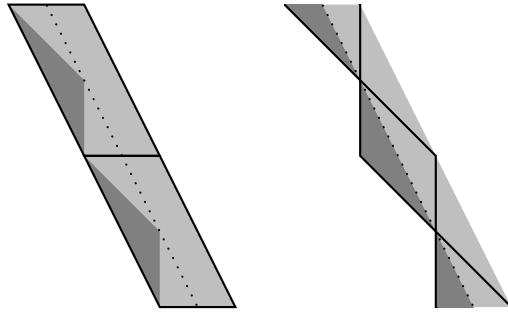


Fig. 6. New algorithm: the recursion base

In contrast with the pivoting-free case, where we could afford to process sufficiently small blocks sequentially, we cannot do so for strips without losing communication efficiency. Therefore, we need to provide a base for our recursion, by specifying a parallel procedure for block-pairwise elimination on a sufficiently narrow strip. Such a procedure is illustrated in Figure 6. For simplicity, the boundaries of the matrix are not shown: the panels represent a “typical” region of the input matrix, far away from the boundaries and not overlapping with the non-eliminated upper triangular part. The processing of the strip at the base of the recursion is performed as follows:

1. Save the initial state of the strip.
2. Partition the strip into parallelogram-shaped blocks, as in Figure 6 (left). In each block, eliminate a triangular “dent” by standard pairwise-pivoting elimination. It is straightforward to check that elimination of the “dents” (as opposed to elimination of the whole lower half-strip) and the computation of the transformation matrix (which in this case is block-diagonal) can be performed using only data local to the parallelogram blocks. After computing the transformation matrix, revert the strip to the state of stage 1.
3. Apply the transformation matrix as in the recursive procedure.
4. Repartition the strip into different parallelogram-shaped blocks, as in Figure 6 (right). Eliminate the lower triangle of each block. Again, it is straightforward to check that the elimination and the computation of the transformation matrix (which is again block-diagonal) can be performed using only data local to the blocks. After computing the transformation matrix, revert the strip to the state of stage 1.
5. Obtain the resulting transformation matrix as the product of the above two transformation matrices.

Note that this procedure is for the recursion base only. It cannot be applied recursively itself, since that would lead to a “ragged” wavefront, something that our recursive procedure has been specifically designed to avoid.

Throughout the algorithm, we multiply banded transformation matrices by sloping matrix strips, as well as by other banded matrices. It remains to verify that such products can be computed efficiently in the BSP model. Consider the

multiplication of two such structured matrices, each of which has size $n \times n$ with $O(nm)$ nonzero entries (here m is the bandwidth or the width of the strip). Using the idea of the McColl–Valiant algorithm (Section 2), we can represent the array of elementary products by an $n \times n \times n$ cube. Out of these n^3 elementary products, only $O(nm^2)$ are nontrivial. Partition the cube into $(n/m)^2 \cdot p$ regular cubic blocks of size $n^{1/3}m^{2/3}/p^{1/3}$. The structure of the matrices is such that only $O(p)$ blocks contain nontrivial elementary products. Therefore, the multiplication can be performed on a BSP computer with computation cost $W = O(nm^2/p)$, communication cost $H = O(n^{2/3}m^{4/3}/p^{2/3})$, and synchronisation cost $O(1)$.

We now describe the allocation of strip processing and matrix updating tasks to the BSP processors. In each level of recursion, the structured matrix multiplication tasks are performed in parallel by all p processors. Each strip processing task is assigned to all p processors, if the strip is large enough. When the strip becomes sufficiently small, we follow the recursion base procedure, allocating every block of the strip to a different processor.

Similarly to the pivoting-free case (Algorithm 1), the depth at which the algorithm switches to the recursion base procedure can be varied, resulting in a tradeoff between the communication and synchronisation costs. As before, we introduce a real parameter α , controlling the depth of recursion. The algorithm is as follows.

Algorithm 2 (Gaussian elimination with pairwise pivoting).

Parameters: integer $n \geq p$; real number α , $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.

Input: $n \times n$ matrix A .

Output: decomposition $D \cdot A = B$, where D is a full-rank $n \times n$ transformation matrix, and B is an $n \times n$ upper triangular matrix. This generic form of decomposition captures e.g. LU decomposition of matrices over a finite field, or QR decomposition of numerical matrices.

Description. Matrix A is embedded in a suitable pseudo-triangular matrix with a leading strip of width $4n$. The computation is then defined by recursion on the width of the strip. Denote the strip width at the current level of recursion by m , keeping $4n$ for the original width. Let $n_0 = n/p^\alpha$. Value $m = n_0$ is the threshold at which the algorithm switches to the recursion base procedure. The recursion proceeds as described earlier in this section.

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(n \cdot m^2/p)$	$O(n_0^3)$
$H(m) =$	$2 \cdot H(m/2) + O(n^{2/3} \cdot m^{4/3}/p^{2/3})$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \quad H = O(n^2/p^\alpha) \quad S = O(p^\alpha) \quad \blacksquare$$

Considerations similar to the ones discussed in Section 3 apply to the choice of a particular value of α .

Similarly to the pivoting-free case, fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The resulting algorithm has BSP cost asymptotically equal to the cost of the fast pivoting-free algorithm presented at the end of the previous section.

5 Conclusions

Parallel algorithm complexity is an area of active research, both from the theoretical and the practical perspective. Dozens of parallel cost models have been proposed and used to analyse thousands of algorithms from various fields. One of the common phenomena, observed both in theory and in practice, is the “parallelism-communication tradeoff”: the finer the granularity of a parallel algorithm, the more communication it requires.

In this paper, we have considered the parallel complexity of two matrix problems: Gaussian elimination with pairwise pivoting, and orthogonal matrix decomposition by Givens rotations. We have defined a common framework that unifies both problems, and presented a new communication-efficient BSP algorithm for their solution. In fact, the communication cost of our algorithm is asymptotically optimal: the lower bound is provided by the communication cost of matrix multiplication. However, the improvement in communication efficiency (relative to previously known algorithms) is offset by a proportional reduction in synchronisation efficiency. Additionally, our method allows one to trade off the costs of communication and synchronisation in a certain range.

Apart from being a useful addition to the growing collection of efficient BSP algorithms, this result (as well as a similar earlier result on pivoting-free Gaussian elimination) can be viewed as a refinement of the classical “parallelism-communication tradeoff”. One of the main strengths of the BSP model is that it allows an independent treatment of pure communication and synchronisation. Our analysis shows that, within a certain range of parameters, moving from coarser-grain to finer-grain computation may actually reduce the amount of pure communication. The tradeoff still exists, but only between parallelism and synchronisation. We have argued that for large problem sizes, the finer-grain, communication-efficient end of this tradeoff may be preferable to the coarser-grain, synchronisation-efficient one.

It remains an open question whether the optimal communication and synchronisation costs for Gaussian elimination (even without pivoting) can be achieved simultaneously.

References

1. A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.
2. J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 21(125):231–236, January 1974.

3. P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Number 315 in Grundlehren der mathematischen Wissenschaften. Springer, 1997.
4. R. Calinescu and D. J. Evans. Bulk-synchronous parallel algorithms for QR and QZ matrix factorisation. *Parallel Algorithms and Applications*, 11:97–112, 1997.
5. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.
6. M. Cosnard and E. M. Daoudi. Optimal algorithms for parallel Givens factorization on a coarse-grained PRAM. *Journal of the ACM*, 41(2):399–421, March 1994.
7. J. W. Demmel, N. J. Higham, and R. S. Schreiber. Block LU factorization. *Numerical Linear Algebra with Applications*, 2(2), 1995.
8. K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.
9. D. Irony and S. Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters*, 12:79–94, 2002.
10. W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1995.
11. W. F. McColl. A BSP realisation of Strassen’s algorithm. In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 43–46. IOS Press, 1996.
12. W. F. McColl. Universal computing. In L. Bougé et al., editors, *Proceedings of Euro-Par (Part I)*, volume 1123 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 1996.
13. J. J. Modi. *Parallel Algorithms and Matrix Computation*. Oxford Applied Mathematics and Computing Science Series. Clarendon Press, 1988.
14. J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, 1988.
15. A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the ACM*, 25(1):81–91, January 1978.
16. A. Schönhage. Unitäre Transformationen großer Matrizen. *Numerische Mathematik*, 20:409–417, 1973.
17. D. C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, C-34(3):274–278, March 1985.
18. A. Tiskin. Bulk-synchronous parallel Gaussian elimination. *Journal of Mathematical Sciences*, 108(6):977–991, 2002.
19. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Alternative Parallelization Strategies in EST Clustering

Nishank Trivedi¹, Kevin T. Pedretti², Terry A. Braun¹, Todd E. Scheetz¹, and Thomas L. Casavant¹

¹ The University of Iowa, Iowa City,
Iowa 52242, USA,
tomc@uiowa.edu

² Sandia National Labs, Albuquerque, New Mexico,
87123, USA

Abstract. One of the fundamental components of large-scale gene discovery projects is that of clustering of Expressed Sequence Tags (ESTs) from complementary DNA (cDNA) clone libraries. Clustering is used to create non-redundant catalogs and indices of these sequences. In particular, clustering of ESTs is frequently used to estimate the number of genes derived from cDNA-based gene discovery efforts. This paper presents a novel parallel extension to an EST clustering program, **UICluster4**, that incorporates alternative splicing information and a new parallelization strategy. The results are compared to other parallelized EST clustering systems in terms of overall processing time and in accuracy of the resulting clustering.

1 Introduction

The sequencing of cDNA libraries is the most common format for gene discovery in higher eukaryotes. The goal of such a project is to utilize the sequences derived from the cDNAs (ESTs; expressed sequence tags) to derive a non-redundant set. This set ideally represents an organisms entire complement of genes. EST-based gene-discovery projects are in progress for numerous species of medical, scientific, and industrial interest. The benefits of EST-based gene discovery include the ability to rapidly identify transcribed genes, the ability to identify exon-intron structure (when coupled with genomic sequence), and information on gene expression. EST data is so useful that the National Center for Biotechnology Information (NCBI) provides a separate division specifically for EST sequences (dbEST) [1].

However, different genes are expressed at different levels. Thus, a given gene's transcript may be present in 0, 1 or many copies within a cell. Because these transcripts are used to generate the cDNAs, both the cDNAs and the ESTs derived from them will also be present in similarly variable levels.

The presence of this redundancy within the EST databases, requires a programmatic method to calculate the complement of genes they represent. These methods (termed clustering) utilize sequence-based comparisons to determine

sets of strongly similar sequences (clusters). The primary difficulty associated with EST-based gene-discovery projects is that ESTs are single-pass sequences, and as such they are relatively error prone (approximately 3% on average [2]).

NCBI also provides a curated and annotated gene index (UniGene) [3] for several species, utilizing the available mRNA and EST sequences to estimate the gene complement. This paper describes and compares several programs that may be used to create non-redundant “UniGene” sets from EST data, and analyzes three different approaches for parallelization of this task.

2 Background

Clustering plays an important role in large scale gene-discovery projects. It not only saves time by identifying redundant (EST) sequences but also provides useful information regarding gene-discovery rate [4]. Another significant use of clustering is to create non-redundant gene indices. As suggested in this paper, clustering can be further used to identify possible alternatively spliced sites in mRNA gene transcripts

There are several varying clustering methods and tools in use today. However, the objective of all such methods is to effectively assess the similarity between all pairs of sequences and place them into equivalence classes. Ideally, these classes correspond, one to one, onto distinct genes. It should be noted that such a procedure based entirely on subsequence similarity cannot achieve perfect fidelity with respect to gene classes. A number of other criteria, not apparent in the primary RNA sequence data, are necessary for such a classification. However, a sequence-based classification is of extremely high usefulness. One of the most widely used clustering tools is NCBI's Unigene clustering [5]. It uses global pairwise sequence comparison, and a stringent protocol for assigning closely related sequences to a common cluster. However, it does not support incremental clustering. Hence, each clustering “build” must begin from the same initial starting point. As the number of known ESTs in homo sapiens currently stands at approximately 5 million, and requires more than one month of computation time, the ability to perform incremental clustering becomes obvious. Also, an EST relating to two different clusters is discarded, overlooking any possible alternative splice sites. The Institute for Genome Research (TIGR) [6], produces gene indices for many organisms. It performs a pairwise alignment of incoming sequences with a template obtained from a database consisting of expressed mRNA transcripts, as well as tentative consensus assemblies of other ESTs, mRNAs and cDNAs. Sequences must qualify through strict identity criteria. Each cluster is finally assembled to produce a consensus sequence. Due to very strict clustering rules, TIGR gene indices discard many under-represented, divergent or low-quality sequences, leading to under-clustering of sequences. The SANBI STACK clustering approach [7] was developed primarily for human databases, but is general purpose. It performs a looser clustering of sequences, but has a strict assembly phase. The clustering is conducted using non-contextual assessment of composition, and a multiplicity of words within each sequence. Typically, the STACK

approach produces larger clusters than Unigene, and has longer consensus sequences for each cluster than TIGR.

All of the above programs are essentially sequential. A parallel clustering method developed at Iowa State University [8], **PaCE**, uses an implementation of suffix trees for sequence comparison. The method is a strict sequence identity match clustering method and performs an $N \times N$ alignment, although in parallel, hence offsetting the high costs associated. The overall method is to construct suffix trees in parallel, perform pairwise alignment for selective sequences, and finally to group them together based on a similarity score. However, the method demands a specific hardware requirement, and overlooks divergent cases within EST sequences. The **UIcluster** family of solutions (both serial and parallel) has been evolving in a production environment at the University of Iowa since 1997. The key characteristics of **UIcluster** are incremental clustering, the maintenance of a “primary” representative element for each cluster, and a hashing scheme to quickly identify potentially meaningful cluster matches for each newly considered input sequence. The stringency of the clustering is a user-definable parameter, although performance is also a sensitive function of this aspect. Parallelization of **UIcluster** has now been performed relative to both the cluster space [9], and input space, which is the main focus of this paper.

The following is a brief description of the underlying approach of **UIcluster**. After an incoming sequence has been read from an input file, it is compared against all existing clusters. The comparison is performed only with the primary element of each cluster, where the primary is a single representative sequence of the entire cluster (usually the longest, and therefore most informative member). If the incoming sequence matches, or “hits” any cluster primary, and further satisfies specified similarity criteria, it is added to that cluster. Otherwise, the incoming sequence itself becomes the primary element of a new cluster. The basic search screening is based on a criteria of n matching positions in a window of length m . Hashing is usually performed on a short motif roughly one quarter of the size of m . The $m - n$ positions in discordance may be either substitution or gap errors. As a practical necessity, a global table of hash values and a map to each cluster containing those values, is used to count hits to the primaries of any given cluster. In many cases, it is possible to avoid actual alignment of a new sequence to the primary of the best cluster hit by employing deduction based on the number of hashes which correspond between the two sequences. The efficiency gained by using hashes to a primary, and thus avoiding alignment is dramatic. Basing our parallel versions of **UIcluster** on this already optimized serial method provides a number of advantages to be described in the next section.

3 Approach and Implementation

The performance of EST clustering is measured both by time as well as by memory resource utilization characteristics. Although the use of hashing, and of a primary sequence for each cluster significantly reduces both these requirements,

space remains a limiting factor for even more efficient and heuristic approaches. Considering the nature of the problem and the size of the data set, parallelization is an obvious choice for the implementation of this process. Computational and memory requirements can be distributed across several computers. This adds to the performance of software so that the program can scale to larger problem sizes.

UIcluster currently implements two different approaches of parallelization, distributing across the cluster space and the input space. The MPI (message passing interface) [10] standard is used for inter-process communications, and distribution is done among multiple UNIX processes.

3.1 Parallelization on Cluster Space

In this scheme of parallelization, implemented in **UIcluster3**, each cluster is stored on exactly one compute node. The clusters are evenly distributed among all nodes. When a sequence is brought in, it is copied to all available nodes and is processed in parallel. Since each node has a different set of clusters, the incoming sequence is compared with the divided cluster space in parallel. For every node, once the local search has been performed, the information about the best matching primary is communicated to all other nodes. Further, the node with the best match adds the sequence to its cluster space. In the case of a non-match, the sequence itself becomes a cluster and is designated to one compute node.

3.2 Parallelization on Input Space

In the cluster space parallelization method, incorporated in **UIcluster4**, the input is the same for all the compute nodes but the clusters being created are distributed over various processors. A variation on this scheme can be implemented by dividing the sequences also in N non-overlapping groups, where N is the number of compute nodes available. Instead of distributing clusters over different compute nodes and processing each sequence at all nodes, in this scheme each node gets an individual sequence. The pool of input sequences is evenly distributed among all compute nodes. This is similar to running sequential version of **UIcluster** in parallel on all nodes, however, with an abridged dataset. Each node computes its own set of primaries. In the second stage, these primaries are compared among themselves and related clusters are merged.

The efficiency of this scheme is heavily dependent on the redundancy within the dataset. If the data has a high rate of redundancy, the clusters being created on each node are more likely to be merged, involving more communication and added processing. On a single node or cluster space parallelization, the redundant data would have converged into a smaller number of clusters. On the contrary, less redundancy amounts to more clusters hence, input space parallelization reduces space and time requirements.

4 Related Issues

4.1 Virtual Primaries

One limitation in early versions of **UIcluster** was the requirement that a single representative sequence (primary sequence) be selected. Even when mRNA transcript sequences are available, they often lack comprehensive coverage of the original transcript (especially the untranslated regions). Therefore, EST sequences generated from the 3' end may contain significant amounts of novel sequence not represented in the mRNA sequences. Other, more complex, processes such as alternative splicing and alternative polyadenylation are also sources of additional novel sequence. To address this issue, we have developed the concept of a virtual representative sequence (virtual primary). A virtual primary is a non-redundant representation of the constituent sequences within a cluster. Utilizing virtual primaries enables sequence comparisons to be performed against only one sequence per cluster, while still searching the entire composite sequence available for each cluster.

Figure 1 illustrates how a set of partial sequences may be combined to construct a virtual primary. Here, alternate shading is used to denote blocks of homologous sequence. On the left is a set of ESTs (A,B,C,D,E) derived from the same gene. The right half shows the effect of adding the sequences into a growing virtual primary. With a single sequence (A) the virtual primary is identical to the EST. As sequences containing novel subsequences (B,C) are added into the cluster, the novel portions are integrated into the virtual primary at the appropriate position (A+B+C). If a sequence contains no novel subsequences (D), the virtual primary is not changed. In the event of a sequence with a novel insertion (with respect to the virtual primary) (E), the novel portion is incorporated into the virtual primary at the congruent position (A+B+C+D+E).

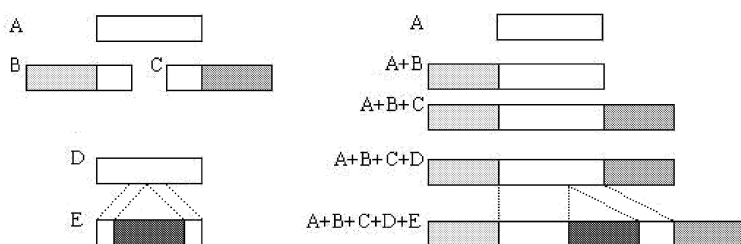


Fig. 1. Construction of a virtual primary. ESTs derived from transcripts of the same gene are shown at left (A, B, C, D, and E). At right the growing virtual primaries are shown as each EST is included. The dashed-lines represent regions of sequence homology.

Incorporating the construction of virtual primaries into the clustering procedure does not affect the strategy used to identify which cluster a sequence belongs

to. The only impact is to alter the method of deriving the primary sequence. This process does add a small overhead into the computation cost clustering, but does not alter the computational complexity of the algorithm.

4.2 Cluster Viewing and Editing

An ancillary program has been developed to aid in the visualization and editing of the resulting clusters. This cluster editor was implemented in Java as both an application and an applet. The applet-based solution makes our clustering method available over the internet to interested users. Search features were integrated into the editor so that clusters with specific features can quickly be identified. Currently supported features include clusters with apparent alternative splicing, and those with weak sequence hits - potentially from gene families. This program facilitated the process of debugging the clustering program, enabling erroneous cases to be visualized. Two issues in the construction of virtual primaries particularly benefited from the use of the cluster editor: the order in which non-redundant sub-sequences were incorporated, and that all non-redundant sub-sequences are included in the virtual primary.

4.3 Order of Inclusion

One factor that can significantly affect the clustering is the order in which sequences are included. **UIcluster's** approach performs smoothly for short EST sequences (400-1,000bp). However, full length mRNA sequences (1000's of bp) may rapidly degrade the performance. As the sequence length increases, so does the probability of finding additional minimally-matching regions, which results in an increase in the number of detailed sequence comparisons. For **UIcluster3**, splitting the longer sequence into smaller overlapping sequences was a potential solution. However, when **UIcluster4** is used with virtual primaries, the order in which the sequences are included affects the computation differently. When longer sequences are included first, although more comparisons are performed, there are fewer cluster primaries to be compared against. Similarly, less computation is spent on updating the virtual primaries, as more of the sequence is provided within the longest sequences. The order of inclusion effect was tested using a dataset of 11,058 sequences with an average length of 460bp, **UIcluster4** was run while including the sequences in two different orders. Adding the sequences in order of descending length, resulted in a clustering run-time of 169 seconds and generated 7485 clusters. In comparison, when adding the sequences in ascending order of sequence length the clustering run-time was 193 seconds and generated 7571 clusters.

5 Results

5.1 Description of Experiment

To evaluate the performance of different clustering methods, several data sets from *Arabidopsis thaliana* and *Homo sapiens* were used. The methods compared

include parallelizing on the cluster space (**UIcluster3**), parallelizing on the input space (**UIcluster4**), and the suffix-tree based method of **PaCE**. The **PaCE** clustering program [8] was included to analyze both parallel speedup and memory requirements. The publically available UniGene clusters from NCBI were used to assess the accuracy of the results. The system used in this comparison was a 16 node, dual processor cluster of 500MHz Pentium III's, each equipped with one Gigabyte of memory. The human EST data set consisted of 41,197 sequences with an average length of 403 bp. The *Arabidopsis thaliana* EST data set contained 81,414 sequences with an average length of 411 bp. The latter data set was used for comparing the accuracy between the programs. The use of *A. thaliana* rather than human ESTs was important in reducing the effect of known genes on the purely sequence-based clustering.

5.2 Accuracy Assessment

Although performance is critical in making the clustering results available, they must also provide an accurate reflection of the underlying mRNA transcripts from which the cDNAs were derived. To assess the accuracy of the clustering methods, two separate comparisons were performed. Both used sets of *Arabidopsis thaliana* sequences. The first data set compared the clustering of 81,414 *A. thaliana* ESTs. This set had previously been clustered with **PaCE** by Srinivas Aluru from Iowa State University. The resulting clusters between **UIcluster4** and **PaCE** were very similar, with 23,642 clusters and 23,995 clusters identified respectively.

A second assessment of clustering accuracy was performed using the complete set of *A. thaliana* ESTs and mRNAs from GenBank. In this assessment, **UIcluster4** was compared to NCBI's UniGene build for *A. thaliana*. The UniGene build contained a total of 27,248 clusters including 9,191 singletons. Similar results were produced with **UIcluster4**, identifying 23,925 clusters of which 6,682 were singletons. This result indicates that **UIcluster4** is more aggressive in merging sequences into the same cluster, resulting in a more conservative estimate of clusters numbers.

5.3 Performance Assessment

Both memory utilization and computation time were measured across these data sets. Table 1 presents the execution time for the same analyses. In this comparison, **UIcluster3** requires approximately one-tenth of the time of **PaCE**. As the number of input sequences increased, the relative difference in computation time between **UIcluster4** and **UIcluster3** decreased. With only 5000 sequences, **UIcluster4** required approximately 60% longer than **UIcluster3** on the same set of sequences. However, on the set of 30,000 sequences, that difference was only 26%. A similar reduction in computation time is observed between **PaCE** and **UIcluster3** with **PaCE** requiring approximately 16 times more computation for 5000 sequences, but only 12 times more in the data set of 20,000 sequences.

The peak memory utilization was assessed on a single node with 1GB of memory, using a subset of the human EST data set. Figure 2 shows the peak memory usage by the three clustering programs. Values were unavailable for PaCE with the 30,000 EST data set, as it exhausted the available memory. Note from this figure that the memory requirements of UIcluster4 increase faster than UIcluster3 as the number of input sequences grows. This is expected, because the likelihood of novel subsequences that must be included in the virtual primary increases as the number of sequences within a cluster increases, Although the PaCE program has to use at least two nodes (one master and one slave node) only the memory utilization for the slave node was measured, because it performs sequence comparisons. If the same computation is run in parallel with UIcluster4, the memory requirement per node is significantly reduced, as there are fewer clusters to be stored.

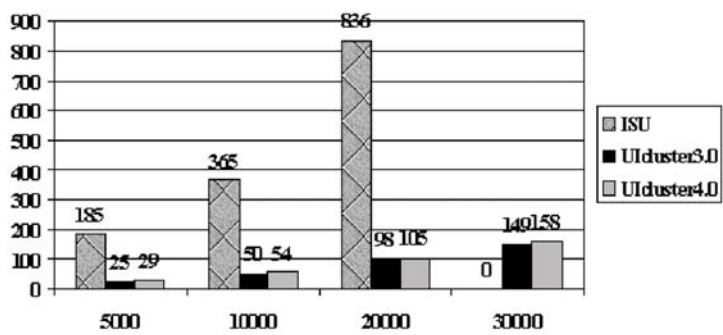


Fig. 2. Memory Utilization

Table 1. Execution time performance comparison.

Num of sequences	PaCE	UIcluster3	UIcluster4
5,000	10 min	37 sec	1 min
10,000	28 min	2 min 7 sec	3 min 28 sec
20,000	1 hr 44 min	8 min 42 sec	12 min
30,000	Out of Mem	20 min	25 min 12 sec

A final performance analysis was performed using the complete set of human EST and mRNA sequences from the human UniGene build. The parallelization on the input space method was used to predict the final number of clusters and the computation time required. This data set contained nearly 4.2 million sequences. The clustering, utilizing 12 nodes, requires an estimated computa-

tion time of 100 hours. For this experiment, the data set was divided into 12 files each containing one twelfth of the ESTs. Thus each file contained roughly 400,000 EST sequences. All of the sequences longer than 1,100 bp were put into a separate file. These thirteen sequence files were first clustered individually. The resulting cluster files were then clustered together to compute the complete set of clusters for the 4.2 million EST sequences. Unfortunately, the final clustering step required more memory than was available. Therefore, the computation time of that component was estimated.

6 Conclusions

An alternative scheme for parallel clustering using **UICluster** has been described in this paper. The concept of a representative sequence made from the non-redundant set of subsequences from a cluster's constituent sequences is also presented. Such representative sequences can provide further information to biologists regarding several features of biological interest that might otherwise be overlooked. The program is comparable in accuracy to other clustering programs, but requires less computation time. Depending upon the nature of the data set, either of the parallelization schemes may be used to optimize the memory or computation requirements.

Acknowledgements. The authors would like to thank Dr. Volker Brendel from Iowa State University for providing us with the test set of 81,141 *A. thaliana* ESTs, Dr. Srinivas Aluru and Anantharaman Kalyanaraman from Iowa State University for their assistance in obtaining and using the PaCE clustering program, and Thomas Bair, Dylan Tack, Jason Grundstad, Jared Bischof, Brian O'Leary and Jesse Walters for their help and suggestions.

References

1. Boguski, M.S., Lowe, T.M., Tolstoshev, C.M.: dbEST – database for 'expressed sequence tags'. *Nature Genetics* **4** (1993) 332–333
2. Hillier, L., Clark, N., Dubuque, T., Elliston, K., Hawkins, M., Holman, M., Hultman, M., Kucaba, T., Le, M., Lennon, G., Marra, M., Parsons, J., Rifkin, L., Rohlfing, T., Soares, M., Tan, F., Trevaskis, E., Waterston, R., Williamson, A., Wohldmann, P., Wilson, R.: Generation and analysis of 280,000 human expressed sequence tags. *Genome Research* **6** (1996) 807–828
3. Schuler, G.D.: Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *Journal of Molecular Medicine* **75** (1997) 694–698
4. Bonaldo, M.F., Lennon, G., Soares, M.B.: Normalization and subtraction: two approaches to facilitate gene discovery. *Genome Research* **6** (1996) 791–806
5. <http://www.ncbi.nlm.nih.gov/UniGene/build.shtml>
6. Adams, M.D., Kerlavage, A.R., Flieshmann, R.D., Fuldner, R.A., Bult, C.J., Lee, N.H., Kirkness, E.F., Weinstock, K.G., Gocayne, J.D., White, O.: Initial assessment of human gene diversity and expression patterns based upon 83 million nucleotides of cDNA sequence. *Nature* **377** (1995) 3–17

7. Miller, R.T., Christoffels, A.G., Gopalakrishnan, C., Burke, J.A., Ptitsyn, A.A., Broveak, T.R., Hide, W.A.: A comprehensive approach to clustering of expressed human gene sequence: The Sequence Tag Alignment and Consensus Knowledgebase. *Genome Research* **9** (1999) 1143–1155
8. Kalyanaraman, A., Aluru, S., Kothari, S.: Space and time efficient parallel algorithms and software for EST clustering. *International Conference on Parallel Processing* (2002) 331
9. Trivedi, N., Bischof, J., Davis, S., Pedretti, K., Scheetz, T.E., Braun, T.A., Roberts, C.A., Robinson, N.L., Sheffield, V.C., Soares, M.B., Casavant, T.L.: Parallel creation of non-redundant gene indices from partial mRNA transcript. *Future Generation Computer Systems* **18** (2002) 863–870
10. Message Passing Interface Form : MPI: A message-passing interface standard. University of Tennessee Technical Report (1994) CS-94230

Protective Laminar Composites Design Optimisation Using Genetic Algorithm and Parallel Processing

Mikhail Alexandrovich Vishnevsky, Vladimir Dmitrievich Koshur,
Alexander Ivanovich Legalov, and Eugenij Moiseevich Mirkes

Krasnoyarsk State Technical University, ul. Kirenskogo 26,
660074, Krasnoyarsk, Russia.
mav@escapenet.ru koshur@fivt.krasn.ru legalov@mail.ru
mirkes@sacaudit.krsn.ru

Abstract. We examined the sound wave dissemination model in laminar composite as one-dimensional elastically dissipative system of nodes and ties. This model is defined by the differential equations system. We used the implemented model in composite structure and its components widths optimisation. The purpose of optimisation is to make the composite to absorb the ultrasound wave with established frequency. We worked out the genetic algorithm modification which we applied in optimisation. We also used the parallel processing.

1 Introduction

Nowadays there are many scientific researches on developing protective covering, which absorbs different noises. In this work we applied modified genetic algorithm (GA) and parallel processing to optimise the laminar composite in order to absorb the ultrasound waves with established frequency.

We implemented the discrete model of wave dissemination in composites. In order to estimate the quality of specified laminar composite the quality functional is calculated using the implemented model. As the optimisation algorithm we used the modified GA. Optimisation parameters are the numbers of materials in the composite and their widths. In our algorithm we use parallel starting of several GA populations which periodically exchange some individual-strangers.

As far as GA is a pseudo-random search method, it doesn't give stable results with expected precision from time to time especially when the function has local extremums. In this case it will obviously increase the probability of acceptable solution if several populations are starting at the same time. The application of the exchange also improves the method. And the experiments proved it. We obtained the result: the structure of the laminar composites for established conditions, which you can see below.

1.1 Problem Relevance

Nowadays we can see a lot of large manufactures, plants and factories with different workshops located not far from some other buildings where people work. In such workshops we can find engineering tools, drilling utilities and other large equipment. This kind of equipment can be a source of different loud noise. It can be unhealthy for people especially for their nervous system. Different ultrasound waves are particularly dangerous because we do not hear them.

It's a problem: how we could isolate rooms to avoid the influence of harmful ultrasound waves. So, one of the goals of this work is to develop and implement the method of finding an optimal laminar composite which absorbs such a noise.

1.2 Problem Setting

Laminar composite is influenced by the ultrasound wave with pulsance equal to 10^6 . Composite can include materials from given assortment. We used 5 materials: steel, copper, aluminium, hard rubber, soft rubber. Materials information required: Yung's module - E , Puason's coefficient - ν , density - ρ . Composite total width is 0,02m. One material layer couldn't be thinner then 0,0001m. Materials are glued strongly. Top and bottom edges are free. Composite absorption of the wave is calculated by the vibrations on the bottom edge. The covering must not let the wave through. So, we need to obtain the structure of the laminar composite which will as much as possible absorb the wave passing through it.

2 Wave Dissemination Model

The model is intended to show the wave dissemination through the composite. The model is considered to have only one dimension. The model construction is based on the continuous medium presentation as a discontinuous system – the number of discrete elements: [1], [2], [3].

The constitution of the laminar composite is shown in Fig.1.

We will define the following parameters for each element numerated $i+1/2$:

- Element mass,
- tension in the element $\sigma_{i+1/2}$,
- Yung's adduced module $E^*_{i+1/2}$,
- Deformation $\varepsilon_{i+1/2}$.

We will define the following parameters for each node numerated as i :

- node coordinate $z_i(t)$,
- node transference $u_i(t)=z_i(t)-z_i(0)$,
- node velocity $v_i(t)$,
- node acceleration $a_i(t)$,
- node mass m_i (it's calculated from the elements masses.)

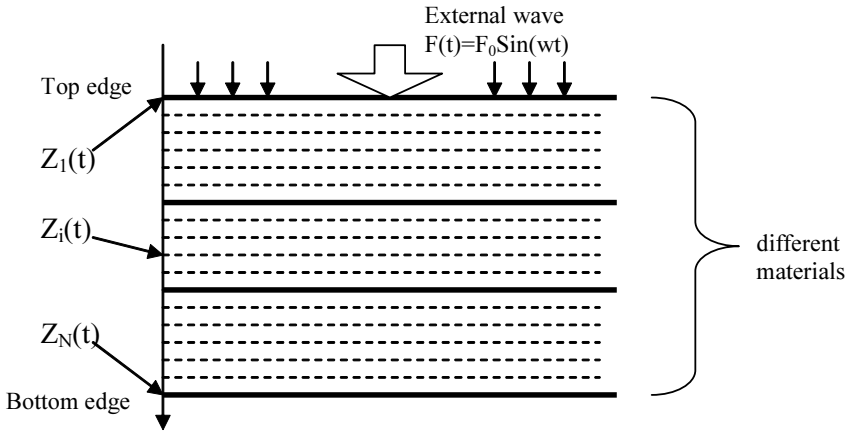


Fig. 1. Laminar composite structure. Each layer is a system of discrete elements. Minimum number of elements is 10

Differential equations system is obtained using virtual velocity principle [3]:

$$\left\{ \begin{array}{l} m_i a_i(t) - \sigma_{i+1/2} + F(t) = 0, \text{ for } i=0, \\ m_i a_i(t) - \sigma_{i+1/2} + \sigma_{i-1/2} = 0, \text{ for } 0 < i < N, \\ m_i a_i(t) + \sigma_{i-1/2} = 0, \text{ for } i=N, \\ \text{where } \sigma_{i+1/2} = E^*_{i+1/2} \varepsilon_{i+1/2}, \\ \varepsilon_{i+1/2} = (z_{i+1}(t) - z_i(t)) / (z_{i+1}(0) - z_i(0)) \end{array} \right. \quad (1)$$

Functional Choice. For measuring the wave passing through the composite we can use the following functionals [3]:

$$\begin{aligned} J_1 &= \int (u_n)^2 dt, \\ J_2 &= \int (v_n)^2 dt, \\ J_3 &= \int (a_n)^2 dt, \\ J_4 &= \int (\sigma_{n-1/2})^2 dt. \end{aligned} \quad (2)$$

Different experiments revealed that the acceptable functional for this problem is J_3 .

Model Verification. We verified the model using various time discretization and various spatial discretization. The testes proved the adequacy of the model. The average inaccuracy is equal to 1,5 %.

Functional Parameters. We must choose the optimisation parameters for the criterion function. Here are the functional optimisation parameters:

$k_1, k_2, \dots, k_N, w_1, w_2, \dots, w_N$, where

k_i – number of material in layer number i . Number of material means number in assortment database,

w_i – width of the layer number i ,

N – maximum quantity of layers in composite.

3 Genetic Algorithm

To optimise the laminar composite we chose the pseudo-random search method – genetic algorithm [4], [5], [6], [7], [8], [9]. It was decided to apply this method because the function is not differentiable. And moreover, some arguments have discrete values.

3.1 GA Modification

In basic GA each gene includes one bit. As far as the optimised function contains real arguments, we suggested the following GA modification.

As usual, chromosome contains number of genes. But each gene includes a real type variable. It means that the individual represents a vector R^n . It is obviously that the traditional crossover doesn't fit. The suggested crossover scheme is the following. We should choose descendants from the hypercube determined with ancestor vectors. The testes proved the advantages of the modified GA.

3.2 GA Using Parallel Processing

GA is a pseudo-random search method. To obtain the reliable result we must start the GA again and again. It is obviously that we can obtain the better result starting several populations at the same time. Then we can implement parallel GA in the following way. Each parallel process starts its own optimisation with its own population.

However, using analogies from biology, we suggest another scheme of parallel GA. Each parallel process starts its own optimisation with its own starting population. But we add to algorithm some kind of individuals-strangers. It means that each population regularly send outwards some best individuals. We will call them strangers. And each population takes back some other strangers from different populations. This feature permits GA to implement «further» crossover. In other words it is the genefund exchange between different populations.

Here is the suggested parallel GA outline:

- Step 0: Generating of the first population.
- Step 1: Evolution step: Crossover, mutation, and natural selection.
- Step 2: Strangers going out.
- Step 3: Stranger reception.
- Step 4: If not terminated, then go to Step 1.

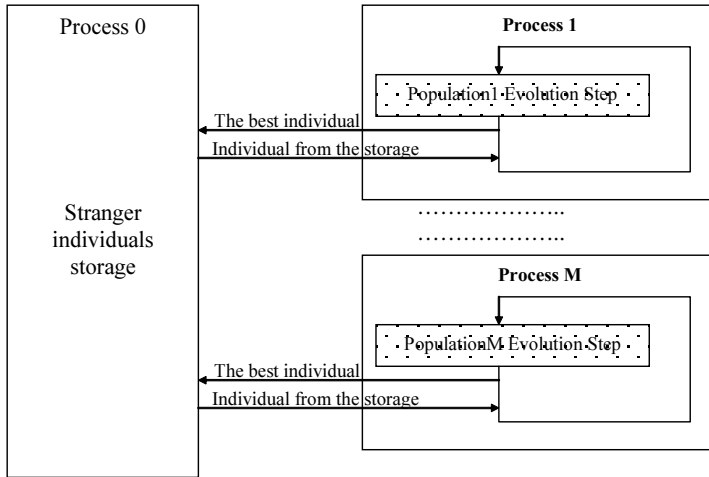


Fig. 2. Implemented GA with parallel populations "with strangers"

Evidently, we don't need any synchronization using this algorithm. The individuals go wandering and settle non- synchronously. We use the process called "storage" to keep them. It provides individuals reception and delivery. The number of strangers is limited but they are displaced by the newcomers.

4 Method Implementation Using MPI. Obtained Results

Paralleled GA was implemented using MPICH on Linux RedHeart. Programming Language – C++. We used 9 computers of the Cluster on the FIVT KGTU. Computer configuration: Intel P4, 256 Mb DDR.

We tested both methods: using strangers and not using strangers. The tests indicated that paralleled GA with strangers is more effective for applying to the problem of optimisation of laminar composites for ultrasound absorption.

The results point to the fact that we can obtain acceptable solution using a suggested method with strangers faster than without of them. Obtained accuracy improvement might be considered insignificant. However, when it takes much time it becomes more important. For example, it took 15 minutes to find each solution in 10 population steps to obtain the results above. For the results below (70 population steps) it took even more then an hour.

We implemented longer test: 70 steps per population. Here are the obtained results.

Laminar composite structure (material width (m) – material name):

0,005167 — Copper,
0,003339 — Soft rubber,
0,001381 — Copper,
0,003925 — Soft rubber,
0,006188 — Copper.

The functional value is $3,56 \cdot 10^{-11}$ for obtained composite. For example, if we use just a steel slice (0,02m), the functional value is $1,5 \cdot 10^{-6}$.

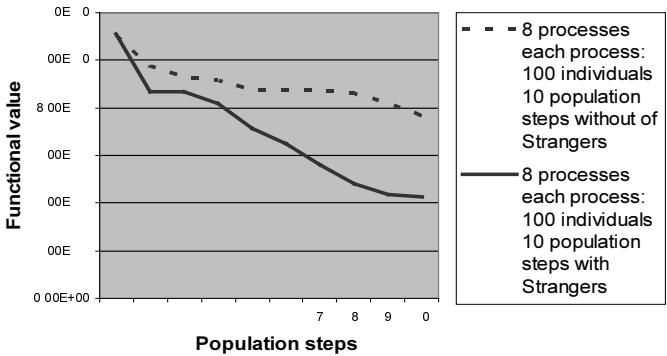


Fig. 3. Parallel GA methods effectiveness

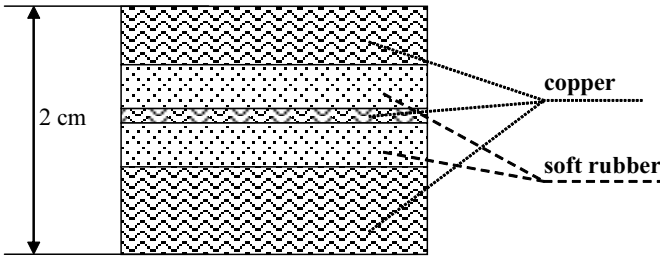


Fig. 4. Obtained laminar composite

So, we developed a method for obtaining optimal composite for ultrasound absorption. The solution for particular conditions is found.

5 Future Method Extensions

- To use much more materials in assortment for optimising the laminar composite and obtain results.
- Method improvement: to combine GA with other search methods and neuronet in order to hasten functional calculating.
- To develop the algorithm to optimise the absorption of the wave spectrum but not the established frequency wave.
- Embedding intelligent elements in composite for adaptation to dynamic waves.

References

1. Koshur V.D., Nemirovsky U.V.: Continuous and discontinuous models of a construction members dynamic deformation. Nauka, Nobosibkrsk (1990)
2. Kanibolotsky M.A., Urzhumtcev U.S.: Laminar constructions optimal design. Nauka, Nobosibkrsk (1990)
3. Koshur V.D.: Differential equations and dynamic systems. Computer lecture version. KGTU, Krasnoyarsk (2001)
4. Isaev S.A.: Genetic algorithm popularly. Web: <http://saisa.chat.ru/ga/ga-pop.html>
5. Isaev S.A.: Genetic algorithm – evolutionary search methods. Web: <http://saisa.chat.ru/ga/text/part1.html>
6. Genetic algorithms. © 1999 NeuroProject Web: <http://www.neuroproject.ru/genealg.htm>
7. Strunkov T.: What are the genetic algorithms. Web: <http://www.neuroproject.ru/gene.htm>
8. Norenkov I.P.: Computer-aided design basics. MGTU, Moscow (2000)
9. Batishev D.I.: Solving extremum problems using genetic algorithms. Voronezh (1995)
10. Nemnugin S.A., Stesik O.L.: Parallel programming for multiprocessor systems. BHV-Peterburg, St. Petersburg (2002)

A Prototype Grid System Using Java and RMI

Martin Alt and Sergei Gorlatch

Technische Universität Berlin, Germany
{mnaalt|gorlatch}@cs.tu-berlin.de

Abstract. Grids aim to combine different kinds of computational resources connected by the Internet and make them easily available to a wide user community. While initial research focused on creating the enabling infra-structure, the challenge of programming the Grid has recently become increasingly important. The difficulties for application programmers lie in the highly heterogeneous and dynamic nature of Grid environments. We address this problem by employing reusable algorithmic patterns, called *skeletons*. Skeletons are used, in addition to the usual library functions, as generic algorithmic building blocks, customizable for particular applications. We describe an experimental Grid programming system, focusing on improving the Java RMI mechanism and the predictability of Java performance in a Grid environment.

1 Introduction

Grid systems aim to combine different kinds of computational resources connected by the Internet and make them easily available to a wide user community. Initial research on Grid computing focused, quite naturally, on developing the enabling infra-structure, systems like Globus, Legion and Condor being the prominent examples presented in the “Gridbook” [1]. Other efforts have addressed important classes of applications and their support tools, like Netsolve [2] and Cactus, and the prediction of resource availability, e. g. in NWS [3].

Some algorithmic and programming methodology aspects appear to have been neglected at this early stage of Grid research and are therefore not yet properly understood. Initial experience has shown that entirely new approaches to software development and programming are required for the Grid [4]; the GrADS [5] project was one of the first to address this need. A common 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. Such systems are commonly referred to as Network Enabled Server (NES) environments [6]. There are several systems, such as NetSolve [7] and Ninf [8], that adopt this approach.

An important challenge in application programming for the Grid is the phase of algorithm design and, in particular, performance prediction early on in the design process. Since the type and configuration of the machine on which the program will be executed is not known in advance, it is difficult to choose the right algorithmic structure and perform architecture-tuned optimizations. The

resulting suboptimality can hardly be compensated in the implementation phase and can thus dramatically worsen the quality of the whole Grid enterprise.

We address programming the Grid by providing application programmers with a set of reusable algorithmic patterns, called *skeletons*. Compute servers in the Grid may provide different, architecture-tuned implementations of the skeletons. Applications composed of skeletons can thus be targeted for execution on particular servers in the Grid with the goal of achieving better performance.

The particular contributions and structure of the paper are as follows:

- We present our prototype Grid environment, which serves as a proof-of-concept programming system and a testbed for experiments (Section 2).
- We describe our implementation of the proposed Grid architecture using Java RMI (Section 3).
- We propose optimizations of the Java RMI mechanism to reduce the overhead of remote calls in our Grid environment (Section 4).
- We present novel methods for estimating the performance of Java bytecodes that are used as parameters of algorithmic skeletons (Section 5).
- We report experimental results that confirm performance improvements and predictability in our Grid system (Section 6).
- We discuss our results in the context of related work (Section 7).

2 The Prototype Grid System

In this section, we present the architecture of our Grid environment, which we use as a proof-of-concept prototype and as an experimental testbed.

2.1 Hardware Architecture

To evaluate our concepts and implementations, we have set up a prototypical Grid system, whose structure is outlined in Fig. 1.

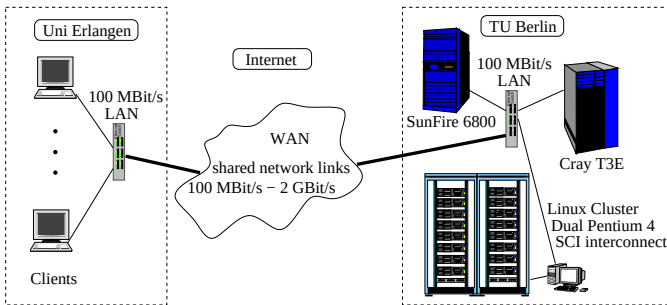


Fig. 1. Structure of the prototypical Grid system

It 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.

We use Berlin as the server side, with three high-performance servers of different architectures: a shared-memory SunFire, a Cray T3E and a distributed-memory Linux cluster with 32 processors. Most of our experiments used a SunFire 6800 SMP system with 12 UltraSparc-III processors running at 750 MHz. Because of the shared-resources operation mode (typical of Grid environments), a maximum of only 8 processors was available for measurements as there were several other applications running on the server machine during our experiments. The client-side role is played by Erlangen, where our clients run on SUN Ultra 5 Workstations with an UltraSparc-III processor running at 360 MHz.

2.2 Programming with Skeletons: The Idea

In our system, application programs are constructed using library functions and/or a set of skeletons (for more details, see [9]). Both the libraries and the skeletons are implemented on the server side and invoked remotely from the client. A skeleton may have several implementations on the Grid, each geared to a particular architecture of a Grid server, e.g. distributed- or shared-memory, multithreaded, etc. This provides potential for achieving portable performance across various target machines. Using skeletons for programming in the Grid has the following advantages:

- As skeletons are implemented on the server side, the implementation can be tuned to the particular server architecture, allowing hardware-specific optimizations.
- The 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. Thus, an application that is expressed as a composition of skeletons runs on any combination of servers implementing the required skeletons, without any hardware-specific adjustments.
- Skeletons provide a reliable model of performance prediction, offering a sound basis for selecting servers.

In an application program for the Grid, skeletons appear as function calls with application-specific parameters. Some of these parameters may in turn be program codes, i.e. skeletons can be formally viewed as higher-order functions. For specific examples of parallel skeletons and details of their use in programming applications, see [9].

There is a difference between using library functions and skeletons. When a library is used, the programmer supplies the structure of the application, the library providing application-independent utility routines. When skeletons are used, they supply the parallel structure of the application, while the user provides application-specific customizing operators (Java bytecodes in our system).

In the remainder of the paper, we use the word “method” for both library functions and skeletons.

2.3 Software Architecture

We propose the following system architecture, consisting of three kinds of components: user machines (*clients*), target machines (*compute servers*) and the central entity, called *lookup service* (see Fig. 2).

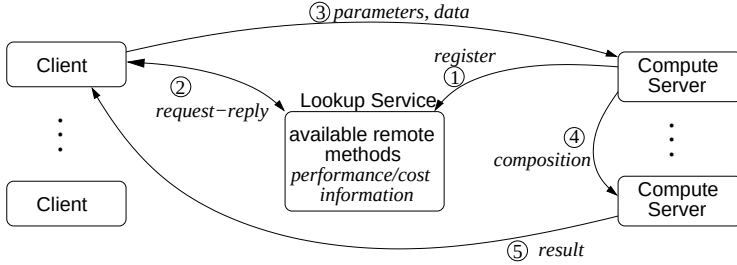


Fig. 2. System architecture and interaction of its parts

Each compute server provides a set of methods that can be invoked remotely from the clients. There are five main steps in the system's functionality, denoted by the circled numbers in Figure 2 (we provide more details below):

- ① Registration: Each server registers the methods it provides with the lookup service to make them accessible to clients. Together with each method, a performance-estimation function (see Section 5) is registered.
- ② Service request-reply: A client asks the lookup service for a method it needs for an application and is returned a list of servers implementing that method. The server combination that will actually be used is selected (using heuristics or tool-driven by the user). For each selected combination, a remote reference to the respective method implementation is obtained from the lookup service.
- ③ Method invocation: During program execution, methods are invoked remotely with application-specific parameters; one method invocation is always performed on only one of the servers.
- ④ Composition: If the application consists of several methods, they may all be executed either on the same server or, alternatively, in a pipelined manner across several servers.
- ⑤ Method completion: When the compute server has completed the invoked method, the result is sent back to the client.

The next section describes how the presented architecture is implemented.

3 Prototype System Implementation in Java

The system sketched in Figure 2 was implemented in Java, using RMI for communication. Java has several advantages for our purposes. First of all, Java byte-codes are portable across a broad range of machines. The method's customizing functional parameters can therefore be used on any of the server machines

without rewriting or recompilation. Moreover, Java and RMI provide simple mechanisms for invoking a method remotely on the server.

The interaction between the system components – client, compute server and lookup server – is realized by implementing a set of remote interfaces known to all components. Figure 3 shows a simplified UML class diagram for the most important classes and interfaces of our implementation. Solid lines connect interfaces and their implementing classes, while dashed lines denote the “uses” relationship.

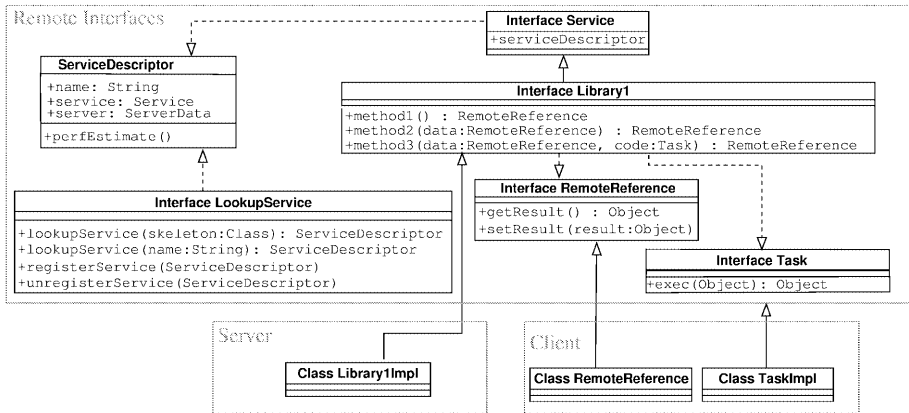


Fig. 3. Simplified class diagram of the implementation

Compute servers: For each library provided by a server, a corresponding interface is implemented. For example, in Figure 3 an interface **Library1** is shown for a library providing three methods. This interface is used by clients to call the methods on the server, where they are implemented by an object of class **Library1Impl**. The client can also provide code to be executed on the server, by implementing a particular interface, e.g. **Task** in the figure. The necessary code shipping is handled transparently by RMI. The system is easily extensible: to add new libraries, an appropriate interface must be specified and copied to the codebase, along with any other necessary interfaces (e.g. functional parameters). The interfaces can then be implemented on the server and registered with the lookup service in the usual manner.

Lookup service: The lookup service has a list of **ServiceDescriptors** (see Fig. 3), one per registered library/skeleton and implementing server. Each **ServiceDescriptor** consists of the library’s name, the implementing server’s address and a remote reference to the implementation on the server side. Clients and servers interact with the lookup service by calling methods of the **LookupService** interface shown in the class diagram: **registerService** is used by the servers to register their methods, and **lookupService** is used by the clients to query for a particular method.

4 Optimizing Java RMI for Grid

In this section, we discuss the specific advantages and disadvantages of Java RMI for remote execution in a Grid environment and present three optimizations that we have implemented to improve RMI for the Grid.

