Lecture Notes in Computer Science 3834

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Dror Feitelson Eitan Frachtenberg
Larry Rudolph Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

11th International Workshop, JSSPP 2005
Cambridge, MA, USA, June 19, 2005
Revised Selected Papers

Springer

Volume Editors

Dror Feitelson
The Hebrew University, School of Computer Science and Engineering
91904 Jerusalem, Israel
E-mail: feit@cs.huji.ac.il

Eitan Frachtenberg
Los Alamos National Laboratory, Computer and Computational Sciences Division
Los Alamos, NM 87545, USA
E-mail: etcs@cs.huji.ac.il

Larry Rudolph
Massachusetts Institute of Technology, CSAIL
32 Vassar Street, Cambridge, MA 02139, USA
E-mail: rudolph@csail.mit.edu

Uwe Schwiegelshohn
University of Dortmund, Robotics Research Institute (IRF-IT)
44221 Dortmund, Germany
E-mail: uwe.schwiegelshohn@udo.edu

Library of Congress Control Number: 2005937592

CR Subject Classification (1998): D.4, D.1.3, F.2.2, C.1.2, B.2.1, B.6, F.1.2

ISSN 0302-9743
ISBN-10 3-540-31024-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-31024-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11605300 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 11th workshop on Job Scheduling Strategies for Parallel Processing. The workshop was held in Boston, MA, on June 19, 2005, in conjunction with the 19th ACM International Conference on Supercomputing (ICS05).

The papers went through a complete review process, with the full version being read and evaluated by an average of five reviewers. We would like to thank the Program Committee members for their willingness to participate in this effort and their excellent, detailed reviews: Su-Hui Chiang, Walfredo Cirne, Allen Downey, Wolfgang Gentzsch, Allan Gottlieb, Moe Jette, Richard Lagerstrom, Virginia Lo, Jose Moreira, Bill Nitzberg, and Mark Squillante. We would also like to thank Sally Lee of MIT for her assistance in the organization of the workshop and the preparation of the pre-conference proceedings.

The papers in this volume cover a wide range of parallel architectures, from distributed grids, through clusters, to massively-parallel supercomputers. The diversity extends to application domains as well, from short, sequential tasks, through interdependent tasks and distributed animation rendering, to classical large-scale parallel workloads. In addition, the methods and metrics used for scheduling and evaluation include not only the usual performance and workload considerations, but also considerations such as security, fairness, and timezones. This wide range of topics attests to the continuing viability of job scheduling research.

The continued interest in this area is reflected by the longevity of this workshop, which has now reached its 11th consecutive year. The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, 2862, and 3277 (and since 1998 they have also been available online).

Finally, we would like to give our warmest thanks to Dror Feitelson and Larry Rudolph, the founding co-organizers of the workshop. Their efforts to promote this field are evidenced by the continuing success of this workshop. Even though they are stepping down from the organization of the workshop, we hope they will continue to lend their expertise and contribution to the workshop and the field as a whole.

August 2005

Eitan Frachtenberg
Uwe Schwiegelshohn

Table of Contents

Modeling User Runtime Estimates

Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson

School of Computer Science and Engineering,
The Hebrew University, 91904 Jerusalem, Israel
{dants, etsman, feit}@cs.huji.ac.il

Abstract. User estimates of job runtimes have emerged as an important component of the workload on parallel machines, and can have a significant impact on how a scheduler treats different jobs, and thus on overall performance. It is therefore highly desirable to have a good model of the relationship between parallel jobs and their associated estimates. We construct such a model based on a detailed analysis of several workload traces. The model incorporates those features that are consistent in all of the logs, most notably the inherently modal nature of estimates (e.g. only 20 different values are used as estimates for about 90% of the jobs). We find that the behavior of users, as manifested through the estimate distributions, is remarkably similar across the different workload traces. Indeed, providing our model with only the maximal allowed estimate value, along with the percentage of jobs that have used it, yields results that are very similar to the original. The remaining difference (if any) is largely eliminated by providing information on one or two additional popular estimates. Consequently, in comparison to previous models, simulations that utilize our model are better in reproducing scheduling behavior similar to that observed when using real estimates.