Intuitively, distributed execution of an application with remote methods should have the following desirable properties:

- **Ease of Programming:** From the programmer's point of view, remote invocation and distributed composition of methods should be expressed in a straightforward manner, resembling normal (local) composition of methods as far as possible.
- **Flexibility:** The assignment of servers should not be hardcoded into the program. Instead, it should be possible for a scheduling entity to change the assignment of servers at runtime to reflect changes in the environment.
- **Low Overhead:** The overhead incurred by invoking methods remotely from the client should be as low as possible.

Java's standard RMI mechanism satisfies the first two requirements: (1) a remote method call is expressed in exactly the same way as a local one, and (2) the server executing the method can be changed at runtime by changing the corresponding remote reference.

The time overhead of RMI for single remote method invocations can be substantial, but it has been drastically reduced thanks to current research efforts like Manta [10] and KaRMI [11]. An additional problem, not covered by these approaches, arises if remote method calls are composed with each other, which is the case in many applications. Let us consider a simple Java code fragment, where the result of `method1` is used as an argument by `method2`, as shown in Fig. 4.

```
... //get remote reference for server1/2
result1 = server1.method1();
result2 = server2.method2(result1);
```

Fig. 4. Sample Java code: composition of two methods

The execution of the code shown in Fig. 4 can be distributed: different methods potentially run on different servers, i. e. different RMI references are assigned to `server1` and `server2`. When such a program is executed on the Grid system of Fig. 2, methods are called remotely on a corresponding server. If a method's result is used as a parameter of other remote methods, the result of the first method should be sent directly to the second server (arrow ④ in Fig. 2). However, using RMI, the result of a remote method is always sent back to the client. We proceed now by first presenting the situation with the standard RMI mechanism (plain RMI) and then describing our optimizations.

Plain RMI: Using plain RMI for calling methods on the server has the advantage that remote methods are called in exactly the same way as local ones. Thus, the code in Fig. 4 would not change at all when using RMI instead of local methods. The only difference would be that `server1` and `server2` are RMI references, i. e. references to RMI stubs instead of “normal” objects.

However, using plain RMI to execute a composition of methods as in Fig. 4 is not time-efficient because the result of a remote method invocation is always sent back directly to the client. Fig. 5(a) demonstrates that assigning two different servers to `server1` and `server2` in our example code leads to the result of `method1` being sent back to the client, and from there to the second server. Furthermore, even if both methods are executed on the same server, the result is still sent first to the client, and from there back to the server again. For typical applications consisting of many composed methods, this feature of RMI results in very high overhead.

To eliminate this overhead of the plain RMI, we propose three optimizations, called lazy, localized and asynchronous RMI:

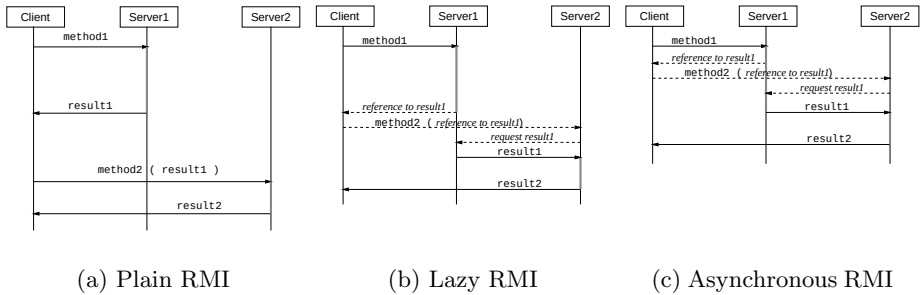


Fig. 5. Timing diagrams for the plain and two improved RMI versions

Lazy RMI: Our first optimization, called *lazy RMI*, aims to reduce the amount of data sent from the server to the client upon method completion. We propose that instead of the result being sent back to the client, an RMI remote reference to the data be returned. The client can then pass this reference on to the next server, which uses the reference to request the result from the previous server. This is shown in Fig. 5(b), with horizontal lines for communication of data, dotted horizontal lines for sending references and thick vertical lines denoting computations.

This mechanism is implemented by wrapping all return values and parameters in objects of the new class `RemoteReference`, which has two methods: `setValue()` is called to set a reference to the result of a call; `getValue()` is used by the next method (or by the client) to retrieve this result and may be called remotely. If `getValue()` is called remotely via RMI, the result is sent over

the network to the next server. Apart from the necessary packing and unpacking of parameters using `getValue` and `setValue`, a distributed composition of methods is expressed in exactly the same way with lazy RMI as with RMI.

Localized RMI: Our next optimization of RMI deals with accesses to the reference which points to the result of the first method in a composition. While there is no real network communication involved, there is still substantial overhead for serializing and deserializing the data and sending it through the local socket. To avoid this overhead, our implementation checks every access to a remote reference, whether it references a local object or not. In the local case, the object is returned directly without issuing an RMI call, thus reducing the runtime.

This is achieved by splitting the remote referencing mechanism into two classes: a remote class `RemoteValue` and a normal class `RemoteReference`. The local class is returned to the client upon method completion. It contains a remote reference to the result on the server, wrapped in a `RemoteValue` object. In addition, it contains a unique id for the object and the server's IP-address. When `getValue` is called at the `RemoteReference`, it first checks if the object is available locally and, if so, it obtains a local reference from a hashtable.

Asynchronous RMI: Since methods in Grid applications are invoked from the client, a method cannot be executed until the remote reference has been passed from the previous server to the client, and from there on to the next server. Returning to our example code in Fig. 4, even if both methods are executed on the same server, the second method cannot be executed until the remote reference for the result of the first has been sent to the client and back once, see Fig. 5(b).

This unnecessary delay offers an additional chance for optimization, which we call *asynchronous RMI*. The idea is that all method invocations immediately return a remote reference to the result. This reference is sent to the client and can be passed on to the next method. All attempts to retrieve the data referenced by this reference are blocked until the data becomes available. Thus, computations and communication between client and server overlap, effectively hiding communication costs. This is shown in Fig. 5(c), with thick vertical lines denoting computations. Since RMI itself does not provide a mechanism for asynchronous method calls, it is up to the implementation of the methods on the server side to make method invocation asynchronous, e.g. by spawning in the client a new thread to carry out computations and returning immediately.

5 Performance Prediction for Java Bytecodes

To achieve an efficient assignment of methods to servers, it is important to have an accurate estimate of a method's runtime on a particular server. There are well-developed performance-prediction functions for skeletons, described in [9]. Such functions usually depend on the size of the input parameters and on the number of processors used. The only remaining problem is how to estimate runtimes for methods that receive user-provided code as parameters: because the runtime of the code passed as a parameter is not known in advance, it is not possible

to estimate a method's runtime *a priori*. Therefore, to achieve realistic time estimates for remote method execution, it is necessary to also predict accurately the runtime of the customizing functional arguments (which are Java bytecodes). This task is complicated even further by the fact that skeletons are not executed locally on the client, but remotely on the server. Thus, it is not sufficient to execute the customizing code once on the client to measure its runtime. While the analysis of Java performance is a widely discussed issue, surprisingly little is known about predicting the performance of Java bytecodes on a remote machine.

We have developed a new approach, whose main feature is that it does not involve communication between client and server: to estimate the customizing function's runtime, we execute it in a special JVM on the client side, counting how often each instruction is invoked. The obtained numbers for each instruction are then multiplied by a time value for that instruction. Measurements lead to a system of linear equations, whose solution yields runtimes for single instructions.

Solving linear equations to obtain runtime estimates will only yield correct results if the runtime of a program is linear in terms of the number of instructions. This is not the case, however, for very small programs containing only a few instructions, as demonstrated by Tab. 1. We measured the times for executing 100 integer additions (along with the indispensable stack-adjustment operations) and 100 integer multiplications in a loop of 10^6 iterations.

Table 1. Runtime for a loop containing addition, multiplication and a combination of both, and runtimes for loops containing two inhomogeneous code sequences, P_1 and P_2 , of approx. 100 instructions.

Instruction	add	mul	add + mul	addmul
Time	286 ms	1429 ms	1715 ms	2106 ms
	P_1	P_2	$P_1 + P_2$	P_{12}
Time	1726 ms	1641 ms	3367 ms	3341 ms

The values obtained are given in the “add” and “mul” columns of the first row of Table 1. The time for executing a loop with both addition and multiplication would be expected to be the sum of the loops containing only addition or multiplication. In fact, a shorter time could be expected, as the combined loop contains more instructions per loop iteration, resulting in less overhead. However, the measured value (“addmul” in the first row of Table 1) of 2106 ms is considerably larger (approx. 23%) than the expected value of 1715 ms. Apparently, the JVM can only optimize loops that contain arithmetic instructions of one type, probably by loading the constant operator to a register before executing the loop, failing to do so for loops containing different instructions. By contrast, when measuring larger code sequences, linearity does hold. In the second row of Table 1, the runtimes taken for two generated code sequences of approx. 100 instructions are shown.

As can be seen, the sum of the execution times for programs P_1 and P_2 is quite close to the time for both codes executed in one program (P_{12}). One requirement for the construction of test programs is therefore that they should

not be too small and homogeneous. Otherwise, the executing JVM can extensively optimize the program, leading to unexpected timing results. Since it is practically impossible to produce “by hand” a sufficiently large number of test programs that satisfy the mentioned requirement, we generate these programs automatically, randomly selecting the bytecode instructions in them.

Our bytecode generator is implemented to automatically produce source files for the Jasmin bytecode assembler ([12]). It generates arbitrarily large bytecode containing randomly selected instructions. For more details about the generation process and the performance prediction method, see [9].

6 Experimental Results

In this section, we report measurements on our prototype system described in Section 2, using SUN’s JDK 1.4.1 and Java HotSpot Client VM in mixed mode (i. e. with JIT compiler enabled).

We compare the performance of plain and improved RMI on a simple example and on a linear system solver, and we demonstrate the accuracy of our bytecode performance prediction using a tridiagonal system solver.

6.1 Improved RMI on a Small Example

Our first series of experiments demonstrates the results of the RMI optimizations described in Section 4. We measured the performance of the small sample program from Fig. 4, with `method1` and `method2` both taking 250 ms, and the amount of data sent over the network ranging from 100 KB to 1 MB.

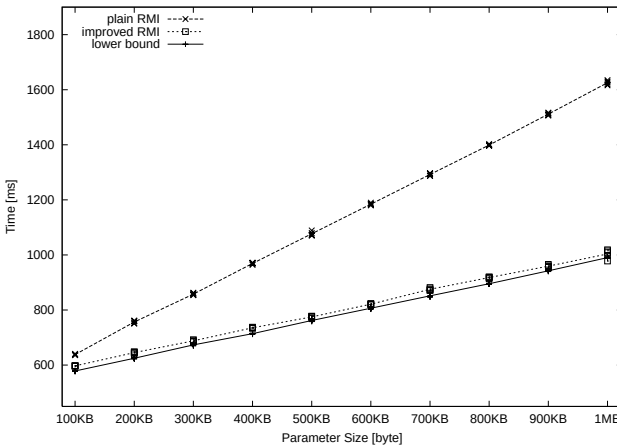


Fig. 6. Runtimes for the example in Fig. 4 using plain and improved RMI

Fig. 6 shows the runtimes for three versions of the program: (1) two calls with plain RMI, (2) two calls with improved RMI, and (3) one call which takes twice

as much time as the original method call. We regard the one-method version as providing an ideal runtime (“lower bound”) for a composition of remote methods. The figure shows five measurements for each version of the program, with the average runtimes for each parameter size connected by lines for the sake of clarity.

The figure shows that the improved RMI version’s runtime is between 40 ms and 620 ms faster than the standard RMI version, depending on the size of the parameters. Considering only communication times (i. e. subtracting 500 ms for computations on the server side), the time for standard RMI is approximately twice as long as for the improved version. This shows clearly that the communication time for the second, composed method call is almost completely hidden owing to the laziness and asynchrony introduced by our optimizations. The composition under improved 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.

6.2 Improved RMI on a Linear System Solver

To study the efficiency of our improved RMI mechanism on a more complex application, we have written a remote wrapper class for the linear algebra library Jama (cf. [13]). As an example application, we have implemented a solver for systems of linear equations. The implementation consists of a sequence of composed library calls for solving a minimalization problem, for matrix multiplication and subtraction to compute the residual.

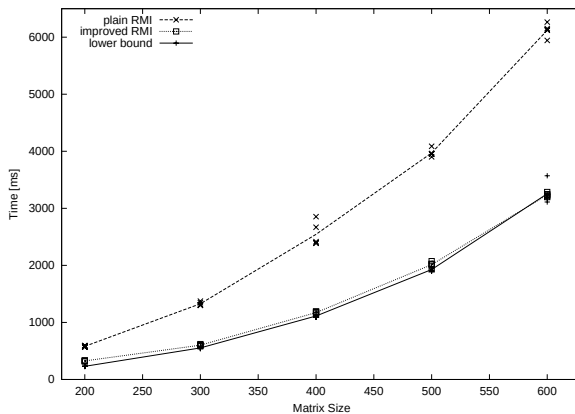


Fig. 7. Measured runtimes for the case study

Fig. 7 shows the runtimes of three versions of the solver: the two versions presented above (plain RMI and improved RMI) and one implementation running completely on the server side (“ideal case”). The measurements for the case

study basically confirm the results already presented for the simple example program in Section 6.1: the improved RMI version is less than 10 % slower than the ideal version, so it eliminates most of the overhead of plain RMI.

6.3 Performance Prediction for a Tridiagonal Solver

To evaluate our approach for predicting the performance of Java bytecodes, we have implemented a solver for tridiagonal equation systems (TDS), using a special divide-and-conquer skeleton called *distributable homomorphism* (DH). This skeleton receives three parameters: a list of input values and two functional arguments (hence called operators). For details about the solver and the DH skeleton, see [9].

As the DH skeleton receives two operators as parameters, its runtime depends not only on the problem size but also on the operators' runtime. We therefore used the approach presented in Section 5 to predict the runtime of the operators and used the obtained estimate to predict the overall runtime of the solver.

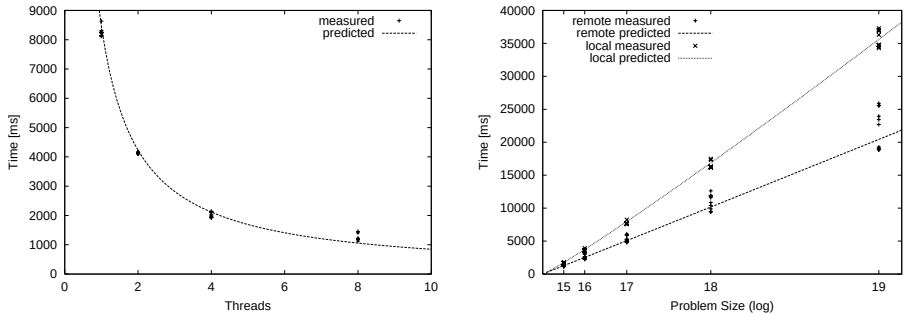


Fig. 8. Left: Execution time of TDS/DH for 2^{18} equations and 1, 2, 4 and 8 threads, executing locally on the server. Right: Execution time of TDS/DH using 8 threads on the server side (“remote”) compared with the local execution time. Problem size varies between 2^{15} and 2^{19} .

Fig. 8 (left) shows the predicted and measured time values for executing the DH skeleton with TDS operators (TDS/DH) locally on the server. The times were measured for 1, 2, 4 and 8 threads and 2^{18} equations. The predicted values correspond very well to the measured values for the skeleton execution.

In Figure 8 (right), predicted and measured runtimes for remote execution are shown, with the client invoking the computations on the server remotely over the WAN and using 8 threads on the server side (“remote”). The second set of values in the figure (“local”) were obtained by executing the skeleton locally on the client side.

Although the predicted and measured values differ to some extent for large problem sizes (up to 21 % for 2^{19}), the predicted values still match the actual

values quite well, all estimates being within the range of the measured values. We assume that the large deviations for the remote execution with 2^{19} elements stem from varying network and server loads.

7 Conclusion

We have introduced an experimental Grid environment, based on Java and RMI, and its programming system.

Our work addresses the challenge of algorithm design for Grids by using two kinds of remote methods: traditional library routines and higher-order, parameterized programming constructs, called skeletons. Java+RMI was chosen to implement our system in order to obtain a highly portable solution. Though Java and RMI performance is still limited, it was substantially improved thanks to JIT compilers and current research efforts like Manta [10] and KaRMI [11].

The novelty of our work on RMI is that whereas previous research dealt with single or repeated RMI calls, we focus on an efficient execution where the result of one call is an argument of another. This situation is highly typical of many Grid applications, and our work has demonstrated several optimizations to improve the performance of such calls. An important advantage of our approach is that it is orthogonal to the underlying RMI implementation and can be used along with faster RMI systems. One drawback of the improved RMI implementation is that static type checking is limited to local methods. This problem can be eliminated by creating a `RemoteReference` class for all classes used, in much the same way that Java RMI uses `rmic` to create stub classes for classes accessed remotely.

The performance analysis of portable code, e.g. Java bytecode, has only recently been studied. Initial research efforts [14,15] are concerned with the high-level analysis of bytecode, i.e. the problem of counting how often an instruction is executed in the worst case. We have presented a novel mechanism for performance estimation using automatically generated test programs. Our experiments confirm the high quality of time estimates, allowing us to predict the performance of Grid programs during the design process and also to control the efficient assignment of remote methods to the compute servers of the Grid.

Acknowledgments. We wish to thank the anonymous referees for their helpful comments and Phil Bacon who helped us to greatly improve the presentation.

References

1. Foster, I., Kesselmann, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (1998)
2. Casanova, H., Dongarra, J.: NetSolve: A network-enabled server for solving computational science problems. *Int. J. of Supercomputing Applications and High Performance Computing* **3** (1997) 212–223
3. 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

4. Kennedy, K., et al.: Toward a framework for preparing and executing adaptive grid programs. (In: IPDPS'2002) To appear.
5. Berman, F., et al.: The GrADS project: Software support for high-level Grid application development. *Int. J. of High Performance Computing Applications* **15** (2001) 327–344
6. Matsuoka, S., Nakada, H., Sato, M., Sekiguchi, S.: Design issues of network enabled server systems for the grid. *GridForum, APM WG whitepaper* (2000)
7. Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Seymour, K., Sagi, K., Shi, Z., Vadhiyar, S.: Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN (2002)
8. Nakada, H., Sato, M., Sekiguchi, S.: Design and implementations of Ninf: towards a global computing infrastructure. *FGCS* **15** (1999) 649–658
9. Alt, M., Bischof, H., Gorlatch, S.: Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters* **12** (2002) 157–174
10. Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., Hofman, R.: Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **23** (2001) 747–775
11. Philippsen, M., Haumacher, B., Nester, C.: More efficient serialization and RMI for Java. *Concurrency: Practice and Experience* **12** (2000) 495–518
12. Meyer, J., Downing, T.: *Java Virtual Machine*. O'Reilly (1997)
13. Hicklin, J., Moler, C., Webb, P., Boisvert, R.F., Miller, B., Pozo, R., Remington, K.: JAMA: A Java matrix package. (<http://math.nist.gov/javanumerics/jama/>)
14. Bate, I., Bernat, G., Murphy, G., Puschner, P.: Low-level analysis of a portable WCET analysis framework. In: *6th IEEE Real-Time Computing Systems and Applications (RTCSA2000)*. (2000) 39–48
15. Bernat, G., Burns, A., Wellings, A.: Portable Worst Case execution time analysis using Java Byte Code. In: *Proc. 12th EUROMICRO conference on Real-time Systems*. (2000)

Design and Implementation of a Cost-Optimal Parallel Tridiagonal System Solver Using Skeletons

Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann

Technische Universität Berlin, Germany
{bischof,gorlatch,jemanuel}@cs.tu-berlin.de

Abstract. We address the problem of systematically designing correct parallel programs and developing their efficient implementations on parallel machines. The design process starts with an intuitive, sequential algorithm and proceeds by expressing it in terms of well-defined, pre-implemented parallel components called *skeletons*. We demonstrate the skeleton-based design process using the tridiagonal system solver as our example application. We develop step by step three provably correct, parallel versions of our application, and finally arrive at a cost-optimal implementation in MPI (Message Passing Interface). The performance of our solutions is demonstrated experimentally on a Cray T3E machine.

1 Introduction

The design of parallel algorithms and their implementation on parallel machines is a complex and error-prone process. Traditionally, application programmers take a sequential algorithm and use their experience to find a parallel implementation in an *ad hoc* manner. A more systematic approach is to use well-defined, reusable components or patterns of parallelism, called *skeletons* [1]. A skeleton can be formally viewed as a higher-order function, customizable for a particular application by means of functional parameters provided by the application programmer. The programmer expresses an application using skeletons as high-level language constructs, whose highly efficient implementations for particular parallel machines are provided by a compiler or library.

The first parallel skeletons studied in the literature were traditional second-order functions known from functional programming: map, reduce, scan, etc. 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. The challenge in skeleton-based program design is to find a systematic way of either adjusting a given application to an available set of skeletons or introducing a new skeleton and developing its efficient implementation.

This paper addresses the task of parallel program design for a practically relevant case study – solving a tridiagonal system of linear equations. Tridiagonal systems have traditionally been considered difficult to parallelize: their sparse structure provides relatively little potential parallelism, while communication demand is relatively high (see [2] for an overview and Section 8 for more details).

The paper’s contribution is that, unlike previous *ad hoc* approaches, we systematically transform an intuitive sequential formulation into a skeleton-based form, ultimately providing an efficient, cost-optimal parallel implementation of our case study in MPI.

The paper is organized as follows:

- We describe a repository containing basic data-parallel skeletons used in the case study (Section 2).
- We express our case study – the tridiagonal system solver – using the basic skeletons and discuss its potential parallelization (Section 3).
- We describe a systematic adjustment of our application to a special divide-and-conquer skeleton DH, thus arriving at a first parallel implementation (Section 4).
- We demonstrate an alternative design option using the double-scan skeleton for the case study (Section 5).
- We further improve our solution by introducing a new intermediate data structure, called *plist*, and finally arrive at a cost-optimal parallel implementation of the tridiagonal solver in MPI (Section 6).
- We experimentally study the performance of the developed MPI implementations on a Cray T3E machine (Section 7).

We conclude the paper by discussing our results in the context of related work.

2 Basic Data-Parallel Skeletons

In this section, we introduce some basic data-parallel skeletons as higher-order functions defined on non-empty lists, 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: Element-wise application of a binary operator $\oplus$ to a pair of lists of equal length:

$$\text{zip}(\oplus)([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1 \oplus y_1), \dots, (x_n \oplus y_n)]$$

- 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” 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 scans are computed sequentially with linear time complexity.

The programming methodology using skeletons involves two groups of programmers, whose tasks complement each other: (1) a system programmer implements skeletons on a target parallel system, and (2) an application programmer expresses applications using available skeletons as implicitly parallel program components. An important advantage of this approach is that the user develops programs without having to consider the particular features of parallel machines.

3 Case Study: Formulation of the Problem

We consider solution of a tridiagonal system of linear equations, $A \cdot x = b$, where A is an $n \times n$ matrix representing coefficients, x a vector of unknowns and b the right-hand-side vector. The only values of matrix A unequal to 0 are on the main diagonal as well as above and below it (we call them the upper and lower diagonal, respectively), as demonstrated by equation (1).

$$\begin{pmatrix} a_{12} & a_{13} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1,1} & a_{n-1,2} & a_{n-1,3} \\ 0 & & & a_{n,1} & a_{n,2} \end{pmatrix} \cdot x = \begin{pmatrix} a_{14} \\ a_{24} \\ \vdots \\ a_{n-1,4} \\ a_{n,4} \end{pmatrix} \quad (1)$$

A typical sequential algorithm for solving a tridiagonal system is Gaussian elimination (see, e.g., [3,4]) which eliminates the lower and upper diagonal of the matrix as shown in Fig. 1. Both the first and last column in the figure consist of fictitious zero elements, introduced for the sake of convenience.

$$\begin{pmatrix} \bullet & \bullet & \bullet & \bullet & 0 \\ & \bullet & \bullet & \bullet & \\ & \ddots & \ddots & \ddots & \\ 0 & & \bullet & \bullet & \bullet \end{pmatrix} \xrightarrow{(1)} \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & 0 \\ \vdots & \bullet & \bullet & \bullet & \\ \bullet & 0 & \bullet & \bullet & \bullet \end{pmatrix} \xrightarrow{(2)} \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \vdots & \bullet & \bullet & \bullet & \vdots \\ \bullet & 0 & \bullet & \bullet & \bullet \end{pmatrix}$$

Fig. 1. The intuitive algorithm for solving a tridiagonal system of equations consists of two stages: (1) elimination of the lower diagonal (2) elimination of the upper diagonal.

The two stages of the algorithm traverse the list of rows, applying operators denoted by ① and ②, which are informally defined below:

1. The first stage eliminates the lower diagonal by traversing matrix A from top to bottom according to the *scanl* skeleton and applying the following operator ② on the rows pairwise:

$$(a_1, a_2, a_3, a_4) \textcircled{2} (b_1, b_2, b_3, b_4) = \left(a_1, a_3 - \frac{b_2 a_2}{b_1}, -\frac{b_3 a_2}{b_1}, a_4 - \frac{b_4 a_2}{b_1} \right)$$

2. The second stage eliminates the upper diagonal of the matrix by a bottom-up traversal, i. e. using the *scanr* skeleton and applying the following operator ① on pairs of rows:

$$(a_1, a_2, a_3, a_4) \textcircled{1} (b_1, b_2, b_3, b_4) = \left(a_1 - \frac{b_1 a_3}{b_2}, a_2, -\frac{b_3 a_3}{b_2}, a_4 - \frac{b_4 a_3}{b_2} \right)$$

Now we can specify the described Gaussian elimination algorithm as function *tds* (*tridiagonal system*), which works on the list of rows in two stages:

$$tds = scanr(\textcircled{1}) \circ scanl(\textcircled{2}) \quad (2)$$

where $\circ$ denotes function composition from right to left, i. e. $(f \circ g) x = f(g(x))$.

In the search for an alternative representation of the algorithm, we can also eliminate first the upper and then the lower diagonal using two new row operators, ③ and ④:

$$\begin{aligned} (a_1, a_2, a_3, a_4) \textcircled{4} (b_1, b_2, b_3, b_4) &= \left(a_1, a_2 - \frac{b_1 a_3}{b_2}, -\frac{b_3 a_3}{b_2}, a_4 - \frac{b_4 a_3}{b_2} \right) \\ (a_1, a_2, a_3, a_4) \textcircled{3} (b_1, b_2, b_3, b_4) &= \left(a_1, -\frac{b_2 a_2}{b_1}, a_3 - \frac{b_3 a_2}{b_1}, a_4 - \frac{b_4 a_2}{b_1} \right) \end{aligned}$$

This alternative version of the algorithm can be specified as follows:

$$tds = scanl(\textcircled{3}) \circ scanr(\textcircled{4}) \quad (3)$$

Neither of the intuitive algorithms (2) and (3) is directly parallelizable because operations ① and ③ are non-associative. Thus both algorithms prescribe strictly sequential execution, and special effort is necessary for parallelization.

4 Version 1: Design by Adjustment to DH

Our first attempt at parallelizing the function *tds* involves expressing it in terms of a known parallel skeleton. We use the DH (distributable homomorphism) skeleton, first introduced in [5]:

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:*

$$\begin{aligned} dh(\oplus, \otimes)[a] &= [a], \\ dh(\oplus, \otimes)(x \uplus y) &= zip(\oplus)(dh\ x, dh\ y) \uplus zip(\otimes)(dh\ x, dh\ y) \end{aligned} \quad (4)$$

The DH skeleton is a special form of the well-known divide-and-conquer paradigm: to compute *dh* on a concatenation of two lists, $x \uplus y$, we apply *dh* to x and y , then combine the results elementwise using *zip* with operators $\oplus$ and $\otimes$ and concatenate them. For this skeleton, there exists a family of generic parallel implementations, directly expressible in MPI [6].

Our adjustment proceeds in two steps: first we consider how the algorithm (2) works on the input list divided according to (4), then we massage the conquer part to fit the DH format.

4.1 Adjustment to DH: Divide Phase

In the first step, we aim at a divide-and-conquer representation of function tds , where the divide phase fits the DH format:

$$tds(x \uplus y) = (tds\ x) \oslash (tds\ y) \quad (5)$$

Here, $\oslash$ is some combine operation whose exact format is of no concern to us at this stage. To find a representation for $\oslash$, we note that applying function tds to the input matrix yields a matrix whose only non-zero elements are on the diagonal and in the first and the last column; see Fig. 2. We call such a matrix the N -matrix because the non-zero elements resemble a letter N.

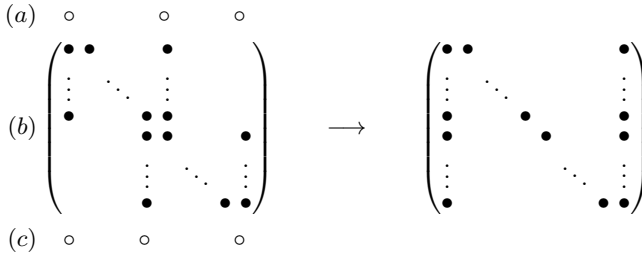


Fig. 2. Combining two N -matrices

The combine operation $\oslash$ of (5) takes two N -matrices and produces an N -matrix of double size, as shown in Fig. 2(b). Therefore, $\oslash$ must eliminate the last column of the first N -matrix and the first column of the second N -matrix.

To eliminate the last column of the first N -matrix, we use a row with non-zero values in the first and last column of the first N -matrix and in the last column of the second N -matrix; these elements are represented by $\circ$ in Fig. 2(a). Such a row can be obtained as $l_1 \oslash f_2$, where l_1 denotes the last row of the first N -matrix and f_2 denotes the first row of the second N -matrix. Now, using the operator $\textcircled{1}$, we can eliminate the last column of the first N -matrix.

Analogously, we can use operator $\textcircled{4}$ to obtain the row shown in Fig. 2(c), which is obtained as $l_1 \textcircled{4} f_2$, and operator $\textcircled{3}$ to eliminate the first column of the second N -matrix.

Function tds of (5) can now be rewritten in the divide-and-conquer format using the introduced row operations $\textcircled{1}$, $\textcircled{2}$, $\textcircled{3}$ and $\textcircled{4}$ as follows:

$$tds(x \uplus y) = \text{map}(g_1)(tds\ x) \uplus \text{map}(g_2)(tds\ y), \quad (6)$$

$$\begin{aligned} \text{where } g_1(a) &= a \textcircled{1} (l_1 \textcircled{2} f_2) & g_2(a) &= (l_1 \textcircled{4} f_2) \textcircled{3} a \\ l_1 &= (\text{last} \circ tds)\ x & f_2 &= (\text{first} \circ tds)\ y \end{aligned} \quad (7)$$

Here, *first* and *last* yield the first and the last element of a list, respectively.

4.2 Adjustment to DH: Conquer Phase

Although our representation (6)-(7) is already in the divide-and-conquer format, its combine operation, i.e. the right-hand side of (6), still does not fit the DH format (4). Its further adjustment is our task in this subsection.

First, we can immediately rewrite (6) by expressing *map* in terms of *zip*:

$$\begin{aligned} tds(x \uplus y) &= zip(g_1 \circ \pi_1)(tds\ x, tds\ y) \uplus zip(g_2 \circ \pi_2)(tds\ x, tds\ y), \\ \text{where } \pi_1(a, b) &= a \text{ and } \pi_2(a, b) = b. \end{aligned} \quad (8)$$

The remaining problem of format (8) is the dependence of its right-hand side on g_1 and g_2 , and thus according to (7) on l_1 and f_2 . A common trick applied in such a situation (see, e.g., [7]) is to add to *tds* an auxiliary function *fl*, which computes both l_1 and f_2 . If the resulting “tupled” function $\langle tds, fl \rangle$ becomes a DH, then it can be computed in parallel, its first component yielding the value of function *tds*, which we wish to compute; in other words, *tds* is a so-called “almost-DH”.

To obtain a DH representation of function *fl*, we use the same divide-and-conquer approach as for *tds* in Sect. 4.1. Let us consider how two pairs of quadruples, representing the first and last row of two N -matrices (left-hand side of Fig. 2), can be transformed into the pair containing the first and last row of the resulting N -matrix (right-hand side of Fig. 2). This computation can be expressed in the DH format using a new operation, $\odot$, as follows:

$$fl(x \uplus y) = zip(\odot)(fl\ x, fl\ y) \uplus zip(\odot)(fl\ x, fl\ y), \quad (9)$$

$$\text{where} \quad \begin{pmatrix} f_1 \\ l_1 \end{pmatrix} \odot \begin{pmatrix} f_2 \\ l_2 \end{pmatrix} = \begin{pmatrix} f_1 \textcircled{1} (l_1 \textcircled{2} f_2) \\ (l_1 \textcircled{4} f_2) \textcircled{3} l_2 \end{pmatrix}$$

From (8) and (9), it follows that the tupled function is itself a DH:

$$\langle tds, fl \rangle = dh(\oplus, \otimes), \quad (10)$$

where

$$\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ f_2 \\ l_2 \end{pmatrix} = \begin{pmatrix} a_1 \textcircled{1} (l_1 \textcircled{2} f_2) \\ f_1 \textcircled{1} (l_1 \textcircled{2} f_2) \\ (l_1 \textcircled{4} f_2) \textcircled{3} l_2 \end{pmatrix} \quad \begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ f_2 \\ l_2 \end{pmatrix} = \begin{pmatrix} (l_1 \textcircled{4} f_2) \textcircled{3} a_2 \\ f_1 \textcircled{1} (l_1 \textcircled{2} f_2) \\ (l_1 \textcircled{4} f_2) \textcircled{3} l_2 \end{pmatrix} \quad (11)$$

To compute our original function *tds*, we note that the tupled function (10) operates on lists of triples of quadruples and that *tds* is its first component:

$$tds = map(\pi_1) \circ dh(\oplus, \otimes) \circ map(triple) \quad (12)$$

Here, function *triple* creates a triple of an element, i.e. $triple\ a = (a, a, a)$, and function π_1 extracts the first element of a triple, i.e. $\pi_1(a, b, c) = a$. As a result, we have proved that function *tds* can be computed according to (12), as a component of the tupled DH function (10).

4.3 Implementation

When studying parallel implementations of skeletons, we assume for the rest of the paper that both the input data and results are distributed blockwise among the processes. A generic implementation schema of the DH skeleton was developed in [5]; its communication pattern is hypercube-like. A generic implementation for this schema is given as MPI pseudocode in Fig. 3:

```

local_dh(data);
for (dim=1; dim<p; dim<=<=1){
  neighbour = my_rank^dim;
  MPI_Sendrecv(data, neighbour, ..., tmp, neighbour, ...);
  if (my_rank < dest) oplus( tmp, data);
  else otimes( tmp, data);
}

```

Fig. 3. MPI pseudocode for the generic implementation of the DH skeleton

The program in Fig. 3 consists of two stages:

1. A sequential computation of the function in all processes simultaneously on their blocks. For the tridiagonal system solver, this obviously takes time $\Theta(n/p)$ if the data of length n is distributed evenly across p processes.
2. The second step is a sequence of $\log p$ 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 process. This second step has a complexity of $\Theta(n/p \cdot \log p)$.

Thus, the overall time taken to solve a tridiagonal system of size n in parallel on p processes using (11)-(12) is $t_{dh} \in \Theta(n/p \cdot \log p)$. While derived systematically in a provably correct manner, our first solution is suboptimal compared with the best solutions known from the literature.

5 Version 2: Adjustment to Double-Scan

An alternative to DH is the *double-scan* skeleton (DS) introduced in [8]:

Definition 2. For binary operators $\oplus$ and $\otimes$, where $\oplus$ is associative, two double-scan (DS) skeletons are defined:

$$\text{scanrl}(\oplus, \otimes) = \text{scanr}(\oplus) \circ \text{scanl}(\otimes) \quad (13)$$

$$\text{scanlr}(\oplus, \otimes) = \text{scanl}(\oplus) \circ \text{scanr}(\otimes) \quad (14)$$

Both double-scan skeletons have two functional parameters, which are the base operators of their constituent scans.

The following theorem provides the sufficient conditions under which the DS skeleton can be expressed using the DH skeleton.

Theorem 1. *Let ①, ②, ③ and ④ be binary operators, where ① and ③ are associative (②, ④ need not be associative). If the following equality holds:*

$$\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{scanlr}(\textcircled{3}, \textcircled{4}) \quad (15)$$

then the following holds:

$$\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{map}(\pi_1) \circ dh(\oplus, \otimes) \circ \text{map}(\text{triple}) \quad (16)$$

where $dh(\oplus, \otimes)$ is a DH with the following operations:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \oplus \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 \textcircled{1} (a_3 \textcircled{2} b_2) \\ a_2 \textcircled{1} (a_3 \textcircled{2} b_2) \\ (a_3 \textcircled{4} b_2) \textcircled{3} b_3 \end{pmatrix} \quad \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} (a_3 \textcircled{4} b_2) \textcircled{3} b_1 \\ a_2 \textcircled{1} (a_3 \textcircled{2} b_2) \\ (a_3 \textcircled{4} b_2) \textcircled{3} b_3 \end{pmatrix} \quad (17)$$

For the theorem's proof, see [9].

Theorem 1 states that all double scans satisfying the theorem's conditions can be implemented using three steps in (16). For the second step, which is the main part of algorithm (16), the generic implementation of the DH skeleton given in Fig. 3 can be used. The two additional adjustment functions before and afterwards (steps one and three) can be implemented by local computation with linear time complexity.

To apply Theorem 1 to our example of a tridiagonal system solver, we must show that operators ① in (2) and ③ in (3) are associative. This is demonstrated below using the associativity of addition and multiplication and the distributivity of multiplication over addition:

$$\begin{aligned} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{1} \left[\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \textcircled{1} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \right] &= \begin{pmatrix} a_1 - \frac{b_1 a_3}{b_2} + \frac{c_1 b_3 a_3}{c_2 b_2} \\ a_2 \\ \frac{c_3 b_3 a_3}{c_2 b_2} \\ a_4 - \frac{b_4 a_3}{b_2} + \frac{c_4 b_3 a_3}{c_2 b_2} \end{pmatrix} = \left[\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{1} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \right] \textcircled{1} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \\ \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{3} \left[\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \textcircled{3} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \right] &= \begin{pmatrix} a_1 \\ \frac{c_2 b_2 a_2}{c_1 b_1} \\ a_3 - \frac{b_3 a_2}{b_1} + \frac{c_3 b_2 a_2}{c_1 b_1} \\ a_4 - \frac{b_4 a_2}{b_1} + \frac{c_4 b_2 a_2}{c_1 b_1} \end{pmatrix} = \left[\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{3} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \right] \textcircled{3} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \end{aligned}$$

Compared with the development of the DH-based solution, the adjustment process to DS is definitely much simpler.

Let us analyze the quality of the obtained parallel solution. An important efficiency criterion is the cost of parallel algorithms, which is defined as the product of the required time and the number of processes used, i.e. $c = t \cdot p$. A parallel implementation is called *cost-optimal* on p processes, iff its cost equals the cost when using one process, i.e. $c_p = p \cdot t_p \in \Theta(t_{seq})$.

Implementation (16) has a time complexity of $\Theta(n/p \cdot \log p)$. Thus it is not cost-optimal:

$$c_p \in \Theta(n \cdot \log p) \neq \Theta(n) = \Theta(t_{seq}) \quad (18)$$

This motivates our further search for a better parallel implementation.

6 Version 3: Towards a Cost-Optimal Solution

In this section, we identify a special case of the double-scan skeleton with a cost-optimal generic implementation and use it for our case study.

6.1 Computing Double-Scan on a Plist

We exploit 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:

$$\begin{array}{ccccccccccc} l_1 & a_1 & l_2 & a_2 & l_3 & a_3 & \dots & a_{k-1} & l_k \\ \hline & \bullet & & \bullet & & \bullet & & \bullet & \end{array}$$

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 functions defined on plists with the letter p , e. g. $pmap$.

On a parallel machine, we partition plists so that each segment and its right “border point” are mapped to one process. The last process 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.

We now develop a parallel implementation for a distributed version of $scanrl$, function $pscanrl$, that computes $scanrl$ on a plist. The following proposition provides a method to compute the distributed version of the double-scan skeleton:

Theorem 2. *Let $\textcircled{1}$, $\textcircled{2}$, $\textcircled{3}$, $\textcircled{4}$ be binary operators, where $\textcircled{1}$ and $\textcircled{3}$ are associative, $scanrl(\textcircled{1}, \textcircled{2}) = scanrl(\textcircled{3}, \textcircled{4})$, and $\textcircled{2}$ is associative modulo $\textcircled{4}$. Moreover, let $\textcircled{5}$ be a three-adic operator working on a pair and an element, for which it holds:*

$$(a, a \textcircled{3} c) \textcircled{5} b = a \textcircled{3} (b \textcircled{1} c) \quad (19)$$

Then, the double-scan skeleton $pscanrl$ on plists can be implemented as follows:

$$\begin{aligned} pscanrl(\textcircled{1}, \textcircled{2}) &= pinmap\_l(\textcircled{5}, \textcircled{1}, \textcircled{3}) \circ pscanrl\_p(\textcircled{1}, \textcircled{2}) \\ &\quad \circ pinmap\_p(\textcircled{2}, \textcircled{4}) \circ (pmap\_l\ scanrl(\textcircled{1}, \textcircled{2})) \end{aligned} \quad (20)$$

Here, a binary operation $\textcircled{*}$ is *associative modulo* $\textcircled{\odot}$, iff for arbitrary elements a, b, c it holds: $(a \textcircled{*} b) \textcircled{*} c = (a \textcircled{\odot} b) \textcircled{*} (b \textcircled{*} c)$. Usual associativity is modulo *first*, which yields the first element of a pair. For the theorem’s proof, see [9].

The right-hand side of (20) consists of four higher-order functions on plists. They are illustrated in Fig. 4 and informally defined as follows:

1. $pmap\_l\ scanrl(\textcircled{1}, \textcircled{2})$ applies function $scanrl(\textcircled{1}, \textcircled{2})$, which operates on usual lists, to all segments of a plist.
2. $pinmap\_p(\textcircled{2}, \textcircled{4})$ modifies each single point of a plist, depending on the last element of the left neighbouring segment and the first element of the right neighbouring segment.
3. $pscanrl\_p(\textcircled{1}, \textcircled{2})$ applies function $scanrl(\textcircled{1}, \textcircled{2})$, defined in (13), to the list containing only the points of the argument plist.
4. $pinmap\_l(\textcircled{5}, \textcircled{1}, \textcircled{3})$ modifies each segment of a plist depending on the neighboring points, using operation $\textcircled{1}$ for the left-most segment, $\textcircled{3}$ for the right-most segment and three-adic operation $\textcircled{5}$ for all inner segments.

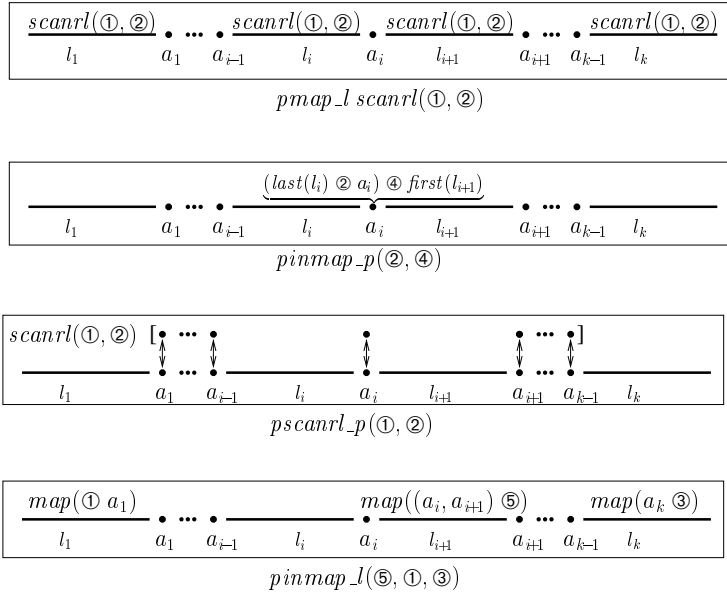


Fig. 4. Graphical illustration of the functions in (20)

6.2 Cost Optimality of the Double-Scan Implementation

Let us analyze the $pscanrl$ implementation provided by Theorem 2. The right-hand side of (20) consists of four stages shown in Fig. 4, which are executed from right to left:

1. $pmap\_l\ scanrl(\textcircled{1}, \textcircled{2})$: If the argument plist is partitioned among p processes as described above, then function $scanrl(\textcircled{1}, \textcircled{2})$ can be applied simultaneously by all processes. If n is the size (number of the elements) of the plist, then the complexity is $\Theta(n/p)$.
2. To compute $pinmap\_p(\textcircled{2}, \textcircled{4})$, each process sends its first element to the preceding process and receives the first element from the next process. Then operations $\textcircled{2}$ and $\textcircled{4}$ are applied to the last element of each process. This results in a complexity of $\Theta(1)$.
3. To compute $pscanrl\_p$, the generic DH implementation provided in Fig. 3 can be used directly, thus leading to a complexity of $\Theta(\log p)$.
4. To compute $pinmap\_l(\textcircled{5}, \textcircled{1}, \textcircled{3})$, each process sends its last element to the next process. Then operation $\textcircled{5}$ is applied to the elements of the “inner” processes. The elements of the first process are manipulated by $\textcircled{1}$, and the elements of the last process by $\textcircled{3}$. The computations in the processes are mutually independent, which results in a complexity of $\Theta(n/p)$.

As an overall time complexity, we obtain:

$$T_p \in \Theta(n/p) + \Theta(\log p) + \Theta(1) + \Theta(n/p) = \Theta(n/p + \log p)$$

which results in the cost $c_p \in \Theta(n + p \cdot \log p)$. Thus, we have proved the following proposition:

Theorem 3. *The parallel implementation (20) of double-scan is cost-optimal on $p \in O(n/\log n)$ processes.*

6.3 Cost-Optimal Tridiagonal System Solver

To apply the result of Theorem 2 to our case study of a tridiagonal system of equations, we must show the associativity of $\textcircled{2}$ modulo $\textcircled{4}$ and find an operation $\textcircled{5}$ so that (19) holds. The following result is helpful for the latter task:

Lemma 1. *If function $(a\textcircled{3})$ is bijective for all a with inverse $(a\textcircled{3})^{-1}$, then operator $\textcircled{5}$, defined as $(a, d) \textcircled{5} b = a \textcircled{3} (b \textcircled{1} (a\textcircled{3})^{-1}(d))$, fulfils (19).*

For the lemma's proof, see [9].

When trying to apply Lemma 1 to find $\textcircled{5}$, we find that function $(a\textcircled{3})$ is non-bijective. The remedy is to normalize the tridiagonal system's matrix, i.e. make all elements of the main diagonal equal to 1, and redefine operations $\textcircled{1}, \dots, \textcircled{4}$ so that they preserve the normalization property – we will call them $\textcircled{1}_{\text{norm}}, \dots, \textcircled{4}_{\text{norm}}$, respectively. The result is the bijective $(a\textcircled{3}_{\text{norm}})$, with

$$(a_1, 1, a_3, a_4) \textcircled{3}_{\text{norm}} (b_1, 1, b_3, b_4) = (-a_1 b_1, 1, b_3 - a_3 b_1, b_4 - a_4 b_1)$$

while the inverse of $\textcircled{3}_{\text{norm}}$ is:

$$(a_1, 1, a_3, a_4) \textcircled{3}_{\text{norm}}^{-1} (b_1, 1, b_3, b_4) = \left(-\frac{b_1}{a_1}, 1, b_3 - \frac{a_3 b_1}{a_1}, b_4 - \frac{a_4 b_1}{a_1}\right) \quad (21)$$

Using (21) and Lemma 1, we obtain the desired operation $\textcircled{5}_{\text{norm}}$ for the tridiagonal system solver.

The normalized operations $\textcircled{1}_{\text{norm}}$ and $\textcircled{3}_{\text{norm}}$ are associative (see [9] for a simple proof). The associativity of $\textcircled{2}_{\text{norm}}$ modulo $\textcircled{4}_{\text{norm}}$ is verified as follows:

$$(a \textcircled{2} b) \textcircled{2} c = \left(\begin{array}{c} -\frac{a_1 b_1 c_1}{b_3 c_1 + a_3 b_1 - 1} \\ 1 \\ c_3 \cdot \frac{a_3 b_1 - 1}{b_3 c_1 + a_3 b_1 - 1} \\ -\frac{c_1(a_4 b_1 - b_4) - c_4(a_3 b_1 - 1)}{b_3 c_1 + a_3 b_1 - 1} \end{array} \right) = (a \textcircled{4} b) \textcircled{2} (b \textcircled{2} c)$$

Now Theorem 2 can be applied, substituting the operations $\textcircled{1}_{\text{norm}}, \dots, \textcircled{5}_{\text{norm}}$ into the generic implementation (20). According to Theorem 3, the obtained parallel implementation for the tridiagonal system solver is cost-optimal on $p \in O(n/\log n)$ processors.

7 Experimental Results

In this section, we briefly report experimental performance results for the three parallel versions of the tridiagonal system solver developed in this paper. The

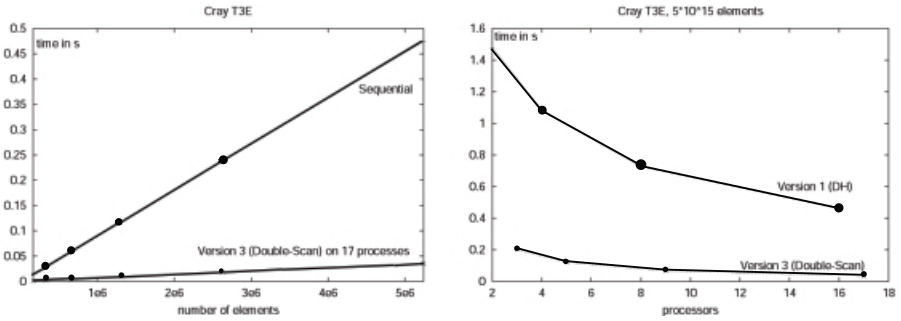


Fig. 5. Runtimes of the tridiagonal system solver. Left: comparison of the sequential with the cost-optimal parallel version (DS). Right: comparison of the cost-optimal (DS) with the non-cost-optimal solution (DH)

measurements were carried out on a Cray T3E machine with 24 processors of type Alpha 21164, 300 MHz, 128 MB, using native MPI implementation.

The two plots in Fig. 5 (left) compare the runtimes of the optimal sequential algorithm with our cost-optimal parallel version depending on the problem size (e.g. $2e6$ stands for $2 \cdot 10^6$). The cost-optimal solution (Version 3) presented in Sect. 6 clearly demonstrates an effective speedup of up to 14 on 17 processors.

In Fig. 5 (right), we compare the runtimes of Version 1 based on the DH skeleton with the cost-optimal Version 3 based on the double-scan skeleton. The achieved time reduction is between 7 and 12 times, depending on the number of processors used. Measurements for these curves were taken for a problem size of approximately $5 \cdot 10^5$.

8 Related Work and Conclusions

The main contribution of this paper is the systematic, step-by-step design of an efficient parallel implementation for a tridiagonal system solver. The most important feature of our design is that it is based on well-defined parallel components (skeletons), which are reusable for different applications. The design process began with an intuitively correct sequential version of the algorithm and proceeded by applying semantically sound transformations. The obtained parallel solutions are therefore provably correct. Furthermore, we proved our final solution to be cost-optimal, i.e. providing not only good runtime but also economical use of processors. The high quality of the obtained solution is confirmed by experiments on a Cray T3E parallel machine.

The paper contributes to the active research area of parallel skeletons – reusable components with prepackaged parallel implementations. In particular, we proposed a new, cost-optimal implementation of the double-scan skeleton based on the new data structure of pointed lists (plists). This implementation can be directly exploited in practical skeleton-based programming systems, including P3L [10] and Skil [11] in the imperative setting, as well as Eden [12] and HDC [13] in the functional world.

Both the complexity of the adjustment process and the target performance depend on the choice of skeletons. Whereas in the version 1 of the tridiagonal solver the user has to adjust the problem to a special divide-and-conquer schema, in version 2 it suffices to find operations ①...④ and prove the associativity of operations ① and ③. However, both versions 1 and 2 are non-cost-optimal. In the cost-optimal version 3, the user additionally has to find operation ⑤ and prove the associativity of ④ modulo ②.

The plist data structure introduced in this paper is new, to the best of our knowledge. Our contribution is in defining and treating explicitly a data structure that has traditionally remained hidden in the design of algorithms.

The parallelization of tridiagonal system solvers is known to be a non-trivial task owing to the sparse structure and restricted amount of potential concurrency. Much research has been done here, López, Zapata [2] and the book by Leighton [3] providing good overviews of the problem and the algorithms used in practice. Classical parallel algorithms for this purpose are Stone's recursive doubling method [14], Hockney and Jesshope's cyclic reduction method [15], originally proposed as a sequential algorithm in [16], and the algorithm proposed by Wang and Mou [17], often called the successive doubling method. It is interesting to observe that our generic algorithm (20) with operations $\textcircled{1}_{norm}, \dots, \textcircled{5}_{norm}$ is very similar to the implementation by Wang and Mou [17], based on Wang's algorithm [18], which is today probably the solution most widely used in practice.

Acknowledgments. We are grateful to the anonymous referees and to Phil Bacon who helped us to greatly improve the presentation.

References

1. Cole, M.I.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. PhD thesis, University of Edinburgh (1988)
2. López, J., Zapata, E.L.: Unified architecture for divide and conquer based tridiagonal system solvers. *IEEE Transactions on Computers* **43** (1994) 1413–1424
3. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publ. (1992)
4. Quinn, M.J.: Parallel Computing. McGraw-Hill, Inc. (1994)
5. 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
6. Gorlatch, S., Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters* **8** (1998) 447–458
7. Gorlatch, S.: Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming* **33** (1998) 1–27
8. 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
9. Bischof, H., Gorlatch, S., Kitzelmann, E.: The double-scan skeleton and its parallelization. Technical Report 2002/06, Technische Universität Berlin (2002)

10. Pelagatti, S.: Structured development of parallel programs. Taylor&Francis (1998)
11. 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
12. Breiting, S., Loogen, R., Ortega-Mallén, Y., Peña, R.: The Eden coordination model for distributed memory systems. In: High-Level Parallel Programming Models and Supportive Environments (HIPS), IEEE Press (1997)
13. Herrmann, C.A., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters* **10** (2000) 239–250
14. Stone, H.S.: An efficient parallel algorithm for the solution of a tridiagonal system of equations. *ACM* **20** (1973) 27–38
15. Hockney, R.W., Jesshope, C.R.: *Parallel Computers*. Adam Hilger, Philadelphia, PA (1988)
16. Hockney, R.W.: A fast direct solution of poisson's equation using fourier analysis. *JACM* **12** (1965) 95–113
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
18. Wang, H.H.: A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software* **7** (1982) 170–183

An Extended ANSI C for Multimedia Processing

Patricio Bulić, Veselko Guštin, and Ljubo Pipan

Faculty of Computer and Information Science, University of Ljubljana,
Tržaška cesta 25, 1000 Ljubljana, Slovenia
{Patricio.Bulic, Veselko.Gustin, Ljubo.Pipan}@fri.uni-lj.si
<http://lra-1.fri.uni-lj.si/index.html>

Abstract. This paper presents the Multimedia C language, which is appropriate for the multimedia extensions included in all modern micro-processors. The paper discusses the language syntax, the implementation of its compiler and its use in developing multimedia applications. The goal was to provide programmers with the most natural way of using multimedia processing facilities in the C language.

1 Introduction

Today's computer architectures are very different from those of a few years ago in terms of complexity and the computational availabilities of the execution units within a processor. Practically all modern processors have facilities that improve performance without placing an additional burden on the software developers—including super-scalar execution, out-of-order execution, speculative execution—as well as those facilities which require support from external entities (i.e. assembler language and compilers) such as multimedia (also called short vector or SIMD within a register) processing ability [18], [20], [23], [33] (i.e. Intel MMX, Intel SSE, Intel SSE2, Motorola AltiVec, SUN VIS, ...), support for multiprocessor architectures, etc. This was reflected in an extension of the assembly languages (extended instruction set).

But if we want to use them in high-level programming languages such as C, then we have to find some way to add these new facilities in some way to the high-level programming languages. In this paper we focus on porting multimedia extension processing facilities into high-level languages, particularly C. So far we have noticed a number of different approaches that are designed to integrate these new facilities into high-level programming languages:

1. The use of assembly languages within high-level programming languages whenever we want to exploit vector processing.
2. The addition of some special libraries [32], [33] that have a wide range of multimedia processing functions coded in assembly language.
3. The use of vectorizing compilers [1], [2], [12], [17], [19]. This is a special class of compilers that can parallelize some simple loops within a high-level programming language code. These compilers are, in general, unable to use some special facilities of vector processing mainly because we cannot describe

these facilities in high-level programming languages (there are no constructs to describe these special facilities). This is the case with, for example, saturation arithmetic. In ordinary C only expressions with modular arithmetic may be used.

4. The last approach is to extend the syntax and semantics of high-level programming languages and to redefine the semantics of existing operators and expressions. We find that this is the best way to migrate new vector processing facilities into high-level programming languages and we agree with those authors [8], [9], [14] [21], [22], [26], [31] who tried to develop such a class of high-level programming languages, although for a different execution model (mainly the large-scale SIMD execution model and general vector processors).

As a consequence of the above we decided to extend the syntax of C and to redefine the existing semantics in such a way that we could use multimedia processing facilities in C. The goal was to provide programmers with the most natural way of using the multimedia processing facilities in the C language. We named this extended C as MMC (MultiMedia C).

This paper is organized as follows: in Section 3 we describe the MMC programming language, in Section 2 we make comparisons with related studies, in Section 4 we describe the implementation of the MMC compiler. Finally, in Section 5 we give real examples from multimedia applications and the performance results.

2 Comparisons with Other Studies

The **C[]** programming language [9], [14], [31] is a Fortran90-like C extension. While preserving all ANSI C syntax and semantics, new powerful facilities for array processing are introduced. In particular, systems with multilevel memory hierarchy and instruction level parallelism are supported. Also, support of array-based computations is provided. The language permits to manipulate arrays as single objects. The key **C[]** provides access to an array as a whole as well as access to both regular and irregular segments of an array, variable-size (dynamic) arrays and variety of elementwise and reduction operators. It introduces a large number of new vector operators are not supported by the existing multimedia hardware extensions. But we found the syntax notation introduced in the **C[]** language the most suitable for MMC expressions of multimedia operations over packed data within a register (for example, we used the **[]** operator to describe most multimedia operations, rather than the **@** operator used in Vector C).

The **Vector C language** [22] was designed and implemented on the CDC Cyber 205 at Purdue University. Vector C extends C by allowing arrays, in effect, to be treated as first-class objects (vectors) by using a special subscripting syntax to select array slices. Vector C targets general vector machines with many vector processing facilities that multimedia-enhanced processors do not have. On the other hand, the operators in Vector C do not cover all processing facilities that are present in multimedia-enhanced processors. The syntax of Vector C

allows periodic scatter/gather operations and compress/expand operations. Two new data types, the vector descriptor (which acts as a pointer to the array but is extended in such a way that it can handle non-stride-1 vectors) and the bit vector, as well as vector function call and multidimensional parallelism are also introduced. The standard C operators act element-wise on vectors and some twenty new, expression operators have been added. It introduces a large number of new, vector operators that have no analogue in ordinary C and are not supported by the existing multimedia hardware extensions. Moreover, Vector C relies on a view of arrays as first-class objects, whereas the confusion of arrays with pointers is essential to the character of the C language. Vector conditional expressions in the Vector C language are handled with the bit vector. Multimedia hardware does not support this kind of operation which depends on the bit vector, thus in the MMC language we had to redefine the act of conditional assignment. Our method generates two vector strips that act as masks. The method was described in Section 4 and in [23].

The C* [26] language is a commercial data-parallel language from Thinking Machine Corporation, which was compiled onto their SIMD CM-2 machine. The main difference between our work and C* is that C* targets large-scale SIMD machines while MMC targets the multimedia extension. C* targets large-scale data-parallel model, which assumes a system with a front-end processor (FE) that controls the overall system and many "processing elements" (PE's). C* extends C by having many processors instead of just one, all executing the same instruction stream. The C* execution model may be summarized as providing the programmer with lots of processors of an conventional nature, operating with a uniform address space in a synchronous execution mode. C* also adds to C additional overloaded meanings of existing operators and new library functions. These overloaded operators provide patterns of communication (i.e. fetching one value from a particular PEs memory, storing one value to the particular PEs memory, broadcasting a value to all PEs, communication among PEs, ...). It also extends the declarations in such a way that we can declare to which memory some variable should be stored. The authors of C* have added two new parallel operators (min, max). Both could easily be expressed in MMC through semantically extended C operators. C* also differs from MMC in adding a new type of statement to C, the selection statement, which is used to activate multiple processors.

And finally, MMC tries to incorporate as much as possible of multimedia processing facilities and in addition to provide as few as possible new operators and type extensions to ANSI C.

3 The MMC Language

MMC language is an extended ANSI C language with multimedia processing facilities. It keeps all the ANSI C syntax plus the syntax rules for vector processing. It extends the ANSI C syntax only in the access possibilities for the array elements and in the new vector operators. The syntax notation and its syntax

is mostly based on the syntax that was first introduced in the C₁₁ programming language [9], [14], [31]. We agree with authors of the C₁₁ language that the C₁₁ syntax offers natural form to express array-based computations which also allows compiler to fully utilize the performance potential of a target platform. But the MMC language aims to support multimedia operations within a processor. Detailed description of the MMC language syntax can be given in [4].

3.1 Arrays

Let us present some basic definitions for an array (vector) and a vector strip in MMC.

Definition 1. *In the MMC language an array (or vector) is a data structure that consists of sequentially allocated elements of the same type with a strictly positive unit step.*

Modern processors with multimedia execution hardware have only vector load/store instructions, which can only move sequentially allocated elements between the memory and the microprocessor. Gather/scatter operations are useful, for example, when multiplying matrices because of the different type of access to the elements in two matrices (in one we access to the column elements and in the other matrix we access to the row elements). But also, these operations are very expensive and we believe that, regarding to the existing multimedia execution hardware, it is better to force the programmer to correctly rearrange the array elements (actually, matrix multiplication can be implemented in a way that doesn't need gather-scatter operations). So, the extension of the array definition to the non-sequentially allocated elements (also called non stride-1 vectors) is redundant for this type of execution model.

3.2 Vector Strips

Because of hardware limitations, especially the multimedia execution hardware and the multimedia register set within a microprocessor, not all the lengths of the array components are permitted. So we will define some notations, which we will use throughout this paper and which represent different vector strips.

Definition 2. *A vector strip is a subset of an array where all of the components have the same type. These components can be as long as 8 bits (or a byte), 16 bits (or a word), 32 bits (or a doubleword), 64 bits (or a quadword) and 128 bits (or a superword). The size of the vector strip is also constant, it is limited to the length of the multimedia register in a microprocessor, and for most modern microprocessors this length is 64 or 128 bits.*

Definition 3. *We can define the following possible vector strips:*

1. A VB vector strip is an array slice composed of 8(16) byte components.
2. A VW vector strip is an array slice composed of 4(8) word components.
3. A VD vector strip is an array slice composed of 2(4) doubleword components.
4. A VQ vector strip is an array slice composed of 1(2) quadword component(s).

5. A *VS vector strip* is an array slice composed of 1 superword component.
6. A *VSF vector strip* is an array slice composed of 4 single-precision floating-point components.
7. A *VDF vector strip* is an array slice composed of 2 double-precision floating-point components.

3.3 Access to the Array Elements

To access the elements of an array we can use one of the following expressions:

1. `expression[expr1]` - with this expression we can access the `expr1`-th element of an array object `expression`. Here, the `expr1` is an integral expression and `expression` has a type "array of type".
2. `expression[expr1:expr2, expr3:expr4]` - with this expression we can access the bits `expr4` through `expr3` of the elements `expr2` to `expr1` of an array object `expression`. Here, the `expr1`, `expr2`, `expr3`, `expr4` are integral expressions and `expression` has a type "array of type". The `expr1` denotes the last accessed element, `expr2` denotes the first accessed element, `expr3` denotes the last accessed bit and `expr4` denotes the first accessed bit. If a programmer specifies something unusual like access to the `array[7:3, 11:4]`, where `array` is of the byte type, the MMC compiler should divide this operation into several memory accesses (actually, the current laboratory version of the MMC compiler will only report an error). We have enabled such irregular access as we believe that the language should be designed for longevity and 'look to the future'. If these multimedia operations are to remain important in the future, some sort of bit scatter/gather hardware will become available on many platforms.
3. `expression[,expr1:expr2]` - with this expression we can access the bits `expr1` through `expr2` of all the elements of an array object `expression`. Here, the `expr1` and `expr2` are integral expressions and `expression` has a type "array of type". The `expr1` denotes the last accessed bit and `expr2` denotes the first accessed bit.
4. `expression[]` - with this expression we can access the whole array object `expression`. Here, the `expression` has a type "array of type". The operator `[]` was first introduced in the C[] language as described in [9]. It is called the *block operator* because blocks (forbids) the conversion of the operand to a pointer. We found it suitable to denote the whole array object and thus avoid any possible confusion of arrays with pointers.

3.4 Operators

Unary Operators. We extended the semantics of the existing ANSI C unary operators `&`, `*`, `+`, `-`, `~`, `!` in the sense that they may now have both scalar- and vector-type operands.

We have also, in a similar way to [9], [14] and [31], added new reduction unary operators `[+]`, `[-]`, `[*]`, `[&]`, `[|]`, `[^]`. These operators are overloaded existing binary operators `+`, `-`, `*`, `&`, `|`, `^` and are only applicable to

the vector operands. These operators perform the given binary operation between the components of the given vector. The result is always a scalar value. Again, we believe that [op] notation, which was introduced in [9], in a more "natural" way indicates that the operation is to be performed over all vector components.

We have also added one new vector operator `|/`, which calculates the square root of each component in the vector (please note, that this works only with floating-point vectors, although the MMC compiler does not perform any type checking, and if we apply this operator to integer vectors the result may be undetermined).

Binary Operators. We have extended the semantics of the existing ANSI C binary operators and the assign operators in such a way that they can now have vector operands. Thus, one or both operands can have an array type. If both operands are arrays of the same length then the result is an array of the same length (note that the length is measured in the number of components and not in the number of bits!). If one array operand has N elements and another array operand has M elements and $N < M$ then the operation is only performed over N elements. If arrays have different types then the MMC compiler reports an error. If one of the operands is of the scalar type then it is internally converted by the MMC compiler into a vector strip of the corresponding type and length. This type of element in the vector strip and its length strongly depends on the processor for which we compile our program. For example, if the array operand consists of word components then for the Intel Pentium processor the scalar operand is converted into the VW vector strip (vector of four 16-bit values).

We have overloaded the existing binary operators with 3 new operators:

- ? this operator overloads the binary operators in such a way that the given binary operator performs the operation with saturation,
- @ this operator overloads the binary add operator in such a way that the given binary operator first performs addition over adjacent vector elements and then averages (shift right one bit) the result,
- ~ this operator overloads the multiply operator in such a way that the result is the high part of the product,
- this operator overloads the multiply operator in such a way that the result is the low part of the product.

Besides the existing binary operators we have added one new, binary operator, which we found to be important in multimedia applications. This operator is applicable only on vector operands (if any operand has a scalar type then it is expanded into an appropriate vector strip) and is as follows:

- | - | absolute difference
(in the grammar denoted as `VEC_SUB_ABS`).

Example 1. The Intel SIMD instruction PSADBW computes the absolute differences of the packed unsigned byte vector strips (VB). Differences are then summed to produce an unsigned word integer result. This can also be written in the MMC language as:

```
unsigned char A[8], B[8]; /* components are 8 bits long */
unsigned short c;
...
c = [+] (A[] |-| B[]) ;
```

Conditional Expression. The conditional operator '?:' which is used in the conditional expression can now have array-type operands. If the first operand is a scalar or an array and the second and third are arrays then the result operand has the same array type as both operands. If the array operands have different array lengths or different types of components then the behavior of the conditional expression is undetermined. If the second or third operand is scalar then it is converted into a vector (the same conversion as for binary operators). If all the operands are arrays of the same length the operation is performed component-wise.

Example 2. The Intel SIMD instruction PMAXUB returns the greater vector components between two byte vectors (VB) . This can also be written in the MMC language as:

```
int A[100, 8], B[100, 8]; /* components are 8 bits long */
int C[100, 8];
...
C[] = (A[] > B[]) ? A[] : B[] ;
```

Tables 1 and 2 summarize the multimedia instruction set supported by the Intel, Motorola and SUN processor families and the associated MMC expression statements.

4 Implementation of the MMC Compiler

The laboratory version of the MMC compiler is implemented for Intel Pentium III and Intel Pentium IV processors. It is implemented as a translator to ordinary C code that is then compiled by an ordinary C compiler (in our example with Intel C++ Compiler for Linux [32]).

The MMC compiler parses input MMC code, performs syntax and semantics analysis, builds its internal representation, and finally translates the internal representation into ANSI C with macros written in a particular assembly language instead of the MMC vector statements.

If we want to compile for another class of microprocessor, we have to use another macro library that is written for that particular class of microprocessor. In such a way we can easily port the programs to another machine. The macro

Table 1. Relations between integer multimedia instructions and MMC expressions.

MMC expression	Intel MMX	SUN UltraSpark VIS	Motorola AltiVec PowerPC
$R[] = A[] + B[]$;	PADDB W D	vis fadd[16, 32]	vec add[8, 16, 32]
$R[] = A[] ?+ B[]$; signed	PADDSB W		vec adds[8, 16, 32]
$R[] = A[] ?+ B[]$; unsigned	PADDUSB W		
$R[] = A[] - B[]$;	PSUBB W D	vis fsub[16, 32]	vec sub[8, 16, 32]
$R[] = A[] ?- B[]$; signed	PSUBSB W D		
$R[] = A[] ?- B[]$; unsigned	PSUBUSB W D		
$R[] = A[] * B[]$; signed	PMULHW		
$R[] = A[] * B[]$; unsigned	PMULHUW		
$R[] = A[] * B[]$; signed	PMULLW		
$R[] = A[] * B[]$; unsigned	PMULLUW		
$A[] = A[] << \text{count}$;	PSLLW PSLLD		vec sl[8, 16, 32]
$A[] = A[] >> \text{count}$;	PSRAW D		vec sra[8, 16, 32]
$A[] = A[] >> \text{count}$;	PSRLW PSRLD		vec srl[8, 16, 32]
$R[] = A[] \text{ op } B[]$;	POR AND XOR ANDN		vec ori andi xor[64]
$R[] = (A[] = B[])? 0xFF : 0$;	PCMPEQB W D	vis fcmpeg[16]	vec cmpeg[8, 16, 32]
$R[] = (A[] > B[])? 0xFF : 0$;	PCMPGTB W D	vis fcmpgt[16]	vec cmpgt[8, 16, 32]
$R[] = A[] > B[] ? A[] : B[]$;	PMAXUB W		vec max[8, 16, 32]
$R[] = A[] < B[] ? A[] : B[]$;	PMINUB W		vec min[8, 16, 32]
$R[] = A[] @+ B[]$;	PAVGB W		vec avg[8, 16, 32]
$R[] = A[] - 0$;			vec abs[8, 16, 32]
$R = [+1 (A[] - B[])$; unsigned	PSADBWB		

op = { |, &, ^, ~&}

Table 2. Relations between floating-point multimedia instructions and MMC expressions.

MMC expression	Intel SSE	SUN VIS	Motorola AltiVec
$R[] = A[] + B[]$;	ADDPS		vec add[4x32]
$R[] = A[] - B[]$;	SUBPS		vec sub[4 x 32]
$R[] = A[] * B[]$;	MULPS		
$R[] = A[] / B[]$;	DIVPS		
$R[] = 1 / A[]$;	RCPPS		vec rc[4 x 32]
$R[] = A[]$;	SQRTPS		
$R[] = 1 / (A[])$;	RSQRT		veci rsorte [4 x 32]
$R[] = A[] \log_{\text{op}} B[]$;	POR, PAND, PXOR, PANDN		vec or and xor andn [4 x 32]
$R[] = (A[] \text{ rel } \text{op } B[])? 0xFF : 0$;	CMPPS, rel op1		vec cmp [rel op2] [4 x 32]
$R[] = A[] > B[] ? A[] : B[]$;	MAXPS		vec max[8, 16, 32]
$R[] = A[] < B[] ? A[] : B[]$;	MINPS		vec min[8, 16, 32]

log_op = { |, &, ^, ~&}

rel_op1 = { =, <, <=, !=, !<, !<=, !? }

rel_op2 = { =, <, <=, !=, >, >= }

libraries for different processors are easily written and all lower-level optimization of the code is done by an ordinary C compiler for a particular microprocessor. In the event that a particular microprocessor does not support some parallel operation written in MMC with special multimedia machine instruction(s) we use a function written in C that executes sequentially instead of multimedia macro.

Example 3. The conditional MMC statement:

$R[] = (A[] > \text{MASK})? A[] : B[]$;

is evaluated during the compilation process into macro IFGTB(MASK, A, B, R); which is defined as:

```
#define IFMGTB(MASK, A, B, R); __asm{ mov eax, B \
    movq mm3, [eax] \
    mov ebx, A \
    movq mm2, [ebx] \
    mov ecx, MASK \
    movq mm1, [ecx] \
    pcmptgb mm1, mm2 \
    movq mm4, mm1 \
    pand mm1, mm3 \
    pandn mm4, mm2 \
    por mm1, mm4 \
    mov edx, R \
    movq [edx], mm1 };
```

5 The Use of the MMC Language to Develop Multimedia Applications

In this section we present the use of MMC language to code some commonly used multimedia kernels. At the end of this section the performance results for some multimedia kernels are presented. Examples 4 and 5 show how the MMC code is translated by the MMC compiler into C code.

Example 4. Finite impulse response (FIR) filters are used in many aspects of present-day technology because filtering is one of the basic tools of information acquisition and manipulation. FIR filters can be expressed by the equation:

$$y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k) \quad (1)$$

where N represents the number of filter coefficients $h(k)$ (or the number of delay elements in the filter cascade), $x(k)$ is the input sample and $y(k)$ is the output sample.

The MMC implementation of the FIR filter is as follows:

```
int j;
double h[FILTER_LENGTH]; // FIR filter coefficients
double delay_line[FILTER_LENGTH]; // delay line
double x[SIGNAL_LENGTH]; // input signal
double y[SIGNAL_LENGTH]; // output signal

for (j=0; j<SIGNAL_LENGTH; j++) {
    delay_line[0] = x[j]; //store input in the delay line

    //calculate FIR:
    y[j] = [+] ( h[] * delay_line[] );
```

```
//shift delay line:
delay_line[FILTER_LENGTH:1] = delay_line[FILTER_LENGTH-1:0];
}
```

This MMC code is translated by the MMC compiler into C code with inserted macros. So, after strip-mining and macro insertion, which is done by the MMC compiler, we have:

```
int j;
float h[FILTER_LENGTH]; // FIR filter coefficients
float delay_line[FILTER_LENGTH]; // delay line
float x[SIGNAL_LENGTH]; // input signal
float y[SIGNAL_LENGTH]; // output signal

// create new symbols:
int i00001, i00002, i00003;

for (j=0; j<SIGNAL_LENGTH; j++) {
    delay_line[0] = x[j]; //store input in the delay line

    //calculate FIR:
    // strip mining and macro insertion:
    for( i00001 = 0; i00001 < FILTER_LENGTH/4; i00001+=4 ) {
        H = h + i00001;      // prepare addresses for macro
        Z = delay_line + i00001;
        OUT = y + j;

        SUMMULTWD(OUT, H, Z); // macro insertion
    }

    for( i00002 = FILTER_LENGTH/4;
        i00002 < FILTER_LENGTH; i00002++ ) {
        y[j] = y[j] + (h[i00002] * delay_line[i00002]) ;
    }

    //shift delay line:
    for( i00003 = FILTER_LENGTH-2; i00003 >= 0; i00003-- ) {
        delay_line[i00003+1] = delay_line[i00003] ;
    }
}
```

Example 5. An Infinite Impulse Response (IIR) filter produces an output, $y(n)$, that is the weighted sum of the current and the past inputs, $x(n)$, and past outputs. IIR filters can be expressed by the equation:

$$y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k) + \sum_{p=1}^{M-1} h'(p) \cdot y(n-p) \quad (2)$$

where N represents the number of forward-filter coefficients $h(k)$ (or the number of delay elements in the forward-filter cascade) and M represents number of backward-filter coefficients $h'(k)$ (or the number of delay elements in the backward-filter cascade), $x(k)$ is the input sample and $y(k)$ is the output sample.

The MMC implementation of the IIR filter is as follows (note that for simplicity in implementation we use the $h'(0)$ coefficient, which is always zero):

```
int j;
float hf[FILTER_LENGTH_F]; // forward IIR filter coefficients
float hb[FILTER_LENGTH_B]; // backward IIR filter coefficients
float in_delay[FILTER_LENGTH]; // input delay line
float out_delay[FILTER_LENGTH]; // output delay line
float x[SIGNAL_LENGTH]; // input signal
float y[SIGNAL_LENGTH]; // output signal

for (j=0; j<SIGNAL_LENGTH; j++) {
    in_delay[0] = x[j]; //store input in the delay line

    //calculate FIR:
    y[j] = [+] ( hf[] * in_delay[] );

    out_delay[0] = y[j]; //store output into the delay line

    //calculate IIR:
    y[j] = y[j] + ( [+] ( hb[] * out_delay[] ) )

    // shift delay lines
    in_delay[FILTER_LENGTH_F-1:1]=in_delay[FILTER_LENGTH_F-1:0];
    out_delay[FILTER_LENGTH_B-1:1]=out_delay[FILTER_LENGTH_B-1:0];
    out_delay[0] = y[j];
}
```

This MMC code is translated into C as follows:

```
int j;
float hf[FILTER_LENGTH_F]; // forward IIR filter coefficients
float hb[FILTER_LENGTH_B]; // backward IIR filter coefficients
float in_delay[FILTER_LENGTH]; // input delay line
float out_delay[FILTER_LENGTH]; // output delay line
float x[SIGNAL_LENGTH]; // input signal
float y[SIGNAL_LENGTH]; // output signal

// create new symbols:
int i00001, i00002, i00003, i00004, i00005;
float temp00001;
```

```

for (j=0; j<SIGNAL_LENGTH; j++) {
    in_delay[0] = x[j]; //store input in the delay line

    //calculate FIR:
    // strip mining and macro insertion:
    for( i00001 = 0; i00001 < FILTER_LENGTH_F/4; i00001+=4 ) {
        HF = hf + i00001;      // prepare addresses for macro
        IND = in_delay + i00001;
        OUT = y + j;

        SUMMULTWD(OUT, HF, IND); // macro insertion
    }

    for( i00002 = FILTER_LENGTH_F/4;
        i00002 < FILTER_LENGTH_B; i00002++ ) {
        output[j] = output[j] + (hf[i00002] * in_delay[i00002]) ;
    }

    //calculate IIR:
    // strip mining and macro insertion:
    for( i00003 = 0; i00003 < FILTER_LENGTH_B/4; i00003+=4 ) {
        HB = hb + i00001;      // prepare addresses for macro
        OUTD = out_delay + i00001;

        SUMMULTWD(temp00001, HB, OUTD); // macro insertion
    }

    y[j] = y[j] + temp00001;

    for( i00004 = FILTER_LENGTH_B/4;
        i00004 < FILTER_LENGTH; i00004++ ) {
        y[j] = y[j] + (hb[i00004] * out_delay[i00004]) ;
    }

    //shift delay lines:
    for( i00005 = FILTER_LENGTH_F-2; i00005 >= 0; i00005-- ) {
        in_delay[i00005+1] = in_delay[i00005] ;
    }
    for( i00005 = FILTER_LENGTH_B-2; i00005 >= 0; i00005-- ) {
        out_delay[i00005+1] = out_delay[i00005] ;
    }
}

```

In Figure 1 we can see the performance improvement when using the MMC instead of the ANSI C for the FIR, IIR, DCT and RGB to YUV kernels. Both

codes, MMC and ANSI C, were finally compiled with an Intel C++ Compiler, and executed on an Intel Pentium III personal computer.

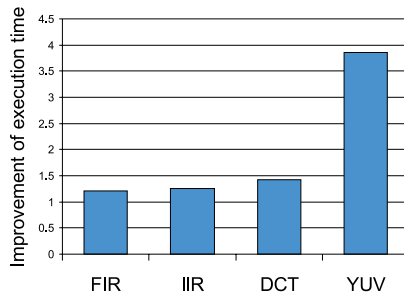


Fig. 1. Speedup on an Intel Pentium III using MMC.

6 Conclusion and Future Work

We have developed a MMC programming language which is able to use hardware-level multimedia execution capabilities. The MMC language is an upward extension of ANSI C and it saves all the ANSI C syntax. In this way it is suitable for use by programmers who want to extract SIMD parallelism in a high-level programming language and also by programmers who do not know anything about multimedia processing facilities and who are using the C language.

We have shown the ease with which it is possible to express some common multimedia kernels with MMC. With MMC we can express these kernels in a more straightforward or 'natural' way. The presented extension to C also preserves the interchangeability of arrays and pointers and adds as few as possible new operators. All added operators have an analogue in ordinary C. The declarations of arrays are left unchanged and also no new types have been added.

We obtained good performance for several application domains. Experiments on scientific and multimedia applications have significant good performance improvements.

References

1. Bacon D.F., Graham S.L., Sharp O.J. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420, 1994.
2. Bik A.J.C., Girkar M., Grey P.M. Tian X.M. Automatic Intra-Register Vectorization for the Intel (R) Architecture. *International Journal of Parallel Programming*. Vol 30., No. 2, pp. 65–98. 2002.

3. Bulić P., Guštin V. Introducing the Vector C. *VECPAR2002. 5th International Conference on High Performance Computing for Computational Science: Selected Paper and Invited Talks. Lecture Notes in Computer Science*. LNCS 2565 pp. 608–622. 2003.
4. Bulić P., Guštin V. An Extended ANSI C for Processors with a Multimedia Extension. *International Journal of Parallel Programming*. Vol. 31, No. 2, pp. 107–136. April 2003.
5. Calland P.Y., Darté A., Robert Y., Vivien F. On the Removal of Anti- and Output-Dependences. *International Journal of Parallel Programming*. Vol. 26, No. 3, pp. 285–312. 1998.
6. Corbera F., Asenjo R., Zapata E. New Shape Analysis and Interprocedural Techniques for Automatic Parallelization of C Codes. *International Journal of Parallel Programming*. Vol. 30, No. 1, pp. 37–63. 2002.
7. Dennis J.B. Machines and Models for Parallel Computing. *International Journal of Parallel Programming*. Vol. 22, No. 1, pp. 44–77, 1994.
8. Fisher R. Compiling for SIMD Within a Register. *Lecture Notes in Computer Science*, No. 1656, pp. 290–304, 1999.
9. Gaissaryan S., Lastovetsky A. An ANSI C for Vector and Superscalar Computers and Its Retargetable Compiler. *Journal of C Language Translation*, 5(3), pp. 183–198, 1994.
10. Griebel M., Feautrier P., Lengauer C. Index Set Splitting. *International Journal of Parallel Programming*. Vol. 28, No. 6, pp. 607–631, 2000.
11. Gupta M., Mukhopadhyay S., Sinha N. Automatic Parallelization of Recursive Procedures. *International Journal of Parallel Programming*. Vol. 28, No. 6, pp. 537–562. 2000.
12. Guštin V., Bulić P. Extracting SIMD Parallelism from "for" Loops. in *Proceedings of the 2001 ICPP Workshop on HPSECA, ICPP Conference, Valencia, Spain, 3–7 September, 2001*, pp. 23–28. 2001.
13. John A., Brown J.C. Compilation of Constraint Programs with Noncyclic and Cyclic Dependences to Procedural Parallel Programs. *International Journal of Parallel Programming*. Vol. 26, No. 1, pp. 65–119, 1998.
14. Kalinov A.Ya., Lastovetsky A.L., Ledovskih I.N., Posypkin M.A.. Refined Description of the C_{||} Language. *Programming and Computer Software*, Vol. 28, No. 6, pp. 333–341, 2000.
15. Kennedy K. Compiler Technology for Machine-Independent Parallel Programming. *International Journal of Parallel Programming*. Vol. 22, No. 1, pp. 79–98, 1994.
16. Kessler C.W., Seidl H. The Fork95 Parallel Programming Language: Design, Implementation, Application. *International Journal of Parallel Programming*. Vol. 25, No. 1, pp. 17–50, 1997.
17. Krall A., Lelait S. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, Vol. 28, No. 4, pp. 347–361, 2000.
18. Kuroda I., Nishitani T. Multimedia Processors. *Proceedings of the IEEE*, Vol. 86, No. 6, pp. 1203–1221, 1998.
19. Larsen S, Amarasinghe S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *Processing of the SIGPLAN'00 Conference on programming Language Design Implementation, Vancouver, B.C., June 2000*. <http://www.cog.lcs.mit.edu/slp/SLP-PLDI-2000.pdf>.
20. Lee R. Accelerating Multimedia with Enhanced Processors. *IEEE Micro*, Vol. 15, No. 2, pp. 22–32, 1995.
21. Li K.C., Schwetman H. Vector C – A Vector Processing Language. *Journal of Parallel and Distributed Computing*, No. 2, pp. 132–169, 1985.

22. Li K.C. A note on the vector C language. *ACM SIGPLAN Notices*, Vol. 21, No. 1, pp. 49–57, 1986.
23. Mitall M., Peleg A., Weiser U. MMX Technology Architecture Overview, *Intel Technology Journal*, 1997.
24. Oberman S., Favor G., Weber F. AMD 3DNow! Technology: Architecture and Implementation. *IEEE Micro*, Vol. 19, No. 2, pp. 37–48, 1999.
25. Peleg A., Weiser U. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, Vol. 16, No. 4, pp. 42–50, 1996.
26. Rose J.R., Steele G.L. C* : An extended C Language for Data Parallel Programming. *Proceedings of the Second International Conference on Supercomputing ICS87, May, 1987*, pp. 2–16, 1987.
27. Sarkar V. Optimized Unrolling of Nested Loops. *International Journal of Parallel Programming*. Vol. 29, No. 5, pp. 545–581. 2001.
28. Sreraman N., Govindarajan R. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*. Vol. 28, No. 4, pp. 363–400, 2000.
29. Tsai J.Y., Jiang Z., Yew P.C. Compiler Techniques for the Superthreaded Architectures. *International Journal of Parallel Programming*. Vol. 27, No. 1, pp. 1–19. 1999.
30. Wolfe M.J., Banerjee U. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*. Vol. 16, No. 2, pp. 137–178, April 1987.
31. -. The C₉₉ Language Specification. <http://www.ispras.ru/~cbr/cbrsp.html>.
32. -. Intel C++ Compiler for Linux 6.0.
<http://www.intel.com/software/products/compilers/c60l/>.
33. -. Pentium (R) II Processor Application Notes, MMX (TM) Technology C Intrinsics, <http://developer.intel.com/technology/collateral/pentiumii/907/907.htm>.

The Parallel Debugging Architecture in the Intel[®] Debugger

Chih-Ping Chen

Software Solution Group
Intel Corporation
110 Spitbrook Rd.
Nashua, NH, 03063, U.S.A.
chih-ping.chen@intel.com

Abstract. In addition to being a quality symbolic debugger for serial IA32 and IPF Linux applications written in C, C++, and Fortran, the Intel[®] Debugger is also capable of debugging parallel applications of Pthreads, OpenMP, and MPI. When debugging a MPI application, the Intel[®] Debugger achieves better startup time and user response time than conventional parallel debuggers by (1) setting up a tree-like debugger network, which has a higher degree of parallelism and scalability than a flat network, and (2) employing a message aggregation mechanism to reduce the amount of data flowing in the network. This parallel debugging architecture can be further enhanced to support the debugging of mixed-mode and heterogeneous parallel applications. Moreover, a generalized version of this architecture can be applied in areas other than debugging, such as performance profiling of parallel applications.

1 Introduction

The Intel[®] Debugger¹ [1] is a programming tool that enables developers to debug their parallel applications that run on Intel IA32 and IPF Linux. The parallel paradigms targeted by idb are

Parallelism via inter-process communication A notable example of this paradigm is MPI [2], which is a standard interface that enables multiple processes to work cooperatively by using a message passing mechanism.

Parallelism in a single process Pthreads and OpenMP [3] applications are the representatives of this paradigm.

In this paper, we give an overview of how idb supports debugging applications in these paradigms and discuss potential enhancements. We put more emphasis on the debugging of MPI applications because the approach used by idb is better than a conventional MPI debugger in terms of performance and scalability.

The rest of the paper is organized as follows: In Section 2, we describe the tree topology used by idb to debug multi-process parallel applications, and compare

¹ We will refer to the Intel[®] Debugger as idb from this point on to avoid clutterness.

it with the traditional linear model. In Sections 3 and 4, we briefly discuss how idb supports debugging threaded applications and mixed-mode applications. In Section 5, we identify several areas for further investigation.

2 Debugging Multi-process Parallel Applications

The conventional approach to debugging a multi-process parallel application uses a linear (or flat) client-server model depicted in Figure 1. Debuggers using this model include P2D2 [4], TotalView [5], Prism [6], and Mantis [7].

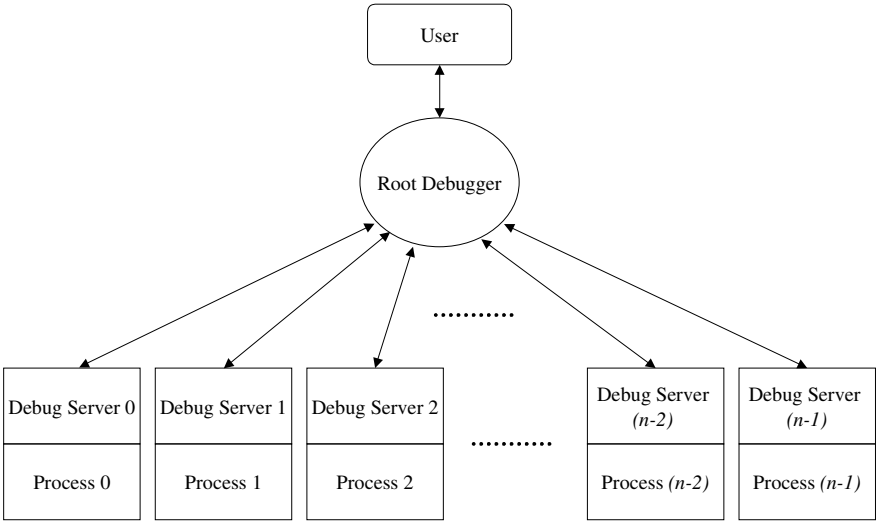


Fig. 1. The linear model. In this model, the debug servers connect to the root debugger directly.

When a debugging session is started, each application process is brought under the control of a debug server, which (1) takes a debugging command from the root debugger (i.e. the client) and applies it to the process it controls, and (2) returns the output, if any, of executing the command back to the root debugger. The root debugger in this model acts as the creator of the servers and the interface between the debugger user and the server.

The major drawback of this approach is that it does not scale well: Both the startup time of a debugging session and the response time of a debugging command have the time complexity $O(n)$, where n is the number of processes in the parallel application. This linear growth in startup time and response time

can be intolerable when (1) there are thousands of application processes, or (2) the output of a debug command is enormous (e.g., a stack trace with thousands of frames). In addition, most operating systems, Linux included, have an upper bound on how many inter-process connections can be opened for a process, which will limit the number of debug servers the root debugger is able to connect to.

2.1 The Architecture

Unlike conventional debuggers, idb sets up a tree topology to debug a parallel application, as shown in Figure 2.

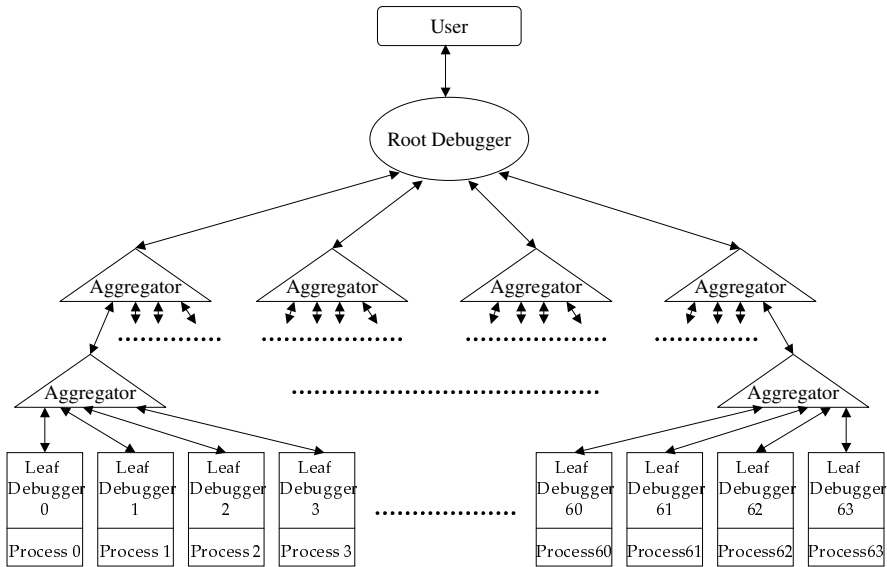


Fig. 2. The tree topology. Idb employs this model to debug a parallel application, connecting the leaf debuggers with the root debugger through levels of aggregators. In this example, idb sets up a 4-level quad-tree for a parallel application with 64 processes.

The tree topology consists of three types of nodes:

1. The root node, called the *root debugger*, acts both as the user interface and as an aggregator.
2. An internal node, called an *aggregator*, propogates the command it receives from its parent to its children, and aggregates the output of its children and sends the aggregation result to its parent.
3. A leaf node, called a *leaf debugger*, takes a command from its parent and executes it, and then sends the result to its parent.

The shape of the tree is determined by two factors: The number of processes in a parallel application n , and the *branching factor* of the tree d , which is the maximal number of children of a node. The simple algorithm currently employed by `ldb` builds a complete tree with $\lceil \log_d(n) \rceil + 1$ levels.

Note that all the nodes that are siblings in a tree can be built in parallel. This means the startup time of a debugging session is in the order of $O(\log(n))$. Furthermore, the number of connections needed for a process is bounded by the branching factor of the tree (with a default value of eight), which is usually much smaller than the system limit on the number of connections. Consequently, the tree topology scales much better than the flat topology in Figure 1.

The tree topology also improves on the flat topology's user response time because it allows a debug command to be broadcast from the root debugger to the leaf debugger in $O(\log(n))$. However, the tree topology alone does not reduce the amount of leaf debugger output received by the root debugger. This is where the aggregators come into play.

An aggregator condenses the output of its children by recognizing the common portion of the children's output, sending only one copy of the common portion, plus descriptions of how each child's output differs from the common portion. This mechanism can often reduce the data traffic from the leaf to the root debuggers substantially, thus further improving the user response time.

This architecture allows the aggregators and leaf debuggers to work in parallel, hence not having a single bottleneck when some action (e.g. a breakpoint being triggered) requires attention. On the other hand, its parallelism is limited by how fast the slowest leaf debugger responds to a user command or by a timeout mechanism. A time-out mechanism employed by the aggregators prevents output stalls through the network should a leaf debugger become slow or unresponsive. `Idb` distributes the aggregators onto the available nodes using a load-balanced scheme that takes into account the number of processes within a given machine as well as how many processes are already running on that machine.

2.2 The User Interface

When debugging a massively parallel application, the user would be overwhelmed if the output of the leaf debuggers were presented verbatim. The aggregation mechanism discussed in the previous section along with a pretty printer in the root debugger can help in this regard by presenting to the user mostly identical leaf debugger messages in a compact format. The root debugger prefixes an aggregated leaf debugger message with (1) the aggregated message ID, and (2) the set of processes that contribute to the aggregated message, as shown in Figure 3. `Idb` currently aggregates only integers (both decimal and hexadecimal) and floating point numbers.

`Idb` provides two commands to inspect the aggregated messages:

1. **show aggregated message** lists the aggregated messages displayed by the root debugger.

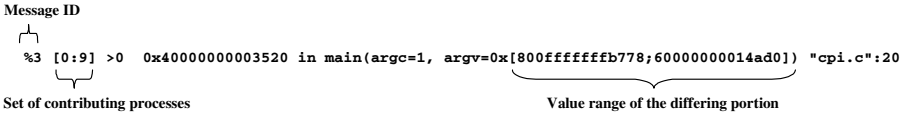


Fig. 3. An aggregated message. This message has the ID number 3, and was received from the leaf debuggers 0 to 9. The value range of the differing portion is enclosed in a pair of square brackets.

- 2. **expand aggregated message** lists the original output from each contributing leaf debugger for the specified aggregated message.

In addition, `idb` supports the **focus** command proposed in [8]. This command changes the active set. As a consequence, subsequent commands will apply to only the processes specified in the new active set. The user can zoom in on his debugging problem by making his active set smaller. `Idb` also extends the process set syntax proposed in [8] with set manipulation operations, including the binary set union (+), the binary set difference (-), and the unary set negation (-).

3 Debugging Threaded Processes

`Idb` supports the debugging of LinuxThreads-based pthreads applications. LinuxThreads [9] is part of the glibc distribution. To debug a threaded application, `idb` uses the debugging interface provided in `thread_db.h` (which is also part of the glibc distribution) to obtain the thread-related information.

Note that the OpenMP support provided by several commercial compilers and non-commercial OpenMP preprocessors are implemented in terms of pthreads. Consequently, `idb` can be used to debug those OpenMP applications as well.

4 Debugging Mixed-Mode Parallel Applications

The two parallel paradigms described in Section 1 are orthogonal to each other and therefore it is logical to use both paradigms to devise a parallel application. A typical mixed-mode parallel application has multiple processes, and one or more of these processes are multi-threaded. The tree topology used by `idb` can handle mixed-mode applications due to the fact that the leaf debuggers in the tree are full-featured debuggers that are thread-aware, as described in the previous section.

To better support mixed-mode debugging, we are considering to generalize the existing process set syntax into a process/thread set syntax, which allows the user to specify certain threads in certain processes conveniently. For example, the following `idb` command sequence


```
focus [30(4:10)]  
where
```

would (1) make the set of Threads 3 in Processes 4 to 10 active, and (2) print the stack trace of those threads only.

Without this syntax, the user would have to enter the sequence of **focus**, **thread**, and **where** commands multiple times, which can be prohibitively inefficient when there are lots of application processes.

5 Future Work

The status of the `idb` reported in this paper is the result of some of our first steps in devising a high-performance, feature-rich parallel debugger. In this section, we identify the directions for future enhancement and investigation.

5.1 Collecting Performance Results

We are currently working on collecting the performance results of `idb`. The preliminary results of using `idb` to debug MPI programs obtained on a 32-node IPF2 cluster with RMS match those reported in [10] for using Ladebug [11] on a Compaq Alpha cluster. Ladebug is the predecessor of `idb` and uses the same tree topology to debug parallel applications. In addition to providing concrete evidence of the superiority of the tree topology, the results may also provide an empirical guide in selecting good default branching factor and time-out delay.

5.2 Supporting Other Parallel Paradigms

We are working on a proposal for a universal debugging interface for implementations of other multi-process parallel paradigms. The basic idea of the proposal will be based on the MPICH debugging interface. If this interface materializes and is used by the implementors, a parallel debugger that supports this interface will be able to set up a debug session for any interface-abiding parallel paradigm implementation in a uniform way.

5.3 Better Message Aggregation/Transformation

We are considering to extend the message aggregation mechanism so that the user can specify (say, using regular expressions) the string patterns to be aggregated. This would give the user complete control of what to aggregate and what not to aggregate. If used properly, this flexibility can further reduce the amount of data percolated from the leaves to the root.

An even more promising direction is to generalize the message aggregation mechanism into a message transformation mechanism that allows the user to specify complex transformation rules like **sed**. This generalization is crucial in broadening the applicability of the tree topology.

5.4 Generalizing the Tree Topology

The tree topology has other applications. For example, we can replace the root debugger in the topology with a profiler user interface and the leaf debuggers with serial profilers to obtain a parallel profiler. We are considering to work on devising an API for the root and the leaf in the topology. Combined with the message transformation mechanism described in Section 5.3, this API will allow tools using it to conveniently exploit the parallelism and message transformation offered by the tree topology.

5.5 Debugging Heterogeneous Applications

The debugger employed at a leaf node in the tree topology does not have to be `ldb`. It can be any debugger as long as it matches the functionality of `ldb`. This flexibility allows the tree topology to be adapted to debug a heterogeneous parallel application, which runs on several different platforms simultaneously.

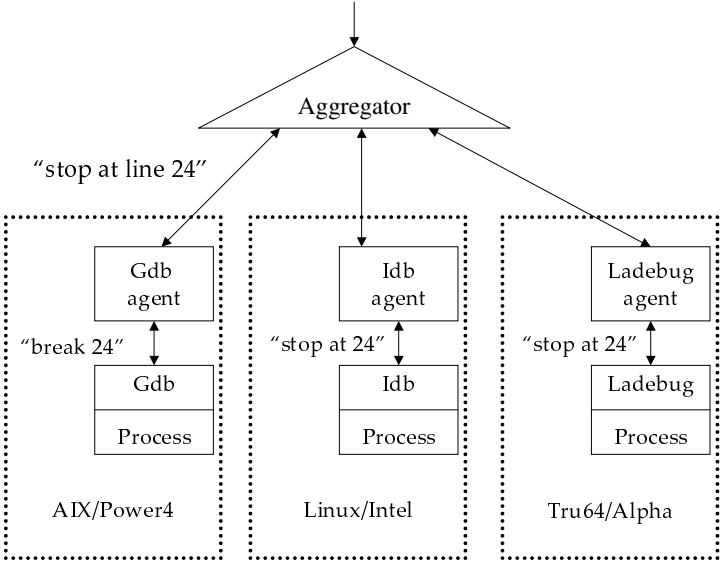


Fig. 4. Heterogeneous debugging. In this example, three processes are spawned on three different platforms. A debugger command “stop at line 24” is translated into the corresponding command for each leaf debugger by a suitable agent.

To enable heterogeneous debugging, what we need is an *agent* that translates `ldb` commands into the corresponding commands for the leaf debugger, and, conversely, the leaf debugger output into the `ldb` output. See Figure 4 for an example.

6 Conclusion

We have described the architecture employed by idb to support the debugging of parallel applications. The two centerpieces of the architecture are the tree topology and the message aggregation mechanism—the former injects more parallelism into the framework while the latter reduces the data traffic and induces a cleaner user interface. They combined give idb better scalability and shorter startup and user response times than conventional parallel debuggers. Equally significant is that this architecture can be generalized into an API so that a developer can use it to rapidly derive a parallel programming tool from an existing serial tool.

References

1. Intel Corporation: Intel Debugger (IDB) Manual (2002)
http://www.intel.com/software/products/compilers/techtopics/iidb.debugger_manual.htm.
2. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 1.1 (1995) <http://www-unix.mcs.anl.gov/mpi/>.
3. OpenMP Architecture Review Board: OpenMP Specifications (2002)
<http://www.openmp.org>.
4. Cheng, D., Hood, R.: A portable debugger for parallel and distributed programs. In: Supercomputing. (1994) 723–732
5. Etnus Inc.: The TotalView Multiprocess Debugger (2000) <http://www.etnus.com>.
6. Thinking Machines Corporation: Prism's User's Guide (1991)
7. Lumetta, S., Culler, D.: The Mantis Parallel Debugger. In: Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools. (1996) 118–126
8. High Performance Debugging Forum: HPD Version 1 Standard: Command Interface for Parallel Debuggers, (Rev. 2.1) (1998) <http://www.ptools.org/hpdf/draft>.
9. Xavier Leroy: The LinuxThreads Library (1998)
<http://pauillac.inria.fr/~xleroy/linuxthreads/>.
10. Balle, S.M., Brett, B.R., Chen, C.P., LaFrance-Linden, D.: A new approach to parallel debugger architecture. In: Proceedings of PARA 2002 (LNCS 2367), Espoo, Finland (2002) 139–149
11. Compaq Corporation: Ladebug Debugger Manual (2001)
http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/V51A_HTML/LADEBUG/TITLE.HTM.

Retargetable and Tuneable Code Generation for High Performance DSP

Anatoliy Doroshenko and Dmitry Ragozin

Institute of Software Systems of the National Academy of Sciences of Ukraine,
Acad. Glushkov prosp., 40, block 5, Kiev 03187, Ukraine
dor@isofts.kiev.ua, dvragozin@hotmail.ru

Abstract. An approach of intelligent retargetable tuneable compiler is introduced to overcome the gap between hardware and software development and to increase performance of embedded systems by enhancing their instruction-level-parallelism. It focuses on high-level model and knowledgeable treatment of code generation where knowledge about target microprocessor architecture and human-level heuristics are integrated into compiler production expert system. XML is used as platform-independent representation of data and knowledge for design process. Structure of an experimental compiler which is developed to support the approach for microprocessors with irregular architecture like DSP and VLIW-DSP is described. A technique to detect optimal processor architecture and instruction-level-parallelism for program execution is presented. Results of code generation experiments are presented for DSPstone benchmarks.

1 Introduction

Rapid evolution of microelectronics yields to significant reducing down microprocessor development life cycle time and allows substantial diversification of hardware product lines of with new architecture features. Most of modern microprocessors are application-specific instruction processors (ASIPs) [1] that usually have tuneable kernel that is expandable with different application-oriented instructions and units. Efficiency of their utilisation strongly depends on effective utilisation of application-specific processor expansions. It can be achieved usually by hand programming in assembler language, because traditional compilers as usual can not handle complex microprocessor instruction set extensions in efficient way especially for digital signal processing and SIMD-in-Register (SIMD-R) extensions.

Digital signal processing (DSP) kernels are the most growing segment of microprocessor market. However DSP kernels have mostly complex architecture, provided instruction level parallelism is very irregular and most of them require programming by hand. Problems of effective utilisation of their architecture supported parallelism can not be solved using only standard compilers and dumb cross-compilers due to their inability to take into account important performance opportunities of new microprocessor architectures and poor software reengineering facilities. The solution should be seen on the way of high-level representation and intelligent manipulation of

knowledge about both software to be designed or reengineered and target architectures. Such approach is assumed in this paper that reflects our results and experience of research and development of retargetable and tuneable compilers for DSP and VLIW processors in our HBPK-2 project (<http://dvragozin.hotbox.ru>).

Retargetable compiler differs from the traditional compiler in its structure as it additionally requires microprocessor description which includes resource and instruction supported architecture (ISA) description according some formal model, e.g. expressed in specially developed mark-up language [2]. Retargetable compilation problem (in aspect of quality code generation for different microprocessors) has arisen 15-20 years ago. The book [1] on retargetable compilation sums up experiences of retargetable compilation and code generation in middle 199x. Up to 1995 two main directions in retargetable compilation were formed; 1) co-development of microprocessor and compiler; 2) development of general-purpose compiler. One of the first retargetable compilation system was MIMOLA developed by group of P. Marvedel [3]. After 1995 up to late 1990s researchers paid main attention to embedded system and on-chip-systems. The first general-purpose compiler RECORD has been built by R. Leupers [4]. Likely systems were built by other researchers, for example, Flexware [5], Chess [1,6].