1 Introduction

EASY Backfilling [19, 21] is probably the most commonly used method for scheduling parallel jobs at the present time [7]. The idea is simple: Whenever the system status changes (a new job arrives or a running job terminates), the scheduler scans the queue of waiting jobs in order of arrival. Upon reaching the first queued job that can not be started immediately (not enough free processors), the scheduler makes a reservation on the job's behalf. This is the earliest time in which enough free processors would accumulate and allow the job to run. The scheduler then continues to scan the queue looking for smaller jobs (require less processors) that have been waiting less, but can be started immediately without interfering with the reservation. The action of selecting smaller jobs for execution before their time is called *backfilling*.

To use backfilling, the scheduler must know in advance the length of each job, that is, how long jobs will run.[1] This information is used when computing the reservation time (requires knowing when processors of currently running

[1] This is true for any backfilling scheduler, not just EASY.

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 1–35, 2005.

jobs will become available) and when determining if a waiting job is eligible for backfilling (must be short enough so as not to interfere with the reservation). As this information is not generally available, users are required to provide runtime estimates for submitted jobs. Obviously, jobs that violate their estimates are killed. This is essential to insure that reservations are respected. Indeed, backfilling is largely based on the assumption that users would be motivated to provide accurate estimates, because jobs would have a better chance to backfill if the estimates are tight, but would be killed if the estimates are too short.

However, empirical investigations of this issue found that user runtime estimates are actually rather inaccurate [21]. Results from four different installations are shown in Fig. 1 (Section 4 discusses the four presented workloads in detail). These graphs are histograms of the estimation accuracy: what percentage of the requested time was actually used. The promising peak at 100% actually reflects jobs that reached their allocated time and were then killed by the system according to the backfilling rules. The hump near zero was conjectured to reflect jobs that failed on startup, based on the fact that all of them are very short (less than 90 seconds). The rest of the jobs, that actually ran successfully, have a rather flat uniform-like histogram.

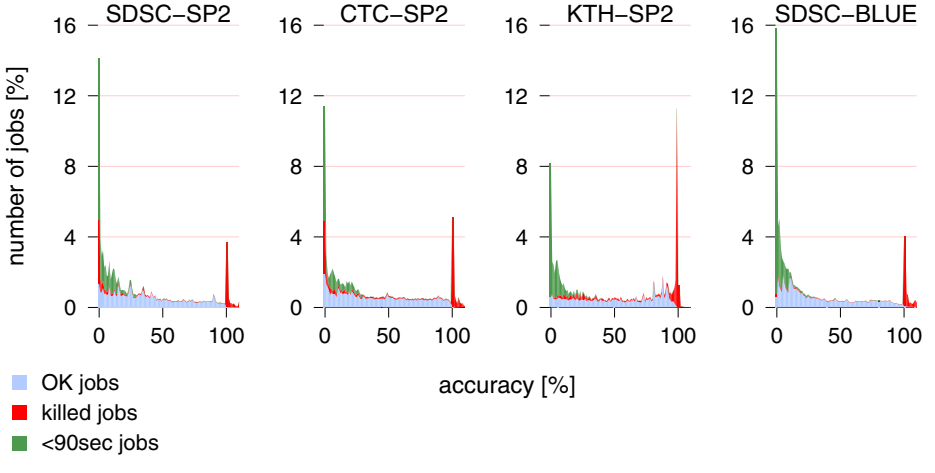


Fig. 1. Accuracy histogram of user runtime estimates: $accuracy = 100 \times \frac{runtime}{estimate}$

The issue of user runtime estimates has since become the focus of intensive research. A number of studies have suggested that inaccurate runtime estimates are actually good, as they provide the scheduler with more flexibility and eventually lead to better performance; as a result, it was even proposed to simply double the user runtime estimates before using them [29, 21], or further, randomizing them [22]. In contrast, other studies contend that accurate runtime estimates are actually better, as they can lead to even better performance if used correctly, e.g. by scheduling in some SJF (shortest job first) based order

[14, 23, 1, 25]. Still other studies have shown that the accuracy of user estimates can have non-trivial effects on the results of performance evaluations [8].