In common case when microprocessor kernel is of RISC type the retargeting procedure is not very complex, usually it comprised of changing a bunch of instructions and register descriptions while migrating to other RISC platforms. As usual RISC microprocessors have orthogonal register file(s) and ISA, so procedure of defining optimal instruction schedule is straightway combinatorial. In other case if microprocessor has complex structure, for example RISC kernel+DSP coprocessor, traditional compilation methods utilise generally only RISC kernel, but not DSP extension while compiling DSP-oriented algorithms [7]. The most unfavourable thing is that ASIPs have irregular architectures, and the compiler have not (or can not extract) knowledge about utilising application-oriented microprocessor unit extensions from its description. Combinatorial code generation algorithms can not be applied directly to irregular architecture descriptions, so utilisation of standard compiling methods in retargetable or processor-specific compiler becomes very inefficient.

Now all efforts in retargetable compiling are concentrating on improving code generation methods for wide processor families – generated code speeding up, code compaction, energy saving, because as usual retargetable compiler accompanies embedded microprocessors. There are special interests in code generation: 1) for exploiting “SIMD-in-a register” commands in embedded processors [8]; 2) low power code generation [9] – some retargetable compilers are oriented to energy-aware code generation (with minimum energy consumption during program execution); 3) exploiting peculiarities in embedded processors [10].

In this paper an approach of retargetable tuneable compiler as design and reengineering tool is proposed to overcome the gap between hardware and software development and to enhance performance of modern DSP supporting instruction level parallelism. Knowledge-oriented techniques are presented which can improve code generation quality applied to irregular microprocessor architectures using some human-like heuristics. In section 2 a simple motivating example of how to increase instruction parallelism at DSP code generation is presented. In section 3 a model and structure of our retargetable compiler are considered. In section 4 code analysis tech-

niques combining iterative code generation and code analysis are described. In section 5 a technique of decision on optimal processor architecture is described. In section 6 knowledge base integration into the compiler is considered and in section 7 numerical results of code generation improvement are presented. Concluding remarks are presented in section 8.

2 Simple Motivating Example

To illustrate possibilities for enhancing parallelism in DSP and problems arising let us consider simple example. Code analysis is important for increasing instruction level parallelism and is based on analysis of generated code parameters, instruction schedule events and iterative code generation [11]. The “generated code parameter” is often obscure concept, and in any particular case parameters may be different. But common their property is that these parameters highly depend on instruction scheduling. For example, traffic analysis between registers and memory can be ugly provided while lexical parsing as each variable reference can be taken as memory reference. But till register allocation and instruction scheduling phases some variables can be kept in registers, some references will be omitted, so precise value can be obtained only via generated code analysis. After retrieving the necessary parameters to enhance performance some attributes of internal program representation should be changed (for example, some variable after analysis must be placed into accumulator register), so even iterative code generation is needed. The code analysis process is dependent on microprocessor architecture. As an example of optimisation for digital signal processors consider distributing variables over Harvard memory architecture for digital signal filtering (convolution):

```
s=0;
for(i=0; i++; i<N) { s+= a[i]*b[i]; }
```

Before instruction scheduling phase, loop is represented as one basic block. It (for most DSPs) consists of six basic instructions, which are supported in most microprocessors: 1) $R_1 = \text{mem}(I_a)$; 2) $I_a = I_a + 1$; 3) $R_2 = \text{mem}(I_b)$; 4) $I_b = I_b + 1$; 5) $R_3 = R_1 * R_2$; 6) $R_5 = R_5 + R_3$. Note that if a processor has not built-in multiplication instruction, there is no sense to use it for DSP mission. Usually (1) and (2), (3) and (4) instructions are executed in parallel, so finally we have 4 instructions in loop basic block: 2 loads from memory with address increment, one multiplication and one addition: 1) $R_1 = \text{mem}(I_a++)$; 2) $R_2 = \text{mem}(I_b++)$; 3) $R_3 = R_1 * R_2$; 4) $R_5 = R_5 + R_3$.

For a microprocessor which does not hold instruction level parallelism these instructions have to execute sequentially from (1) to (4). Harvard architecture has two memory spaces and can execute them in parallel by means of software pipeline increasing instruction level parallelism (superscript denotes loop iteration number, i is iteration number, $i = 3 \div N$):

$(1)^1 (2)^1$ – loop prolog
 $(1)^2 (2)^2 (3)^1$ – loop prolog
 $(1)^i (2)^i (3)^{i-1} (4)^{i-2}$ – loop body,
 $(3)^N (4)^{N-1}$ – loop epilog
 $(4)^N$ – loop epilog

The loop body takes one cycle for execution, because (1) and (2) instructions uses dual memory access and takes two words from different memory spaces. But if arrays are located in one memory space, instruction schedule of the loop body should be:

$$\begin{array}{l} (1)^i (3)^{i-1} (4)^{i-2} - \text{loop body;} \\ (2)^i - \text{loop body} \end{array}$$

which takes twice time of original loop.

While looking through these instructions in the simplest case distribution can be resolved during instruction scheduling process. But if in the loop body has more than two arrays referenced, accurate information can be got only after collecting information about memory reference conflicts. In the example above while instruction scheduling instructions (1) and (2) can not be scheduled in one processor instruction, because variables are placed in one memory space by default.

3 A Model and Structure of Retargetable and Tuneable Compiler

Conventional code generation paradigm presumes some definite microprocessor architecture type, for example: SISC, RISC, classical DSP, VLIW, RISC+SIMD-R or other ones. Compiler can produce well-formed object code for microprocessors, as their internal architecture is regular and well adopted for compilers code generation paradigm. However in case of modern DSP architectures or Programmable Logic-based processors we can not represent the underlying microprocessor architecture class so purely. Therefore code generator must be able to use mixed paradigm dependent on certain features of processor architecture. So the task is to find more comprehensive solution for retargetable code generation problem. Below retargetable compiler prototype HBPK-2 [11] structure is reviewed (shown in Fig. 1).

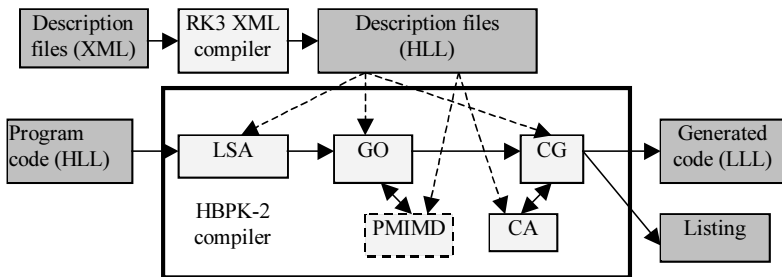


Fig. 1. Structure of retargetable compiler prototype HBPK-2.

The compiler consists of four major modules: Lexical and Syntax Analyser (LSA), Global Optimiser (GO), Code Generator (CG), Code Analyser (CA); optional – Pre-processor for MIMD cluster (PMIMD), and one external utility RK3 — XML-to-C compiler. Description files are XML files where all directives and tunings for compiler parts are collected. In spite of existing description languages with other formats like ISDL [2], XML tag model has excellent abilities to organise irregular hierarchical information and can provide platform-independent, structured, machine and human-

readable code. All information about code generation process (target architecture, sets of global optimisations, expert knowledge) is expressed as hierarchical XML tags.

Syntax analyser and global optimiser have well-known architectures [12]. These modules are independent from processor architecture, in some cases special program analysis is provided while global optimisation for special optimisations for particular processor. Syntax and lexical analyser module forms a hierarchical graph for data and control flow (HG) derived from program representation in a programming language code. Analyser is constructed in such way that module can support any programming language. Wirth diagrams are used for keeping language syntax. While constructing program graph a set of unified graph generation procedures are used. An example of HG of a sample program and its C code are presented in Fig. 2.

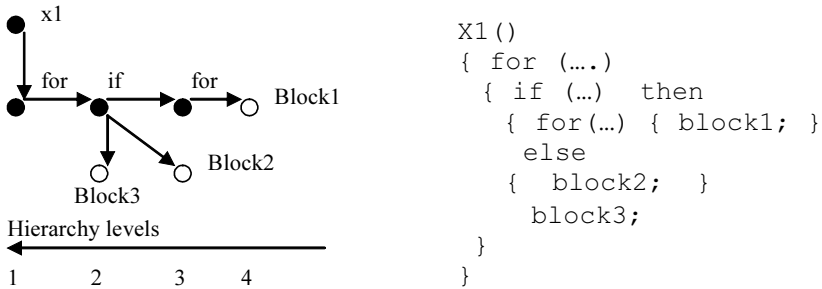


Fig. 2. Example of sample program and formed HG.

Program hierarchical graph consists of vertices of two types: recognisers and transformers. Each transformer is $H=(T,G)$, where T is a tree of hierarchy and G is a sequence of acyclic oriented graphs; H represents linear part of program or basic block. Each recogniser controls the flow of program and has one or less son on current hierarchy level and its “body” at lower hierarchy level. All loops, jumps, condition operators are represented as recognisers. Such HGs allow to describe optimisations of programs code as graph transformation productions, so global optimiser can sequentially apply them to HG. As program structure is clearly expressed optimiser can identify optimisation possibilities in regular and simple way. All non-desired optimisation changes of control flow are expressed as links between different hierarchy levels.

Global optimiser incorporates processes of local (at basic block level) and global (at some higher level) optimisations. At the HG level global optimisations can be expressed as graph grammar. Graph grammar is a set of rules [8] (graph productions) which would be applied iteratively to HG. Graph production is a set (L,R,E,C) , where L and R are two graphs, left and right parts of production, E is transforming mechanism, C is a condition of production applicability. Production p is applied to hierarchy graph G in following way of graph rewriting: 1) optimiser tries to find out L entries in G (if C is true), 2) part of G , which corresponds to L , would be deleted and some context graph D is retrieved; 3) R is built into D by mechanism E , final graph H is retrieved. We use designation $G \Rightarrow pH$ if we can retrieve H from G using p .

In the HBPK-2 compiler graph productions are used as a uniform mechanism for optimisation of HGs. Graph grammar in HBPK-2 successfully extracts index variables, looping variables (needed to express circular buffers widely used in DSP), triangle variables, index transformation expressions and finds out aggregate variables (reductions).

4 Code Generation Process

In HBPK-2 code generation process is divided into two parts: instruction selection step and instruction scheduling and register allocation step. Code generator uses descriptions of microprocessor command set, memory and register resources, and expert system. After code generation the code analysis step is performed using code analysis module. Before code generation additional optimisation can be done.

HBPK-2 compiler's code generator module is built without using any RISC-based generator code. The main idea is to make code generation process intelligent by means of expandable code generation and to provide post-generation optimisation. If classic DSP has a small amount of scalar parallelism then it can be treated as VLIW but with irregular architecture. Some commands take all instruction word, in other case five commands can be placed in one instruction word. While instruction scheduling code generator must choose the best variant from available long instruction words. This approach is quite straightforward because DSPs have a lot of other "irregular" features like non-orthogonal register files, two or more memory spaces, and different computation modes. Some of these problems can be solved using enhanced register and memory allocation techniques. Another solution could be DSP-C extension to C language standard that specifies keywords to describe additional attributes for data items at DSP but this approach is avoided here due to DSP-C non-portable keywords.

HBPK-2 compiler uses common methods for code generation, quite similar to ideas in MIMOLA, FlexWare and CHESS [1]. However a potential of HBPK-2 is much wider and can cover hardware/software codesign. HBPK-2 uses description of processor instructions and resources in terms of register files and instruction. If it would be a need in using the compiler in HW/SW codesign system, XML-descriptions can suit well. As programmers are inclined not to care in mind existing hardware issues, like structure of register ports, multiplexers, etc. human thinking in terms of available data transformations, abstract resources and constraints over resources and instructions seems much more natural. Code generation in HBPK-2 uses high-level model (likely an microprocessor model used by expert programmer) so can produce highly-optimised code due to prescribed processor model. Also HBPK-2 uses combined instruction scheduling and register allocation step which allows avoid extra register spill code if register pressure is high.

Code generator deals with target architecture and provides possible machine dependent optimisations. For efficient code generation it needs not only information about processor architecture but also about principles of optimisation of hierarchical graphs. Code generator optimisations can include (list is expandable):

- speculative code execution;
- "shortening" – replacing complex logical expression in conditional statements with sequence of conditional jump instructions;
- optimisation for parameter passing into functions;
- trace optimisation;
- replacing conditional statements with predicate executed instructions;
- utilising of delayed jump/call/return instructions;
- cycle unrolling and software pipelining;
- interprocedural values caching;
- optimisation of variable location in non-orthogonal register banks;
- DSP mode switching;
- ASIP instructions support.

For supporting different processors kernels retargeting is applied.

None of traditional compilers (like freeware GCC [13]) can be ported to DSP architecture and produce efficient DSP low level code because RISC and DSP processors have different programming paradigms. After migration to DSP architecture programmer must learn not only new names for registers and new instruction mnemonics but new style of code generation, as DSP is strongly oriented to speed up only certain algorithms like convolution, Fourier/Hartley transformations, matrix multiplication etc. For other algorithms DSP architecture only can improve instruction level parallelism. That is why for each architecture and ISA we have to define knowledge on "how code must be generated to 100% utilise the processor".

As in general case no solution can be taken using only existing information about program graph, so we use another technique based on iterative program graph analysis which can improve code generation results for software pipelining, data clustering and distributing data into memory spaces for Harvard processor architecture.

5 Compiler Expert System Productions

There is a strong need in unified representation for the compilation knowledge base and defining logical deductive system behaviour to get counsels about particular code generation problems from the base. Therefore production expert system utilisation within compiler is offered here for intelligent compilation. This approach is quite simple comparing to neural network or frame based models. It is suitable for particular realisation especially because "expert knowledge" have simple and unified form. Developed knowledge representation has following general form:

```
NAME <production_name> [ DEFAULT <default_value> ]
[ IF <predicate> OF <condition_1>[, <condition_2>[, ... , [condition_N]...]]
  DO <action> ]
[... other IF-DO rules]
```

where *production_name* – unique expert production name, used for external (from compiler modules) access; *default_value* – default value (if production returns value), used if production is not defined by user, but is needed for the compilation process;

predicate can be *ONE*, *ALL*, *MOST*, number of true conditions or production name and defines how many further conditions must be true in order to production action should be proceed; *condition_x* – conditions (represented as functions), which result is Boolean; *action* – action that should be performed if predicated conditions are true.

Further explanation is superfluous – general form of “IF-THEN” rule is described. The set of productions must be complete to specify all features, but whole “expert system” is not integral – each production serves in particular code generation procedure. Although, at this point of view, the set can’t be called “expert system”, but it is imagined by compiler as integral knowledge base, so we called it “expert system”.

This expert system may be less complex than common case [14] – without options for uncertainties and complex logical deduction machine, because logical deduction as usual in the compiler must not be deep. Generally the production set is a good formalism for representing knowledge because in comparison with other approaches (like neural networks and frames) it is simple and high-speed. High speed is a good reason, because expert system would be often referenced during code generation. Expert system productions meets two targets: a) they describe processor features in unified form; b) code generation process in controlled by a set of expert productions.

The production set is organised like associative memory, productions are accessed by name from compiler modules.

Basic expert production types [11,16] are following.

1. *Expert variables*. It’s a set of variables, defining basic properties of described processor: *NAME <name> [DEFAULT <value>] IF ALL { return 1; } DO { return <value>; }*. These variables, for example, define: a) machine word and bus width; b) stack top and frame address registers; c) flag register; d) addressing type of global data segment (via index register or not); e) local variables placement; f) existence of predicated instructions, delayed branches, zero-overhead cycles; g) stack type; h) procedure parameter placement in registers; i) existence of instructions with immediate operands; j) fundamental addressing modes; k) function inlining modes; l) global code generation advices; m) peculiarities of some processor instructions; n) processor resources definition.
2. *Optimising processor-dependent transformations*. These productions are used for architectures with acceleration instructions, which can perform complex operation as one instruction: $|a+b|$, $|a-b|$, $(a+b)/2$: *NAME <name> IF <expression-pattern-found-in-program-graph> DO <change-pattern-to-target-instruction>*. During optimisation initial pattern should be changed to instruction.
3. *Templates*. Often ASIPs have instructions, which can execute parts of complex operations, for example partial division and partial square root. In the source code initial instruction must be changed into procedure, which calculates this function via partial operations. For examples, square root at ADSP-21060:

```
Y=partroot(X);
for (i=1; i<=3; i++) Y=0.5*Y*(3-X*(Y*Y));
```

The production notation is like previous.

4. *Tables of advices*. In common code generation cases usually the compiler uses tables of cases. For example, these tables are useful for code generation for data structures access. Consider $x[i].p \rightarrow t \rightarrow k$: it consists of several atomic parts – $x; [i]; .p; \rightarrow t; \rightarrow k$. These parts has several types, for example: base ar-

ray reference (x); array reference ([i]), structure field reference(.p), structure field reference by pointer (->t, ->k). For such types the table with code generations procedures is formed. Advice tables are made for data access, function prolog/epilog generation, stack frame forming, array access.

5. *Pragmas (compiler directives)*. The pragmas are used for compiler behaviour tooling at run-time (by programmer). Compiler directives *#pragma* are used for interaction between the programmer and the compiler. Internally pragma likes the expert variable with only difference that is can be changed by the programmer.

Expert productions are atomic representation of different knowledge about code generation. But, the main problem expresses as “how to unite separate productions into integral system”.

6 Integrating Expert System into Compiler

In general case of code generation (when we have to transform internal representation of basic block into sequence of machine instructions) combinatorial algorithms work good, especially if there are no complex constrains dictated by hardware architecture. Common algorithms deals with microprocessor description like with some orthogonal RISC, any deviations lead to decreasing quality of generated code. Human mind has excellent abilities to solve this problem but usually it is good only if degree of instruction level parallelism provided by processors is small. Computer speed is advantageous in case of the VLIW/EPIC processors. The problem is good illustrated with current state-of-art inefficient compilers for DSP architectures.

First of all superblock code generation must be improved. Expert productions allow to apply microprocessor-specified optimisations and templates. Complex instructions (like addition+multiplication) are completely supported. To prevent frequent computation modes switching (normal/saturated computations) instruction clusterisation by required modes is used. During code generation all instructions (RISC-like and combined) are described in unified graph format, derived from program HG, so during instruction scheduling compiler processes unified graph with instruction uniformly without paying attention to specific architecture options. The most important task is to increase instruction level parallelism. Code generation phases: register allocation, instruction scheduling are joined into single generation phase, as it is done in modern research [15]. Single phase is the only possible way to provide retargetable code generation for “unknown” processor, especially applying expert system consultations.

Special attention is paid to extended register files utilisation. Many microprocessors (including DSPs) have special registers for extended accuracy data processing, but the programmer only sometimes indicates to the compiler that some variables must have extended accuracy. As example we can point to DSP’s accumulator registers. Usually during fixed point computation only two integer data types are used in programs – short and long, often 16-bit and 32(24)-bit. Accumulator registers have capacity from 48 up to 80 bits, but usually there is no sense to store long integers into accumulator registers because only small amount of operation can be applied to accumulators. So, the compiler must extract additional information from program

statements to handle additional register banks effectively, for example very often reduction variables is stored in accumulators. Some researches try to extend programming language with DSP extensions, for example DSP-C [16], but it likes that this method is quite ugly.

Clustered register files became recently a novel approach in microprocessor architectures. In spite of very complex register multiplexer and wiring for big register files (32bit*64 and more registers), full commutations between units, operands and results can be made only within some register file part, for example $\frac{1}{4}$, $\frac{1}{2}$ of whole file. Data stream between register file chunks may be only one word per cycle, because as usual chunks are wired with constraints (4 clusters in ADSP-21k, 2 clusters in TMS320C60). Dynamic register-cluster utilisation methods, register allocation methods with rolling back [11] are used to utilise such architectures. Expert system can be used to consult compiler how register chunks can be used providing coefficients cluster allocation for code generation procedure (like minimal ratio scheduled commands to unscheduled, optimal basic block length) so that compiler can choose right way to use clustered register file.

Expert system also is useful while register allocation if processor have specialised register set. For example, if processor has built-in loop instructions, expert system can have rule "loop counter must be held in register R_COUNT", so if the loop counter register location changes between microprocessors, expert production answers how to generate code.

Further, ASIPs have many architecture options, for example effective long instructions with large ILP can work only with some addressing modes, control transfer instructions can execute command from following basic block, instructions can be predicated, control transfers can be delayed (decreasing pipeline stalls). The list is not exhaustive, and usually the compiler has a set of different code generation procedures, which must be executed only if processor has appropriate architectural extensions. Therefore, expert system holds knowledge about applicable code generation methods and optimisations which must be applied in prescribed cases. For example, Analog Devices DSP ADSP-21k has only one form of command where dual reference to both memory spaces can be made:

$R_x = DM(I_x, M_x), R_y = PM(I_y, M_y), <arithmetic\ instruction>.$

The command is formed so that it is suitable for most DSP kernel procedures, but it uses only special form of addressing – postincrement with increment value held into step registers M_y . So, to utilise dual memory access possibility the compiler must have addresses be placed in particular registers to schedule instructions properly. In the retargetable compiler expert system can resolve such optimisations, than collect code generation routines for each microprocessor.

Sometimes expert system information is insufficient for optimisations. For example, exploiting Harvard memory (and increasing instruction level parallelism by memory access) architecture is inefficient during instruction scheduling. Previously done research gives us [17] very rough methods to exploit possibilities for data processing, especially for DSP, so quite expensive hardware options the compiler utilises inefficiently. There are many cases there we can not utilise hardware options while instruction scheduling process. The global optimiser acts independently of processor architecture, so architecture options are not under considerations at this phase. Appointed variables distribution to different memory spaces in Harvard architecture can

not be provided efficiently during code generation because exact information about variable distribution can be received only from conflicts while loading values to registers from memory. This information is inaccessible while instruction scheduling because command sequence is not yet formed. So, this optimisation requires at least two cycles of the code generation – first one to get information about the conflicts, second one to generate improved schedule with help of received information.

Below a code analysis method is presented, which combines iterative code generation process and analysis of generated code quality.

7 Code and Processor Architecture Analysis

To solve the distribution problem next scheme is proposed. For memory locations set $M=\{m_i\}$, which consists of all program variables and memory referenced constants, we define set $M_C \in M \times M$, $M_C = \{c_{ij} \mid i < j\}$, $c_{ij} \in N$; where c_{ij} – cost of conflict between variables (in they are located in one memory space) m_i and m_j . M_C can be represented as upper triangle matrix. Initially M_C hold zeroes, further information about conflicts is added while instruction scheduling. Each time if the conflict between m_i and m_j exists, value 10^D is added to c_{ij} , where D – the loop nest degree of conflicting instructions (if out of loops $D=0$, $10^D=1$).

After collection completion M_C is sorted in descending order. For biggest values c_{ij} variables m_i and m_j must be placed into different memory spaces, after resolving c_{ij} will be deleted (set to 0). Algorithm implementation details can be changed from compiler to compiler, but selection of proper memory space for variables don't require complex algorithm.

Other important tasks are variable clustering and optimal software pipeline. The variable clustering is used for microprocessors (usually DSP) which have small possibilities of indexed variable addressing for accessing local variables (which addresses from frame pointer). Usually effective addressing is possible only for small offsets, to 8 or 16 words. Thus it is useful to place jointly used variables in adjacent memory cells. Unfortunately, best variables fit can be resolved only after scheduling phase, so the problem also requires iterative code generation.

However, the most important problem is optimal software pipeline. One of the best software pipeline algorithms – Enhanced Pipeline Scheduling (EPS) [18] requires loop unrolling before processing loops. But, one can not say accurately in advance what unrolling coefficient is the best for pipelining. Of course, iterative procedure is necessary here, but before it, especially if microprocessor has SIMD-R commands, it would be useful to consult to expert system what set of unrolling coefficient would be good to apply for reducing code generation time.

EPS algorithm uses data-flow model, which allows to extract information on all Instruction Level Parallelism. Initially EPS method is dedicated for hypothetical VLIW microprocessors with infinite register resources and instruction word width. In reality, there are strong constraints in this model, but its properties are still useful for processor architecture analysis. Consider a program code that is part (loop body) of the MPEG-2 decoder which executes about 2/3 of all decoding time.

```

for(j=0; j<h; j++)
{
  for(i=0; i<16; i++)
  {
    v=((unsigned int) (p1[i]+p1[i+1]+1)>>1)-p2[i];
    if (v>=0) s+=v; else s-=v;
    s+=abs(v);
  } p1+=1x; p2+=1x;
}

```

With widely used DSP ADSP-21k the loop body can be executed in 6 cycles:

```

R1 = DM(I1,1); // (1)1
LCNTR=N-2, do L0002 until LCE;
R1 = R3+R1, R3 = R1; // (2)i(3)i
R1 = R1+1; // (4)i
R1 = LSHIFT R1 BY 1, R2 = DM(I2,1); // (5)i(6)i
R1 = R1-R2; // (7)i
R1 = ABS R1; // (8)i
L0002: R4 = R4+R1, R1 = DM(I1,1); // (9)i(1)i+1

```

And traditional DSP kernel can not provide enough instruction level parallelism to execute the body faster. As EPS software pipeline method works without taking into account availability of microprocessor resources, one can find out maximum available parallelism of loop body:

```

(1)1 – loop prolog
(2)1(3)1(1)2 – loop prolog
(4)1(2)2(3)2(1)3 – loop prolog
(5)1(6)1(4)2(2)3(3)3(1)4 – loop prolog
(7)1(5)2(6)2(4)3(2)4(3)4(1)5 – loop prolog
(8)1(7)2(5)3(6)3(4)4(2)5(3)5(1)6 – loop prolog
(9)i-6(8)i-5(7)i-4(5)i-3(6)i-3(4)i-2(2)i-1(3)i-1(1)i – loop body, i=7÷N
(9)i-5(8)i-4(7)i-3(5)i-2(6)i-2(4)i-1(2)i(3)i – loop epilog
(9)i-4(8)i-3(7)i-2(5)i-1(6)i-1(4)i – loop epilog
(9)i-3(8)i-2(7)i-1(5)i(6)i – loop epilog
(9)i-2(8)i-1(7)i – loop epilog
(9)i-1(8)i – loop epilog
(9)i – loop epilog

```

Analysis shows that to execute the pipelined loop body in 1 cycle processor must have: 1) 12 general purpose registers; 2) a shifter, 3 adders, an incrementor, a functional unit “absolute value”; 3) 9 slots wide instruction word. Today these requirements can be met. So, using EPS method without resource constraints, we can define which processor architecture is most suitable for executing such program. In case of sectioned VLIW or TTA-architecture we can slightly change number of functional devices in processor, that is excellent solution for Programmable Logic-based microprocessors. In this case we can “measure” needs in processor resources and make hardware which is the best for some tasks (but highly specialised).

Such analysis can be provided during compiling – for most nested loop bodies or for loop bodies, pointed out by user with compiler directives (pragmas). Some analysis details must be guided with expert productions, for example maximal loop unroll-

ing coefficients. Analysis report would be showed to user (as tables or graphs) after compilation.

Some of described techniques were tested using retargetable compiler prototype HBPK-2. Here are some results of code generation on standardised benchmarks DSPstone[19] (Tables 1 and 2). In left column results for ADSP-21060 are presented, in right column – for neuroprocessor L1879VM1 (NM6403).

Table 1. Code size reduction as a result of knowledge base integration into compiler

Test program	Generated code length for permanent compiler (in words)		Generated code length for HBPK-2 compiler (in words)		Code size reduction, %	
Real_update	5	31	5	28	0	9,7
N_real_updates	16	79	10	36	37,5	54,4
Complex_update	18	149	10	92	44,4	38,2
Complex_updates	31	224	19	95	38,7	57,5
Dot_product	20	102	8	29	60	71,5
FIR (filtering)	20	148	11	38	45	74,3
Convolution	18	103	10	24	44,4	76,7
Matrix	44	265	20	61	54,5	76,9
Matrix_1x3	32	158	13	43	59,3	72,7
FIR2DIM	63	387	21	61	66,6	84,2
IIR_one_biquad	23	189	12	135	47,8	28,5
IIR_n_biquad	26	343	17	139	34,6	59,4
LMS	30	243	20	62	33,3	74,4
FFT	54	817	33	144	38,9	82,3

It can be seen that the most speedup is achieved at some tasks: filtering, convolution, FFT, matrix multiplication which uses highest instruction level parallelism provided by DSP architecture. Such optimisation level cannot be achieved with useful “classic” optimisations but those results presented in the tables (50-300% speedup) is achieved owing to expert system utilisation.

8 Conclusion

An approach of retargetable and tuneable compilation that gives improvements in code generation process and enhances instruction level parallelism for DSP and VLIW processors is developed and demonstrated on the base of HBPK-2 compiler. The results are due to: 1) integration of code generation paradigm into compiler and improve utilisation of microprocessor devices; 2) optimisation which use iterative code generation like variables distribution over memory spaces; 3) analysis target processor architecture to get to know how processor units can be used for best results.

Table 2. Code speed improvement as result of knowledge base integration into compiler

Test program	Generated code speed for permanent compiler (in cycles)		Generated code speed for HBPK-2 compiler (in cycles)		Code speed improvement, %	
Real_update	5	31	5	28	0	9,7
N_real_updates	1006	8068	406	2605	59,6	67,7
Complex_update	18	132	10	84	44,4	36,3
Complex_updates	2110	21864	811	8316	61,5	61,9
Dot_product	31	175	8	51	74,1	70,8
FIR (filtering)	91	968	20	276	78	71,4
Convolution	126	760	19	71	84,9	90,6
Matrix	11618	102964	1844	6253	84,1	93,9
Matrix_1x3	137	912	37	247	72,9	72,9
FIR2DIM	12147	19194	2906	2919	76	84,7
IIR_one_biquad	23	181	12	118	47,8	34,8
IIR_n_biquad	211	2669	80	842	62	68,4
LMS	111	1832	35	213	68,4	88,3
FFT	2271	55555	1178	7827	48,1	85,9

Proposed methods expand compiler retargetability over larger range of microprocessors and can be applied not only for regular processor architectures but also for widely used ASIPs which usually have irregular instruction supported architecture. The compiler is in progress and involve some new directions for DSP-oriented processors including adaptation of the compiler to industrial microprocessor clusters with shared memory and support of highly-specialised DSPs and VLIWs, like neuroprocessors.

Further development of compiler can be integration of some multiprocessing paradigm (for example, OpenMP) to meet modern tendencies into DSP development – 4, 6 or 8 chips into SMP cluster with shared memory space. Main goal here can be expressed as achieving higher performance of code by using multiprocessor programming paradigm for clusters.

References

1. Marwedel, P., Goossens, G.: Code Generation for Embedded Processors. Kluwer Academic Publishers, Dordrecht Boston London Lancaster (1995)
2. Hadjiyiannis, G., Hanono, S., and Devadas, S.: ISDL: An Instruction Set Description Language for Retargetability. In: Proc. 34th Design Automation Conference. ACM Press, New York (1997) 299–302
3. Marwedel, P.: The MIMOLA Design System: Tools for Design of Digital Processors. In: Proc. 21th Design Automation Conference. ACM Press, New York (1984) 587–593

4. Leupers, R.: Retargetable Code Generation for Digital Signal Processors.– Kluwer Academic Publishers, Dordrecht Boston London Lancaster (1997)
5. Liem C., Paulin P.: FlexWare – a Flexible Firmware Development Environment. In Proc. European Design and Test Conference. ACM Press, New York (1994) 31–37
6. Fauth, A., Praet, J. V., Freericks, M.: Describing Instruction Set Processors Using nML. In: Proc. European Design and Test Conference.- ACM Press, Paris. (March 1995) 503–507
7. Bashford, S.: Code Generation Techniques for Irregular Architectures. Tech. Rep. 596, Universitat Dortmund (1995)
8. Fisher, R.J., Dietz, H.G.: The Scc compiler: SWARing at MMX and 3Dnow!. In: Annual WS on Lang. and Compilers for Parallel Computing. ACM Press, New York (1999)
9. Steinke, S., Schwarz, R., Wehmeyer, L., Marwedel, P.: Low power code generation for a RISC processor by Register Pipelining. Technical Report 754, University of Dortmund, Dept. of Computer Science XII (2001)
10. Leupers, R., Kotte, D.: Variable Partitioning for dual memory banks DSPs. In ICASSP, Salt Lake City (USA). ACM Press, New York. (May 2001)
11. Ragozin, D.V.: Retargetable code generation methods for irregular long instruction word microprocessor architectures. PhD thesis. Institute of Software Systems, National Academy of Sciences of Ukraine, Kiev. (2002) (in Ukrainian)
12. Aho, A. V., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques and Tools. Reading, Mass: Addison-Wesley. (1986)
13. Stallman, R.: Using and Porting the GNU Compiler Collection. Addison-Wesley Publishing, New York. (2000)
14. Michie, D. (ed): Expert Systems in the Microelectronic age. Edinburgh University Press. (1979)
15. Doroshenko, A.Yu., Kuivashev (Ragozin), D.V.: Intelligent compacting compilers for VLIW microprocessors. Problems of Programming, №1-2. (2001) 138–151. (in Ukrainian)
16. Leary, K., Waddington. W.: DSP-C. A Standard High Level Language for DSP and Numeric Processing. In: Proc. of ICASSP-90. ACM Press, New York. (April 1990) 1065–1068
17. Saghir, M., Chow, P., Lee, C.: Exploiting Dual Data-Memory Banks in Digital Signal Processors. In: Proc. of 8th Int. Conf. on Architectural Support for Programming Languages and Operation Systems. ACM Press, New York. (1996) 234–243
18. Ebcioğlu, K., Nakatani, T.: A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In: Languages and Compilers for Parallel Computer. MIT Press, Cambridge, MA. (1990) 213–229
19. Zivojnovic, V., Schraut, H., Willems, M., Schoenen, R.: DSPs, GPPs, and Multimedia Applications - an Evaluation Using DSPstone. In: Proc. Int. Conf. on Signal Processing Applications and Technology. DSP Associates, Boston, Mass. (1995) 1779–1783

The Instruction Register File

Bernard Goossens

LIAFA, Université Paris 7, 175 rue du Chevaleret, 75013 Paris, France
bg@liafa.jussieu.fr

Abstract. We present the Instruction Register File (IRF) coupled with a basic block translator, aiming to deliver a high instruction fetch rate. The IRF has one write port to load instruction cache blocks into registers. It also has p read ports to fetch up to p basic blocks per cycle from up to p registers. The translator predicts up to p on-path basic blocks per cycle and translates their start address into an IRF reference. The references are used in the fetch stage to read the registers and the basic blocks limits serve to merge the accessed registers into a dynamically predicted trace line. The IRF coupled with basic block descriptor tables avoid the need to cache traces as in the trace cache micro-architecture. Moreover, the IRF places the instruction memory hierarchy out of the cycle determining path, as does the data register file with the data memory hierarchy. The IRF performance is estimated with a SimpleScalar based simulator run on the Mediabench benchmark suite and compared to the trace cache performance on the same benchmarks. We show that on this benchmark suite, an IRF-based processor fetching up to 3 basic blocks per cycle outperforms a trace-cache-based processor fetching 16 instructions long traces by 25% on the average.

1 Introduction

Fetch bandwidth is an essential issue in actual processors. Two factors tend to limit the processor fetch capacity per cycle:

1. The presence of control flow instructions breaks the sequential nature of fetching.
2. The reduction of the cycle width leads to an increased latency of the fetch path, which increases the mis-predict penalty.

The trace cache [16,24] is the main micro-architectural technique proposed to overcome the first factor (another micro-architectural technique is multi-path fetch and execution [1]; predicated Instruction Set Architectures [14] and block-structured ISA [9] being the main contributions on the architectural side; branch alignment [4] and trace scheduling [6] are compiler optimizations to improve fetch bandwidth). Even though the trace cache is a recent proposal (1994 for the patent from Peleg and Weiser and 1996 for the main performance estimations from Rotenberg, Bennett and Smith), it has already been integrated to the Pentium IV [19]. This shows that industry architects consider it as a major innovation in the processor front-end design.

However, the main problem with the trace cache is its redundancy [21], firstly because instructions may be duplicated in the trace cache itself. Some constraints on the trace cache content like forbidding path associativity (paths AB and AC may not reside in the cache simultaneously) and basic blocks splitting tend to diminish the redundancy. Still, the trace cache is redundant with the instruction memory hierarchy.

The block-based trace cache [2] was proposed by Black, Rychlik and Shen to prevent duplications in the trace cache. The basic blocks (in short BBs) instead of the traces are cached. A trace table is constituted, each of its entries binding a set of n BB descriptors. The fetcher predicts a next trace id (next BB start address and the predicted directions of its conditional branches) and gets its constituting BBs limits from the trace table. The referenced BBs are then fetched from the BB cache and merged to form the fetched trace line. By replacing the trace cache with a BB cache, the block-based trace cache micro-architecture decreases the instruction redundancy. However, it moves the fetched trace construction from the dispatch stage back to the fetch stage. Moreover, the BB cache is still redundant with the instruction cache and it may also waste some space due to the different lengths of the BBs.

Concerning the second limitation in the fetch bandwidth (due to the cycle width reduction), some new proposals are arising. The FTB (Fetch Target Buffer) [22] is a multi-level BTB (Branch Target Buffer [17]) decoupled from the fetch path to allow a fast next basic block prediction. However, the instruction cache remains in the fetch critical path and today yet must be pipelined to scale with the cycle (today, a fetch has a two cycles latency). Prefetching techniques have also been proposed to hide instruction cache miss latency [18,5,23,26].

In this paper, we propose to remove both the trace cache and the BB cache to confine instruction redundancy to the Instruction Register File space. The fetcher dynamically constructs the fetched trace line (as in the block-based trace cache micro-architecture) from parallel register file accesses. The register file pointers are translations from BB descriptor tables such as the BBTB proposed in [27]. These translations are enqueued by a prefetcher (like the FTB in [22] which enqueues one BB descriptor per cycle in an FTQ). The IRF-based micro-architecture removes any cache from the fetch path which in the future should turn out to be our main contribution. Instead, the fetch path fetches from a register file of the same size than the data register file. Hence, the instruction path is organized on the model of the data path, with a main access point in the registers and a secondary access point in the hierarchical memory.

The paper is organized as follows: section 2 recalls the trace cache micro-architecture and explains why the IRF design should perform better. Section 3 describes the IRF fetcher and the BB descriptors prefetcher. Section 4 is devoted to the performance measures of the IRF-based micro-architecture, performed on an adapted version of the SimpleScalar [3] simulator applied to the Mediabench [12] benchmark suite. Section 5 compares these results to the performance of the trace cache micro-architecture on the same benchmark suite. Then comes a summary.

2 The Trace Cache Micro-Architecture

A trace cache (see figure 1) is composed of a fill unit, a cache of trace lines and a trace path predictor. The fill unit is placed in the dispatch stage of the pipeline (an alternative is to place it in the commit stage; in the former case, speculative traces are cached; in the latter only committed traces are; another alternative is trace preconstruction [11]). While instructions are in-order dispatched to the reservation stations (or any equivalent structure) they also fill a trace line buffer which is cached when full (we will define what *full* means soon). Multiple BBs may take place in the buffer (we will also give a precise definition of a BB in the context of the trace cache soon). The fetch stage reads the trace line indexed by the program counter (PC) from the trace cache. It cuts the fetched line at the longest prefix according to the trace path predictor outcome and delivers this prefix to the dispatch stage. This is known as “partial matching” [24]. In case of a trace cache miss, the instruction memory hierarchy is requested (in fact, both the trace cache and the L1 cache are accessed; if the former hits, the latter access is cancelled). At the other end of the pipeline at control flow instruction retire time, the processor core may correct a mis-predicted path by sending a correcting address to PC. It also adjusts the trace path predictor according to the control flow instruction computed direction and target.

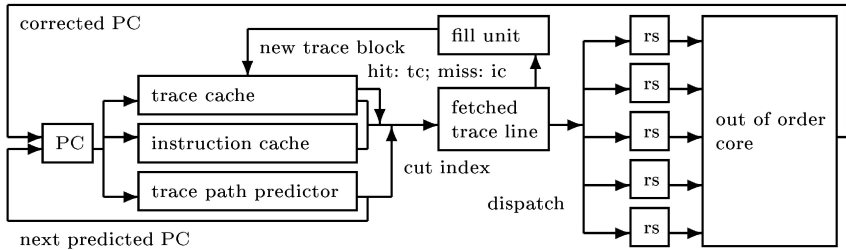


Fig. 1. The trace cache micro-architecture

In the context of the trace cache, a BB is defined by the three following rules:

1. A BB has at most one control flow instruction.
2. If a BB has a control flow instruction, it is the ending one.
3. A BB has at most n instructions (n is the trace cache block size).

The third rule ensures that a BB always fits in a trace cache line. A succession of $kn+r$, $r < n$, $k > 0$ instructions with a single ending control flow instruction is splitted into $k+1$ BBs (or k if $r = 0$). So, a BB is either ended by a control flow instruction or has length n . It is worth to notice that this definition does not invoke control flow instructions targets. A consequence is that two BBs may share a common suffix (e.g. the BB entering a loop and the BB starting the loop body). Two BBs may also share a common middle part (a first length n

one composed of chunks A and B and a second one composed of chunks B and C; B is the starting chunk of a loop body). However, two different BBs may not start with the same prefix.

A trace line is the concatenation of full BBs. No BB may span multiple trace lines. A new BB is integrated by the fill unit to the buffer containing the trace line under construction if all the following conditions hold:

1. The BB fits in the buffer.
2. The buffer does not contain any indirect jump or trap instruction.
3. If the incoming BB is ended by a conditional branch instruction, the buffer contains at most $q - 1$ other conditional branch ended BBs.

The first condition (don't cut a BB) serves to limit the redundancy in the trace cache (cutting BBs also increases their total number). The second condition (at most one indirect jump per line) comes from the fact that indirect jumps may be followed in the run trace by different BBs, which would lead to multiple trace lines with a common prefix. To furthermore limit the redundancy in the trace cache, path associativity (two different lines starting with a common BB) is not allowed (a single trace line in the trace cache for a given trace line start address). In the third condition (at most q conditional branches per trace line) the value of q is fixed according to the average number of branches in a full trace (for traces of $n = 16$ instructions, $q = 3$). Notice that the last conditional branch of the line may be followed by multiple other BBs all ended by unconditional jumps (the last one may be ended by an indirect jump).

The major advantage of the trace cache over the standard instruction cache is that it contains dynamic traces rather than static code. Hence, it is able to deliver multiple BBs fastly to the dispatch stage (a multiported instruction cache can also deliver multiple BBs to the dispatch stage with a multiple-block ahead branch predictor as the one proposed in [25] but not fastly: the multiported cache is big and so has a long latency; moreover, the concatenation of the BBs is a long process because the limits of the BBs are known lately after cache read and BB inspection).

The IRF micro-architecture described in the next section should perform better than the trace cache for the five following reasons:

1. Partial matching does not apply in the IRF context. A BB is predicted and then fetched. In the trace cache micro-architecture, BBs are fetched and possibly predicted to be discarded. Inactive issue [7] which issues and tags the fetched but predicted not on-path BBs (they are committed if the prediction turns out wrong) may partially overcome this loss of fetch bandwidth, but only when the predictor is wrong, which we hope is rare.
2. The IRF is less restrictive than the trace cache concerning path associativity. For example, if path A is ended by a conditional branch and followed either by path B (taken branch path) or by path C (not taken branch path), the trace cache can keep paths AB and C or AC and B but not AB and AC. If the predictor predicts AC and the trace cache holds AB and C, it will fetch A and one cycle later C. The IRF can hold A, B and C in one, two or three registers. If AB is requested, two pointers on A and B allow their fetching

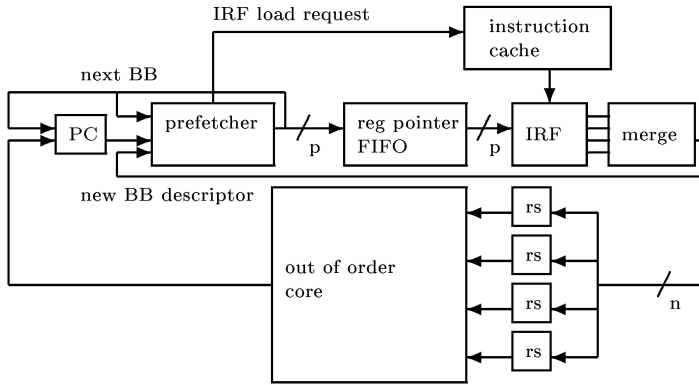


Fig. 2. The IRF micro-architecture

in the same cycle. If later, the predicted path is A and C, two new pointers allow the fetching of A and C also in the same cycle.

3. The IRF allows the fetcher to cut an ending BB. The fetcher fetches BBs until its fetch capacity is reached. If the last fetched BB exceeds the fetch capacity, its pointer remains in the queue with an updated start offset and length. In the trace cache design, if a line fetched is not full, the trace cache fetcher has no way to fill it. Moreover, if the prediction does not match the fetched trace, the trace line is cut. Inactive issue may once more partially overcome this fetch bandwidth loss but still only when the predictor is wrong.
4. The IRF fetcher can build trace lines across indirect jump boundaries and the trace cache fetcher cannot.
5. Decoupling the BBs selection from the fetch leads to non blocking IRF load requests. Fetch is idle while the instruction register is loaded but prefetch may go on and send a new load request every cycle. Any design including a BB prefetcher as in the FTB micro-architecture offers the same advantage.

3 The Instruction Register File Machine

Figure 2 is a general view of the IRF micro-architecture. Its components are a prefetcher devoted to BBs descriptors search and translation (at most p descriptors are searched and translated into register references per cycle), a fetcher that reads (up to p) registers from the IRF and merges them, a dispatch stage which dispatches the fetched instructions to the reservation stations (at most n instructions fetched and dispatched per cycle) and an out-of-order core that executes the instructions. The dispatch stage contains a BB descriptor builder. New descriptors are added to the BB descriptor tables held by the prefetcher. In the IRF context the BB definition includes a fourth rule: a BB may not span two registers (its ending instruction is either a control flow one or is followed by a register aligned instruction).

3.1 The Fetcher and the IRF

The fetch path can be built on the model of the data path. The processor core is able to execute multiple instructions per cycle because each gets its data from a register file. Modern ISA are register oriented rather than memory oriented because a register file has the advantage over an L1 data cache that it can be reduced to a few multiported entries. By limiting the number of registers in the file, each instruction can designate at least two sources and one destination. The same structure can be applied to the instruction path to allow multiple BBs fetches by placing an instruction register file between the instruction memory hierarchy and the fetch part of the processor core. In the data register file context, the instruction controls the data register file access with its sources and destination fields. In the instruction register file context, BBs are read from the register file using register pointers prepared by a prefetcher.

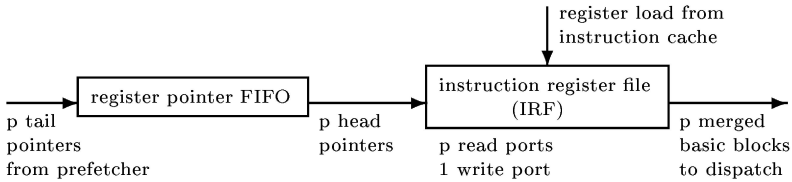


Fig. 3. The instruction register file and the register pointer FIFO

If the register file has p read ports, up to p BBs can be read and merged by the fetcher to form a dynamic trace, provided that p pointers are available (a pointer may be available but point to a loading register; in this case, the fetcher stays idle until the register is loaded). It must be noticed that the merging operation is far much simpler than what is required with a multiported cache and a multi-block fetcher. The BBs need not be inspected to find their ending instruction. Their start address and length are prepared by the prefetcher and given to the fetcher with the register reference. The merging process is just a matter of shifting and multiplexing the register file outputs. If the p BBs accumulated length overtakes the fetch capacity, the BB that has not been fully fetched remains in the pointer FIFO with an adjusted offset and length. In the reverse case, if the p fetched BBs accumulated length is far under the fetch capacity, the fixed number of read ports on the register file does not allow any further adding (in this particular case, the trace cache could perform better: a trace line may contain any number of BBs as long as the maximum of q conditional branches is not exceeded; hopefully for the IRF micro-architecture, as the simulation has shown, the case is rare). Figure 3 depicts the fetcher design.

3.2 The Prefetcher and the BB Descriptor Tables

The second part of the proposed micro-architecture is the prefetcher (see figure 4). It is responsible of the instruction register pointers production. For this

purpose, it uses a register allocation table (TRALL). This table keeps for each register of the IRF the aligned address of the instructions it contains. Translating a BB address into a register pointer is a parallel search in this table. If the BB aligned address is found, the hit entry number gives the desired pointer. In case of a miss, a free register is allocated (a register is free if no waiting-to-be-fetched BB refers to it; table entry e contains a field pointing to the newest enqueued waiting-to-be-fetched BB referring to register e ; a FIFO head entry h referencing e frees register e if when dequeued by the fetcher, the TRALL table entry e still points to h). If no register is free, prefetching stops. When a free register is allocated, a register load is requested (load from the instruction memory hierarchy). As we assume a single read port L1 instruction cache, we stop prefetching for the cycle (hence, the prefetcher can initiate at most one load per cycle).

The translations wait in a register pointer FIFO until they are consumed by the fetcher. Each FIFO entry contains a register pointer field, an offset field (offset of the BB first instruction in the register), a length field (length of the BB), an address field (base address of the BB; this is useful for the BB instructions executions) and a prediction field (used for conditional branch ended BBs; the prediction made is forwarded to the core to allow its verification).

The BB address the prefetcher translates is obtained from a table search. The search may provide the BB address itself (conditional branch and immediate jump) or a way to get it (return: the return address is given by the return address stack; indirect jump: the target is searched in a BTB). The prefetcher uses three tables of BB descriptors, each table caching descriptors according to the BB ending instruction. There is one table for the conditional branch ended BBs (TCOND), one for the indirect jump ended BBs (TIND; it includes returns) and one for the immediate jump ended BBs (TIMM). The tables are separated because the descriptors they hold are not identical. In every descriptor we find a tag field (the upper part of the BB start address, the lower part being used to address the tables; as the tables have different sizes, the tag field width varies from one table to another). There is also a length field. In the TIND and the TIMM table, we find a target field (for branches, it contains the computed target instead of the displacement; this loses space but accelerates the prefetching process). In the TIMM and TIND tables, we have a link bit (indicating whether the ending jump instruction is a jump and link or not). In the TIND table, we have a return bit (indicating whether the ending jump instruction is a return or not). Each descriptor in the TCOND and TIMM tables should be approximately 8 bytes wide. Each descriptor in the TIND table should be approximately 4 bytes wide (no target address field).

The current BB start address is the PC. It addresses the three BBs tables and the TRALL table in parallel. It also accesses the conditional branch predictor and the indirect jumps BTB (not represented on figure 4). The current BB is translated and its register pointer translation is enqueued in the register pointer FIFO. The prefetch stops if the queue is full. In parallel with the translation, one of the BBs tables may hit and fix the next BB address. If the TCOND table hits, the branch prediction is used. If the TIND table hits, its return field indicates whether the target address should be obtained from the top of the return address stack or from the BTB. If all the tables miss, the next BB address

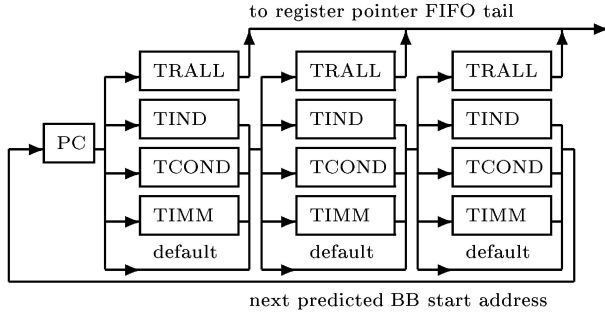


Fig. 4. The prefetcher path with 3 searches per cycle

is the next aligned address (such a miss either comes from a BB that had to be cut not to span two registers or from an unknown BB; in the latter case, the BB may contain a never yet encountered control flow instruction; it will be seen at dispatch time, the sequence of instructions will be cut into two BBs at the control flow boundary and the prefix BB will have its descriptor written in the appropriate BB descriptor table; eventually, at retire time the missed control flow instruction will restart the prefetcher).

If the IRF has p read ports, the fetcher consumes pointers at a peak rate of p per cycle. The prefetcher must produce them at that same rate which means that each table must be accessed p times in the cycle (the last access to the tables in the cycle fixes next PC). If p is high and the cycle is short, it may be a problem for the TCOND table, due to its size (in the simulation described in section 4, we assumed a 2048 entry TCOND table which represents 16KB of storage; the TIMM and TIND tables were assumed to have 256 entries each which represents 2KB and 1KB of storage). An alternative design is to have multi-descriptors tables as in the block-based trace cache micro-architecture (a table gives multiple BB addresses in a single access; the addresses can then be translated in parallel). The table content relies on a branch prediction. Rakvic, Black and Shen [20] have proposed a way to build it at completion time. However, binding multiple BBs together has bad side effects such as wrong path prefetching and descriptor redundancy. Another option is to have a two level TCOND table as in the FTB design. We do not further investigate such design options in this paper to focus on the IRF design and performance.

Figure 4 depicts the prefetcher. The simulated micro-architecture assumed 3 read ports on the IRF ($p = 3$). On the figure, the different tables are assumed to have 3 identical copies each. Different implementations options are possible (among which we can mention multi-ported tables or wave-pipelined accesses [8]) but are not furthermore investigated in the paper.

4 The IRF Micro-Architecture Performance

In order to measure both the fetch bandwidth and the IPC performance of an IRF micro-architecture, we have adapted the SimpleScalar simulator to an IRF-based fetcher and a BB descriptor prefetcher and translator. The simulator was run on the Mediabench benchmark suite. Table 1 gives the main characteristics of the programs in the Mediabench suite (mainly the number of instructions in a full run, limited to the 500 millions first instructions and the number of instructions per branch (IPB)).

Table 1. The Mediabench suite

<i>program</i>	<i>run size</i>	<i>IPB</i>
rawcaudio	6,618,178	3.7357
rawdaudio	5,459,855	3.3642
epic	52,901,786	6.7474
unepic	6,988,923	4.7280
cjpeg	15,510,101	6.3729
djpeg	4,649,119	19.6027
pegwitencode	32,157,921	8.5496
pegwitdecode	18,182,963	9.0270
g721encode	275,165,274	4.3445
g721decode	268,411,763	4.3072
gs	500,000,000	4.6841
mipmap	68,415,518	6.2431
osdemo	22,712,467	5.4228
texgen	99,441,965	7.5833
toast	234,070,244	20.4992
untoast	73,621,328	5.5866
mpeg2encode	500,000,000	5.7330
mpeg2decode	171,225,367	8.4695
average		7.50

The IRF micro-architecture design parameters for the simulation were fixed as follows: The IRF register file has 8 registers of 32 instructions each, 3 read ports and one write port. The maximum fetch length is 16 instructions. In the prefetcher, the register pointer FIFO has 16 entries. The TIND and the TIMM tables have 256 entries each (1KB and 2KB of storage). The TCOND table has 2048 entries (16KB of storage). The return stack has 64 entries. The indirect jumps BTB has 256 entries (2KB). The branch predictor is a bimodal one with 4096×2 bits counters (1KB). As the IRF removes the instruction cache from the fetch path, we have modeled a big and long latency L1 instruction cache (as in the HP PA8500 [10]). It has a capacity of 512KB and a 3 cycles latency. Because each register holds 32 instructions (twice the fetch peak bandwidth), to allow a full load in one shot, each L1 cache line holds 128 bytes. The associativity is 4. The L1 data cache has a capacity of 64KB, with blocks of 64 bytes and an

Table 2. Performance of the IRF micro-architecture

<i>program</i>	<i>IRF PPC</i>	<i>IRF IPC</i>
rawaudio	9.7320	1.7403
rawdaudio	9.5707	2.6249
epic	15.5377	6.6439
unepic	8.3403	3.6559
cjpeg	12.4742	5.0975
djpeg	15.1504	7.3157
pegwitencode	16.4982	3.5058
pegwitdecode	16.8531	3.7092
g721encode	12.1354	3.7108
g721decode	11.7895	4.1403
gs	9.6364	3.6196
mipmap	15.3304	5.8612
osdemo	15.5832	8.2981
texgen	19.4091	6.3602
toast	12.2308	7.5460
untoast	11.3206	6.7441
mpeg2encode	14.6591	2.5157
mpeg2decode	23.9409	6.1446
average	13.8996	4.9574

associativity degree of 4. The latency is 2 cycles. The L2 cache is unified, 1MB, 128 bytes blocks, associativity 4 and a latency of 8 cycles. The memory has a first word access time of 50 cycles and a following words access time of 2 cycles. The bus width is 16 bytes. There is a 16 entries ITLB and a 32 entries DTLB both with an 8 cycles access latency.

The size of the IRF ($8 \times 32 \times 4$ bytes, 3 read ports, 1 write port) has been chosen to keep its die occupation area close to a standard 32×8 bytes data register file with 8 read ports and 4 write ports.

Table 2 gives the measured PPC (instructions prefetched per cycle; this is the sum of all prefetched BBs lengths divided by the number of cycles) and IPC. The PPC value can be greater than the maximum fetch length. However, at most 16 instructions are fetched, the last BB being cut by the fetcher and fetched in two cycles.

Table 3 gives the IRF miss rate and the branch predictor miss rate. As the storage devoted to the IRF and to the branch predictor is limited by the cycle width, the miss rates are sometimes high.

5 Comparison with the Trace Cache Micro-Architecture Performance

We have derived a second simulator from the SimpleScalar tool set implementing a baseline trace cache (i.e. not including enhancements such as trace packing and branch promotion [15] nor inactive issue [7]). The simulator was run on the same

Table 3. Miss rates of the IRF micro-architecture

<i>program</i>	<i>IRF miss</i>	<i>b.pred. miss</i>
rawaudio	0.04%	29.34%
rawaudio	0.10%	19.53%
epic	0.87%	4.38%
unepic	0.41%	6.97%
cjpeg	5.41%	7.24%
djpeg	4.71%	8.64%
pegwitencode	7.01%	14.17%
pegwitdecode	6.74%	14.27%
g721encode	5.10%	10.27%
g721decode	4.88%	8.45%
gs	15.03%	5.09%
mipmap	26.18%	1.25%
osdemo	8.32%	1.73%
texgen	18.28%	5.51%
toast	13.66%	7.30%
untoast	1.55%	2.42%
mpeg2encode	0.48%	24.34%
mpeg2decode	2.60%	10.04%
average	6.74%	10.05%

Table 4. Performance of the trace cache

<i>program</i>	<i>tc IPC</i>	<i>improvement</i>
rawaudio	1.8148	-4.11%
rawaudio	2.5017	4.92%
epic	5.6007	18.63%
unepic	2.9247	25.00%
cjpeg	4.2303	20.50%
djpeg	6.6510	9.99%
pegwitencode	3.0784	13.88%
pegwitdecode	3.2759	13.23%
g721encode	2.7669	34.11%
g721decode	2.8850	43.51%
gs	3.1903	13.46%
mipmap	3.4519	69.80%
osdemo	2.7875	197.69%
texgen	4.0244	58.04%
toast	7.2792	3.67%
untoast	6.7687	-0.36%
mpeg2encode	2.4262	3.69%
mpeg2decode	5.3741	14.34%
average	3.9462	25.62%

benchmarks. Because the trace cache also removes the instruction cache from the fetch path (but it replaces it by the trace cache itself), we have modeled the same big and long latency instruction cache than for the IRF micro-architecture. The only difference was in the block length fixed to 64 bytes in the trace cache design. The branch predictor is a two level predictor [13] accessed at most three times per cycle (this should give an optimistic hit rate compared to the accuracy of a true trace path predictor) with $32K \times 2$ bits counters at the second level and $32K \times 15$ bits indexes at the first level. All other parameters are the same than the ones chosen for the IRF micro-architecture.

Table 4 shows the performance of the trace cache design and the improvement of the IRF micro-architecture over the trace cache one. The IRF design outperforms the trace cache design except in two cases (“rawaudio” and “untoast”), even though its predictor is smaller and the fetch is limited to 3 BBs by the number of read ports on the IRF (the trace cache micro-architecture can fetch more than 3 BBs every time a fourth BB is not ended by a conditional branch and fits in the trace line). The maximum improvement (close to 200%) is obtained with “osdemo”. This example is a very good illustration of what can go wrong in the trace cache: most of the time, the predictor predicts that the first conditional branch in the fetched trace line should go in a direction opposite to the one traced. Whatever the hit rate, the trace has to be cut after a short prefix. Maybe inactive issue could have some impact on “osdemo” performance. We did not measure this. A good score is also obtained with “mipmap” (70%) and “texgen” (58%). In both cases, the IRF miss rate is high (this explains that the gain is not as good as in “osdemo”) but the branch predictor miss rate is low for “texgen” and very low for “mipmap”. These examples show all the potential of the IRF design if coupled with an accurate branch predictor. At the other end, we find “rawaudio” and “untoast” for which the trace cache performs a little better than the IRF (4%). In “rawaudio”, it comes from a bad IRF performance due to a very high mis-prediction rate (close to 30%). The reason is clear when we see that the IPB is only 3.74. For “rawaudio” this time it is the IRF that performs 4% better than the trace cache. The IPB is also very low (3.36 instructions per branch). However the predictor miss rate is lower (20%) which explains why the IRF has a better score. For “untoast” even though the miss rates are very low, the trace cache has a higher IPC. It comes from a good performance of the trace cache as shows the fact that it is the only case for which the IPC is greater than the IPB (we run more than one basic block per cycle). In the case of “djpeg” and “toast”, the IPB is very high (20 instructions per branch). In such a condition, fetching multiple BBs does not give any advantage. The two architectures are quite close, with a slight advantage to the IRF (10% and 4%).

6 Summary

In this paper we have presented the IRF micro-architecture. It is composed of an instruction register file which is the first instruction storage on the fetch path. Its multi-ported organization and the basic blocks limits delivered by the prefetcher allow a fast multi-block fetching and merging which provides a higher fetch rate than in the trace cache micro-architecture. Moreover, the IRF does not

require the redundant storage of the traces or even the basic blocks. The IRF also removes any cache from the fetch path, allowing fetching to scale with the cycle reduction without requiring new fetch pipeline stages. We have shown that on the Mediabench benchmark suite, the IRF micro-architecture when parameterized to fetch up to 3 basic blocks per cycle outperforms a 16 instructions per cycle trace cache based micro-architecture of an average 25%.

Table 5. Variation of the number of read ports

<i>program</i>	<i>IPC 2</i>	<i>IPC 4</i>
rawcaudio	1.6835	1.7766
rawdaudio	2.4143	2.7070
epic	5.9511	6.5090
unepic	3.4757	3.5977
cjpeg	4.6668	5.1155
djpeg	7.2522	7.2734
pegwitencode	3.4423	3.5081
pegwitdecode	3.6176	3.7209
g721encode	3.5036	3.7992
g721decode	3.8675	4.2665
gs	3.3859	3.7167
mipmap	5.3869	5.6694
osdemo	7.1942	7.4454
texgen	5.8945	6.2776
toast	7.4022	7.5668
untoast	6.6037	6.7495
mpeg2encode	2.4150	2.5348
mpeg2decode	5.8224	6.1290
average	4.6655	4.9091

In this design, the prefetch path should be the most critical one because it requires multiple sequential accesses to basic block descriptor tables and control flow predictors. We have suggested various design options to allow up to 3 descriptor translations within a cycle, among which we find a hierarchical organization of the tables as in the FTB micro-architecture, a multiple descriptors binding as in the block-based trace cache or limiting the prefetch to 2 BBs per cycle which gives an IPC only 7% less than the IPC obtained when fetching up to 3 BBs (see table 5: the left values are the IPC of the design with 2 BBs read per cycle and the right ones are the IPC with 4 reads per cycle). This is still 19% better than the trace cache performance.

A future work is to investigate the prefetcher different design options mentioned in the paper (e.g. hierarchical descriptor tables and predictors or completion time binding of BBs descriptors) in relation with the cycle width. In particular for the experiments reported in this paper, we have placed the IRF and the trace cache on the same level concerning pipeline latency. In fact, because with

the IRF no cache is anymore accessed in the critical path, the pipeline should be shorter than in a trace cache design in which accessing the cache should require at least 2 cycles today and probably more tomorrow.

References

1. P.S. Ahuja, K. Skadron, M. Martonosi, D.W. Clark: Multipath execution: opportunities and limits. ICS12 (1998)
2. B. Black, B. Rychlik, J.P. Shen: The block-based trace cache. ISCA26 (1999)
3. D. Burger, T.M. Austin: The SimpleScalar tool set, version 2.0. Technical report 1342, University of Wisconsin-Madison (june 1997)
4. B. Calder and D. Grunwald: Reducing branch costs via branch alignment. ASPLOS6 (1994)
5. I.K. Chen, C.C. Lee and T.N. Mudge: Instruction prefetching using branch prediction information. ICCD'97 (1997)
6. J.A. Fisher: Trace scheduling: a technique for global microcode compaction. IEEE Trans. on Computers, C30(7) (1981), 478–490
7. D.H. Friendly, S.J. Patel, Y.N. Patt: Alternative fetch and issue policies for the trace cache fetch mechanism. Micro30 (1997)
8. C.T. Gray, W. Liu and R.K. Cain III: Wave pipelining: theory and CMOS implementation. Kluwer academic publishers, Norwell (1993)
9. E. Hao, P. Chang, M. Evers and Y. Patt: Increasing the prediction fetch rate via block-structured instruction set architectures. Micro 29 (1996)
10. <ftp://www.hotchips.org/pub/hot7to11cd/hc98/pdf.1up/hc98-1a-johnson.1up.pdf>
11. Q. Jacobson and J.E. Smith: Trace preconstruction. ISCA27 (2000)
12. C. Lee, M. Potkonjak, W.H. Mangione-Smith: Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. Micro30 (1997)
13. S. Mc Farling: Combining branch predictors. Technical report TN-36, DEC-WRL (june 1993)
14. S.A. Mahlke et al.: Characterizing the impact of predicated execution on branch prediction. Micro27 (1994)
15. S.J. Patel, M. Evers, Y.N. Patt: Improving trace cache effectiveness with branch promotion and trace packing. ISCA25 (1998)
16. A. Peleg, U. Weiser: Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. patent 5–381–533 (1994)
17. C.H. Perleberg and A.J. Smith: Branch target buffer design and optimization. IEEE Trans. on Computers, 42(4) (1993), 396–412
18. J. Pierce and T. Mudge: Wrong-path instruction prefetching. Micro29 (1996)
19. <ftp://download.intel.com/pentium4/download/netburstdetail.pdf>
20. R. Rakvic, B. Black and J.P. Shen: Completion time multiple branch prediction for enhancing trace cache performance. ISCA27 (2000)
21. A. Ramirez, J. Larriba-Rey, M. Valero: Trace cache redundancy: red and blue traces. HPCA6 (2000)
22. G. Reinmann, T. Austin, B. Calder: A scalable front-end architecture for fast instruction delivery. ISCA26 (1999)
23. G. Reinmann, B. Calder, T. Austin: Fetch directed instruction prefetching. Micro32 (1999)

24. E. Rotenberg, S. Bennett, J.E. Smith: Trace cache: a low latency approach to high bandwidth instruction fetching. *Micro29* (1996)
25. A. Seznec, S. Jourdan, P. Sainrat, P. Michaud: Multiple-block ahead branch predictors. *ASPLOS7* (1996)
26. A. Veidenbaum, Q. Zhao, A. Shameer: Non sequential instruction cache prefetching for multiple-issue processors. *Int. Journal of Highspeed Computing*, 10(1), (1999), 115–140
27. T. Yeh and Y. Patt: A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *Micro25* (1992)

A High Performance and Low Cost Cluster-Based E-mail System

Woo-Chul Jeun¹, Yang-Suk Kee¹, Jin-Soo Kim², and Soonhoi Ha¹

¹ School of Electrical Engineering and Computer Science,
Seoul National University,
San 56-1, Sinlim-dong, Gwanak-gu, Seoul 151-744, Korea
{wcjeun, yskee, sha}@iris.snu.ac.kr

² Division of Computer Science,
KAIST,
Daejeon 305-701, Korea
jinsoo@cs.kaist.ac.kr

Abstract. A large-scale e-mail service provider requests a highly scalable and available e-mail system to accommodate the increasing volume of e-mail traffic as well as the increasing number of e-mail users. To reduce the system development and maintenance cost, it is requested to make the system modular using off-the-shelf components. In this paper, we propose a cluster-based e-mail system architecture to achieve the goals of high scalability and availability, and low development and maintenance cost. We adopt the internal structure of a typical Internet e-mail system for a single server, called the MTA-MDA structure, to the proposed system architecture for the low cost requirements. We have implemented four different system configurations with the MTA-MDA structure and compare their performances. Experimental results show that the proposed system architecture achieves all the design objectives.

1 Introduction

The growth of the Internet has led to an explosion in volumes of e-mail traffic and in number of the users of e-mail service. At the same time, large-scale e-mail service providers have appeared. They have hundreds of millions of subscribers and process billions of messages: for example, in May 2001, Hotmail had over 100 million users and Yahoo! Mail, in March 1999, served 45 million users with 3.6 billion mail messages [1][2].

E-mail systems can be evaluated using various criteria [2] among which we are concerned about the following four in this paper: scalability, availability, flexibility, and extensibility. A system is highly scalable if the message throughput of the system increases linearly with the cluster size. As the cluster size increases, the probability of node failure also increases, so that making the system highly available is crucial to the service provider. An available system isolates a local failure from the system operation to avoid global outage. Considering these performance requirements, a cluster-based system architecture appears more suit-

able for the large-scale e-mail systems than a single server system. Thus, our proposed e-mail system architecture is a cluster system architecture.

Flexibility and extensibility are related with the system development and maintenance cost. A system is called flexible if it has a modular structure that consists of replaceable components with only a little modification if any. An extensible system allows one to improve the system performance easily by upgrading some components. Considering these requirements, we adopt a structure that we call as “*MTA-MDA structure*” for short. MTA (Message Transfer Agent) and MDA (Message Delivery Agent) are server agents in a typical single-server e-mail system [3]: MTA receives an e-mail via standardized SMTP (Simple Mail Transfer Protocol) [4] and MDA stores it in a repository to be retrieved later by user’s request. Even though the MTA-MDA structure is not a standardized structure, it has benefits of extensibility and flexibility: an e-mail system can be easily constructed by using off-the-shelf components for the MTA and the MDA.

Cluster-based e-mail systems can be classified into two approaches by their internal structure. One approach is to let each cluster node preserve the MTA-MDA structure except the modification to store an incoming mail to a remote node. Christenson et al. developed a cluster e-mail system using NFS (Network File System) for remote delivery and showed good scalability, flexibility, and extensibility [5]. However, it fails to meet the availability requirement.

The other approach is to make its own structure supporting standardized protocols for e-mail service: POP (Post Office Protocol) [6] and IMAP (Internet Message Access Protocol) [7] for e-mail retrieval and SMTP for e-mail exchange [2][8]. In this approach, they could successfully design scalable and available e-mail systems with a proprietary internal structure. But, it has serious drawbacks to avoid: lack of flexibility and extensibility. Without using existent off-the-shelf components, it takes long time and much effort to develop the system.

In this paper, we present a novel cluster-based e-mail system architecture using the MTA-MDA structure. Its modular structure provides many opportunities to improve the performance easily by using new off-the-shelf components. Moreover, this system architecture meets both scalability and availability requirements.

In section 2, we review the MTA-MDA structure of a typical e-mail system and overview some cluster-based e-mail systems classified by their internal structure. In section 3, the proposed cluster-based e-mail system architecture is explained. Section 4 presents the implementation of four different system configurations based on the MTA-MDA structure. Section 5 shows the experimental results and section 6 concludes the paper.

2 Backgrounds and Related Work

Fig. 1 shows the structure of a typical e-mail system for a single server, focusing on the receiving process of e-mail messages. The server consists of three agent programs: MTA, MDA, and MRA (Mail Retrieval Agent) [3]. MTA is a server program that transfers e-mails between machines on the Internet via the SMTP

protocol. Three well-known examples of MTAs are ‘*sendmail*’, ‘*qmail*’ and ‘*postfix*’. When MTA receives an e-mail message, MTA invokes an appropriate MDA to store the e-mail in the repository. If the message is destined for a user that has an account on the local system, MTA calls a local MDA that writes the message to the recipient’s mailbox. Otherwise, MTA calls another MDA to reroute the message to the destination MTA. The message can be filtered by mail filtering programs on the way to the mailbox. Some examples of local MDAs in UNIX systems are ‘*procmail*’, ‘*/bin/mail*’, and ‘*mail.local*’. While the mailbox format is not standardized, the most commonly used format in a single server system is ‘*mbox*’.

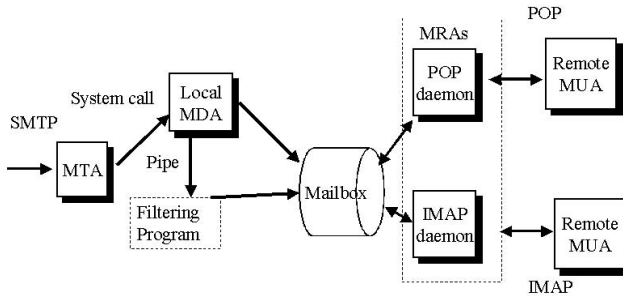


Fig. 1. The structure of a typical Internet mail system for a single server

Although accessing the mailbox directly can retrieve stored e-mail messages, MRA allows one to read the messages across the Internet. Upon request, an MRA accesses the user’s mailbox. Two Internet protocols have been proposed for MRAs: they are the older and simpler POP and the newer and more complex IMAP. MUA (Mail User Agent) is a client program used by a user to send or receive e-mails. The Outlook Express of Microsoft, Inc. is an example.

This modular MTA-MDA structure allows a typical e-mail system to be constructed as a collection of loosely connected components that are developed independently. Therefore, some cluster-based e-mail systems have been developed to adopt the MTA-MDA structure on each node for reducing the development cost and preserving the benefits of flexibility and extensibility. On the other hand, others use a different proprietary architecture. Now, we briefly overview some existent cluster-based systems on each category.

2.1 E-mail Systems with the MTA-MDA Structure

Christenson et al. proposed a scalable e-mail system using the MTA-MDA structure in EarthLink Network, Inc [5]. Fig. 2 shows the message delivery and retrieval process in the case that the recipient’s mailbox exists on a remote node. When a message arrives, the MTA forks a local MDA. Then, the local MDA

queries an authentication SQL DB about the recipient information. If the recipient's mailbox exists on the local node, the MDA stores the message into the mailbox. If it exists on a remote node, the local MDA transfers the message to the remote node by NFS mechanism. EarthLink system uses the '*sendmail*' as MTA with slight modification to solve a file-locking problem. They modified the '*mail.local*' to obtain the user information from SQL DB instead of the '*passwd*' file. Compared with the basic MTA-MDA structure of Fig. 1, the NFS module plays the role of an interface module between a local MDA and the remote mailbox.

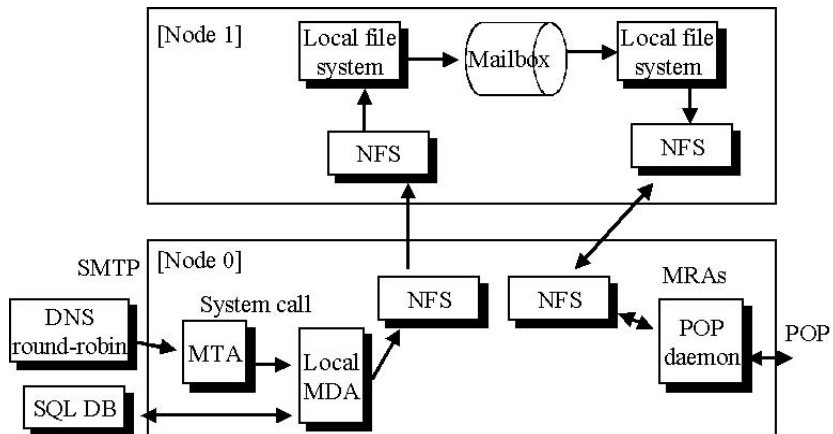


Fig. 2. The architecture of the EarthLink system

The scalability of this system depends on the performance of the NFS and the SQL DB. As processing power of each node and the network performance both increase, the system shows good scalability, flexibility, and extensibility. However, it has a serious drawback: if any node fails, the whole system stops the operation, soon. If an MDA process sends an NFS request to the failed node, the process sleeps until receiving the reply. As the number of sleeping processes increases, the load average of the node also increases. Then, high load average makes the '*sendmail*' MTA refuse all SMTP requests including the messages headed for other available nodes. This means that the system stops e-mail service until the failed node is recovered.

M. Grubb presented a scalable e-mail system that is deployed in Duke University [9]. The system provides aliased mail addresses whose real addresses are mapped to the e-mail servers geographically distributed in the campus. Thus the e-mail system plays the role of distributing the incoming e-mails to the distributed single servers after translating the aliases addresses to the real addresses. The system has the MTA-MDA structure. It is considered as a distributed e-mail system rather than a cluster-based system though it may be classified as a loosely coupled cluster system.

2.2 E-mail Systems with a Proprietary Structure

Several cluster-based e-mail systems with a proprietary structure have been developed among which Porcupine [8] and NinjaMail [2] are two representative systems. Y. Saito et al. developed the Porcupine system as a scalable and highly available e-mail system. The system partitions the user information and the user mailboxes across the nodes and replicates them to achieve high availability. Since they do not preserve the MTA-MDA structure, no off-the-shelf components could be used and significant effort has been paid to build the system. Their idea of caching the user information on the main memory for high performance is adopted in our proposed system.

The UC Berkeley's NinjaMail is built on top of UC Berkeley's Ninja software infrastructure [10], which supports scalable and highly available Internet services, and OceanStore [11] wide-area data storage architecture. Thus, flexibility and extensibility is limited within the Ninja infrastructure. We do not know performance number for the system.

3 Proposed E-Mail System Architecture

In this section, we explain two versions of the proposed cluster system architecture. The differences between the proposed system and the existent systems are summarized in Table 1. Similarly to the EarthLink system, the proposed system architecture augments an interface module between an MDA and the remote mailbox in the basic MTA-MDA structure. Fig. 3 shows the first version of the proposed e-mail system architecture. Message delivery process is similar to the EarthLink system until a local MDA is forked by the MTA. We modify the local MDA, '*mail.local*', to only forward the incoming message to the interface module via a UNIX domain socket. Now, the message delivery role is delegated to the interface module that stores the input message into the local file system or transfers it to the interface module of the remote node. If the remote node fails, the interface module detects the failure immediately at the connection establishment of TCP socket for remote delivery. Then, the interface module signals an error to the caller MDA and eventually to the sender. Unlike the NFS module, the interface module can service other e-mail deliveries to make the system available.

The version 1 system, however, has a performance overhead since there is a redundant message transfer from a local MDA to the interface module for remote delivery. We make a local MDA transfer the e-mail messages directly to the interface module of the remote node without intervention of the local interface module in version 2 system as shown in Fig. 4.

In the proposed system, we use the off-the-shelf components for the MTA and the MDA: '*sendmail*' or '*postfix*' for the MTA and '*mail.local*' for the MDA in the current implementations. Therefore our main effort to build the system is confined to the design of the interface module.

The proposed cluster system is implemented as a web-mail system. Therefore, we define the following basic roles that the interface module should serve:

Table 1. E-mail system comparison. (* : Availability feature is not supported in the current implementation.)

	Scalability	Availability	Flexibility	Extensibility
EarthLink	O	X	O	O
Porcupine	O	O	X	X
NinjaMail	O	O	X	X
Version 1	O	O*	O	O
Version 2	O	O	O	O

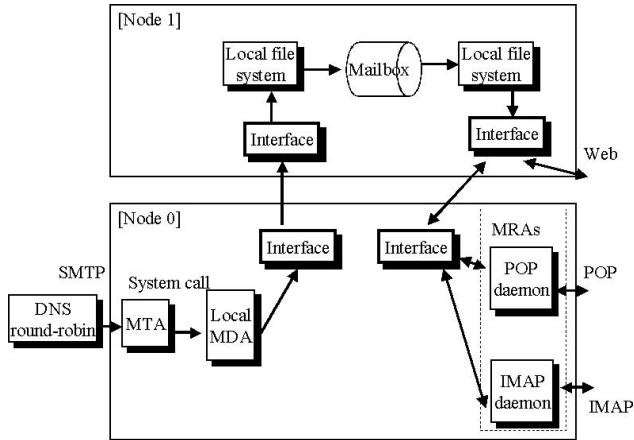


Fig. 3. The architecture of the proposed e-mail cluster system (version 1)

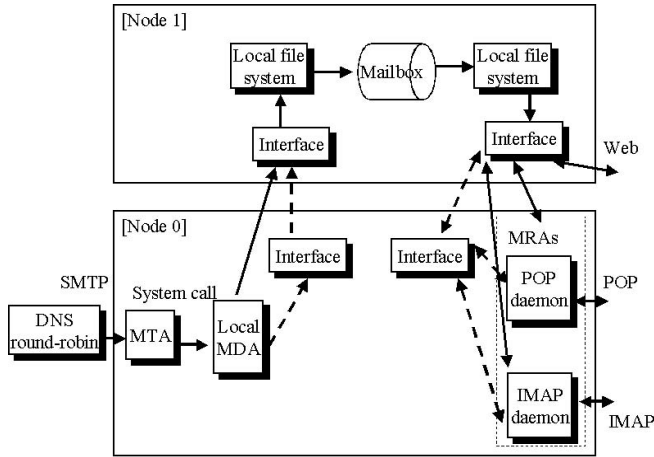


Fig. 4. The architecture of the proposed e-mail cluster system (version 2). For comparison, we also display the message delivery path of the version 1 system (*dotted lines*)

- E-mail message delivery to the local mailboxes
- User authentication
- Web-mail service for compact message summary
- Web-mail service for user information handling
- Web-mail service for user log-on request

Therefore, we define five different subsystems as displayed at the bottom of Fig. 5 and Fig. 6. A subsystem is a collection of functions. An ‘*auth subsystem*’ manages user authentication information using an SQL DB. We cache the user authentication information to the memory as a hash table, borrowing the idea of the Porcupine’s work for faster user authentication [8]. A ‘*logon info subsystem*’ keeps a status of user log-on information for checking an illegal logon trial. A ‘*user info subsystem*’ manages additional information of users such as name, address, phone number, and so on for the web-mail service. A ‘*mailbox subsystem*’ manipulates the mailboxes of users in the directory structure as described earlier. A ‘*message info subsystem*’ keeps additional information of messages such as sender, recipient, date, size, and so on. We expect that these modular subsystems make us easy to replace and improve them.

3.1 Interface Module, Version I

Fig. 5 shows the structure of the interface module, version 1. Four kinds of threads compose the interface module, among which the ‘*operation threads*’ are the central threads that process the e-mail messages and the web-service requests using the subsystems. A ‘*node thread*’ is a thread that receives a message from the local MDA or from the other nodes. The ‘*node thread*’ encapsulates the message in an ‘*op-entry*’ structure and puts it into the central ‘*work-list*’ queue. A ‘*data thread*’ is a thread that receives a request from the web, a POP daemon, or an IMAP daemon and puts the request in an ‘*op-entry*’ structure into the ‘*work-list*’ queue. An ‘*auth thread*’ is a thread that receives and processes a user authentication request, from the web, a POP daemon, or an IMAP daemon. This thread puts the request in the ‘*auth list queue*’ and processes it by calling functions in the ‘*auth subsystem*’. After the request completes, this thread replies to the requesting MRA. An available ‘*operation thread*’ fetches an ‘*op-entry*’ out of the ‘*work-list queue*’ and examines whether the recipient is a valid user and where the recipient’s mailbox is located by calling functions in the ‘*auth subsystem*’. If the recipient’s mailbox exists on this node, the ‘*operation thread*’ stores the message into the recipient’s mailbox through function calls in the ‘*mailbox subsystem*’. At the same time, it stores the compact summary of the e-mail message in the ‘*message info subsystem*’. If the mailbox exists on a remote node, it sends the op-entry structure to the remote node where a ‘*node thread*’ receives and puts into the ‘*work-list queue*’. After completing the message saving, the correspondent ‘*operation thread*’ of the remote node acknowledges to the requesting node.

The number of available ‘*operation thread*’s is determined a priori considering the trade-offs between the parallel processing benefits and thread scheduling

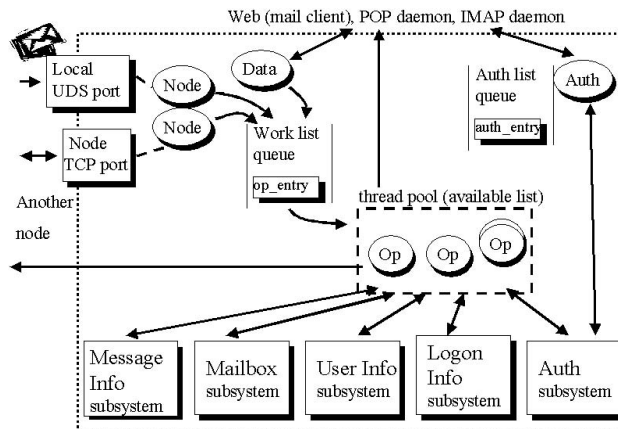


Fig. 5. The interface module of the proposed e-mail cluster, version 1. Interaction between four kinds of threads and five subsystems is depicted.

overheads. An ‘*operation thread*’ is assigned to each outstanding e-mail message or each web-service request. If there are multiple messages destined for the same recipient, there can be more than one outstanding ‘*operation thread*’s that try to access the same mailbox. To minimize the mailbox synchronization overhead, we allow only one ‘*operation thread*’ to be active for each user by locking mechanism. Extensive experiments reveal that separation of the ‘*node thread*’ and the ‘*operation thread*’ incurs non-negligible message copy overhead. In addition, too many ‘*operation thread*’s may degrade the performance due to the thread scheduling overhead. The second version of the interface module overcomes these drawbacks.

3.2 Interface Module, Version 2

Fig. 6 shows the structure of our improved interface module. To remove an additional message copy overhead, the local MDA sends the message to the remote node directly without intervention of the interface module of the local node. To make such decision, however, the MDA should inquire the interface module where the recipient’s mailbox is located. Therefore, the interface module of the proposed e-mail cluster version 2 has a new thread named ‘*location query thread*’ as shown in Fig. 6. With the information on the recipient’s ID, the ‘*location query thread*’ obtains the destination node using the ‘*auth subsystem*’. If the recipient’s mailbox exists on the local node, MDA transfer the message to a ‘*node thread*’ in the interface module. The ‘*node-thread*’ of the second version implementation takes the roles of both a ‘*node-thread*’ and an ‘*operation-thread*’ in the first version. We create as many ‘*node-thread*’s as the number of nodes in the cluster. In fact, a ‘*node-thread*’ is dedicated to each node including the local node. The ‘*node thread*’ dedicated to the local node receives a message by a UNIX domain socket while a ‘*node thread*’ assigned to a remote node, receives a message by a TCP socket. Such coalescing of two threads implies

that the messages or requests for a certain node are served in sequence in the second version of implementation. Therefore, we do not need the central ‘work-list queue’ in the second version. As a result, the second version greatly reduces the implementation complexity while improving the performance.

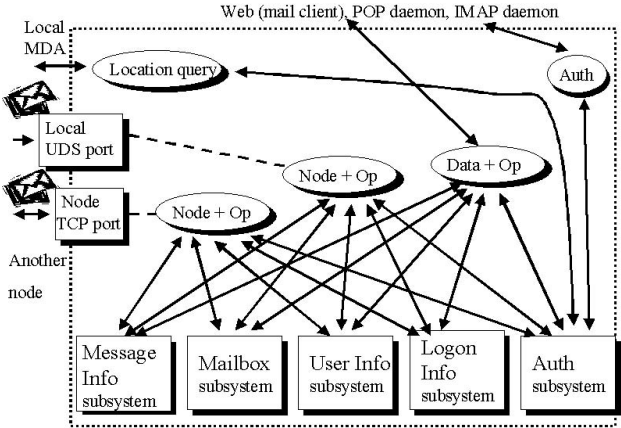


Fig. 6. The interface module of the proposed e-mail cluster, version 2. Interaction between four kinds of threads and five subsystems is depicted.

4 Implementation

The modular structure of the proposed system architecture allows one to change the system configuration easily by replacing a component with another. We have implemented three different configurations of the proposed system architecture. And, we have also implemented a simple e-mail cluster system based on the NFS mechanism, which is similar to the EarthLink system, for comparison purpose. We choose the EarthLink system for performance comparison because it is the only known e-mail cluster to us with an MTA-MDA structure. In this section, we explain some implementation details of those systems that are used for experiments in the next section. Four system configurations including the EarthLink configuration are summarized in Table 2.

We could easily implement an EarthLink system using off-the-shelf components such as ‘sendmail’ [12], ‘mail.local’, and the NFS. We used the NFS version 3 with the options of hard mount, asynchronous I/O, and 8KB read/write size. Notwithstanding our best efforts to replicate the described system in the paper, we admit that there may be discrepancy between the implemented one and the original one [5]. We use the ‘mail.local’ as the local MDA, and make slight modification. We replace the code calling ‘getpwnam()’, ‘getpwuid()’ functions with a code querying the authentication SQL DB. We use MySQL 3.23.41 as the authentication SQL DB. And, we increase the “max connection” value of MySQL

Table 2. Experiment configurations

Configuration	MTA	Interface Module
EarthLink	Sendmail	NFS
Version1 (Sendmail)	Sendmail	Version 1
Version2 (Sendmail)	Sendmail	Version 2
Version2 (Postfix)	Postfix	Version 2

daemon from 50 as default value to 500 to avoid “*too many connections*” error [13].

For the proposed system architecture, we have implemented three different configurations as listed in Table 2. For the systems using the ‘*sendmail*’, we had to adjust some configuration parameters in the ‘*sendmail.cf*’ file. First, we remove the “*w*” flag in the entry for the local delivery agent, which prevents the ‘*sendmail 8.11.1*’ from using the ‘*passwd*’ file for user authentication [5]. For as large as hundreds of thousands of users, linear search of the ‘*passwd*’ file takes prohibitively long. And we increase the values of ‘*QueueLA*’ and ‘*RefuseLA*’ from 8 and 12 as default values to 46 and 50 in the ‘*sendmail.cf*’ file. This makes the ‘*sendmail*’ receive SMTP requests at high load average value until saturated [14]. The default values make the ‘*sendmail*’ reject new incoming mails too early before saturated.

For the last configuration with the ‘*postfix*’, we set the ‘*mailbox_command*’ parameter to ‘*/usr/bin/mail.local "\$USER"*’ in the ‘*main.cf*’ file. This makes ‘*postfix 1.1.11*’ call ‘*mail.local*’ to deliver received message. Next, we replace the code calling ‘*mypwnam()*’ function with a code passing recipient’s ID to the ‘*mail.local*’.

For load balancing of the cluster nodes, the DNS round-robin mechanism is used to determine which cluster node receives a new SMTP request. And user mailboxes are distributed randomly and uniformly across the nodes. In each node, user mailboxes are grouped and stored in directories whose locations are defined as “*/home/node#{a serial number of a user ID mod 300}/user ID*”: for example, if a user ID is “*s0003123*” and his mailbox exists on node-0, the mailbox location becomes “*/home/00/123/s0003123/mbox*”. Such grouping reduces the time to find and access a user mailbox from a given user ID [5].

5 Experiments

We compare the four system configurations of Table 2 in terms of the message throughput, the message latency, cluster scalability, and availability. Even though SPECmail2001 is recently released as an industrial standard benchmark [15], it is not suitable to measure the raw performance of the system such as the peak message throughput of the system. Therefore, we created our own experimental method.

The experimental environment consists of a test cluster, workload generators, a DNS server, and a SQL DB server as shown in Fig. 7. Our test cluster has four nodes of Red Hat 7.3 Linux 2.4.18-3 kernel with the following hardware attributes: 550MHz Pentium III CPU, UDMA66 40GB IDE disk, 512MB main memory, two 100Mb Ethernet interface cards, and ext3 file system. We need at least as many as the workload generators as the number of cluster nodes to generate e-mail messages enough to saturate the cluster. We use four Linux machines of a similar kind but with 600MHz Pentium III CPU as the workload generators. The DNS server is a Red Hat 6.2 Linux 2.2.14-5.0 server machine with 166MHz Pentium MMX CPU, 2GB IDE disk, and 128MB main memory. The SQL DB server is a Red Hat 7.2 Linux 2.4.7-10 machine with 1GHz Pentium III CPU, 40GB IDE disk, and 256MB main memory. The test cluster nodes and a SQL DB server are interconnected via a switched 100Mb Ethernet network. The cluster, workload generators, and the DNS server are connected by another switched 100Mb Ethernet network. The version of BIND is 8.2.2-P5. Each node has the mailboxes of 50,000 users. We use a constant message size of 8KB for a workload to evaluate the system. This size is known to be the mean or the median value on the workload characterization of mail servers [16].

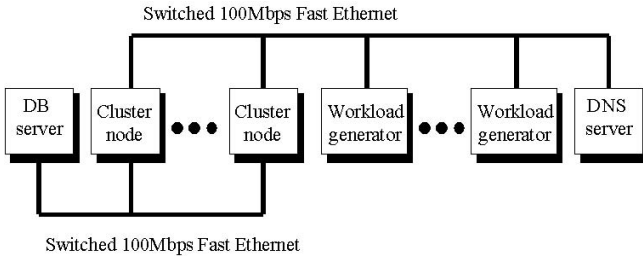


Fig. 7. Experimental environment that includes the e-mail cluster system and workload generators

5.1 Message Throughput

We define the message throughput of an e-mail cluster system as the number of messages that the system can process maximally in a second. The workload generators generate a certain number of e-mail messages in 60 seconds at a constant rate. And the total elapsed time for the system to process all messages is measured. Then, the number of messages divided by the measured time becomes the average number of messages, which a system processes for a second. For example, if we send 1200 messages to a system in 60 seconds and the system takes 180 seconds to process all the messages, the average number of messages processed for a second becomes $1200/180=6.7$ (messages/second). Increasing the number of generated messages by 60 messages (one message per second), we

repeat these experiments until finding the maximum average number of messages processed for a second. Then, the value is regarded as the message throughput of the system. We used the mean value of 3 sets of experiments for the same number of messages in 60 seconds.

Table 3 presents the message throughput of four e-mail system configurations we have implemented. All configurations, except for ‘*Version2 (postfix)*’, scale well up to 4-node cases. For a single node case, both version 1 system and version 2 system have a slightly higher throughput than EarthLink system. For 2 and 4 nodes, ‘*Version2 (Sendmail)*’ has the highest performance among all four system configurations even though the difference is not significant.

Table 3. Message throughput of e-mail system configurations (messages/second)

Configuration	1-node	2-nodes	4-nodes
EarthLink	7.8	16.9	30.9
Version1 (Sendmail)	9.0	17.7	33.6
Version2 (Sendmail)	9.0	19.4	35.0
Version2 (Postfix)	10.8	21.4	30.1

Also, we have set up another experimental environment to examine the scalability with a larger cluster system before implementing the second version of the proposed system with ‘*sendmail*’. The cluster consists of 16 nodes of Red Hat 7.2 Linux 2.4.7-10 kernel with 1.7GHz Pentium4 CPU, UDMA66, 40GB IDE disk, 256MB main memory and 2 100Mb Ethernet interface cards. We compared the scalability of the proposed system version 1 and the EarthLink system and show the result in Table 4. Both systems are fairly scalable up to 16 nodes while the version 1 system degrades its performance a little bit more. In short, the proposed system and the EarthLink system both possess good scalability.

Table 4. Message throughput of the EarthLink system and the proposed sytem, version 1 (messages/second)

Configuration	1-node	2-nodes	4-nodes	8-nodes	16-nodes
EarthLink	9.8	19.9	37.6	80.4	178.1
Version1 (Sendmail)	11.5	19.2	42.5	79.5	175.0

5.2 Message Latency

We define the message latency of a system as a time interval from receiving a SMTP request to storing the message on the recipient’s mailbox. We compute

the average value out of 100 experimental results excluding both upper 10% results and lower 10% results to compensate the run-time variances of the system behavior.

Fig. 8 and Fig. 9 show the message latencies that are divided into sections for a single node and for two nodes clusters respectively. The ‘*MTA*’ section (*MTA*) means a time interval for MTA to process a message. The ‘*setup*’ section (*setup*) means a time interval for MDA to prepare store the received message. The ‘*store*’ section (*store*) means a time interval for MDA to store the received message into temporary file. The ‘*transfer*’ section (*transfer*) means a time interval for the interface to complete the message delivery process.

The message latency of each system is nearly constant except for the ‘*transfer*’ section, independently of the cluster size. The version 1 system has longer latency than the version 2 system because it adds additional message copy and thread switch overhead for message storage as explained in the previous section. For remote delivery case, the performance degradation of the version 1 system becomes significant. On the other hand, the version 2 system with ‘*postfix*’ has the shortest latency among all four systems.

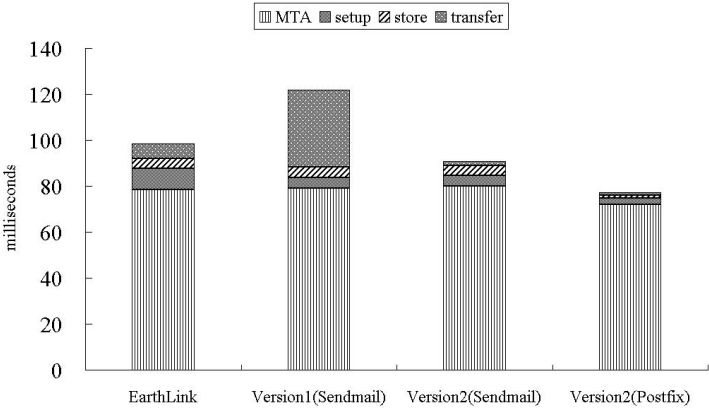


Fig. 8. The message latencies of single-node e-mail cluster systems.

5.3 Availability

To test the availability, we disconnect a cable from a node or turn the power off a node suddenly at run time. Then, we examine whether the system operates properly, until all messages are received. The EarthLink system has not been able to accept any SMTP connection in a few seconds after a fault occurs. However, ‘*version 2*’ system survives by making a local failure cause a local, not global outage.

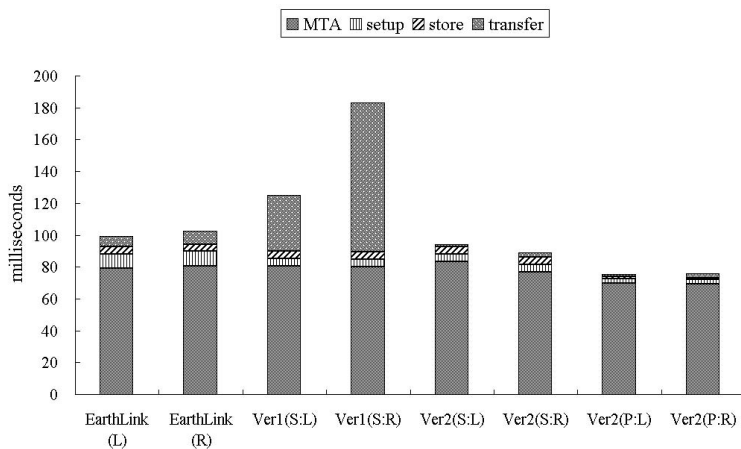


Fig. 9. The message latency of two-node e-mail cluster systems. Local delivery (*L*) means that recipient's mailbox exists on local node. Remote delivery (*R*) means that recipient's mailbox exists on other node (*L*: local delivery, *R*: remote delivery, *S*: send-mail, *P*: postfix, *Ver1*: version1, *Ver2*: version2)

6 Conclusion

We have presented a novel architecture for cluster-based e-mail systems with the MTA-MDA structure to achieve the goals of high scalability and availability, and the low development and maintenance cost. To demonstrate the flexibility and the extensibility of the proposed architecture, we implemented four systems with the MTA-MDA structure. Experimental results show that all systems are scalable in the sense the peak message throughput. Preliminary experiments show that one of our implementations (*version 2*) makes a local failure cause a local, not global outage.

Although we use our own experimental method to size a system, standardized benchmark is necessary to compare mail systems formally. Then, we plan to evaluate our system using the new SPECmail benchmark. Because the benchmark uses POP and IMAP, we extend the functionalities of 'version 2' system for POP and IMAP services.

Acknowledgement. This work was supported by National Research Laboratory Program (number M1-0104-00-0015) and Brain Korea 21 Project. The RIACT at Seoul National University provided research facilities for this study.

References

1. Microsoft.: MSN Hotmail Tops 100 Million User Milestone, REDMOND, Washington, 2001.

2. J.Robert von Behren, Steven Czerwinski, Anthony D. Joseph, Eric A. Brewer, and Jonh Kubiawicz.: NinjaMail: the Design of a High-Performance Clustered, Distributed E-mail System, In Proceeding of International Workshops on Parallel Processing 2000, Toronto, Canada, August 21–24, 2000, pp. 151–158.
3. David Wood.: Programming Internet Email, CA:O'Reilly & Associates, Inc., Sebastopol, 1999.
4. Postel, Jonathan.: RFC 821: Simple Mail Transfer Protocol, 1982.
5. Nick Christenson, Tim Bosserman, David Beckemeyer, EarthLink Network, Inc.: A Highly Scalable Electronic Mail Service Using Open Systems, Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Dec 1997.
6. J. Myers and M. Rose.: RFC 1939: Post Office Protocol – Version 3, May 1996.
7. M. Crispin.: RFC 2060: Internet Message Access Protocol – Version 4rev1, Dec 1996.
8. Yasushi Saito, Brian N. Bershad, and Henry M. Levy.: Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service, 17th ACM symposium on Operating System Review, Dec 1999, 34(5) pp. 1–15.
9. Michael Grubb.: How to Get There From Here: Scaling the Enterprise – Wide Mail Infrastructure, In the proceedings of the Tenth USENIX Systems Administration Conference (LISA '96), Chicago, IL, 1996, pp.131–138.
10. U.C. Berkeley.: Ninja project, <http://ninja.cs.berkeley.edu>
11. U.C. Berkeley.: OceanStore project, <http://oceanstore.cs.berkeley.edu>
12. Eric Allman.: SENDMAIL – An Internetwork Mail Router, BSD UNIX documentation Set , University of California, Berkeley, CA, 1986.
13. Randy Jay Yarger, George Reese, Tim King.: MySQL & mSQL, CA: O'Reilly & Associates, Inc., Sebastopol, 1999.
14. Bryan Costales with Eric Allman.: sendmail ; Second Edition, CA:O'Reilly & Associates, Inc., Sebastopol, 1997.
15. SPECmail2001.: <http://www.spec.org/osg/mail2001>
16. Laura Bertolotti – Maria Carla Calzarossa.: WORKLOAD CHARACTERIZATION OF MAIL SERVERS , In the proceedings of SPECT'2000, Vancouver, Canada, July 16–20, 2000.

The Presentation of Information in mpC Workshop Parallel Debugger

A. Kalinov, K. Karganov, V. Khatzkevich, K. Khorenko, I. Ledovskikh,
D. Morozov, and S. Savchenko

Institute for System Programming, Russian Academy of Sciences, 25
B.Kommunisticheskaya str., Moscow 109004, Russia
{ka, kostik, khvapist, kostja, il, dmor, stasaha}@ispras.ru

Abstract. The paper presents the mpC Workshop parallel debugger and its approach to data presentation. The debugger interface concepts are described that make the interface simple and easy to use, offering a convenient tool for parallel programming.

1 Introduction

The debugger is a key component of any development environment. The availability of a high-quality debugger is especially actual for parallel programming because of its inherent complexity.

In general, there are two types of debugging operations: managing the program execution and examining and controlling the program state. In the case of parallel program the amount of information to be controlled and displayed grows enormously and debugger should provide certain means for filtering the displayed information and space-efficient data presentation. This means are the most important issues of parallel debugger user interface and it greatly influences the simplicity and convenience of the debugger use.

This paper presents the new way of parallel program data presentation that is used in mpC Workshop parallel debugger. It is organized as follows. Section 2 describes the main problems concerning data presentation in parallel debuggers, section 3 overviews the approaches used in most popular parallel debuggers. In section 4 the mpC Workshop parallel debugger is presented and section 5 summarizes the results and draws the conclusion.

2 The Problems of Data Presentation in Parallel Debuggers

A debugger is a tool for exploring the state of the program being developed, its internal data and behavior. Even for sequential program the full program state includes data and stack segments, registers, opened files and so on — an enormous amount of data, that is impossible to understand as a whole. The goal of debugger user interface is to restrict the displayed information to a reasonable amount and to provide convenient means for its manipulation.

For sequential debugging the user interface design concepts are well-known and common, as ones of Microsoft Visual Studio or Borland products. For parallel debuggers it is not so because of the growing gap between the desirable interface simplicity and the internal complexity of the parallel program. If the parallel program consists of N processes, then the amount of information is more than N times larger, if compared to sequential case. Assuming that N could be large it is necessary that the interface should be scalable. To simplify the usage of the debugger an easy way for controlling the debug process and displaying data should be provided. It should be flexible enough to restrict the superfluous data from being displayed and should provide rapid focusing on data, that are necessary at the moment.

Since it is very difficult to fit all these requirements simultaneously, many debuggers suggest their own interface design concepts, but still there is no common and universally recognized approach.

3 Existing Approaches

As an example of existing approaches we will review three powerful debuggers — TotalView, Prism and p2d2, and describe the main concepts of data presentation each of these debuggers use.

One of the most powerful parallel debuggers is *TotalView* [4] from Etnus. It is designed to be universal debugging tool and allows to develop any kind of parallel program in several languages. It makes no assumptions on program source structure and thus it has to show each process in separate window. This approach makes possible to display the maximum details on each process, but complicates the process control and does not allow quick focusing on necessary data thus providing rather complicated interface.

As for information specific for parallel program, TotalView has a tool for displaying a call-tree (the extension of a call stack notion for the parallel program) and message queue graphs.

Another well-known parallel debugger is *Prism* [6,7] from Sun Microsystems. It was designed to be scalable as much as possible and to provide the interface, similar to sequential debuggers. It has a powerful command-line control language together with GUI and diverse visualization tools.

Prism introduces the notion of *pset* that is a group of processes that can be treated as a single entity. User can define custom psets and work with them as with single processes, applying to them control and display commands. Pset membership can be evaluated dynamically during the debugging session, that gives very flexible and powerful way for controlling large process sets as single processes.

Prism debugger has advanced tools for program call-tree display and array visualization. Along with several graphical representation modes it allows pset-based filtering the displayed components of distributed values. Its main drawback, that for the historical reasons most functions of the debugger are available

only from the command line. It gives certain flexibility, but drastically hardens the debugging control.

The intermediate approach is implemented in *p2d2* [5] debugger from NASA. It has three levels of abstraction of the program state information — all processes, focus group and focus process and displays all levels simultaneously. This approach allows to see the program state at a glance, does not overload user with unneeded information and provides easy switching to necessary data.

At the top level of abstraction *p2d2* shows the grid of all processes that provides simple visual representation of each process state and allows quick selection of necessary processes or groups. Customized process icons help to see which processes are in certain state and need to be inspected more carefully.

Focus group specifies a set of processes which states will be displayed in more detail. Each process of the focus group has a single-line description of its state that can include process identifier, the name of the host process is running at and current location in source file.

The most detailed information is displayed on focus process. The main window shows the program source code and the current process call stack. To simplify the comparison of components of distributed values another process can be defined as second focus process. This allows to see two components of the distributed value (of the primary and secondary focus processes) at the same time. Such approach provides average scalability, but very fast focusing on necessary data and simple user interface.

4 Approach Used in mpC Workshop Debugger

When designing the mpC Workshop debugger several objectives were taken into consideration. Debugger interface should be:

- as close to sequential as possible,
- aimed to debug only mpC programs,
- scalable,
- easy to use.

This has lead us to the design that is shown at the Fig. 1.

The main IDE window contains subwindows for displaying project files, program variables, debug cursors and other useful information. The main window shows the source code of the program being debugged and displays execution control elements, such as breakpoints or watchpoints.

To simplify the display of current positions and control over the program execution the notion of debug cursors is introduced. Debug cursor corresponds to the set of processes that stay at the same position of source file and have the same color, representing process state. For better understanding the process state a traffic light style is chosen. If some cursor is red it means that all its processes are blocked at a communication point or synchronization barrier. Such cursor can not perform any actions and report its state. If the cursor is green it is ready to execute commands. And if the cursor is colored with amber it means that it

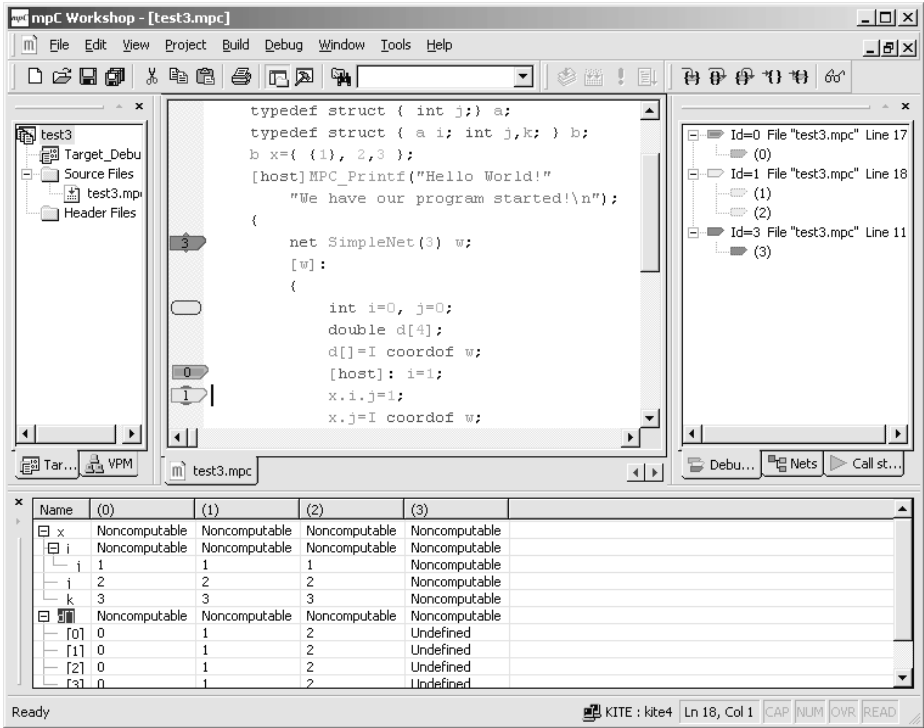


Fig. 1. The main window of mpC Workshop debugger

is held down by user and should not move during step commands. Processes are grouped to cursors automatically that provides a scalable way of controlling the program execution.

At the Fig. 1 you can see three cursors with identifiers 0, 1 and 3 at the source code lines 17, 18 and 11 respectively. Debug cursors window at the right shows all cursors and the processes, they contain. It also supports many cursor control actions such as splitting, joining or recoloring cursors.

At the bottom of the debugger window you can see the watch window that displays values of variables. Each line represents the scalar component of the value and each column is related to some process of parallel program. For example, on Fig. 1 two displays are set on values of structure *x* and array *d*. The process with the identifier 3 is blocked at the synchronization point and thus can not report the values.

Another control window displays mpC specific information about current structure of computing space. It displays the virtual processors available and the tree of networks and subnetworks already created.

To provide the scalability and simplify the usage of the mpC Workshop debugger a new concept was introduced — the concept of network filters. The network filter helps to control the display of superfluous information - it de-

scribes which components should be displayed in a window and which should not. You can set the filter by selecting the processes you are interested in or by selecting the networks created in your program. The use of internal program information, such as a network structure, gives a great advantage to the user interface, raising its flexibility and usability. The situation when the program performs some computations on a certain network is common in mpC and there is no sense to display all components of distributed values since only the components that belong to the network have a meaningful information. In this case, the user can just click on the current network in a network filter setup dialog and apply this filter to the display window.

On the Fig. 2 the main ideas are shown by the example of an active watchpoint. During the program execution the value of the variable i had changed and the watchpoint in line 14 of the source code became active. "Break info" pop-up window appeared and the watchpoint details dialog that allows to see the distributed value of the watchpoint expression at several nodes was invoked. Then the network filter setup dialog was opened. At the Fig. 1 you can see that the host process (number 0) are to execute the statement `[host]: i=1;` and at the "Break info details" window you can see that the 0-th component of i has been really changed to 1.

Figure 2 also shows the network filter configuration dialog opened. By default the "Break info details" window displays only the values that have been changed, but here the filter was set to display the components related to processes 0 and 2. As a result, the details window displays only two components of i and only two processes.

Owing to such interface organization the mpC Workshop debugger fits all the requirements mentioned above that makes it a useful tool for parallel program development. It uses the compromise approach between showing maximum available information and difficult configuring the presentation of exactly what is needed. Debug cursors and network filters introduced in mpC Workshop debugger provide a clear and powerful way of controlling the program execution and displaying the program state information. They ensure the interface scalability and let to configure data presentation in a couple of mouse clicks.

The mpC Workshop debugger does not allow to control the execution threads or to examine the registers because they are not the objects an mpC program works with. It uses the high-level language abstractions and displays the information in the same terms that the programmer uses to write the program. It makes the debugging much simpler for the programmer.

5 Conclusions

Because of the high complexity of parallel programming, the design of a good parallel debugger is a challenging task. The developers of parallel debuggers face a lot of problems when designing the user interface concepts. Several parallel debuggers do exist but still there is no one convenient and simple enough to become a wide-spread or de-facto standard. The mpC Workshop parallel debugger

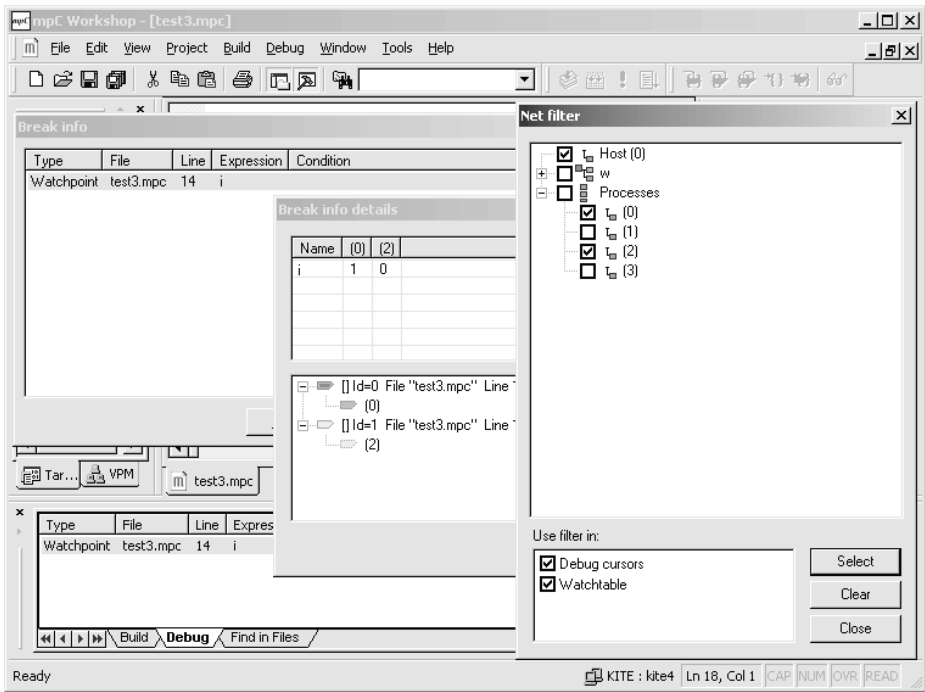


Fig. 2. Watchpoint details network filter

is trying to reach a golden mean of existing approaches combining the convenient interface with powerful data display capabilities.

References

1. Lastovetsky A., Arapov D., Kalinov A., Ledovskih I.: A parallel language and its programming system for heterogeneous networks. *Concurrency: practice and experience* (2000)
2. Kalinov A., Ledovskikh I.: The mpC parallel debugger. *Proc. of PDPTA'2001, CSREA Press*, (2001)
3. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, version 1.1 (1995)
4. Etnus: TotalView Users Guide. Version 5.0 (2001)
5. Robert Hood: The p2d2 Project: Building a Portable Distributed Debugger. *Proc. of SPDT'96* (1996)
6. Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simmons, Rich Title: A Scalable Debugger for Massively Parallel Message Passing Programs. *Debugging and performance tuning for parallel computing systems*, IEEE press (1996)
7. Sun Microsystems: Prism 6.2 User's Guide (2001)

Grid-Based Parallel and Distributed Simulation Environment^{*}

Chang-Hoon Kim, Tae-Dong Lee, Sun-Chul Hwang, and Chang-Sung Jeong

Department of Electronics Engineering Graduate School, Korea University
5-ka, Anam-dong, Sungbuk-ku, Seoul, Korea

{manipp, lyadlove}@snoopy.korea.ac.kr, csjeong@charlie.korea.ac.kr

Abstract. Although parallel and distributed computing for a large-scale simulation has many advantages in speed and efficiency, it is difficult for parallel and distributed application to achieve its expected performance, because of some obstacles such as deficient computing powers, weakness in fault and security problem. Motivated by these concerns, we present a Grid-based Parallel and Distributed Simulation environment(GPDS) which not only addresses the problems but also supports transparency and scalability using Grid technologies. GPDS supports a 3-tier architecture which consists of clients at front end, interaction server at the middle, and a network of computing resources at back-end. Grid and simulation agents in the interaction server enables client to transparently perform a large-scale object-oriented simulation by automatically distributing the relevant simulation objects among the computing resources while supporting scalability and fault tolerance by load balancing and dynamic migration schemes.

1 Introduction

Simulations are playing an increasingly critical role in all areas of science and engineering. As the complexity and computational cost of these simulations grows, it has become important for scientists and engineers to achieve its complex computation more rapidly and efficiently[1]. Although the parallel and distributed computing for a large-scale simulation has many advantages in speed and efficiency, it is difficult for parallel and distributed application to achieve its expected performance due to several inherent problems : First, the supply of resources for large-scale computation needs many efforts. In recent years, as the importance of scientific computing grows steadily, the scale of simulation is dramatically on the increase. The expansion of the scale essentially requires enormous computing power. The bigger the required computing powers are, the harder gathering available resource is. Moreover, the use of resources is closely related to security problem. Second, the robustness of the whole system can be

^{*} This work has been supported by KOSEF in 2003, KIPA-Information Technology Research Center, University research program by Ministry of Information & Communication, and Brain Korea 21 projects in 2003

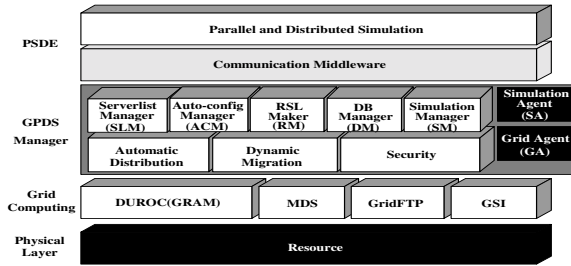


Fig. 1. The Layer Architecture of the GPDS

easily destroyed by the fault of a system. Since parallel and distributed simulation(PADS) is based on the frequent and accurate interactions among distributed entities, the failure of one host may cause an interruption of data communication, resulting in the halt of the entire simulation. In this paper, we present a Grid-based Parallel and Distributed Simulation environment(GPDS) which not only addresses the problems but also supports transparency, performance and scalability using grid technologies.

Advances in high speed network and computing power make it possible to construct a large scale high performance distributed computing environments, called a grid, which uses a network of computer as a single unified computing resource[1]. GPDS achieves the design goal of transparency, scalability, performance, and fault tolerance by integrating a parallel and distributed simulation environment(PDSE) onto grid computing environment(GCE). PDSE can be considered as a networked virtual environment(NVE) which uses a simulation-specific middleware for executing distributed simulation, while GCE supports a common set of services and capabilities that are deployed across resources. GPDS makes remarkable improvements over the existing PDSE by exploiting GCE.

The outline of our paper is as follows: In section 2, we illustrate the architecture of GPDS, and in section 3, describe the detailed services of GPDS. In section 4, we give a conclusion.

2 GPDS Architecture

GPDS consists of several layers: physical layer, grid computing layer(GCL), GPDS manager layer(GML), PSDE layer(PL) as in figure 1. GCL is composed of various services as offered in Globus toolkit to manage a set of resources in physical layer. GCL comprises four modules: DUROC and GRAM which allocate and execute the job in the remote hosts, MDS(Meta Directory Service) which provides information services, GridFTP which is used to access and transfer files, and GSI(Grid Security Infrastructure) which enables the authentication via single sign-on using a proxy[2]. PL consists of parallel and distributed simulation application and simulation-specific middleware. A parallel and distributed

simulation is performed by making simulation objects interact with each other through the simulation-specific middleware which provides services such as communication between simulation objects, interest management, data filtering, and time management required to achieve stable and efficient simulation.

GPDS manager layer is an intermediate layer between PL and GCL which is in the charge of a bridge between them in order to support PDSE over GCE. It is composed of Grid Agent(GA) and Simulation Agent(SA) which allow PSDE and GCE to harmoniously interact with each other. The layered architecture provides modularity and extensibility by each layer interacting with each other using the uniform interfaces. As shown in figure 1, GA supports automatic distribution, dynamic migration, and security services using the modules offered on GCL, which shall be explained in detail in section 3. Simulation Agent(SA) consists of five modules : Serverlist Manager(SLM), RSL Manager(RM), Auto-configuration Manager(ACM), Simulation Manager(SM) and DB Manager(DM). SLM makes the list of resources available in the corresponding virtual organization(VO)[4]. The number and performance of available hosts have a great effect on the configuration of PADS(Parallel and Distributed Simulation). SLM periodically updates the severlist of available resources which are referenced by other managers RM, ACM, SM. RM dynamically creates a RSL code for allocating resources to meet the status of simulation and the requirements of GA. ACM automatically makes the configuration files to provide information needed to initiate the PSDE according to RSL code. SM has three missions: First, it establishes a connection to the client, receives the commands from the client, and returns the simulation results to the client. Second, it periodically receives and monitors simulation data from a simulation process in SA, and stores them in DB by delivering to DM so that simulation data of each host shall be used in the recovery of the fault. Third, it enables automatic and dynamic features by interacting with GA via the frequent exchange of the necessary information between them.

3 GPDS Service

In this section, we shall describe about the services offered by GPDS Manager to meet the design goals of GPDS : transparency, scalability, fault tolerance, and performance.

3.1 Automatic Distribution Service

Automatic distribution service enables the automatic execution of PADS by allocating computing resources, transferring the executable files and carrying out them on the allocated computing resources. The automatic distribution allows the transparent use of computing resources, and the dynamic configuration used in resource allocation for load balancing enables the better utilization of computing resources with the enhancement of scalability. The service is composed of three steps as follows:

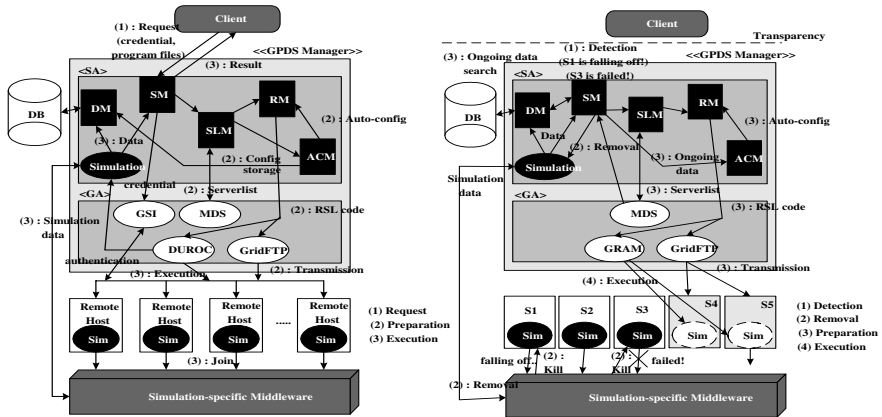


Fig. 2. (a)Automatic Distribution Service (b) Dynamic Migration Service

Request: A client sends a connection request to GPDS manager in interaction server, and SA in GPDS manager establishes a connection with the client. Then, the client submits a job to SA with information on the executable file and input file including a list of simulation objects.

Preparation: This step prepares for creating simulation processes in remote servers at back-end. It consists of four stages: server list production, configuration, storage, and transmission. In serverlist production stage, SLM in SA creates and maintains a server list about the available resources by making use of metadata on hosts which are registered using GIIS(Grid Index Information Service) of MDS in GCL[4]. In configuration stage, ACM automatically creates a configuration file including all the information required to initialize each simulation process in remote servers and to evenly distribute the simulation objects according to the number of available remote servers. RM automatically generates a RSL (Resource Specification Language) for resource allocation. In storage stage, the configuration file for each remote server is saved in DB by DB Manager(DM) for later use in dynamic migration, and in transmission stage, the program files and configuration file are sent to the corresponding remote servers through GridFTP service by GA[5].

Execution: GA simultaneously activates simulation processes on the allocated remote servers as indicated in RSL code through DUROC service in Globus toolkit. DUROC also provides the barrier which guarantees that all remote servers start successfully. Simulation processes in remote servers are initiated by referencing the configuration file, and they interact with each other and periodically delivers its own data to the simulation process in SA through simulation-specific middleware. The simulation process in SA stores the simulation data collected from remote servers in DB through DM, or return to the client through SM. Fig.2(a) illustrates each step of automatic distribution service.

3.2 Dynamic Migration Service

Dynamic migration service allows the simulation process on one host to be transferred to another host. It can achieve two design goals of GPDS : fault tolerance and performance improvement by transferring the process in the failed server or the server with poor performance to the new server with the better one. Dynamic migration service has four steps as follows :

Detection: Simulation Manager(SM) of SA detects the fault of remote servers by time mechanism, or finds out the remote servers with the degrading performance based on the information obtained by regularly retrieving the current status of remote servers using GIIS of MDS. Figure 2(b) shows remote servers S1, S2, and S3 allocated at the preparation step of automatic distribution service. Suppose that SM perceives the fault of S3 or the performance degradation of S1 through GA, and S4 or S5 are remote servers which have not been used yet but with better performance than S1.

Removal: At this step, the simulation process in the failed server or bad server is removed to keep the whole system from being halted or degraded. The details of this step are omitted, since it is dependent on simulation-specific middleware.

Preparation: GPDS Manager prepares the creation of a new simulation process in remote server. This step is similar to the preparation step in automatic distribution service, but that ACM retrieves the simulation data of the transferred simulation processes from DB through DM, and makes the configuration files for the transferred simulation processes, and transmits them to the allocated servers through GridFTP by GA so that the new simulation process has the same configuration as the old one.

Execution: This step is executed in the same manner as the one in automatic distribution service. A new simulation process succeeds the job of the previous server by referencing the information in the configuration file. Figure 2(b) shows the execution of the new simulation processes in S4 and S5 by GA.

3.3 Security Service

The importance of security issues is rapidly increasing in the last tens of years. Especially, the security problem has constricted the use of required resources in other organizations. GPDS Manager addresses this issue by using GSI(Grid Security Infrastructure) service of Globus toolkit[3]. GSI service provides single sign-on and delegation. Single sign-on allows a user to authenticate once and have access to grid resources without further user intervention. GSI implements it by generating a user proxy, an entity that acts on behalf of the user. Because the user proxy accomplishes the authentication process for remote control instead of the user, the authentication process is transparent to the user, and single sign-on can satisfy the requirements of multiple authentication. Also, GSI uses the delegation mechanism. When the user delegates his own credentials to one site, the site can access another site without further intervention of the user. This delegation process provides convenient interface to the user. The client sends its

own credential to GA, which in turn performs single sign-on by generating a user proxy based on the credential, allowing the efficient use of any resources in the corresponding virtual organization.

4 Conclusion

In this paper, we have presented a new Grid-based Parallel and Distributed Simulation Environment, called GPDS which supports a large-scale parallel and distributed simulation, and have shown that the nice integration of PSDE onto GCE allows GPDS to achieve its design goals of transparency, scalability, performance, and fault tolerance.

The key element of GPDS is a GPDS manager composed of two agents: grid agent(GA) and simulation agent(SA). The cooperative work of SA and GA enables client to transparently perform a large-scale object-oriented simulation by automatically distributing the relevant simulation objects among the computing resources while supporting scalability, fault tolerance and performance improvement by load balancing and dynamic migration schemes. The automatic distribution provides dynamic configuration used in resource allocation, enabling the better utilization of computing resources with the enhancement of load balancing and scalability. Dynamic migration service achieves two design goals of GPDS, fault tolerance and performance improvement by transferring the process in the failed server or the server with poor performance to the new server with the better one.

References

1. I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, 15(3), 2001.
2. I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, "A Security Architecture for Computational Grids," *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pp. 83–92, 1998.
3. I. Foster, and C. Kesselman, "The Globus Project: A Status Report," *Heterogeneous Computing Workshop*, pp. 4–18, 1998.
4. K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid Information Services for Distributed Resource Sharing," *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, IEEE Press, August 2001.
5. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, S. Tuecke, "GridFTP Protocol Specification," *GGF GridFTP Working Group Document*, September 2002.
6. K. Czajkowski, I. Foster, and C. Kesselman, "Resource Co-Allocation in Computational Grids," *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pp. 219–228, 1999.

Distributed Object-Oriented Web-Based Simulation^{*}

Tae-Dong Lee, Sun-Chul Hwang, Jin-Lip Jeong, and Chang-Sung Jeong

School of Electrical Engineering in Korea University 1-5ka, Anam-Dong,
Sungbuk-Ku, Seoul 136-701, Republic of Korea
{csjeong}@charlie.korea.ac.kr

Abstract. This paper presents the design and implementation of Distributed Object-oriented Web-based Simulation (DOWS). DOWS is an object-oriented simulation system based on a new concept, a director-actor model. It also models a distributed simulation as a collection of actor objects running concurrently at different nodes. DOWS is implemented using Java objects which interact with each other through Java RMI. Each director object is implemented as web client which downloads and executes the proper applet from a HTTP server, and hence DOWS provides a web-enabled simulation environment which allows users to easily instantiate the simulation model using the HTTP server. The whole simulation can be speed up by decomposing the simulation model into several submodels and mapping them onto the actors. DOWS also provides an efficient virtual real time simulation environment which integrates the coordination among actors by supporting time synchronization, simulation message transfer, and network fault detection.

1 Introduction

Simulation has been used in a variety of fields of science, engineering, military, business, and entertainment applications. Simulation of a large complex system requires intensive computation time, high development and maintenance costs. Recently, advances in hardware, networking, and software technology have provided the ability to consider more cost effective, interactive and distributed simulations.

Distributed simulation [1,2] attempts to reduce the time needed to perform a simulation by spreading its execution over multiple processes. Distributed simulation is of particular military interest because it offers a way to integrate large scale interactive training in real time or to simulate several tactical simulation models together. Even though distributed computing is a powerful tool in many high performance applications, it is particularly relevant when it can be developed easily, reconfigured easily, and have reusable components. Object oriented simulation provides this structure. With the advent of object-oriented programming like Java, more elegant methodology can be used for the specification and

^{*} This work has been supported by KOSEF in 2003, KIPA-Information Technology Research Center, and Brain Korea 21 projects in 2003

implementation of distributed simulation model on a network. Especially, an integration of the web and Java provides a new approach for simulation modeling which enables the dynamic widespread use of the common simulation model.

This paper presents the design and implementation of Distributed Object-oriented Web-based Simulation environment(DOWS). DOWS is an object-oriented simulation system based on a new concept, a director-actor model. It also models a distributed simulation as a collection of actor objects running concurrently at different nodes. DOWS is implemented using Java objects which interact with each other through Java RMI [3]. Each director object is implemented as web client which downloads and executes the proper applet from a HTTP server [4], and hence DOWS provides a web-enabled simulation environment which allows users to easily instantiate the simulation model using the HTTP server. The whole simulation can be speed up by decomposing the simulation model into several submodels and mapping them onto the actors. DOWS also provides an efficient virtual real time simulation environment which integrates the coordination among actors by supporting time synchronization, simulation message transfer, and network fault detection.

The outline of our paper is as follows: Section 2 presents the architecture of DOWS, and describe the function of each component. Section 3 describes the implementation of DOWS. Finally, in section 4, we give a conclusion.

2 System Architecture

2.1 System Model

DOWS is a distributed environment for simulating large and complex applications. It is based on director-actor model which can be mapped efficiently on object-oriented and distributed simulation. Object-oriented simulation allows fast and easy changes to the simulation application as well as the underlying structures, while distributed simulation can significantly reduce the time required for the applications. Director-actor model consists of actors and directors which interacts with others. Actor represents a simulation entity or a submodel in the simulation, and director is a participant in the simulation which controls the actors. Actors are partitioned into several subgroups, and each subgroup is associated with an unique director. Each director is connected to all the actors in its corresponding subgroup. Each director generates commands or events to its associated actors which in turn activate the actual simulations by interacting with other actors in the same or different subgroups. A set of actors may be designated as actor group so that each director can issue commands to each member in the actor group simultaneously. In director-actor model, simulation is carried out by actors interacting with each other. As we see in figure 1, director-actor model can be easily implemented in object-oriented and distributed simulation by mapping actor entities into objects and then assigning them into logical processes in a distributed environment.

Director-actor model has advantages of expressing various kinds of simulations in a simple and efficient way. It allows users to construct multiple simu-

lation models with the abstract actor object which provides common and basic external interfaces, and is easily extended to meet the requirements of a specific simulation model. Each actor can represent a simulation entity participating in one simulation model or a submodel in the whole simulation, and each director provides graphic user interface to instantiate and execute the simulation model in batch or interactive mode.

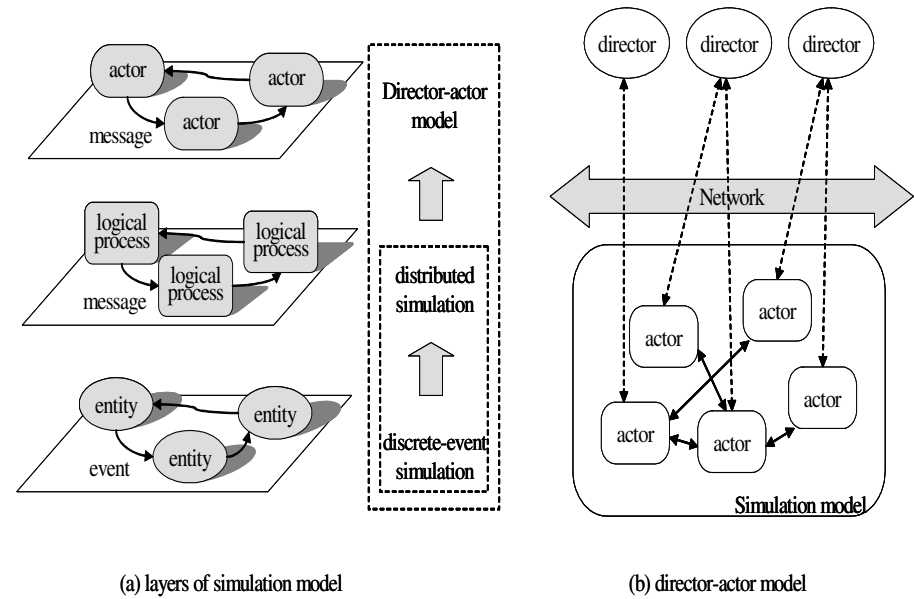


Fig. 1. Director-Actor Model

2.2 Architecture

DOWS consists of four major components which interacts with each other on a distributed environment: director, actor, agent and coordinator.

1) Director

Directors instantiate the simulation model and participate in the simulation concurrently through the interactive communication with the agents which are in charge of the parallel simulation of its associated submodels using several actors.

2) Actor

An actor is a basic execution unit for a simulation model. Each actor corresponds to a simulation entity of the simulation model or a submodel which

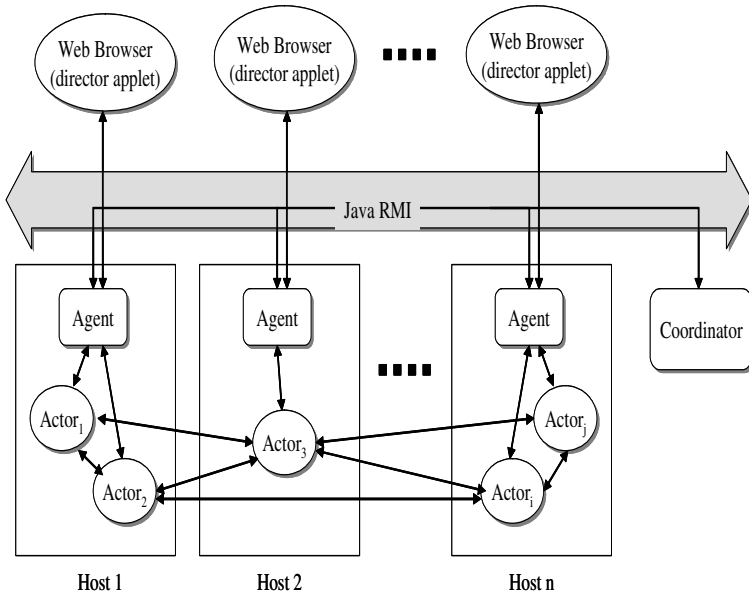


Fig. 2. DOWS architecture

is a part of the whole simulation model. Several actors can be executed either in one processor or several processors interconnected through a network on a distributed environment.

Each actor consists of simulation engine, channel, and simulation clock and local calendar of events. Figure 3 illustrates the architecture of an actor. Simulation engine carries out a simulation of its corresponding submodel using local calendar of events and local virtual simulation clock. In order to execute the whole simulation, actors interact with each other by remote method invocations. Each actor sends an event message to the channel of the other actor, and maintains the synchronization of the local simulation with the whole one by its own channel keeping track of messages sent to itself. Channel prevents a causality error by making use of conservative method.

3) Agent

The agent provides an efficient virtual real time simulation environment which integrates the coordination among directors and actors by supporting time synchronization, simulation message transfer, and network fault detection [5]. Since applets can neither create a server socket to accept any incoming connection nor connect other hosts except their originating hosts, each agent also plays a role of a message router, and coordinates each other to forward messages between actors and directors. Figure 4 illustrates various components of agent. Agent consists of *input thread*, *output thread*, *watch thread*, *time thread*, *input/output buffer*, *clients table*, *wall clock*, *blocking flag*. Input and output

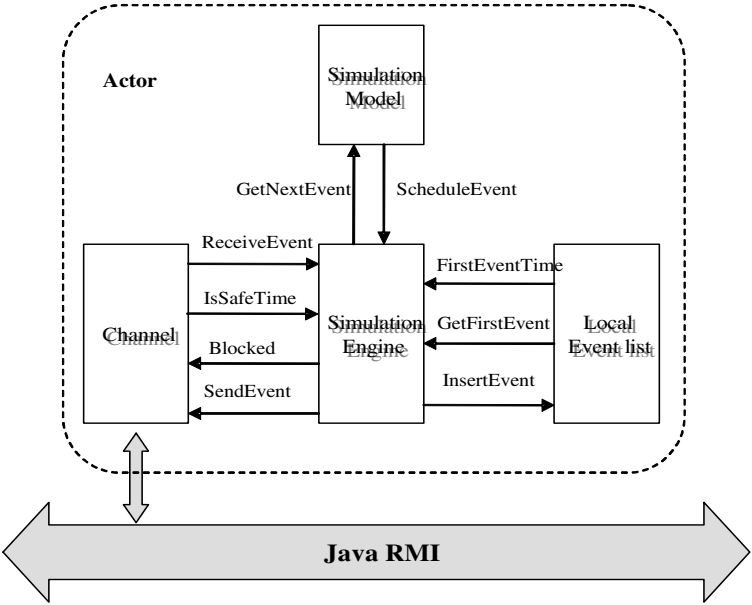


Fig. 3. Structure of Actor

threads manage data transfer between directors and actors. Input thread receives messages from director and transfers them to actors or other agents. Output thread multicasts messages from the actor to all the agents which in turn send the messages to their directors. Both of input and output threads transfers blocking signal to the actors and directors respectively.

Time thread is in charge of the correct advancement of virtual real time, called wall clock [6]. It can scale the wall clock for simulations slower or faster than real time, and synchronizes it with the simulation clock of the actors

Watch thread checks the state of director and network periodically. If it detects network faults or director program multifunctions, it generates a blocking signal to the agent, and transmit a warning message together with the source of errors to the coordinator.

4) Coordinator

The coordinator runs the whole simulation by sending start message to each actor, or suspend it by sending a block signal to the agents whose input thread in turn notifies its corresponding actors to block. As mentioned in the previous subsection, the simulation may be blocked by a watch thread in the agent, too. The coordinator can resume the simulation, and set up the initial conditions of the simulation such as duration, speed, resolution. Directors also can start, suspend, resume, ignore and stop the simulation through Java AWT using the coordinator.

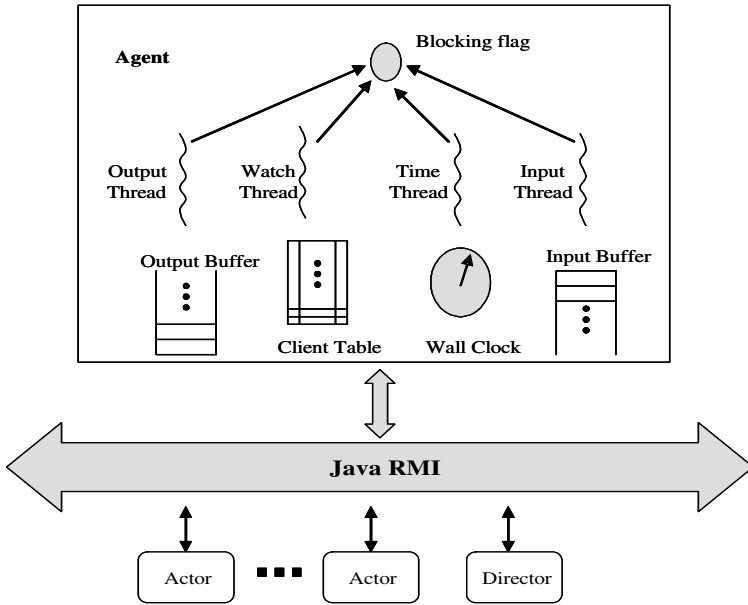


Fig. 4. Structure of Agent

3 Implementation

This section presents an object-oriented implementation of DOWS. Each component is implemented using Java. Java provides an object-oriented dynamic programming environment compared to a static, text-driven one prior to Java, and used to implement the applet for director interfaces on the web and the components in DOWS.

3.1 Web-Based Simulation Environment

Director is implemented by Java applets which can be posted to a Web site so that any director with a Java-enabled Web browser can run DOWS in accordance with to the paradigm "write once, run everywhere". Java applets for director provides a uniform and easy-to-use graphical user interface to enable the user to launch the simulation models of his interest in the host using CGI script. Figure 4 illustrates the web-based simulation model [7,8].

The process of the web-based simulation can be described more in detail as follows:

- An HTML page which represents a simulation model of interests is downloaded, and a Java applet for director embedded in the HTML page is retrieved from HTTP server and executed by web browser.

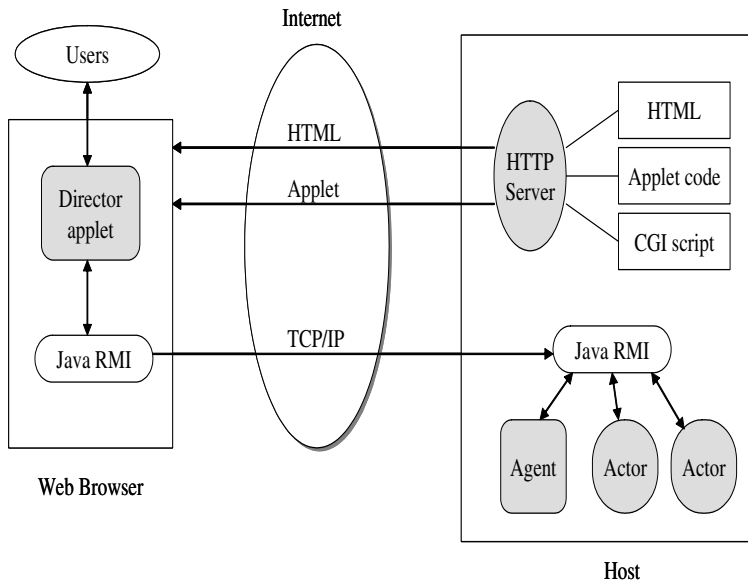


Fig. 5. Web-based Simulation

- Director execute the simulation model of interest by creating processes for its corresponding actors and agent. Directors generate and transfer event messages to the actors objects through its associated agent.
- Director can either execute the simulation model in batch or interactive mode. In batch mode, director issues the initiation command to the agent, and get the result after the simulation finishes. In interactive mode, director generates commands for event generation to the actors through agent during the simulation.

3.2 Java-Based Simulation

Each component in DOWS is implemented as Java object, and communicates with each other by RMI method across the network. RMI is implemented in an extremely transparent fashion such that the methods of remote objects can be invoked only through their references irrespective of whether they reside in local or remote machines.

DOWS supports concurrent programming using Java threads. Agent consists of several threads such as input, output, watch, time threads which are executed simultaneously. Each actor consists of simulation engine and server threads. While each simulation engine thread processes its associated events sequentially, simulation engine threads residing in multiple actors are executed simultaneously for the concurrent event processing in the overall simulation. Actors are created as threads, and interact with each other by exchanging objects for event messages. Serialization in Java enables the easy transfer of complex

local objects between actors transparently by supporting automatic marshalling and unmarshalling [9].

The real-world system is modeled as a collection of discrete-event processes called physical processes(PPs). The state of PP is changed at discrete points of time by exchanging event messages with other PP. The simulation of the real-world system is implemented by mapping each actor onto PP. Each actor comprises application-specific states and behaviors which describe a submodel of the entire simulation, and maintains its own local simulation clock and internal event list. Physical processes in the real world system are simulated by a collection of actors which exchange event messages with each other.

3.3 Class Libraries

Director is inherited from *UnicastRemoteObject*, and also includes functions as remote server to deal with callbacks from agent objects.

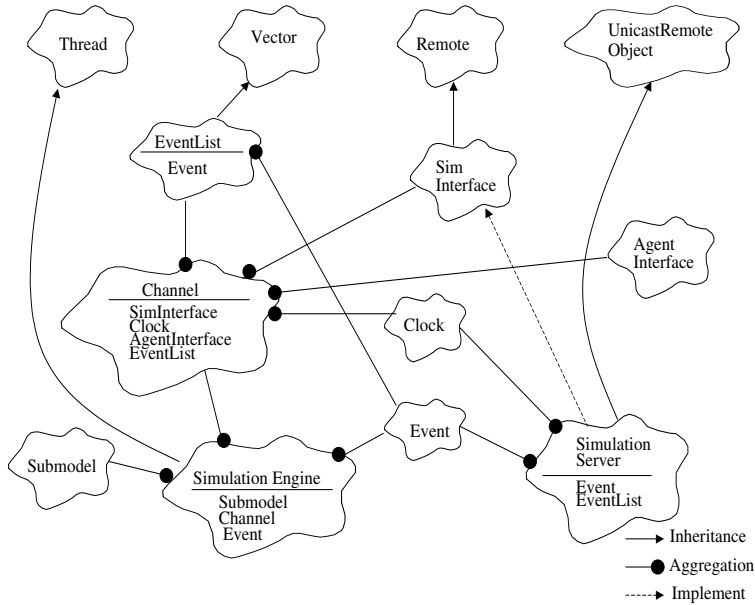


Fig. 6. Class diagram of Actor

Figure 6 illustrates the class diagram of an actor for simulation engine, clock, eventlist used in the distributed simulation algorithm, and classes for Java RMI. *SimEngine* and *EventList* classes are inherited from Java *Thread* and *Vector*, respectively. *ChannelInterface* and *AgentInterface* are inherited from *Remote* interface. *Channel* is inherited from *RemoteUnicastObject*. It sets up a connection to other channels in the different actor as well as to an agent.

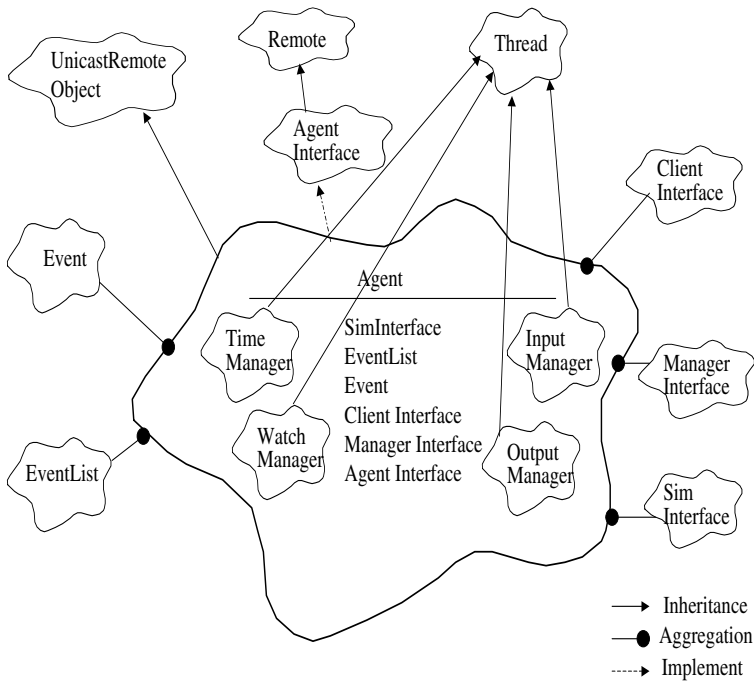


Fig. 7. Class diagram of Agent

Figure 7 illustrates the class diagram of an agent. *Agent* class represents a RMI server, inherited from *RemoteUnicastObject*. It has four nested classes: *TimeManager*, *WatchManager*, *InputManager*, *OutputManager* classes. The nested classes support time management, data management and network management. *EventList* is used to receive messages from directors or actors.

4 Conclusion

This paper has presented the design and implementation of Distributed Object-Oriented Simulation Environment(DOWS). DOWS is a distributed object-oriented system based on a new concept, a director-actor model. It models a distributed simulation as a collection of distributed director and actor objects running concurrently at different nodes. Collaboration among the objects are achieved by interaction between actors and directors, or between actors or directors. Each director corresponding to the participant in the distributed simulation generates commands to the actors which in turn activate the actual simulation by interacting with other actors or directors. The whole simulation is designed based on object oriented and distributed simulation model by mapping actor and director entities into objects and then assigning them into several logical processes on a distributed environment.

DOWS consists of four major components which interacts with each other in real time on a distributed environment: director, actor, actor manager, simulation manager. Director participates in the simulation concurrently through the interactive communication with one or more agents which are in charge of the parallel simulation of its associated simulation submodels using several logical processes. Coordinator controls the status of the simulation for one or a set of submodels. DOWS is implemented using *Java* objects which interact with each other through *Java* RMI. Each director is implemented as web client which downloads and execute the proper applet from HTTP server, and hence DOWS provides a web-enabled simulation environment which allows users to easily instantiate and modify the simulation model in HTTP server. The whole simulation can be speed up by decomposing the simulation model into several submodels, each of which consists of logical processes. The agents provide an efficient virtual real time simulation environment which integrates the coordination among user groups, logical processes, simulation manager based on a client-server model by supporting time synchronization, simulation message transfer, and network fault detection.

References

1. Fujimoto, R.M.: Parallel Discrete Event Simulation. *Commun. ACM*33, 10(Oct.), 30–53, 1990.
2. Jayadev Misra: Distributed Discrete-event Simulation. *ACM Computing Surveys*, vol.18, no.1, 39–65, March, 1986.
3. Sun Microsystems Inc. *Java Remote Method Invocation Specification*. 1998.
4. Hwa-Chun Lin, Chien-Hsing Wang: Distributed network management by HTTP-based remote invocation. *Global Telecommunications Conference, 1999. GLOBE-COM'99*, Volume: 3, 1999, 1889–1893
5. Fujimoto, R.M.: Time Management in the High Level Architecture. *Simulation* Vol. 71, No. 6, 388–400, December 1998.
6. Jefferson, D.R.: virtual time, *ACM Trans. Program. Lang. Syst.* 7, 3 (July), 1985, 404–425.
7. Arnold Buss: Web-based Simulation Modeling. *International Conference on Web-based Modeling & Simulation*, 1998.
8. Kevin J. Healy: Simulation Modeling Methodology and WWW. *International Conference on Web-based Modeling & Simulation*, 1998.
9. J.G. Park, A.H. Lee: Specializing the Java object serialization using partial evaluation for a faster RMI [remote method invocation]. *Parallel and Distributed Systems, ICPADS 2001 Proceedings*, 451–458

GEPARD – General Parallel Debugger for MVS-1000/M

V.E. Malyshkin and A.A. Romanenko

Novosibirsk State University
Chair of Parallel Computing, Russia
`malysh@ssd.sscs.ru`, `arom@ccfit.nsu.ru`

1 Introduction

Installed in Akademgorok (Novosibirsk) multicomputer MVS-1000/M [1] is now in intensive use. MVS-1000/M is the multicomputer of cluster architecture. It consists of 11 nodes with coupled Alpha-21264 processors and used mostly for the development and debugging of application programs. Its architecture and software are fully identical to the MVS-1000/M installation in Moscow [2] (more than 350 nodes with coupled Alpha-21264 processors) where real large-scale numerical modeling is accomplished.

One of the problems we have is debugging of application parallel programs. Unfortunately only sequential debuggers and proofing tools are installed on master computer of MVS-1000/M. It's highly important to have specialized parallel MVS-1000/M debugger that should take into account the peculiarities as hardware and software of MVS-1000/M as application area.

2 Parallel Program Debuggers Overview

Unfortunately, considered parallel program debuggers (TotalView [3], RAMPA [4], AIMS [5], Vampir [6], JumpShot [7], ParadyN [8], etc.) have strict functionality, which makes it not suitable for usage on MVS-1000/M. Interactive tools, like TotalView, are not suitable because interactions bring distortion in program behavior negligible at non-parallel program debugging. Some tools, for example Jumpshot, accumulate only information on communication operations, though it is not enough sometimes. The other programs know nothing about MPI (e.g. CXpref) or don't support Alpha architecture, for example ParadyN.

The desirable debugger for MVS-1000/M should meet the following requirements:

- flexibility of the system for debugging interprocesses communications,
- specialization for debugging of numerical models.

The debugger should be designed for usage on multicomputer.

3 Objectives of GEPARD Development

Basing on above mentioned requirements the project of GEPARD debugger development was initiated that provide:

- minimal influence on the behavior of an application program,
- flexible debugging data gathering and analysis,
- correspondence to the specification of MVS-1000/M.

4 Choice of Debugger Type

A parallel program for multicomputer is represented as a system of sequential asynchronous communicating processes [9,10]. Therefore, the development of a parallel program is carried out in two stages. From the beginning, sequential algorithms development, testing and debugging of implementing their sequential procedures are performed. Then the whole parallel program, assembled out of these procedures and communications, is debugged. In debugging the total correctness of all the interprocesses communications and the whole program should be checked. Not excluded that new errors in separately debugged components (sequential procedures) can be recognized.

Interactive and monitoring debuggers represent two different approaches to the implementation of the debugging tools. Interactive tools allow a user to stop the program, inspect values, and perform step-by-step execution. Monitoring systems are used if there is no possibility to use interactive tools or usage of them takes much time or heavily distorts time diagram of monitored system. Monitoring systems accumulate information to be analyzed on-line or after completion of the program execution. Since the influence of the debugger on real behavior of parallel program should be minimized the monitoring mode was chosen for GEPARD.

Two main ways to implement the collection of the debugging information are known. One of them is to execute a program under external trace tool. It might result that program runtime is substantially increased because program execution context is changed for every executable statement. Another one is to insert instructions into the source code in order to collect only the information required. In this case the debugger less influences on the program behavior, but the program's source code should be changed.

For GEPARD the second strategy was chosen. Gathered data are partially processed (collected, buffered, transferred to the trace file, etc.) by the external process. Data analysis is done after program completion that results in reduction of the debugger's influence on the program runtime.

5 GEPARD

GEPARD consists of the following components:

- debug language,
- data gathering system,
- visualization and analysis system.

An analysis of the program behavior is based on the information gathered at program runtime. The gathered information falls into three groups:

- state of the operations of communication (send/receive/synchronized),
- state of the program (e.g. value of some its variables),
- state of the program executing environment.

5.1 Debug Language

Two levels of information gathering are defined. By default only information on the state of the operations of communications is gathered (MPI function names, source/destination ranks, function call times, procedures run time and position of the called MPI function in source code). In order to get additional information a user should explicitly point out what info is required to be gathered. This is expressed in special debug language.

The debug language consists of instructions that are inserted into debugged program. Programmer performs insertion of the instructions into the source code. Each instruction is a comment of the C language. Its format is:

```
/*GPRD <instruction> */,
```

where debugger instruction defines description of the additional information to be gathered or function of the debugger.

This approach allows us not to rewrite program code. A user should only insert some comments that are processed by debugger's preprocessor. For instance, in order to count the number of loop iterations, user should place the following line as the first statement of a loop's body:

```
/*GPRD count */.
```

Similarly, user can count the number of calls of a certain function.

Expected interprocesses communications (IPC) also can be described in order the debugger could compare at runtime the described/expected and real program behavior. Communications define the relation on the set of all the processes. All the pairs of type (source_process_rank, destination_process_rank) are included into the relation. This relation is described by the instructions of the debug language. Here is a small example that shows the way to point out that the *i*-th process can sends messages to the next and previous processes in a line system of processes.

```
/*GPRD SCOMMSET $i SEND ($i-1, $i+1) */
```

The IPC system is not static and can be changed at the program runtime.

Debugger preprocessor recognizes these instructions and substitutes for them the proper statements of programming language. MPI function calls are replaced with wrapper-functions, which gather debug information. For example, MPI_Init function call is substituted for the following wrapper-function:

```
/*GPRD mod.MPI_Init(int *, char ***, int) */,
```

where the last parameter defines position of the function call in the source code. Preprocessor provides this information.

For now the only supported language is C with MPI-1.

5.2 Gathering of the Debug Information

Data gathering system consists of monitors (system processes, one for every virtual processor). Before execution of MPI_Init function each process of the parallel program creates monitor (MON) using fork (2) system call. MON is joined to their parent process with pipe. On completion of the monitor creation debugged process calls MPI_Init function. In accordance to the instructions, inserted into the source code, information is gathered and transferred to the MON. So, no activities for storing and processing information are performed by debugged program. It is done by MONs. MONs should put debug information into their internal buffers, gather information referred to program runtime environment, and perform initial data analysis. For example, if a user has described interprocesses communications, monitors compare this description with how the system of communications was really accomplished. The information on recognized mismatch is stored in the buffer too. When the buffer becomes full, debug info is put into the trace file.

Within a wrapper function debug information is sent to the MON twice – before and after replaced function call. It allows the debugger to keep track of information on functions lock. So, the debugger can recognize the situation of making receive call without proper send call.

5.3 Data Analysis and Visualization

In process of trace file creation, or after termination of the debugged program the gathered data can be visualized. The analysis system helps user to find the cause of an error. It is allowed to apply filters, to view different kinds of statistics. Analysis system points out different disparities; for example, mismatching quantities of send and receive calls, unbalanced load of processes, etc.

The visualization part of the analysis system has user-friendly interface in order a user doesn't spend time for study basic operations.

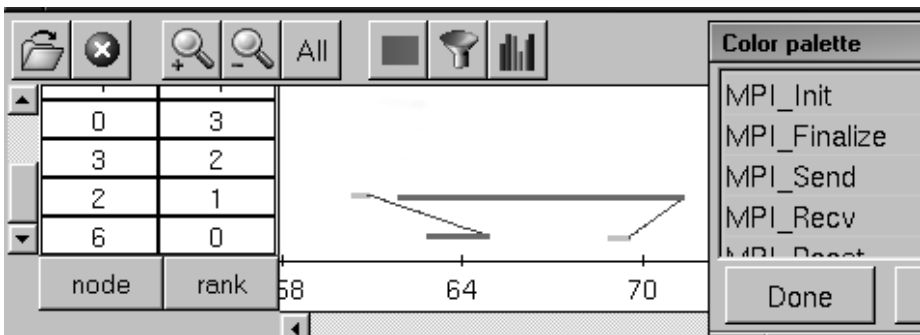


Fig. 1. The main window of visualization program

Event graph is displayed in the main window of visualization program. In the event graph each executing/running process is associated with a line, along which

the time is laid off. Events at process runtime are represented as the segments of the line. The segment length is equal to event duration. The fact of interaction between processes is represented as a line joining two segments. Gaps between segments of a process on the picture should be regarded as program execution.

Analysis of different program behavior with GEPARD enables us to recognize specific peculiarities of MPICH implementation. So, in Fig. 1 one can see that MPI_Send call (light segments) from thread 1 terminates before MPI_Recv function (dark segments) in thread 0 is called. It means, that data to be transferred with MPI_Send call is first stored into an internal buffer.

6 Conclusion

GEPARD demonstrates good ability for program debugging. It was in particular successfully applied for the analysis of the behavior of post-accidents states search parallel program developed for the Russian Energetic Company.

The debugger is now under permanent development. First, the analysis of the gathered information should be improved. Comparison of the different program executions is also planned. It is also planned to extend the debug language by the statements for description of mass operation like "to collect the information on all the function's calls located between statement_1 and statement_2". FORTRAN program debugging also should be supported.

References

1. Official site of the Novosibirsk Supercomputer Software Department. www.ssd2-new.sccc.ru
2. Official site of the Moscow Joint Supercomputer Center. www.jscc.ru
3. University of Karlsruhe: Parallel debugger TotalView. www.uni-karlsruhe.de/~SP
4. Krukov, V.A., Pozdnjakov, L.A., Zadykhailo I.B.: RAMPA – CASE for portable parallel programs development. Proceedings of the Third International Conference on Parallel Computing Technologies, Vol. 3, IPPE, Obninck, Russia (1993)
5. Yan, J.C., Sarukkai, S.R., Mehra, P.: Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. In: Software Practice & Experience. Vol. 25, No. 4 (April 1995)
6. Pallas GmbH: Vampirtrace User's and Installation-Guide. www.pallas.com (1999)
7. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. www-unix.mcs.anl.gov/perfvis/software/viewers/jumpshot-2/paper.html
8. Miller, B.P., Callaghan, M.D., Cargille, J.M.: The Paradyn Parallel Performance Measurement Tools. In: Special issue on performance evaluation tools for parallel and distributed computer systems. IEEE Computer 28, No. 11 (November 1995)
9. Valkovskii, V.A., Malyshekin, V.E.: Parallel Program and System Synthesis on the basis of Computational models. Nauka, Novosibirsk (1988) [in Russian].
10. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)

Development of Distributed Simulation System*

Victor Okol'nishnikov and Sergey Rudometov

Institute of Computational Mathematics and Mathematical Geophysics
of Siberian Branch of Russian Academy of Sciences,
prospect Lavrent'eva, 6, 630090 Novosibirsk, Russia
`okoln@kti.nsc.ru`

Abstract. Problems of development of distributed simulation system are discussed in this paper. The architecture and the realization of distributed simulation system DSS that is realized for parallel computer RM600-E30 are described. Directions for the further development of this system are defined.

1 Introduction

The simulation of the dynamics of complex systems is one of the fields of application of computers that requires a great deal of computational resources. The use of parallel or distributed multiprocessor systems can satisfy more and more increasing requirements of computational resources for simulation.

The sequential event-driven discrete simulation is based on quasiparallel execution of a set of submodels of the simulation model in logical (simulation, virtual) time. Two large steps for going from quasiparallel performance to parallel or distributed one have been made by the simulation. The first step was the development of the concepts and language forms for simulation tools. The second step was the development of Run Time Systems.

On the first step global variables in a simulation model were eliminated. As a result of this elimination a generation of event-driven discrete simulation languages, packages were developed where the interaction of submodels is realized uniquely with the help of message passing. The programs of simulation models designed with the help of these tools don't require algorithms of models to be changed for parallel performance.

On the second step a global clock indicating the current time of simulation model was eliminated for increasing the efficiency of parallel performance. As a result of this elimination the current time of simulation model is equal to the minimum of current times of each submodels. But a new problem for such asynchronous simulation has arisen. A submodel A can send a message to another submodel B and the current time of A is less then the current time of B.

Such irregular message breaks the correctness of parallel execution of simulation model. The correctness implies that events should be performed in the

* This research was supported by the Russian Foundation for Basic Research (Grant 02-01-00688)

strict chronological order for each submodel. It means that any submodel can not receive the irregular message. This constraint is usually called a "local casualty constraint" [1].

In order to satisfy this constraint special protocols of synchronization of message passing were developed. These protocols are realized within the Run Time System. The methods of synchronization are divided on conservative and optimistic ones [2].

A conservative method prevents a generation of irregular message by suspension of submodel B execution. An optimistic method permits a generation of irregular message, but it invokes a "rollback" mechanism for submodel B.

On this basis the distributed simulation system (DSS) was realized for SMP computer RM600-E30. This system was used as a prototype for a presently being developed generation of DSS for the parallel Supercomputer MBC-1000M.

2 Architecture of DSS

DSS models have a hierarchical structure with three levels of hierarchy: subjects, systems, and submodels.

Model processes, located at the lower level of hierarchy, are referred as subjects. Subjects present some simulated entities. A subject has input and output ports for passing messages, local data, and a program of its activity. The program of subject activity can include the following special operations:

- *hold* – delay of subject execution for some time;
- *waitSignal* – waiting of message arrival to some input port;
- *receive* – reading a message from some input port;
- *send* – writing a message to some output port.

Collections of subjects are referred as systems. Generally a system consists of subjects and perhaps of nested systems. The message passing between the subjects what are included in different systems is carried out implicitly with the help of input and output ports of systems. The connection between ports of subjects and systems is defined in a separate part of the simulation model program. The messages are passed asynchronously.

The systems located at a high level of hierarchy are referred as submodels. A submodel is performed in local logical time according to the concept of sequential event-driven discrete simulation. If the number of submodels is more than 1 then submodels can be performed in parallel on a multiprocessor system.

DSS Run Time System is divided into two parts: a simulation engine, and a communication system. The simulation engine drives the performance of each submodel by cyclic advancing local time of the submodel to the lowest time stamp of an event in the local (internal) event list. Also the simulation engine schedules activities of submodel subjects.

An arrival of a message is an external event for some submodel. The message passing between submodels is carrying out with the help of a communication

system. The communication system performs delivery of messages and synchronization of submodel local times to prevent a local causality constraint. The communication system interacts with submodels. The interaction is performed by passing additional special service message. The method of synchronization (conservative or optimistic) is realized within the communication system.

Architecture of DSS allows using various synchronization methods. At present a conservative algorithm is realized within the communication system of DSS. This algorithm is based on the sending of "nullmessages".

Model execution consists of 2 phases: a building phase and an execution phase. Actions in building phase: creation of subject and system objects (instances of corresponding classes), creation of input and output ports for them; connecting of ports of subjects and systems, setting of start model time, including subjects in systems, setting the final time of simulation.

When building phase has been carried out, the system goes in the execution phase. This phase is carried out when the model time is exceeded or there is no event in the local event list (all subjects are waiting subjects or system contains no subjects).

In the case of a distributed model there is another procedure of model creation. A user should create not only subjects, but system and submodel classes as well. A user must define a distributed topology and mapping between submodel names and names of processors.

3 Realization of DSS

The goal of developing of DSS is to obtain the great portable, high- performance system. This goal is achieved by using recent approaches in system design, by using recent portable and progressive techniques. These techniques are threads and MPI (Message Passing Interface).

Threads were standardized relatively recently. At present almost all operating system has a support of this standard. It will guarantee the portability of source codes that allow using DSS on distinct parallel and distributed architectures. Thread-enabled applications have more then one points of execution, and can be placed by operating system on more than one processor to allow using a resources of computer more effective. But using only threads cannot solve the problem of really distributed model execution.

MPI is intended for creating of distributed systems that needs a message passing between processes. Also MPI interface application can make barrier synchronization that will be used for starting of distributed simulation model. MPI can use all well-known and well-defined cases of message sending: synchronous, asynchronous, buffered, combined (complex). DSS uses non-blocking buffered data sending method with waiting of data transmission completion (MPI_IbSEND with MPI_Wait) that gives more qualified control of data transmission process and data buffering process. DSS uses MPI to send messages between submodels. The message (basic or server) is a data packet in an envelope with a timestamp that is the logical time of sender.

The source language of DSS is C++ based, process-oriented, discrete simulation language. The language provides following capabilities: interaction of processes between themselves through message passing, building of hierarchical models, dynamical change of the structure of the model, and parallel execution. DSS is intended for large-scale simulation of complex systems.

The source language of DSS is an extension of the language for quasi-parallel version of DSS for Windows (Chimera [3]). The Windows version has serious difficulties with porting it on other operating systems because it uses low-level operations. Thus the decision was made of modifying the kernel of Chimera so it can work using more portable and progressive techniques. There was a draft version of DSS that is written on Java [4].

After the realization of the draft version on Java the realization of DSS on UNIX-like system has been made. DSS has been realized for SMP computer RM600-E30 on C++ programming language using operation system Reliant-UNIX v.5.44, threads library SIthreads V5.44C for "pthreads", and MPICH V1.2.0 for MPI.

4 Conclusion

DSS is developed so that it provides means for the further development. This development is supposed to be made in the following directions:

- to port DSS on parallel Supercomputer MBC 1000M;
- to realize a library of various methods of synchronization both conservative and optimistic. It is also intended to provide the capability to choose a method that is more suitable on performance for a concrete class of models;
- to use the time management services defined in HLA (High Level Architecture). The HLA is the successor of distributed interactive simulation [5]. It supports the basic concepts and adds many new capabilities and functionality.

References

1. Fujimoto, R. M.: Parallel Discrete Event Simulation. *Communication of the ACM*, 33(10) (1990) 30–53
2. Ferscha, A.: Parallel and Distributed Simulation of Discrete Event Systems. *Parallel and Distributed Computing Handbook*, McGraw-Hill (1996) 1003–1041
3. Okol'nishnikov, V.V., Iakimovitch, D.A.: Visual Interactive Industrial Simulation Environment. In *Proc. of 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Vol. 5. Berlin (1997) 391–395
4. Okol'nishnikov, V.V., Rudometov, S.W.: Distributed Manufacturing Simulation Environment. In *Proc. of 16th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*. Eds. M. Deville, R. Owens. Dept. of Computer Sci. Rutgers University New Brunswick (2000) No. 611–5
5. Fujimoto, R.M.: Time Management in the High Level Architecture. *Simulation*, Vol. 71, No. 6 (1998) 388–400

CMDE: A Channel Memory Based Dynamic Environment for Fault-Tolerant Message Passing Based on MPICH-V Architecture

Anton Selikhov and Cécile Germain

Université Paris-Sud, Laboratoire de Recherche en Informatique,
Bâtiment 490, F-91405 Orsay Cedex, France
{Anton.Selikhov,Cecile.Germain}@lri.fr

Abstract. Utilization of computing power of idle workstations and tolerating failures of computing nodes running parallel message-passing applications is the research area attracting many research groups in Computer Science. A Channel Memory based approach has shown its capabilities to tolerate faults of tasks of parallel applications. The first work utilizing such approach in conjunction with a specially designed checkpointing and recovery protocol has been resulted in MPICH-V architecture. In this paper, we present Channel Memory based Dynamic Environment (CMDE) – a stand-alone distributed program system based on MPICH-V architecture. We also present an approach to tolerate faults of Channel Memories, based on CMDE architecture and on a Limited Replication of Channel Memories algorithm, introduced in this paper.

1 Introduction

Parallel execution models have been designed with a traditional machine model in mind, which can be summarized as strongly coupled: machines are reliable, and information flows reliably across computing entities (processes or threads). This assumption becomes less realistic with present computing infrastructures. The first case is very large clusters, which become increasingly represented in the most powerful installed machines (e.g., those from Top500 list). Even for high-end clusters, the MTBF is considered as typically less than one day. With current architectures, a failure of one cluster processing component has a minor impact on the productivity in a *farm of computers*: the farm runs a large number of sequential tasks. In contrast, large *clusters* target parallel or distributed applications. The challenge is then to design an execution environment that provides fault tolerance of parallel applications. The second case is Global Computing and P2P systems, which gather idle time of low-end desktops. In this case, disconnections of computing nodes are very frequent.

Previous research in the MPICH-V project has defined a protocol for transparent fault tolerance for message-passing programs, and has realized an implementation [1]. The core of MPICH-V is to define a recovery protocol [2] which allows to checkpoint and restart each MPI process independently. This feature

ensures that computation may progress even in presence of frequent faults. The protocol is based on uncoordinated checkpointing and distributed pessimistic message logging on dedicated architecture elements – Channel Memory Servers.

This paper presents CMDE, a parallel execution environment targeted at fault tolerance of MPI applications and based on the MPICH-V architecture. CMDE is a stand-alone environment for running parallel applications. It can manage several parallel applications simultaneously; each application may also contain different binary codes for different computing nodes. Features of CMDE architecture allow for cooperation with other systems providing their own environment for resource management, such as Condor [3] or XtremWeb [4].

As well as in [1], CMDE tolerates faults of its computing elements running task codes. It enhances its flexibility by tolerating faults of Channel Memories on the base of the Limited Replication of Channel Memories algorithm (LRCM). It has been shown [1] that the number of Channel Memory Servers is critical for performance of a user application. Having fault tolerance mechanisms, the Channel Memory Servers can be picked in the pool of unreliable resources, allowing for a large number of them if the expected communication pattern needs it. Only a small number of hosts is required by CMDE to be reliable in order to build an actual system.

The rest of the paper is organized as follows. Section 2 gives an overview of an execution model and of software components of the Environment. Section 3 describes the architecture and interaction of the components in more detail. Section 4 focuses on LRCM and its features. Section 5 presents some results of performance evaluation of different parts of CMDE.

Finally, Section 6 provides some conclusions and future work directions of the system.

2 Overview of the Environment

Execution model. CMDE consists of one distinguished component, the Dispatcher, and of components of three other types: Checkpoint Servers (CP Servers), Channel Memory Servers (CM Servers), and Workers. The Dispatcher monitors resource availability and schedules tasks to Workers. CM Servers remotely buffer all communications between tasks. CP Servers store checkpoint memory images of tasks running on the Workers. Finally, Workers run tasks which are MPI processes executed on volatile resources. CM and CP Server resources are shared by all applications, while, at a given point in time, a Worker runs one task of only one application. The Dispatcher and CP Servers have to be hosted on reliable machines with special requirements (network connection and disk space). CM Servers and Workers can be freely allocated on available resources, depending on the communication requirements.

An application in CMDE is considered as a set of executable files, input files and command lines. All applications are stored in the Dispatcher. An application is launched when enough Workers and CM Servers become available. The choice of an application to be launched by the Dispatcher depends on the scheduling policy. The Dispatcher monitors execution of the application until completion.

If faults lead to an insufficient number of resources for a running application, it is stalled until new resources appear.

The user interface to CMDE is provided by a separate component, the Client, which is external to the CMDE resource pool, but can connect to the Dispatcher.

Dynamic components. All the components join to CMDE dynamically and in any order by registering in the Dispatcher. The dynamic components (CM Servers and Workers) are allowed to disappear from the Environment at any time, without any prior notification. Faults are considered as fatal: if a faulty component reconnects later, all the information it stored before the fault is considered as lost. Because all the components which communicate are permanently connected to the Dispatcher through TCP/IP sockets, there is no need for a soft-state registration protocol: the faults are detected by broken connections. This allows for local detection of faults, and a distributed recovery protocol in the case of LRCM algorithm support.

MPICH role. CMDE targets applications based on MPI, and uses the MPICH implementation of MPI (currently MPICH 1.x). In MPICH, MPI communications are implemented on top of a communication *device*, which provides an interface between the MPI user-level functions and a low-level communication protocol (TCP/IP or other). We defined and implemented a *ch_cm* device, which targets the Channel Memories protocol, for reliable (non faulty) Channel Memories in [5]. In order to take into account failures of CM Servers, a new version of MPICH device, *ch_cmde*, has been designed and implemented.

3 Components of the Environment

All components initially register to the Environment by connecting to the Dispatcher using its IP address and a port number corresponding to the type of the component. Each component receives a rank, unique for each group of components, which acts as a unique resource identifier in the system. The following functionality depends on the component type. After registration, a permanent connection is opened between a component and the Dispatcher. Illustration of a sample CMDE configuration is presented in the Fig. 1.

3.1 Clients

A Client is implemented as a simple console application using an *application configuration file*. The file has a simple text format, describing the number of computing nodes required for the application, one or more executable binary files of the application with the required command lines and all input files. The minimal number of CM and CP Servers may also be pointed. While submitting an application to the Dispatcher, a Client checks the Configuration file, uploads the application files and notifies about the result of the submission. The Client is considered as an implementation of interface to the Dispatcher for submission of applications and may be redesigned utilising other user interfaces, e.g. special graphic or web interfaces.

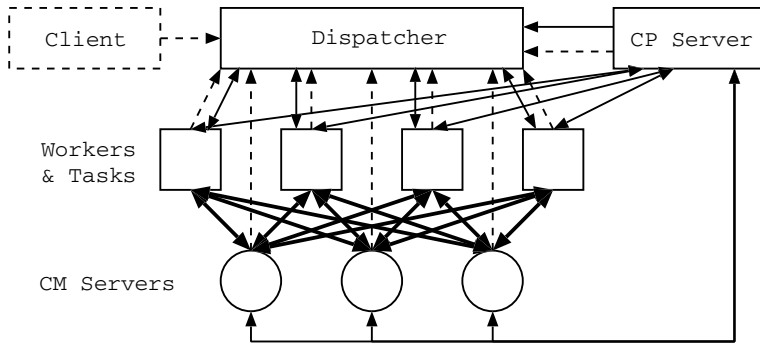


Fig. 1. An example of CMDE configuration for a parallel application with 4 tasks. Dashed arrows – connections of components, solid arrows – application support communications, bold arrows – communications between the application tasks and CM Servers during execution of an application

3.2 Dispatcher

Besides accepting and storing applications before and during their execution, the Dispatcher monitors execution of tasks on the Workers, and availability of CM and CP Servers. To achieve scalability, the role of the Dispatcher should be as limited as possible. Currently, the Dispatcher is deeply involved in monitoring computations, and only slightly in monitoring CM Servers. Faults of CM Servers are handled by a distributed algorithm described in the Section 4). To provide better performance, the Dispatcher is implemented as a multithread server.

Monitoring computations on Workers aimed to recovering of failed tasks. When a Worker fails, the Dispatcher chooses an available Worker from the current pool of free Workers and uploads the code and input files to this new Worker with a flag indicating that this task should be restarted from the last checkpoint, and also with a reference to that checkpoint. Each successful checkpointing of a task is registered by the Dispatcher and marked in the corresponding task structure, so that the Dispatcher can set the restart flag. From this moment, the control of the recovery process goes to the Worker. If there is no free Worker available, the failed task is placed in the FIFO queue common for all applications.

3.3 Checkpoint Servers

The main purpose of a CP Server is to store checkpoint image files of executed tasks and sending them back on a request. Its functionality is very simple, however, its disconnection cannot be tolerated in CMDE, thus the number of CP Servers should be minimal to provide a good checkpointing performance.

In addition to registration in CMDE and receiving/sending checkpoint images, a CP Server supports MPICH-V checkpointing protocol by communicating with CM Servers and with the Dispatcher to notify them about successful receipt of a checkpoint image of a task. Each checkpoint image is stored with unique name composed by the unique identifier of the task which has sent this image.

3.4 Channel Memory Servers

Detailed description of purposes and principles of functioning of a CM Server was presented in [5]. The main purpose of a CM Server is to handle CMs – the storages of MPI messages being in transit between source and destination nodes of a parallel application. Because of required ability to handle communications for many tasks, a CM Server is implemented as a multithread server with piping long messages passing through a CM Server from one task to another.

Mapping between tasks of an application and the CM Servers of CMDE is described in a CM Server Table (CMS Table) created by the Dispatcher and maintained by CM Servers and by *ch_cmde* devices. The CMS Table defines the *owning* relation: a CM server *owns* a task for which it stores all messages addressed to this task (i.e. CM of this task). This relation is used in reconfiguration of existing CMS Tables of applications when a new CM Server appears in the system or one existing CM Server disappears because of a fault. Description of the algorithm and communication protocols supporting dynamic addition/removing of CM Servers in CMDE are described in the Section 4.

The CMS Table of an application also defines a *ring* of CM Servers. Communications between CM servers are limited to the two nearest neighbors in the ring. This allows to reduce the number of communications between CM Servers while supporting LRCM algorithm.

3.5 Workers

Worker represents a computing resource in CMDE. Its main function is to launch an application task locally. In addition, a Worker supports checkpoint facilities of the task by running an additional checkpoint alarm process.

Functionality of a Worker is based on a number of processes and threads, dedicated to communication with the Dispatcher, launching a task and providing checkpointing mechanism.

4 Fault Tolerance Mechanisms

Providing fault tolerance for parallel applications in CMDE is based on uncoordinated (asynchronous) checkpointing of application tasks and on pessimistic logging of messages being transferred between the tasks. Checkpointing of parallel tasks is performed using Condor Stand-Alone Checkpointing library (CSCL) [6], while message logging is based on utilisation of Channel Memories and a specially designed MPICH-V protocol [2], making possible to perform checkpointing asynchronously. Extensive analysis of this approach in the case of reliable Channel Memories is resulted in [1]. In comparison with other fault tolerance mechanisms, it has two main advantages. First, it allows to exclude global checkpoint synchronisation, used e.g. in Cocheck[7] and Starfish[8]. Next, it performs checkpointing automatically, being implemented in the low-level MPICH communication device, in comparison with API-based tools for user-defined checkpointing, used e.g. in Clip[10] and FT-MPI[11]. A disadvantage of this approach

is using external Channel Memories to store messages and logs. MPICH-V protocol implies that the Channel Memories are reliable and always available for communicating tasks.

In this section, we focus on fault tolerance for Channel Memory Servers. The basic hypothesis is that there is only one failure at a time. More precisely, a failure implies a recovery process involving the Workers and the CM Servers. If another failure happens while this process is not complete, CMDE will have to restart the whole application from the beginning.

4.1 MPICH Communication Device

Message passing through unreliable communication media requires to verify each communication on success by involving additional ack messages. According to this requirement, the communication device used by MPICH library [5] in tasks managed by CMDE has been redesigned in comparison with that used in MPICH-V. Each of three basic communication function implementing higher layers [9] of MPICH begins with a request to the CM Server being the owning CM Server of the destination of the communication. If one CM Server disconnected and remaining CM Servers have to be (have been already) reconfigured, the result of this request leads to reconfiguration of the local CMS Table, receiving a confirmation about successful reconfiguration from all remaining CM Servers (which blocks until the reconfiguration has been done), reconnection to a new own CM Server if the old one has been disconnected and restarting the communication from the beginning. All of this is performed by the communication device of tasks independently from other tasks and asynchronously. All communications requested by communication devices of tasks during reconfiguration are delayed until the reconfiguration has been finished. Notifications about finishing the reconfiguration are sent to all waiting tasks by CM Servers immediately, while all other tasks receive this on their first upcoming communication.

4.2 CM Servers

The possibility to tolerate faults of CM Servers is based on restoring all Channel Memories and corresponding message logs, handled by the server. LRCM algorithm is designed to manage both disappearing and appearing of CM Servers, providing dynamism of CM Server components in CMDE.

LRCM Algorithm. The main idea of LRCM algorithm is replication of CMs (and their logs) handled by a CM Server on a *mirroring* CM Server being chosen appropriately. The case when each CM Server is mirrored by one other CM Server (one-by-one, "limited" mirroring) is considered. This approach has a constant overhead depending on the number of replicated Channel Memories. A CM Server mirroring the given CM Server is chosen on the base of CMDE global ranking of components: CM Server with a rank r mirrors (has replicas of CMs of) CM Server with the rank $r + 1$ in round-robin fashion. Each CM Server uses the LRCM algorithm locally, communicating with only one mirroring and one mirrored CM Servers being neighbors on the application CMS Table ring.

The following main steps of the LRCM algorithm may be outlined.

1) *Assigning of CM Servers to an application.* CM Servers are assigned to applications automatically with nearly equal number of CMs handled by each CM Server and fixed in CMS Table of an application. According to this Table, each CM Server determines mirroring and mirrored CM Servers.

2) *Replication of Channel Memories.* Replication of a CM means replication of each message it contains and of its message log. It is performed by sending each arriving message to the mirroring CM Server *in parallel* while receiving of the message, utilising thread-based pipeline. To ensure replication, a task is acknowledged on each message received from it by a CM Server only when this message is successfully replicated to its mirroring CM Server.

3) *Handling of appearing/disappearing of CM Servers.* A fault of a CM Server with rank f leads to excluding it from CMS Tables of all applications it was included and to creating of mirroring dependences between CM Servers $f - 1$ and $f + 1$. Appearance of a new CM Server leads to including it in place of a failed CM Server or between CM Servers with 0 and maximal ranks, if the number of CM Servers in the CMS Table is less than the number of nodes of the application and to reconfiguring of mirroring dependences. All the changes lead to reconfiguration of CMS Tables of corresponding CM Servers.

Correctness of LRCM algorithm with respect to message passing is based on proved correctness of tasks checkpoint and recovery protocol [2] and on absence of messages being in transit during execution of the algorithm.

One-by-one mirroring suggested by LRCM algorithm allows tolerate only one CM Server fault at a moment of time, until reconfiguration of CM Servers has been finished. It also induces an additional overhead in communications between tasks, which seems to be a reasonable payoff for a possibility to tolerate faults of CM Servers and to increase the number of CM Servers improving performance of the whole CMDE.

4.3 The System Fault Recovery Levels

To summarise all fault tolerance mechanisms implemented in CMDE, one can consider the levels of recovery from faults of various system components.

A fault of a Worker and, as a consequence, of a task of an application is handled by allocating a new free Worker for the task if such a Worker is available and do not stop the application. If there is no free Worker, all other tasks of the application are paused since the moment of waiting for the failed task messages.

A fault of a CM Server leads to interruption of all tasks of all applications handled by this CM Server since the next communication of each task. In the case of two or more CM Servers registered in CMDE before the failure, tasks of all the applications continue their communications right after reconfiguration of remaining CM Server(s). Otherwise, all the applications will be restarted from the beginning as soon as the first two CM Servers will be registered in CMDE.

Because the CP Server is expected to be reliable for the current CMDE architecture, its fault leads to cancellation of all applications it handled. These applications will be restarted from the beginning using another CP Server if any or as soon as the first CP Server will be registered in CMDE.

The Dispatcher is considered to be single in the current CMDE architecture and to be reliable, therefore a fault of the Dispatcher leads to cancellation of all applications and disconnection of all CMDE components. At the current CMDE implementation, restarting of the Dispatcher requires reconnection of all other components and resubmission of all applications being queued and executed.

5 Performance of CMDE

Because of the target functionality of CMDE, we consider performance of the Environment in the sense of its ability to decrease all overheads obviously induced on execution of an application to achieve its fault tolerance. All other performance characteristics of CMDE like a time of submission of an application or a time of registration of new components or even a time of fault recovery process are out of consideration in this paper.

To estimate these important overheads, performance of the same test applications being run on top of MPICH for NOW (with *ch_p4* communication device) is used as a standard.

Utilization of Channel Memories have the main contribution to an application execution overhead. Performance overheads of this approach were investigated in [1,5]. It was shown, that the time of blocking communication through Channel Memory is two time longer in comparison with *ch_p4* implementation. This is explained by the fact of two actual communications through CM-based communication device instead of one through *ch_p4* device. In CMDE, a new implementation of the communication device, *ch_cmde*, is performed, using pipelining of simultaneous send-receives and transferring a message by fixed-size blocks and by multiple threads. Fig. 2 illustrates results of round-trip time test for this improved CMDE communication device in the case of two application nodes communicating through two Channel Memory Servers. According to these results, the overhead is very low at the message sizes less than 220 kB and becomes nearly 75% of *ch_p4* performance for bigger messages. The throughput of the Channel Memory based communication channel obtained on 100 Mbit Ethernet network is 1,66 MB/sec on 1-kB messages (1,8 MB/sec for *ch_p4*), 5,4 MB/sec for 100-kB messages (5.45 Mb/sec for *ch_p4*) and 4,3 Mb/sec for 500-kB – 1-Mb messages (5,5 Mb/sec for *ch_p4*).

An advantage of fault-tolerant communications between tasks in CMDE obviously brings an overhead in the same manner as TCP/IP protocol brings an overhead by implementing reliable low-level communications in comparison with UDP protocol. Implementation of the LRCM algorithm includes two more phases of each communication of a task: probing of availability of a Channel Memory and receiving a confirmation about the communication result. The first phase involves request-response messages passing, the last one implemented as receiving of a short message. All these additional transactions use fixed-length messages and therefore bring a constant latency overhead. Current implementation is characterised by latency of around 1 ms for communication device supporting LRCM (*ch_cmde*), 0.4 - 0.6 ms – without LRCM support (*ch_cm*), in comparison with 0.16 ms for MPICH with *ch_p4* device.

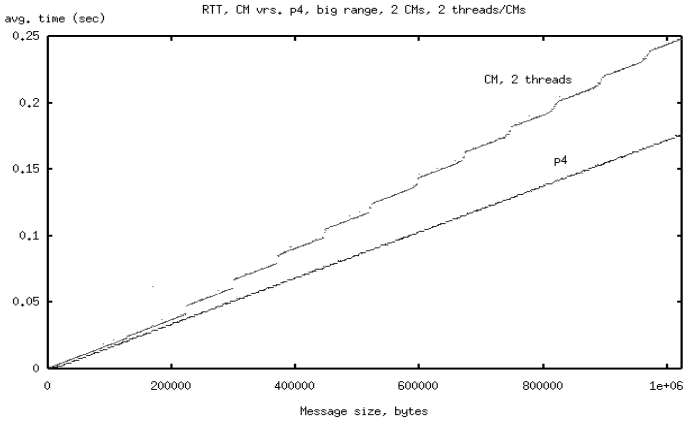


Fig. 2. Round-trip time test results for point-to-point communications between tasks through CM Servers. Two tasks communicated through two CM Servers

The last significant performance factor is implementation of checkpointing mechanism. Like MPICH-V implementation, the checkpointing is performed by a specially created parallel process to minimise an impact on the task process. In CMDE, a multithread implementation of Checkpoint Server is used to maximise its throughput. An optimal checkpointing frequency could minimise an overhead in utilisation of the same physical network channels, but this possibility as well as the fault rate influence seems to be a subject of a separate deep investigation.

6 Conclusion and Future Work

CMDE, a new dynamic environment based on Channel Memory approach for message passing in MPI applications, is presented in the paper. It is a result of stand-alone implementation of MPICH-V architecture complemented by a simple LRCM algorithm to tolerate faults of Channel Memory Servers. CMDE dynamic changing the number of most CMDE components. As in many similar projects, the Condor Stand-Alone Checkpointing library is used in CMDE to checkpoint application tasks. A simple interface to submit parallel MPI applications to CMDE is implemented with possibility to use different task codes.

CMDE is in the beginning of its development, however all functionality concerning submission of a parallel application to CMDE, dynamical connection of CMDE components, checkpointing and recovery from faults of tasks are implemented. CMDE can be used as a stand-alone environment, but after improvement of its modularity and adaptation to OGSA [12] specifications, CMDE may be also configured as a service for Global Computing systems. A possibility of running more than one task by a Worker is considered for the nearest enhancement of the system in order to obtain more efficient utilisation of SMP hosts. Improvement of modularity of CMDE will allow to use its Dispatcher

and Workers to run usual MPICH-based and distributed applications as well. Finally, CMDE may be improved to provide message passing service for some other message-based communication interfaces.

References

1. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Hérault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In Proc. IEEE/ACM SC2002 Conf., Baltimore, Maryland (2002)
2. Hérault T., Lemarinier P.: A rollback-recovery protocol on peer to peer systems. In Proc. of MOVEP'2002 Summer School (2002) 313–319
3. Raman R., Livny M.: High throughput resource management. Chapter 13 in The Grid: Blueprint for aNew Computing Infrastructure, Morgan Kaufmann, San Francisco, California (1999)
4. Fedak, G., Germain, C., Neri, V., Cappello, F.: XtremWeb: a generic global computing platform. IEEE/ACM CCGRID'2001. IEEE Press (2001) 582–587
5. Selikhov, A., Bosilca, G., Germain, C., Fedak, G., Cappello, F.: MPICH-CM: A communication library design for a P2P MPI implementation. Proc. 9th European PVM/MPI User's Group Meeting, Linz, Austria, September/October 2002, LNCS, Vol. 2474. Springer-Verlag, Berlin Heidelberg (2002) 323–330
6. Condor Manuals, Chapter 4.2.1. <http://www.cs.wisc.edu/condor/manual/>
7. Stellner, G.: CoCheck: Checkpointing and proces migration for MPI. Proc. 10th International Parallel Processing Symposium (IPPS'96), Hawaii (1996) 526–531
8. Agbaria, A., Friedman, R.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. Proc. 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99) (1999) 167–176
9. Gropp, W., Lusk, E.: MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory (1995)
10. Chen, Y., Plank, J. S., Li, K.: CLIP: A checkpointing tool for message-passing parallel programs. Int. Conf. on High Performance Networking and Computing (SC'97) ACM Press (1997)
11. Fagg, G., Dongarra, J.: FT-MPI: fault-tolerant MPI, supporting dynamic applications in a dynamic world. Proc. 7-th EuroPVM/MPI User's Group Meeting, LNCS, Vol. 1908 Springer-Verlag, Berlin Heidelberg (2000) 346–353
12. Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project (2002) <http://www.globus.org/research/papers/ogsa.pdf>

DxML: A Program for Distributed Computation of Phylogenetic Trees Based on Load Managed CORBA^{*}

Alexandros P. Stamatakis¹, Markus Lindermeier¹, Michael Ott¹,
Thomas Ludwig², and Harald Meier¹

¹ Technical University of Munich, Department of Computer Science
Boltzmannstr. 3, 85748 Garching b. München, Germany
{stamatak, linderme, ottmi, meierh}@in.tum.de
wwwbode.cs.tum.edu

² Ruprecht-Karls University, Department of Computer Science
Im Neuenheimer Feld 348, 69120 Heidelberg, Germany
thomas.ludwig@informatik.uni-heidelberg.de
pvs.iwr.uni-heidelberg.de

Abstract. High performance computing in bioinformatics has led to important progress in the field of genome analysis. Due to the huge amount of data and the complexity of the underlying algorithms many problems can only be solved by using supercomputers. In this paper we present **DxML**, a program for the distributed computation of evolutionary trees. In contrast to prior approaches **DxML** runs on a cluster of workstations instead of an expensive supercomputer. For this purpose we transformed **PxML**, a fast parallel phylogeny program incorporating novel algorithmic optimizations, into a distributed application. **DxML** uses modern object-oriented middleware instead of message-passing communication in order to reduce the development and maintenance costs. Our goal is to provide **DxML** to a broad range of users, in particular those who do not have supercomputers at their disposal. We ensure high performance and scalability by applying a high-level load management service called **LMC** (**Load Managed CORBA**). **LMC** provides transparent system level load management by integrating the load management functionality directly into the ORB. In this paper we demonstrate the simplicity of integrating **LMC** into a real-world application and how it enhances the performance and scalability of **DxML**.

1 Introduction

Within the framework of the **ParBaum** project at the TUM (Technische Universität München) work is conducted in the area of high performance bioinformatics

^{*} This work is partially sponsored under the project ID **ParBaum**, within the framework of the “Competence Network for Technical, Scientific High Performance Computing in Bavaria”: KONWIHR (Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern). KONWIHR is funded by means of “High-Tech-Offensive Bayern”.

in order to design novel parallel and distributed systems as well as algorithms for large-scale phylogenetic (evolutionary) tree computations based on the maximum likelihood method.

Phylogenetic trees describe the relative evolutionary distances between organisms and are calculated using information from their genetic sequences. The rRNA (ribosomal RiboNucleic Acid) is a distinguished, highly conserved region of an organisms genetic sequence and is therefore apt for determining evolutionary relationships.

Our work relies on sequence data provided by the ARB [16] (Latin, “arbor” = tree) rRNA-sequence database, which provides a huge amount of high quality sequence data and is a joint development of the LRR (Lehrstuhl für Rechner-technik und Rechnerorganisation) and the “Department of Microbiology” of the TUM. The ARB software is a graphically oriented package comprising various tools for sequence database handling and data analysis. A central database of processed (aligned) sequences and any type of additional data linked to the respective sequence entries is structured according to phylogenetic or other user defined criteria.

The maximum likelihood method renders evolutionary trees of high quality. A recent result by Korber *et al.* that times the evolution of the HIV-1 virus [3] demonstrates that maximum likelihood techniques can be effective and important for solving scientific problems in medicine and biology. However computing evolutionary trees based on this model is extremely computationally expensive. Thus, only relatively small trees (≈ 500 sequences [14],[15]), compared to the huge amount of data available (≈ 20000 sequences in today’s databases), have been calculated on supercomputers so far.

Within this context we investigate different approaches for handling the complexity of the problem. In this paper we focus on the distributed computation of large phylogenetic trees.

An important property of existing parallel phylogeny programs, such as **parallel fastDNAmI** [14] or **PxML** [8], [10], [11], [12], [13], is that they are well suited for distributed computation, since the largest part of computation time is consumed by the workers during tree evaluation and comparatively small amounts of data are communicated in a simple string format. Furthermore, at each step of the computation there is a large amount of independent tasks that can easily be distributed among the workers.

For handling the complexity and heterogeneity of todays computing environments and to exploit the vast amount of unused resources one can typically find in organizations, such as universities or research laboratories the distributed object-oriented programming paradigm is the most adequate mechanism, especially coupled with a powerful load balancing tool. With **DAXML** (**D**istributed **A(x)**ccelerated **M**aximum **L**ikelihood) we present a new approach for the calculation of large phylogenetic trees exploiting the advantages of the distributed object-oriented programming paradigm through the integration of the powerful load management tool **LMC** paired with the very fast tree evaluation function of **PxML** [10], [13], which is based on novel algorithmic optimizations.

2 The Load Management System

Nowadays applications do not reside on a single host anymore - they are distributed all over the world and interact through well defined protocols. Global interaction is accomplished by so called middleware architectures. The most common middleware architectures for distributed object-oriented applications are the CORBA (Common Object Request Broker Architecture) and the DCOM (Distributed Component Object Model). Environments like CORBA and DCOM cause new problems because of their distribution. A significant problem is load imbalance. As application objects are distributed over multiple hosts, the slowest host determines the overall performance of an application. Load management services intend to compensate load imbalance by distributing workload. This guarantees both, high performance, as well as scalability of distributed applications.

Our load management concept uses objects as load distribution entities and hosts as load distribution targets. Workload is distributed by initial placement, migration, and replication.

- *Initial Placement* stands for the creation of an object on a host that has sufficient computing resources in order to efficiently execute the object.
- *Migration* means moving an existing object to another host that promises a more expeditious execution.
- *Replication* is similar to migration but the original object is not removed, such that identical objects called replicas are created. Further requests to the object are split up among its replicas in order to distribute the workload (requests) among them.

There are two kinds of overload in distributed object-oriented systems - background overload and request overload. Background load is caused by applications that are not controlled by the load management system. Request overload means that an object is not capable to efficiently process all requests it receives. Migration is an adequate technique for handling background load but the scalability attained by migration is limited. Replication helps to break this limitations and is an adequate technique for handling request overload.

We implemented these concepts in the **LMC** system [5]. **LMC** is a load management system for CORBA. The main components of **LMC** are shown in Figure 1. These components fulfill different tasks and work at different abstraction levels. The load monitoring component offers both, information on available computing resources and their utilization, as well as information on application objects and their resource usage. This data has to be provided dynamically, i.e. at runtime, in order to obtain information about the runtime environment and the respective objects. Load distribution provides the functionality for distributing workload by initial placement, migration, or replication of objects. Finally, the load evaluation component decides about load distribution based on information provided by load monitoring. Those decisions can be attained by a variety of strategies, which are discussed in detail in [6].

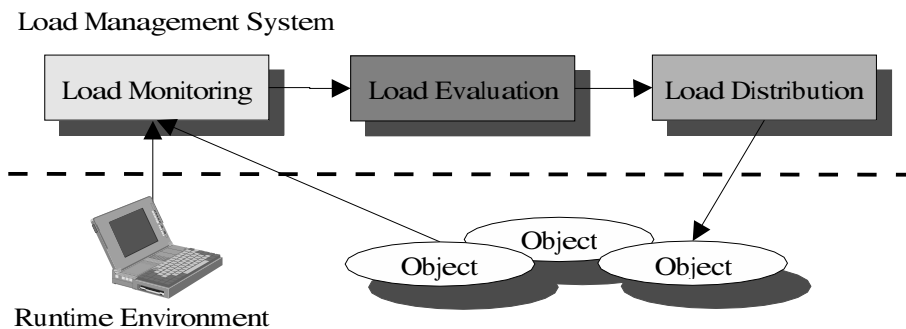


Fig. 1. The components of the Load Management System LMC

LMC is completely transparent on the client-side because it uses CORBA's Location Forward mechanism to distribute requests among replicas. On the server-side minor changes to the existing code are necessary for integrating load management functionality into the application. These changes mainly affect the configuration of the Portable Object Adapter (POA). All extensions are seamlessly integrated into the CORBA programming model. Thus, only a minor additional effort is required by the application programmer for the integration of the services provided by **LMC**.

For a detailed description of the load management system as well as the initial placement, migration and replication policies used see [6].

3 The Application

DAXML is based on **PAXML**, which is in turn a derivative of the latest release of **parallel fastDNaml** (version 1.2.2). The essential difference between **parallel fastDNaml** and **PAXML** are several novel algorithmic optimizations of the topology evaluation function, which, depending on the input data set and the processor architecture, lead to global run time improvements between 27% and 65% [11], [13]. Note that **PAXML** scales particularly well on PC processor architectures, which are the main target platforms for **DAXML**. Since our optimizations are purely algorithmic **PAXML/DAXML** render exactly the same results as **parallel fastDNaml**.

The attained level of run time improvement attained by our algorithmic optimizations is significant (see Figure 3), because our work focuses on computations of huge phylogenetic trees.

Since the algorithmic optimizations introduced by **DAXML** are on a fine granularity level and do not affect the parallelization concept, we will restrain our analysis to a brief description of the sequential "stepwise addition algorithm" which was introduced by J. Felsenstein [2] and implemented with some modifications in **fastDNaml** [7]. Furthermore, we will shortly outline the parallel algorithm of **parallel fastDNaml** and **PAXML**.

The calculation of the optimal phylogenetic tree for a set of rRNA input sequences based on the maximum likelihood method is NP-complete, due to the exponential growth in the number of possible tree topologies (e.g. there exist over 2 million possible topologies for 10 sequences). Thus, heuristics have to be introduced, in order to reduce the search space, i.e. the number of evaluated tree topologies. Suppose we have a set of n input sequences. A phylogenetic tree is an unrooted binary tree with the sequences at its leaves and with $2n - 3$ inner nodes (each node of the tree has either degree 3 or degree 1).

The sequential algorithm, works as follows:

Suppose we have found the best tree t_k of size k , i.e. with k sequences at its leaves, according to the heuristics. Sequence $k + 1$ consisting of a new branch with a new inner node and the sequence at either end, is then inserted into all $2k - 3$ branches of t_k and the likelihood of the in that manner generated topologies $t_{k+1,1}, \dots, t_{k+1,s}$, $s = 2k - 3$ is calculated. After this step, local and/or global rearrangements of the best tree drawn from the set $t_{k+1,1}, \dots, t_{k+1,s}$ are performed and evaluated, if the respective program option is set, in order to further improve the quality of the tree. The tree with the best likelihood of size $k + 1$ is then used for insertion of sequence $k + 2$. We call all tree topologies of size k , that are evaluated by the algorithm: “topology class of size k ”.

The algorithm starts with the only possible tree topology of size 3 using the first 3 sequences of the input data set and subsequently adds the remaining sequences as described above.

Since the most cost intensive part of the computation is the calculation of the likelihood value for each tree topology analyzed ($\approx 95\%$ of total computation time in the sequential program), the parallelization is straight-forward.

The parallel algorithm consists of a master, which is responsible for initialization, distribution of the input data, generation of tree topologies and gathering results.

The worker component simply performs the evaluation of a specific tree topology obtained by the master, i.e. computes its likelihood value. The topology to be evaluated is transformed into a simple, relatively short, string representation by the master and sent to a worker. Thus, especially since topology evaluation times increase with tree size k , $k = 4, \dots, n$, the communication overhead is neglectible for the calculation of huge phylogenetic trees and the problem is well-suited for distributed computation (see Figure 4).

In **parallel fastDNaml** an additional foreman component has been inserted between master and workers for error-handling.

4 Implementation

For designing **DAxML** we initially simplified **PAxML** by removing the foreman component entirely from the system, since error handling can more easily be handled directly by **LMC**.

Furthermore, we changed the program structure, such as to create all tasks of size k , i.e. all topologies with k leaves, that can be evaluated independently

at once, and queue them in their string representation. This transformation was performed, in order to provide a means for issuing simultaneous topology evaluation requests (see below). Note, that several sets of trees of topology class

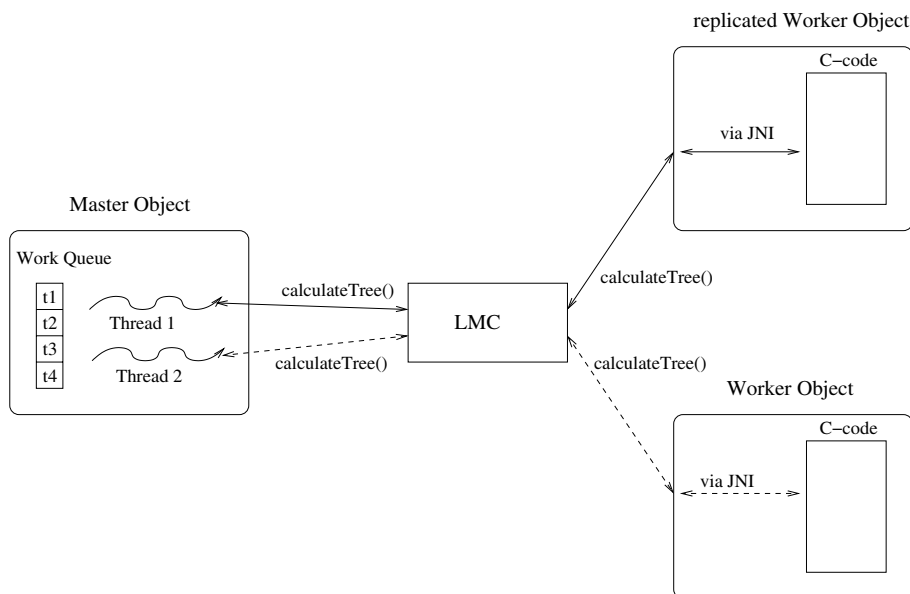


Fig. 2. System architecture of DAxML

k , that have to be evaluated in sequential order, may be generated, depending on the selected program options of **DAxML**.

Those sets are sufficiently large, such that they do not create a synchronization problem at the respective transition points.

The overhead induced by first creating and storing *all* topologies before invoking the evaluation function is neglectible, since the invocation of the topology evaluation function consumes by far the greatest portion of execution time.

Because **LMC** is based on a modified **JacORB** [1] version and only provides services for JAVA/CORBA applications, we initially transformed the simplified code into a sequential JAVA program using JNI (JAVA Native Interface). We designed two JAVA classes **Master** and **Worker** providing analogous functionalities as their counterparts in **PAxML**. The basic service provided by the **Worker** class is a method called **calculateTree()**, for evaluating a specific tree topology, which in turn invokes the fast native C evaluation function via JNI.

The **Master** component loads and parses the sequence file, passes the input data to the **Worker**, generates tree topologies and gathers results.

The transformation of the sequential JAVA code into a **LMC**-based application was straight-forward, since its class layout already complied with the structure of the distributed application. The **Worker** class is encapsulated as CORBA worker object, and provides its topology evaluation function as CORBA service. The state of the CORBA **Worker** object consists only of the sequence data, which can be loaded via NFS or directly from the **Master** when the **Worker** object is created either by initial placement, migration or replication.

Thus, since the sequence data is not modified during tree calculation, replications and migrations of worker objects do not induce any consistency problems.

In the main work-loop of the **Master**, a number of threads corresponding to the number of available hosts controlled by **LMC** is created, in order to perform simultaneous topology evaluation requests. This enables **LMC** to correctly distribute tree evaluation requests among worker objects on distinct hosts and to ensure optimal distribution granularity. The system architecture of **DaxML** is outlined in Figure 2 for a simple configuration with two worker objects.

5 Results

We conducted performance analysis tests on 4 Ethernet connected Sun-Blade-1000 machines of the SUN cluster at the LRR using sufficiently large test sets of 20, 30, 40 and 50 sequences, extracted from the ARB database, in order to evaluate the behavior of **DaxML** and **LMC** in terms of CORBA/JNI overhead, impact of the algorithmic optimizations, and automatic worker object replication/migration.

In Figure 3 we demonstrate the impact of the algorithmic optimizations on the speed of the tree evaluation function including JNI and CORBA overhead. We conducted two **DaxML** test runs with a single worker object, using the standard and optimized tree evaluation function and measured the average tree evaluation time per topology class (see section 3) for a test set of 40 sequences. The algorithmic optimizations show analogous performance improvements as measured for the parallel and sequential program [11]. All subsequent tests were performed using our novel optimized evaluation function.

Another important aspect is the overhead induced by the integration of CORBA and JNI into **DaxML**. As previously mentioned the communication overhead decreases with increasing tree size, due to the fact, that average evaluation time per tree increases during the computation as depicted in Figure 3, whereas the amount of communicated data per topology class remains practically constant. For the same reasons and despite the fact, that we have used some heavy-weight JNI mechanisms such as JAVA callbacks from C, the JNI overhead becomes neglectable as the tree grows, since only small amounts of data are passed through JNI.

We measured average C, JNI, and CORBA tree evaluation times for selected topology classes of size 4, 10, 20, 30 and 40. As can be seen in Figure 4 during the initial phase of the computation, i.e. for size 4 and 10, the CORBA overhead is relatively high but decreases significantly for increasing topology size.

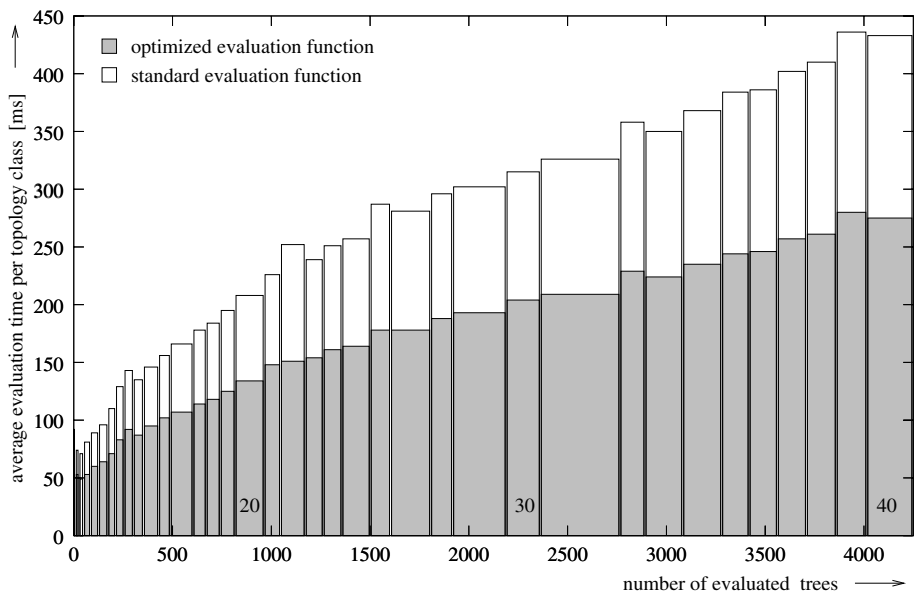


Fig. 3. Average evaluation time improvement per topology class: DAXML vs. parallel fastDNAML evaluation function

In order to demonstrate the efficiency and soundness of **LMC** we performed test runs using worker object replication and migration.

Figure 5 depicts the correct response of **LMC** to an increase of background load on a worker object host. We performed two test runs with 40 sequences and a single worker object, i.e. the replication mechanism was switched off, located on the same initially unloaded node and measured the evaluation time per topology.

Around the evaluation of the 1750th tree topology during the first test run we produced external load on the worker object host, which lead to a significant increase in topology evaluation time. The unfavorable situation is correctly resolved by the load balancer and a migration of the worker object to an unloaded host is performed.

Finally, Figure 6 demonstrates how the average evaluation time per topology class is progressively being improved by 3 subsequent automatic worker object replications performed by **LMC**, compared to a run with automatic replication switched off.

6 Future Work

Current work focuses mainly on building a seti@home-like [9] distributed phylogeny program based on the `http` protocol and the novel randomized/distributed tree inference algorithm described in [13]. A parallel MPI-based

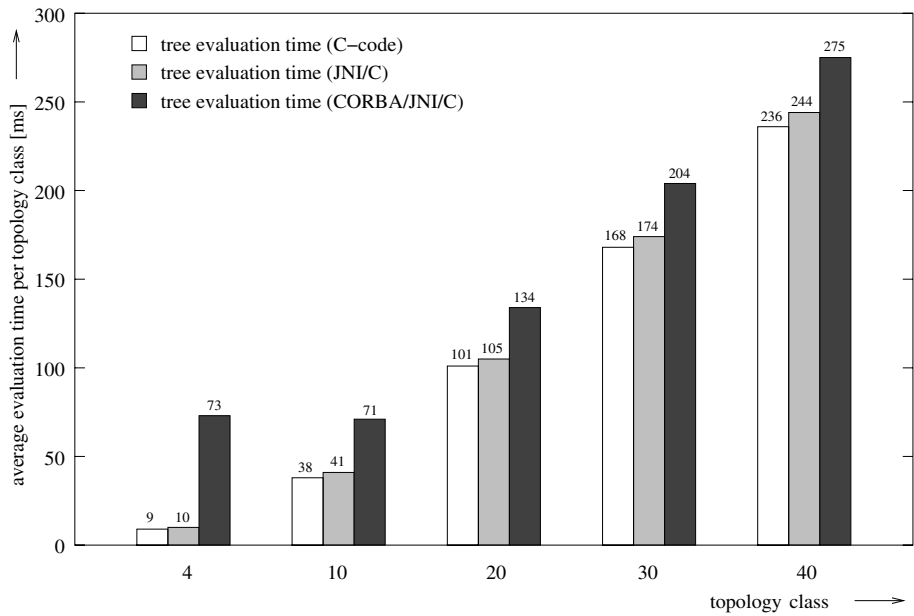


Fig. 4. JNI and CORBA-communication overhead

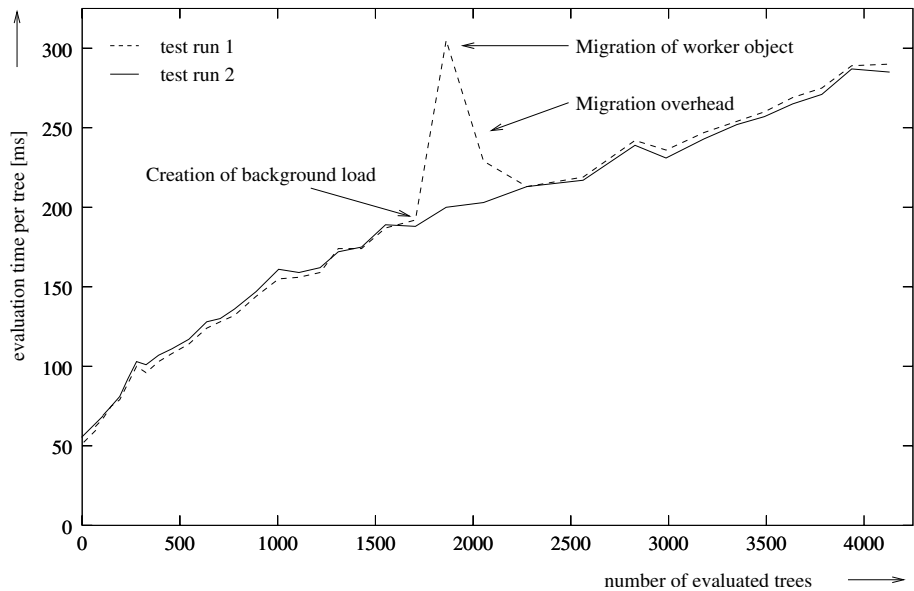


Fig. 5. Worker object migration after creation of background load on its host

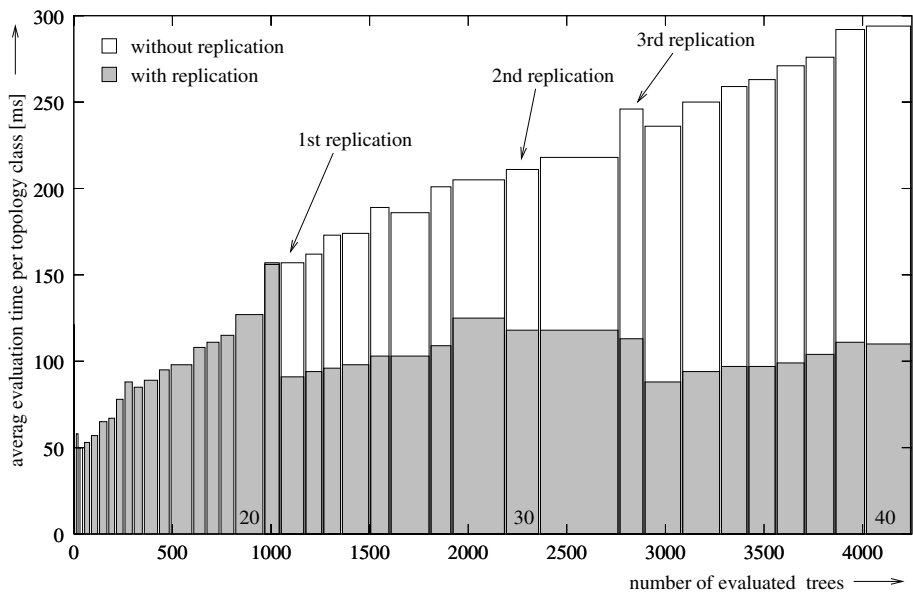


Fig. 6. Impact of 3 subsequent automatic worker object replications

prototype is already being evaluated. Within this context we plan to run big distributed phylogenetic tree calculations with data sets from the ARB database, using the available resources at the TUM. Furthermore, we will work on further improving our randomized tree inference algorithm, by extracting additional information from the set of trees calculated during the initial phase of the algorithm. Within this context we have already integrated the consensus tree program CONSENSE [4] into our parallel prototype.

References

1. Brose, G.: JacORB: Implementation and Design of a Java ORB. International Conference on Distributed Applications and Interoperable Systems (DAIS'97). Chapman & Hal (1997)
2. Felsenstein, J.: Evolutionary trees from DNA sequences: A maximum likelihood approach. *J. Mol. Evol.*, Vol. 17. (1981) 368–376
3. Korber, B., Muldoon, M., Theiler, J., Gao, F., Gupta, R., Lapedes, A., Hahn, B.H., Wolinsky, S., Bhattacharya, T.: Timing the ancestor of the HIV-1 pandemic strains. *Science*, Vol. 288. (2000) 1789–1796
4. Jermini, L.S., Olsen, G.J., Mengersen, K.L., Easteal, S.: Majority-rule consensus of phylogenetic trees obtained by maximum-likelihood analysis. *Mol. Biol. Evol.*, Vol. 14. (1997) 1297–1302

5. Lindermeier, M.: Load Management for Distributed Object-Oriented Environments. Proceedings of 2nd International Symposium on Distributed Objects and Applications (DOA'00). IEEE Computer Society, (2000) 59–68
6. Lindermeier, M.: Ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen (A concept for load management in distributed object-oriented systems). Ph.D. thesis. Technical University of Munich (2002)
7. Olsen, G.J., Matsuda, H., Hagstrom, R., Overbeek, R.: fastDNAm1: A tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Comput. Appl. Biosci.*, Vol. 10. (1994) 41–48
8. **ParBaum** homepage, **PxML** download:
<http://www.bode.in.tum.de/~stamatak/research.html>
9. Search for Extraterrestrial Intelligence at Home:
<http://setiathome.ssl.berkeley.edu/>
10. Stamatakis, A.P., Ludwig, T., Meier, H., Wolf, M.J.: **AxML**: A Fast Program for Sequential and Parallel Phylogenetic Tree Calculations Based on the Maximum Likelihood Method. Proceedings of 1st IEEE Computer Society Bioinformatics Conference (CSB 2002). IEEE Computer Society (2002)
11. Stamatakis, A.P., Ludwig, T., Meier, H., Wolf, M.J.: Accelerating Parallel Maximum Likelihood-based Phylogenetic Tree Computations using Subtree Equality Vectors. Proceedings of Supercomputing Conference (SC2002). IEEE Computer Society (2002)
12. Stamatakis, A.P., Ludwig, T., Meier, H.: Adapting **PxML** to the Hitachi SR8000-F1 Supercomputer. Proceedings of 1. Joint HLRB and KONWIHR Workshop. (2002)
13. Stamatakis, A.P., Ludwig, T.: Phylogenetic Tree Inference on PC Architectures with AxML/PxML. Proceedings of IPDPS2003, High Performance Computational Biology Workshop (HICOMB). IEEE Computer Society (2003)
14. Stewart, C.A., Hart, D., Berry, D.K., Olsen, G.J., Wernert, E., Fischer, W.: Parallel implementation and performance of fastDNAm1 – a program for maximum likelihood phylogenetic inference. Proceedings of Supercomputing Conference 2001 (SC2001). IEEE Computer Society (2001)
15. Stewart, C.A., Tan, T.W., Buchhorn, M., Hart, D., Berry, D., Zhang, L., Wernert, E., Sakharkar, M., Fisher, W., McMullen, D.: Evolutionary biology and computational grids. IBM CASCON 1999 Computational Biology Workshop: Software Tools for Computational Biology. (1999)
16. The ARB project: <http://www.arb-home.de>

D-SAB: A Sparse Matrix Benchmark Suite

Pyrrhos Stathis, Stamatis Vassiliadis, and Sorin Cotofana

Computer Engineering Laboratory, Electrical Engineering Department,
Delft University of Technology,
2600 GA Delft, The Netherlands
`{pyrrhos,stamatis,sorin}@dutepp0.et.tudelft.nl`

Abstract. In this paper we present the Delft Sparse Architecture Benchmark (D-SAB) Suite for evaluating sparse matrix architectures. The focus is on providing a benchmark suite which is flexible and easy to port on (novel) systems, yet complete enough to expose the main difficulties which are encountered when dealing with sparse matrices. The novelty compared to previous benchmarks is that it is not limited by the need for a compiler. The D-SAB comprises of two parts: (1) the benchmark algorithms and (2) the sparse matrix set. The benchmark algorithms (operations) are categorized in (a) value related operations and (b) position related operations.

1 Introduction

Dealing with sparse matrices has always been problematic in the scientific computing world. The reason for this, simply put, is that computers, and especially vector computers, are best in dealing with regularity. One of the problems associated with sparse matrices is the determination of a common way to evaluate new architectural features. In this paper we introduce D-SAB, a benchmark suite to be used in early architectural developments. The contributions of this paper can be summarized as follows:

- We propose the Delft Sparse Architecture Benchmark (D-SAB), a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB is not dependent on the existence of a compiler on the benchmarked system.
- Although keeping the code simple D-SAB maintain coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility of the implementation.
- Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing automatically generated matrices.

The remainder of the paper is organized as follows: In the next section, Section 2 we discuss previous work on the field, and give our motivation and goals for the development of D-SAB. Subsequently, in Sections 3 and 4 we describe the operations and the matrix collection that comprise the benchmark. Finally, in Section 5 we give some conclusions.

2 Previous Work, Motivation, and Goals

Up to now, several efforts have been made that address the problem of benchmarking the performance of various architectures on sparse matrix operations. Some of the most important are listed below:

- The **Perfect Club** [2] is a collection of 13 full applications from engineering and scientific computing written in Fortran. A number of these include code involving operations on sparse matrices, mainly linear iterative solvers.
- The **NAS Parallel Benchmarks** [1] are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics applications, consist of five kernels and three pseudo-applications and aim at providing a performance metric for both dense and sparse systems.
- **SparseBench: a Sparse Iterative Benchmark** This benchmark [6] uses common iterative methods, preconditioners, and storage schemes to evaluate machine performance on typical sparse operations. The benchmark components are: Conjugate Gradient and GMRES iterative methods, Jacobi and ILU preconditioners.
- **SPARK: A benchmark package for sparse computations** [9] (see also [10]) is a benchmark developed by Saad and Wijshoff to evaluate the behavior of various architectures on the field of sparse computations. The main rationale behind the SPARK approach for designing the benchmark is to try and capture the main kernels that expose the problems encountered in sparse computing.

All the above mentioned benchmarks (except the NAS benchmark) assume a fully functioning system including a compiler. Although this is a very useful approach that reflects real system settings and is easy to use, it cannot be used for architectures in an early stage of development that don't have a compiler or architectures that simply don't make use of a compiler. Furthermore, the benchmarks (except the NAS benchmark) define the storage method of the sparse matrices. Although the storage methods utilized are usually the most commonly utilized in the scientific world, this approach may fail to reveal the full potential of an architecture since the matrix format determines the way it will be accessed and processed. Therefore the flexibility of these benchmarks is limited and does not allow for a novel way of storing, accessing and operating on the sparse matrix. Additionally, the above benchmarks use matrices that are automatically generated. This is mainly done for reasons of memory efficiency. However, we believe that the parameters that are used to generate those matrices cannot capture the diversity of sparsity patterns that are observed in matrices obtained from actual applications.

Our proposed benchmark aims removing the above shortcomings of the currently existing sparse matrix benchmarks while keeping their benefits. Our benchmark is partly inspired by the NAS and SPARK benchmarks regarding their design philosophy.

3 The Benchmark Operations

To construct the benchmark we need to construct a set of algorithms that can cover the basic operations that make up most of the sparse matrix related applications. We have examined a number of toolkits and packages to extract the operations including the following: SPARSEKIT [8], The NIST Sparse BLAS [7, 4], LAPACK, and Sparselib++ [5].

We have observed that although most packages offer an extensive set of functions, there is a plurality of functions performing the same operation. After an initial analysis of the packages we have chosen to divide the basic operations in two parts:

- 1. *Value Related Operations* (VROs). These operations include arithmetic operations such as multiplication, addition, inner product, etc.
- 2. *Position Related Operations* (PROs). These include operations for which the actual values of the elements are not important for the outcome, such as element searching, element insertion, matrix transposition, etc.

We have chosen 5 operations from each of the VROs and PROs which we believe represent the most basic operations of sparse matrix applications and are listed in Tables 1 and 2 respectively.

Table 1. Value Related benchmark operations

Name	Operation	Description
1. Multiplication	$C = AB$	Multiplication of two sparse matrices. This operation has a high degree of value reuse and indicates <u>how a method can deal with this fact.</u>
2. Addition	$C = A + B$	Matrix addition exposes fill-in (i.e. the addition of extra nonzero elements in a sparse matrix)
3. SMVM	$y = Av$	Sparse Matrix - dense Vector Multiplication. This operation is one of the most important in sparse matrix computations in terms of execution time.
4. Gaussian Elimination, Pivoting	see text	Operations used in Direct Methods for linear system solving and the construction of preconditioners
5. (Bi)Conjugate Gradient	see Fig 1	(Bi)CG, 2 iterative solvers, typical sparse matrix applications. The main benchmark for most existing sparse matrix benchmarks

Operations 1 till 3 are self explanatory. Figure 1 depicts the code for the BiCG algorithm where x , p , z , q , $\tilde{p}$, $\tilde{z}$ and $\tilde{q}$ denote dense vectors and Greek letters denote scalars. The $\Rightarrow$ signs indicate the code lines of interest since they form the asymptotic execution of the code. Therefore only this code needs to

be executed for benchmarking. We have included the BiCG code along the CG code because it includes SMVM with both the A and A^T . Performing both in the same code is considered troublesome and is avoided in practice in spite the fact that algorithmically it can offer advantages. However we believe that this is precisely the reason to include it in our benchmark.

```

Compute  $r_0 = b - Ax_0$  using initial guess  $x_0$ 
 $\tilde{r}_0 = r_0$ 
for  $i = 1, 2, 3, \dots$ 
  if  $i = 1$ 
     $p_i = z_{i-1}$ 
     $\tilde{p}_i = \tilde{z}_{i-1}$ 
  else
 $\Rightarrow p_i = z_{i-1} + \beta p_{i-1}$ 
 $\Rightarrow \tilde{p}_i = \tilde{z}_{i-1} + \beta_{i-1} \tilde{p}_{i-1}$ 
  endif
 $\Rightarrow q_i = Ap_i$ 
 $\Rightarrow \tilde{q}_i = A^T \tilde{p}_i$ 
 $\Rightarrow \alpha_i = \rho_{i-1} / \tilde{p}_i^T q_i$ 
 $\Rightarrow x_i = x_{i-1} + \alpha_i p_i$ 
 $\Rightarrow r_i = r_{i-1} + \alpha_i q_i$ 
 $\Rightarrow \tilde{r}_i = \tilde{r}_{i-1} + \alpha_i \tilde{q}_i$ 
    check convergence; continue if necessary
end for

```

Fig. 1. The Non-preconditioned Bi-Conjugate Gradient Iterative Algorithm.

Pivoting and Gaussian elimination are operations that relate to the direct methods for solving linear systems as well as for constructing preconditioners for iterative methods. Several methods exist to perform these operations. For D-SAB we have chosen to use the most straightforward version described below:

```

for  $j = 0$  to  $n - 2$  do
 $\Rightarrow$  Find position ( $q$ ) of max abs in column  $C_j$ 
 $\Rightarrow$  if  $q <> j$  then exchange rows( $j, q$ )
  for  $i = j$  to  $n - 2$  do
     $R_{j+1} = R_{j+1} - a_{ij} a_{jj} R_j$ 
  end for
end for

```

where C_k denotes the k^{th} row, $R_k = (a_{1k}, a_{2k}, \dots, a_{nk})$, R_k denotes the k^{th} row, $R_k = (a_{k1}, a_{k2}, \dots, a_{kn})$ and a_{ij} the element of A at position (i, j) . The part of the algorithm that is used for pivoting is denoted by the $\Rightarrow$ sign.

Position Related Operations:

All ten named benchmarks are to be executed using the benchmark matrices that are listed in the following section. Wherever a second matrix is needed (i.e.

Table 2. Position Related benchmark operations

Name	Description
6. Sub-matrix Extraction	Create a new matrix from by extracting a sub-matrix from matrix A. Start from position (5,10) Use sizes 10x10, 100x100, 1000x1000, 10000x10000 if the original matrix size permits to do so.
7. Transposition (A^T)	Create a new matrix that is the transpose of the original.
8. Get element from matrix	Return the time needed to access an element in the matrix averaged over 50 values randomly chosen over the whole matrix. At least 10 should return a non-zero value.
9. Extract Lower Triangular Part	Create a new matrix that comprises only of the elements a_{ij} of matrix A where $i \geq j$.
10. Insert or Modify Element	Modify a non-zero Element in the matrix or Insert an element in the matrix (modify a zero entry) .

the B Matrix in Addition and Multiplication) we construct it as follows: The B matrix is the A matrix mirrored around the second diagonal, that is, the top right to bottom left diagonal. We have chosen to do so because the sparse matrix suits from which we have chosen our benchmark matrices do not offer pairs of matrices of the same dimensions. Therefore, to avoid the fill-in free addition of the matrix with itself we mirror the matrix at the diagonal where most of the matrices are not symmetric.

4 The Sparse Matrix Suit

The benchmark matrices for the D-SAB suite were chosen from a wide variety of matrices that are available from the Matrix Market Collection [3]. The collection offers 551 matrices collected from various applications and includes several other collections of sparse matrices and is therefore the most complete we could get access to. Of these matrices we have selected 132 matrices taking care not to select similar matrices in terms of application, size and sparsity patterns in order to reduce the number of matrices while keeping the variety intact. The 132 matrices matrices have been sorted using 3 different criteria that relate to various matrix properties. For an extensive discussion of the criteria refer to [11].

Sorting the matrices by the three named criteria resulted in three sets. From each of these sets ten matrices have been chosen to represent the set due to space limitation. The steps are constant in a logarithmic scale since we observed from the data that the distributions after sorting for each criterion was logarithmic

rather than linear. Therefore for instance for the matrix size criterion each matrix is approximately 3.5 times larger than the previous one. See [11] for the precise list of the used matrices.

5 Conclusions

In this paper we introduced the Delft Sparse Architecture Benchmark (D-SAB) suite, a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB does not depend on the existence of a compiler meant to map the benchmark code on the benchmarked system. Although keeping the code simple D-SAB maintain coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility for the way the operation is implemented. Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing synthetic matrices.

References

1. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
2. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
3. R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman & Hall.
4. S. Carney. A revised proposal for a sparse blas toolkit, 1994.
5. J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures, 1994.
6. J. J. Dongarra and H. A. Van der Vorst. Performance of various computers using standard linear equations software in a Fortran environment. *Supercomputer*, 9(5):17–30, Sept. 1992.
7. K. Remington and R. Pozo. Nist sparse blas: user’s guide, 1996.
8. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.
9. Y. Saad. Wijshoff: Spark: A benchmark package for sparse computations, 1990.
10. Y. Saad and H. Wijshoff. A benchmark package for sparse matrix computations. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 500–509, St. Malo, France, 1988.
11. P. Stathis, S. Vassiliadis, and S. Cotofana. D-sab: Delft sparse architecture benchmark, <http://ce.et.tudelft.nl/iiliad/d-sab/>, 2003.

DOVE-G: Design and Implementation of Distributed Object-Oriented Virtual Environment on Grid^{*}

Young-Je Woo and Chang-Sung Jeong

Department of Electronics and Computer Engineering,
Korea University

Anamdong 5-ga, Sungbuk-gu, Seoul 136-701, Korea
gossamor@snoopy.korea.ac.kr, csjeong@charlie.korea.ac.kr

Abstract. In this paper, we address the design and implementation of DOVE-G (Distributed Object-oriented Virtual Computing Environment on Grid). DOVE-G is designed to integrate application and Grid services by encapsulating not only application modules but also Grid services as DOVE-G objects, and hence supports an enriched parallel programming environment on Grid by enabling users to build a parallel program as a collection of concurrent DOVE-G objects. DOVE-G is built as a multi-layered runtime system which can support a unified programming view for application and Grid services. Each layer of DOVE-G is implemented in C++ class library, and interacts with one another using the interfaces in the class library to provide system modularity and extensibility.

1 Introduction

The explosive growth of the Internet and the availability of computing powers and high speed networks has led to the possibility of using networks of computers as a single unified resource, forming what is called Grid[1,4]. The Grid computing is started to link supercomputing sites, but now used to enable programmers and application developers to aggregate various resources scattered around the world in many several applications including distributed supercomputing, collaborative engineering, data exploration, high throughput computing[3,4,5]. Grid must be able to operate on top of the whole spectrum of current and emerging hardware and software technologies. A user of Grid does not want to be bothered with details of its underlying hardware and software infrastructure. A user is really only interested in running their application to the appropriate resources and getting the results back in a timely fashion on Grid. However, it is still difficult for application developers to efficiently make use of Grid due to the incompatibility between Grid services and commodity technologies.

^{*} This work has been supported by KIPA-Information Technology Research Center, University research program by Ministry of Information & Communication, and Brain Korea 21 projects in 2003

In this paper, we present the design and implementation of DOVE-G (Distributed Object-oriented Virtual Computing Environment on Grid) to address the problem. DOVE-G is designed to integrate application and Grid services by encapsulating not only application modules but also Grid services as DOVE-G objects, and hence supports an enriched parallel programming environment on Grid by enabling users to build a parallel program as a collection of concurrent DOVE-G objects. DOVE-G is built as a multilayered runtime system which can support a unified programming view for application and Grid services. Each layer of DOVE-G is implemented in C++ class library, and interacts with one another using the interfaces in the class library to provide system modularity and extensibility.

The outline of our paper is as follows: In section 2, we give related works. In section 3, we describe the design of DOVE-G, and in section 4, the implementation of DOVE-G. Finally, a conclusion will be given in section 5.

2 Previous Work

The programs that can be run on Grid is largely made in MPICH-G2. it communicates using not RMI of object-oriented programming but message passing mechanism. MPICH-G2 [9] provides a program launcher that executes programs on remote host on Grid. Its application is the same as one of a general MPI. So MPICH-G2 provides the only allocation services out of Grid interfaces. Besides MPICH-G2, there are diverse programming environments for Grid that have been developed, such as CONDOR-G, Legion, and so on. Condor-G [7] plays a part in Condor as a job manager using the Globus Toolkit(tm) [2] only to start the job on the remote machine. For this, Condor-G provides a "window to the Grid" with which users can both access resources and manage jobs running on remote resources. But Condor-G does not provide an application developer with the interface to Grid services else than resource allocation. Legion [8] presents a new idea called Grid OS in their community. it provides various services for Grid, but it is not simple to use.

The main purpose of DOVE-G usage is to provide a user with easy-to-use programming environment as well as parallelism encapsulated within distributed objects. The code for parallel program has little difference from sequential one, and an efficient parallelism is supported by the diverse method invocation schemes and the multiple method invocation to an object group. In addition to these features, heterogeneity, object group, object life management and naming service of object manager are supported to provide a transparent programming environment for parallel applications by integrating Dove objects onto Grid.

3 DOVE-G Design

In this section, we address the design of DOVE-G by first describing distributed object model and concurrency model, and then its architecture in detail.

3.1 DOVE-G Distributed Object Model

DOVE-G is based on distributed object model that consists of several distributed objects interacting with each other using method invocation mechanism (See Fig. 1). It models a parallel application as a collection of distributed objects which run concurrently at the same or different hosts to execute the subtasks partitioned from the given task.

Interface and implementation objects: DOVE-G object A contains two types of object : implementation object IM_A for A and interface object IN_B for other remote DOVE-G object B . The interface object IN_B provides an interaction point to its corresponding implementation object IM_B of B . Therefore, the implementation object IM_A makes a method invocation to remote other DOVE-G object B by calling a member function of the interface object IN_B of B as if it resides as local object.

Stub and skeleton objects: The interface and implementation objects are connected to stub and skeleton objects respectively. A method invocation to the interface object is converted to the invocation message in its stub object, and sent to the corresponding implementation object by DOVE-G runtime system. On the side of implementation object, this message is dispatched to invoke the matched member function of implementation object through the skeleton object. The reply message for the member function is sent from implementation object back to the stub object, and returned like a normal function call. This mechanism allows transparent access to the DOVE-G object irrespective of whether it resides in local or remote site.

Peer-to-peer paradigm: In a DOVE-G application, user can use the peer-to-peer programming paradigm as well as client-server programming paradigm. In other words, each DOVE-G object is associated with its own sub and skeleton objects, and by using them behaves either as client or server to interact with other DOVE-G objects.

3.2 DOVE-G Concurrency Model

For developing high-performance parallel applications, it is essential to exploit concurrency among the distributed objects. In DOVE-G, various synchronization schemes are provided for the enhancement of concurrency : synchronous, deferred synchronous and asynchronous method invocations. In synchronous method invocation, sender is blocked until the corresponding reply is arrived. In deferred synchronous call, the sender can proceed on immediately without awaiting the reply of the remote method invocation, and later at some point the process must wait for the reply in order to use the return values. In asynchronous method invocation, the sender can proceed without awaiting the reply similarly as in deferred synchronous one, but the it upcall function registered when this method invocation is invoked on the arrival of its reply. These communication types may be

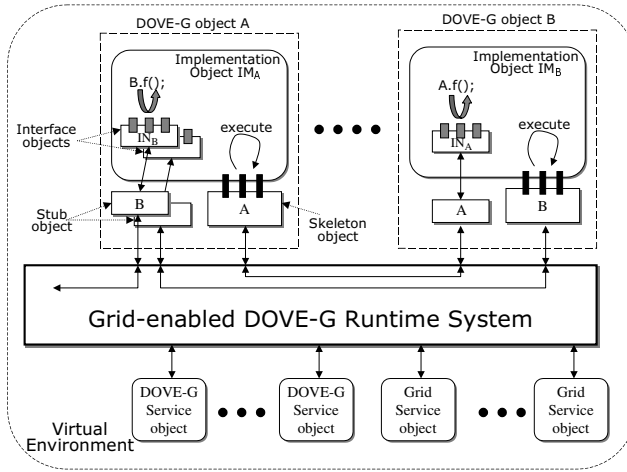


Fig. 1. DOVE-G Distributed Object Model

used to acquire high performance in distributed system. In later two synchronization schemes, communication and computation are overlapped.

In addition to the various synchronization schemes described in the previous subsection, DOVE-G supports another concurrency enhancement method by using the concept of object group. In distributed systems, group communication pattern is often used, since it provides simple and powerful abstraction for parallelism. Group communication pattern can be used in distributed object model using object group as a basic unit for the method invocation. A remote method invocation issued to an object group is transparently multicast to each object in the object group. A client stub for the object group has the same interfaces as the one to the single object, and provides an interaction point with multiple objects in the object group so that user treats it just like a single object. Therefore, the concept of object group allows users to do more simple programming, and to have chance to get better performance if the underlying communication layer supports multicasting facilities. Even though the communication layer does not support any multicasting functions, a method invocation to the object group can be emulated by iterative invocations to each object in the object group with some loss in performance. In DOVE-G, three types of multiple method invocations are supported for remote method invocation to the object group: *multicast/select*, *multicast/gather*, *scatter/gather* and *scatter/select*. A *multicast/select* invocation is returned with only one reply that is obtained by applying selecting operations such as MIN and MAX to the replies from objects in the group. A *multicast/gather* invocation multicasts the requests of same parameters to each member in the object group, while A *scatter/gather* and *scatter/select* invocation multicast the different requests to each member by storing them in the array. Both of *multicast/gather* and *scatter/gather* method invocations are

returned with arrays that store each reply from the members in the object group. A selecting function specified at method invocation is applied to this arrays. Object group is also used as a basic unit of inter-object synchronization. In table 1, all selecting functions for each multiple method invocation are shown. At *FIRST_m_ARRIVED*, m is no more than the number of member belong to its group object.

Table 1. Selecting functions for Multiple Method Invocation

Multiple method invocation	Selecting functions
<i>multicast/select</i>	FIRST_ARRIVED, MIN, MAX, AVERAGE, MEAN and SUM
<i>multicast/gather</i>	FIRST_m_ARRIVED
<i>scatter/gather</i>	FIRST_m_ARRIVED
<i>scatter/select</i>	FIRST_ARRIVED, MIN, MAX, AVERAGE, MEAN and SUM

3.3 DOVE-G Architecture

In this section we describe a portable and flexible DOVE-G architecture which encapsulates Grid services into DOVE-G object, and hence provides a unified view of distributed object model for application and Grid environment as if they are merged into one single environment. This provide users with an easy-to-use transparent parallel programming environment on Grid by supporting efficient parallelism encapsulated and distributed over DOVE-G objects, while allowing to use various Grid services.

1) Application Layer

From a user point of view, an application program is composed of application objects interacting with each other. Application layer consists of application objects which is defined and coded by a user, while the other objects are provided by DOVE-G system.

2) DOVE-G Service Layer

DOVE-G service layer comprises object manager and monitor objects which provide various system services such as object creation, naming services and resource allocation between application layer and DOVE-G runtime system layer.

Object Manager: Object manager exists per host in D , and provides crucial services such as object creation and naming services, to build a transparent and easy-to-use computing environment. A set of object managers constitutes a single object group which determines the domain of the parallel program environment.

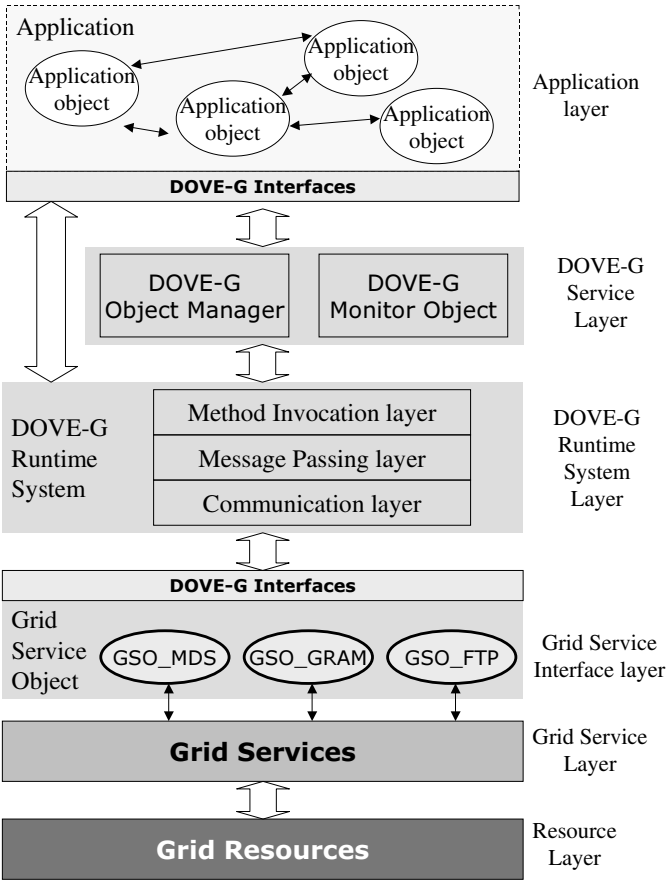


Fig. 2. DOVE-G Architecture

Monitor Object: Monitor object determines a set of hosts to be allocated in D by interacting with GSO_MDS which retrieves information from MDS server, and creates a object manager in each host of D by accessing GSO_GRAM at a time. Monitor object may change the number of hosts in D dynamically for load balancing according to the resource information collected through GSO_MDS.

DOVE-G integrates application and Grid services by encapsulating not only application modules but also Grid services as DOVE-G objects, and hence provides a unified view of object model as a collection of DOVE-G objects for application and Grid services. Therefore, DOVE-G runtime system can support a unified programming view by internally executing a Grid service using the interface object for its corresponding Grid Service Object. Moreover, DOVE-G application may also use a Grid service by simply generating a method invocation to Grid Service Object, and hence DOVE-G supports an enriched programming

environment which allows users to directly access to Grid services. However, most of the Grid services for a DOVE-G application are handled within DOVE-G runtime system in order to prevent the complicated programming burden from users.

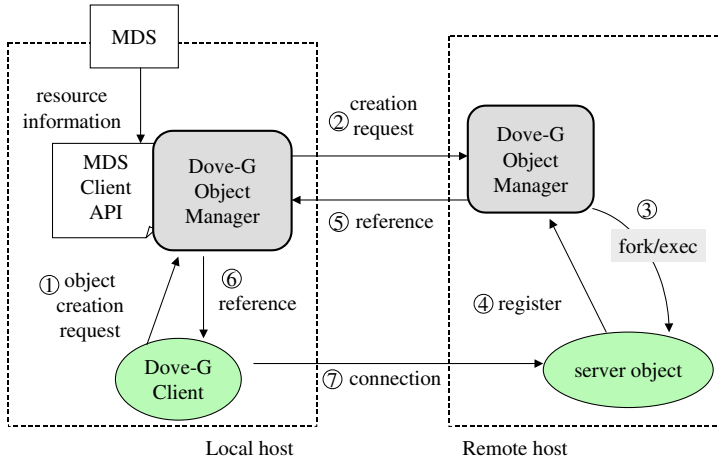


Fig. 3. Creation of DOVE-G Object

Object Creation: DOVE-G object is instantiated as a process by an object manager. When an interface object is newly created, its local object manager determines a remote host where its implementation object is created by retrieving the information about the available hosts which have been collected through GSO_MDS, and then automatically creates its corresponding implementation object on a remote site by cooperating with the other object manager at the remote site. After the implementation object is created, object reference of the object is returned to the interface object, and used to connect to the implementation object. The object reference represents the physical address of the implementation object. Internally, object reference is obtained from the name service of object manager, and used to connect to the existing implementation object (See Fig. 3).

Naming service: An object may have a name given by user, i.e. an alias represented by user-defined string when it is created. It is much easier and more user friendly to use an alias for object instead of its object reference. Every named object is identified by its object identifier which consists of its name and class name from which it is instantiated. Binding from object identifier to its

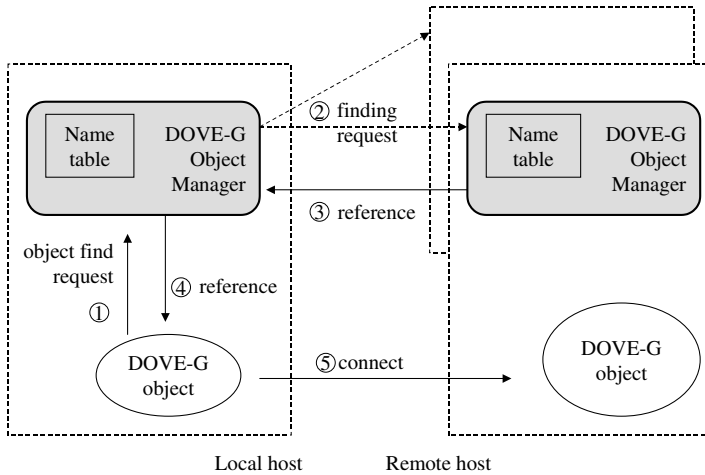


Fig. 4. Naming Service in DOVE-G

object reference is represented as simple triple: an object name, a class name, an object reference. Naming service makes remote objects appear to users as the ones in one virtual computer by encapsulating binding operations from users, and hence provides an easy-to-use programming environment. Object manager keeps track of binding information about objects and object groups on its local host. Each object has its local cache to store binding information about its currently accessing objects, and first consult its local cache for binding information (See Fig. 4). If it fails, it invokes the *get_binding()* method to its local object manager. If the object manager does not contain the binding information, it multicasts the *get_binding()* method to the object manager group to get the binding information. With this kind of hierarchical naming service where the binding information is distributed in the local cache, local object manager and remote object managers, DOVE can be more scalable.

3) DOVE-G Runtime System layer

DOVE-G system consists of three layers: method invocation layer, message passing layer and communication layer (See Fig. 5). It provides a set of interfaces that are platform independent by decoupling method invocation layer from communication layer.

Normally, a user program interacts with DOVE-G through stub object and skeleton object. Distributed objects are defined using Interface Description Language (IDL). An IDL compiler is developed to automatically generate codes for stub and skeleton objects. Automatic generation of stub and skeleton objects

including codes for marshal and unmarshal methods by IDL compiler provides users with an easy-to-use programming environment on a heterogeneous distributed system. User can make use of different interfaces for the same operation to support diverse method invocations and group method invocations. A method invoked through stub object is converted into an invocation structure which is then passed to method invocation layer.

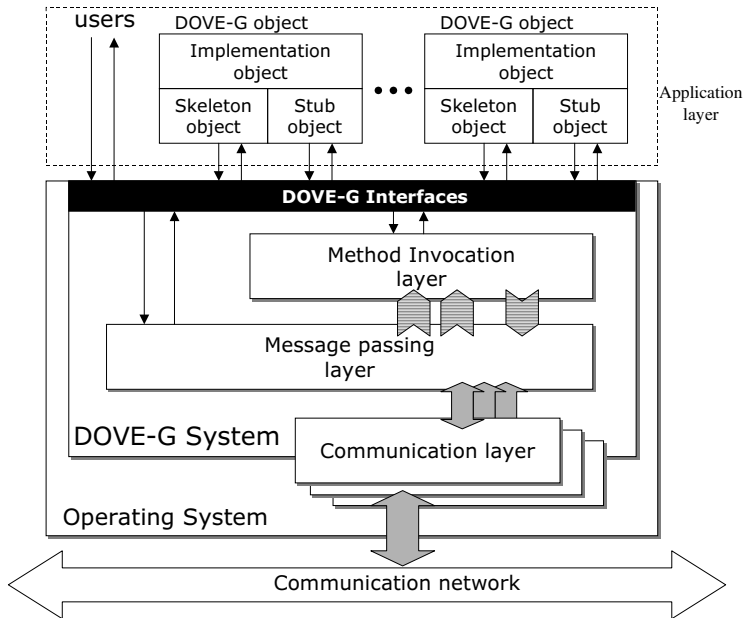


Fig. 5. Multi-layered Architecture of DOVE-G System

Method invocation layer marshals each invocation structure into an invocation message, and then passes it to message passing layer after registering it at invocation table in order to deal with the reply message properly. When it receives an invocation message from remote site, it creates a new thread for executing the method of the object which corresponds to the invocation message. When a reply message is returned to method passing layer, it is unmarshalled into an invocation structure, and then matched with the previously registered one by scanning through the invocation table to perform the proper action according to the semantic of the method invocation.

Message passing layer carries out object name binding to physical address, and delivers messages to distributed object using one of the underlying several communication layers. Each communication layer can be implemented using any distributed protocol which makes use of different interfaces and naming schemes.

The main purpose of message passing layer is to encapsulate method invocation layer from communication layer. Message passing layer behaves like an adaptor that can be plugged into specific communication layer, and decouples method invocation layer from communication layer in order to provide method invocation layer with uniform interfaces for message delivery and object group membership. Message passing layer stores information about group membership, and takes care of remote method invocation to the object group by using multicasting function if the communication layer supports it; otherwise by iterative execution of unicast function in the communication layer.

In addition, we add an acknowledgement message to remote invocation protocol. After receiving an invocation request from the client of a DOVE-G Object, a DOVE-G object immediately sends an acknowledgement message to its client. After the execution of the function, a DOVE-G object sends reply of the result of that function. The handling of invocation request, the sending of an acknowledgement message, and the reply of invocation are executed by extra thread in a DOVE-G object. So a DOVE-G object handles simultaneously many invocation requests from DOVE-G object clients.

In order to achieve the full functionality of DOVE-G system, the communication layer should be equipped with reliable unicast, reliable multicast and process group management. The minimal requirement for communication layer is reliable unicast likes TCP/IP. Currently, DOVE-G has installed two communication layers, one for reliable unicast using TCP/IP and another for reliable and totally ordered group communication using IP multicast. Each layer of DOVE-G system has its own thread of control and works concurrently. Multi-layered architecture of DOVE-G provides user with more extensibility, since each layer is implemented as an independent module which interacts with other layers through uniform interfaces.

4) Grid Service Interface Layer

This layer consists of several Grid service objects such as GSO.GRAM, GSO.MDS and GSO.FTP which directly interact with grid services offered by Globus toolkit [11]. GSO.GRAM handles a request for resource allocation by using GRAM service which allows a user to run a job remotely [3]. GSO.MDS provides the information about Grid resources by using MDS(Meta Directory Service). GSO.FTP transfer user programs or files to remote a host by exploiting GASS (Global Access of Secondary Storage) service. Therefore, the grid service objects provide grid services to other objects by encapsulating them internally.

4 DOVE-G Implementation

DOVE-G is composed of C++ class library for runtime system and server classes for DOVE-G service layer and Grid service layer. A user application is linked with the class library, and get necessary runtime system or Grid services from server classes. The class library provides a set of interfaces which can make an

application independent on the underlying platform including operating system and communication system.

Server classes for DOVE-G service layer comprises Monitor object and object manager, and server classes for Grid service layer consists of several Grid service objects : GSO_GRAM, GSO_MDS and GSO_FTP. Each Grid service object encapsulates Grid services internally by having interfaces to the Grid service offered by Globus Toolkit[2,4]. Globus toolkit establishes a software framework for Grid infrastructure by providing a meta computer toolkit, and becomes a de facto standard for Grid services. Grid Service objects are inherited from the same base class, **GridObject** which keeps a Grid service name and its version information. **GRAM** class has an interface of gatekeeper in Globus Toolkit, and generates a RSL(Resource Specification Language) for launching a job. **GSO_GRAM** class is inherited from **GRAM**, and make use of its interface to decouple gatekeeper interface from DOVE-G objects. **LDAP** class implements a client of general LDAP server with which MDS server of Globus Toolkit is designed. **GSO_MDS** class inherits from **LDAP**, and provide the information about status of Grid resources, to server objects such as Object Manager and Monitor object as well as DOVE-G application objects. **GridFTP** class implements a function of client for Grid FTP server. **GSO_FTP** class derives from **GridFTP**, and delivers the transfer request from DOVE-G object to Grid FTP server. since Grid Service objects are identical to DOVE-G application objects, they are easily generated using DOVE-G IDL, and allowed to interact with other DOVE-G application object directly, thus providing a unified view of application and Grid services.

C++ class library consists of classes for each layer of runtime system and classes for stub and skeletons. **MIL** class is designed for method invocation layer, **MPL** for message passing layer, and **Comm** for communication. **MIL** provides interfaces for invoking methods, returning replies, creating a thread for the execution of the implementation. **MPL** has a set of interfaces for asynchronous message passing between **MIL** and **Comm**. Class **Comm** encapsulates the underlying protocols for message passing layer so that **Comm** provides **MPL** a uniform interfaces for sending and receiving data.

DoveObject and **Servant** have been designed for stub and skeleton object respectively. **DoveObject** is a base class for stub objects. Every stub object for each type of distributed object must be derived from **DoveObject** class, which provides basic functions such as connecting to and disconnecting from distributed object, generating an invocation structure and invoking a remote method to the distributed object by passing the invocation structure to the method invocation layer. **Servant** is a base class for skeleton object. Every class for skeleton object should inherit **Servant** class, and redefine *dispatch()* method for proper execution of user-defined methods in distributed object. **Servant** class exports methods for activation and deactivation of distributed object.

Activated objects are registered into servant table in the method invocation layer. When a request message is received, it is matched with the previously registered one, and the *dispatch()* method is executed. DOVE-G IDL compiler generates code for stub and skeleton objects automatically. This stub and server

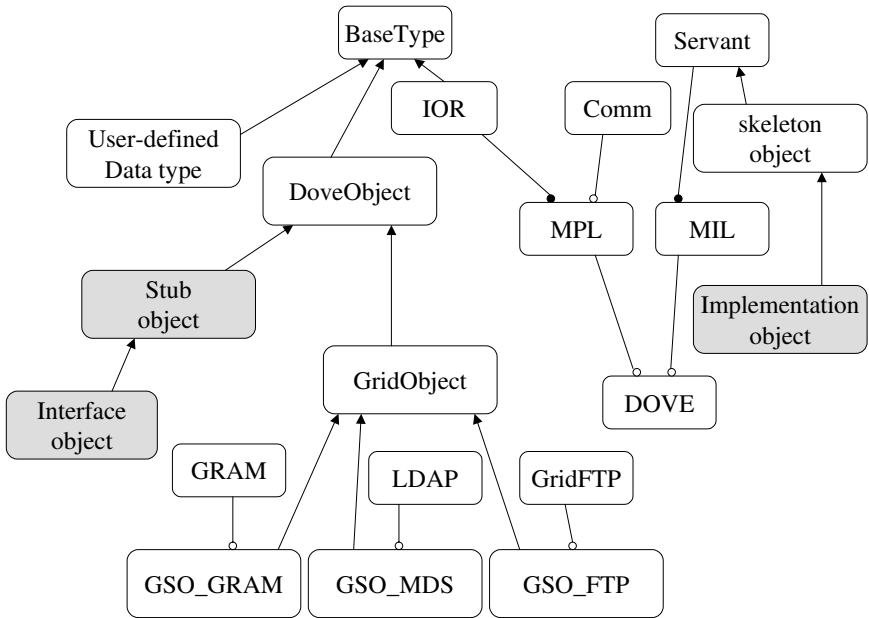


Fig. 6. Class hierarchy of DOVE-G

skeleton include codes for marshal and unmarshal methods, and so DOVE-G provides users with an easy-to-use programming environment on a heterogeneous distributed system. Since user and each layer of DOVE-G interacts with one another using the interfaces in the class library without directly accessing the underlying system, it can be built on various heterogeneous machine without any change in its implementation. Therefore, DOVE-G provides a portable and flexible virtual environment which can be easily extended and adapted to the new technology.

5 Conclusion

In this paper, we have presented the design and implementation of DOVE-G(Distributed Object-oriented Virtual Computing Environment on Grid). DOVE-G have been designed to integrate application and Grid services by encapsulating not only application modules but also Grid services as DOVE-G objects, and hence supports an enriched parallel programming environment on Grid by enabling users to build a parallel program as a collection of concurrent DOVE-G objects. DOVE-G incorporates a concurrency model to provide various synchronization schemes and object group for the enhancement of concurrency. DOVE-G is built as a multilayered architecture which consists of application

layer, DOVE-G service layer, DOVE-G Runtime layer, Grid service layer, and resource layer to provide system modularity and extensibility.

DOVE-G has been implemented in C++ class library and server classes, where the former consists of classes for each layer of runtime system and classes for stub and skeletons, and the latter for DOVE-G service layer and Grid service layer. The identical automatic generation of application objects, DOVE-G service object and Grid service objects by IDL allows application objects to interact with DOVE-G service objects and Grid service objects in the same way as other application objects, thus providing a unified view of the integrated environment comprising application, runtime services and Grid services. We believe that DOVE-G can be a powerful virtual programming environment for developing distributed/parallel applications on Grid.

References

1. I. Foster, C. Kesselman, S. Tuecke: "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, 15(3), 2001.
2. I. Foster, C. Kesselman: "Globus: A Metacomputing Infrastructure Toolkit," *Intl J. Supercomputer Applications*, 11(2):115-128, 1997.
3. I. Foster, A. Roy, V. Sander: "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation," 8th International Workshop on Quality of Service, 2000.
4. I. Foster, C. Kesselman: "The Globus Project: A Status Report," *Proc. IPPS. IPPS/SPDP'98 Heterogenous Computing Workshop*, pp. 4-18, 1998
5. H. J. Kim, S. U. Joe, C. S. Jeong, "Fast Parallel Algorithm for Volume Rendering and its Experiment on Computational Grid" ,*ICCS 2003*, June 2003.
6. H. D. Kim and C. S. Jeong, "Object Clustering for High Performance Parallel Computin," *Journal of Supercomputing*, 19, pp. 267-283, 2001.
7. Francesco Giacomini, Francesco Prelz, Massimo Sgaravatto, Igor Terekhov, Gabriele Garzoglio, and Todd Tannenbaum: "Planning on the Grid: A Status Report [DRAFT]," *Techical Report PPDG-20*, Particle Physics Data Grid collaboration, October 2002.
8. Anand Natrajan, Marty A. Humphrey, Andrew S. Grimshaw: "Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context," Presented at High Performance Computing Systems, June 2001
9. G. Mahinthakumar, F. M. Hoffman, W. W. Hargrove, and N. Karonis: "Multivariate Geographic Clustering in a Metacomputing Environment Using Globus," *Proc. SC99*, no page numbers available, Portland, OR, November 1999.
10. Globus Toolkits: <http://globus.org/toolkit/>
11. J. Frey, S. Graham, C. Kesselman: "Grid Service Specification. S. Tuecke, K. Czajkowski, I. Foster," Open Grid Service Infrastructure WG, Global Grid Forum, Draft 2, 7/17/2002.
12. T. Whitted, An Improved Illumination Model for Shaded Display, *Communication of ACM* 23, No. 6 (June 1980).
13. H. Chen, N. S. Flann, and D. W. Watson, Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm, *IEEE Transactions on Parallel and Distributed Systems* 9, No. 2 (February 1998).

Author Index

- Adutskevich, Evgeniya V. 1
Alevizos, Panagiotis D. 336
Alt, Martin 401
- Bandini, Stefania 10
Bandman, Olga 20
Bashkin, Vladimir A. 35
Bessonov, Oleg 345
Bischof, Holger 415
Bodei, Chiara 49
Bolvin, Hervé 185
Boutsinas, Basilis 336
Brand, Per 324
Braun, Terry A. 384
Bulić Patricio 429
- Calzarossa, Maria 197
Casavant, Thomas L. 384
Chaly, Dmitry J. 66
Chambarel, André 185
Chen, Chih-Ping 444
Choi, Jin-Young 180, 253
Cotofana, Sorin 549
- Defour, David 207
Degano, Pierpaolo 49
Dinechin, Florent de 207
Doroshenko, Anatoliy 452
- Fakas, George 304
Focardi, Riccardo 49
Fougère, Dominique 185, 345
- Gava, Frédéric 215
Gergel, V.P. 76
Germain, Cécile 528
Gladkikh, Petr 185
Goossens, Bernard 467
Gorlatch, Sergei 401, 415
Grelck, Clemens 230
Guštin, Veselko 429
Guzev, Vadim 236
- Ha, Soonhoi 482
Han, Zongfen 270
Haridi, Seif 324
- Hurson, Ali R. 276
Hwang, Sun-Chul 503, 509
- Il'in, V.P. 89
- Jeong, Chang-Sung 244, 503, 509, 555
Jeong, Jin-Lip 509
Jeun, Woo-Chul 482
Jin, Hai 270
- Kalinov, A. 497
Karganov, K. 497
Karpov, Yuri G. 100
Kee, Yang-Suk 482
Khatzkevich, V. 497
Khorenko, K. 497
Kim, Chang-Hoon 503
Kim, Hyung-Jun 244
Kim, Jin-Soo 482
Kim, Sung-Jae 253
Kitzelmann, Emanuel 415
Koshur, Vladimir Dmitrievich 394
Kuksheva, Elvira A. 354
Kwon, Yong-Won 244
- Lastovetsky, Alexey 117
Ledovskikh, I. 497
Lee, DongWoo 259
Lee, Tae-Dong 503, 509
Legalov, Alexander Ivanovich 394
Li, Guo 270
Likhoded, Nickolai A. 1
Lim, Joford T. 276
Lindermeier, Markus 538
Litvinenko, S.A. 89
Lomazova, Irina A. 35
Lopes, Luís 316
Loulergue, Frédéric 215
Ludwig, Thomas 538
- Malyshkin, Viktor E. 354, 519
Manzoni, Sara 10
Marques, Pedro 316
Massari, Luisa 197
Meier, Harald 538
Mirkes, Eugenij Moiseevich 394

- Morozov, D. 497
 Mostefaoui, Achour 130

 Nepomniaschaya, Anna S. 141
 Nikitin, Serguei A. 354

 Okol'nishnikov, Victor 524
 Ott, Michael 538

 Papadopoulos, George A. 291, 304
 Park, Jong-Koo 332
 Paulino, Hervé 316
 Pedretti, Kevin T. 384
 Pipan, Ljubo 429
 Popov, Konstantin 324
 Priami, Corrado 49
 Pritchett, Larry D. 276

 Ragozin, Dmitry 452
 Rajsbaum, Sergio 130
 Ramakrishna, R.S. 259
 Raynal, Michel 130, 151
 Reddy, Ravi 117
 Romanenko, A.A. 519
 Roux, Bernard 345
 Roy, Matthieu 130
 Rudometov, Sergey 524
 Ryu, So-Hyun 244

 Savchenko, S. 497
 Schamberger, Stefan 165
 Scheetz, Todd E. 384
 Scholz, Sven-Bodo 230

 Selikhov, Anton 528
 Serdyuk, Yury 236
 Silva, Fernando 316
 Simone, Carla 10
 Snytnikov, Alexei V. 354
 Snytnikov, Valery N. 354
 Sokolov, Valery A. 66
 Song, Sung-Keun 332
 Sotnikov, Dmitry 100
 Stamatakis, Alexandros P. 538
 Stathis, Pyrrhos 549
 Strongin, R.G. 76
 Sveshnikov, V.M. 89

 Tasoulis, Dimitris K. 336
 Tessera, Daniele 197
 Tiskin, Alexander 369
 Trivedi, Nishank 384

 Vasconcelos, Vasco 316
 Vassiliadis, Stamatis 549
 Vishnevsky, Mikhail Alexandrovich 394
 Vlassov, Vladimir 324
 Vrahatis, Michael N. 336
 Vshivkov, Vitalii A. 354

 Wierum, Jens-Michael 165
 Woo, Yong-Je 244
 Woo, Young-Je 555

 Yoo, Hee-Jun 180
 Youn, Hee-Yong 332