1.1 Motivation

All this activity spurred a search for ways to model user runtime estimates. Such a model is needed for three reasons. First, it is useful as part of a general workload model that can be used to study different job scheduling schemes, e.g. by means of simulation. Second, it is often the case that existing log files from production systems (used to drive simulations) are missing this information; a model can help in artificially manufacturing it. Third, a model may provide insights that will be useful in the study of whether and how the inaccuracy of estimates may be exploited by the scheduler.

We would like to make it clear that this paper targets the first two reasons mentioned above, that is, we aim to model and reflect reality, not to make it better. Indeed, in a different study, we show how backfilling schedulers can produce and utilize better runtime predictions that dramatically improve performance [25]. But even this novel technique often relies on user estimates under various conditions. Additionally, recall that user estimates have a role that is different than just serving as approximated runtimes, as they are also part of the user contract: the system guarantees a job will never be killed before its user estimate is reached. Consequently, system generated predictions (or other conceivable future mechanisms that are similar) can't "just" replace estimates.

At the same time, estimates ensure that jobs will indeed be killed at some point. Systems with no user estimates at all (that is, no runtime upper bound) are also undesirable, as these will allow jobs to run indefinitely, potentially overwhelming the system. At the very least we would expect users to choose some runtime upper-bound from a predefined set of values. However, this scenario is rather similar to reality, in which most users are already limiting themselves to very few canonical "round" estimates (as will be shown below), and jobs that exceed their estimates are immediately killed. It turns out there is actually no fundamental difference between allowing users to choose "any value", or from within a limited set.

Therefore, regardless of any possible scheduling improvements or changes, it seems a parallel workload model will not be complete if realistic user estimates are not included. Importantly, we will show that systems perform better if real user estimates are replaced with artificial ones, generated by existing models. This uncaptured "badness" quality of real user estimates constitutes a serious deficiency of existing models, as the purpose of these is to reflect reality, not to paint a brighter (false) picture. While counter intuitive, our goal in this paper is to produce estimates such that performance is worsened, not improved. Only when such a model is available, we can take the next step and consider ways to improve performance, based on a truly representative workload.

In the reminder of this section we survey the estimate models that have been proposed, and point out their shortcomings. This motivates the quest for a better model, which we propose in this paper.

1.2 Existing Models

The simplest possible model is to assume that user estimates are accurate. For example, such a model was used by Feitelson in [8]. This approach has two advantages: it is extremely simple, and it avoids the murky issue of how to model user estimates correctly. However, as witnessed by the data in Fig. 1, it is far from the truth.

A generalization of this model is to assume that a job's estimate is uniformly distributed within $[R, (f+1)R]$, where R is the job's runtime, and f is some non negative factor (f can't be negative because jobs are killed once their estimates are reached). If $f = 0$, this means that the estimates are identical to runtimes; if $f = 4$, they are distributed between R and $5R$, with an average of $3R$. Arguably, higher f values model increasingly inaccurate users. This model, which we call the "f-model", was proposed by Mu'alem and Feitelson [11] and several variants of it were used to investigate the effects of inaccuracy [29, 21, 1]. It was also used by several researchers in simulations using workloads that did not contain estimates data [13, 8]. The main problem with this model is that the estimates it creates are overly correlated with the real runtimes, so it actually gives the scheduler considerable amount of valuable information that is unavailable when real user estimates are used. In particular, it enables the scheduler to effectively identify shorter jobs and select them for backfilling, leading to SJF-like behavior. For example, under this model, a one-hour job will always appear longer than a one-minute job (in reality, this is often not the case). This leads to better performance results than those observed when using real user estimates.

A third model, also proposed by Mu'alem and Feitelson, attempts to reproduce the histograms of Fig. 1. These flat histograms imply that $R/E = u$, i.e. that the ratio of the actual runtime R to the estimate E can be modeled as a uniformly distributed random variable ($u \in [0, 1]$). By changing sides we find that given a runtime R divided by u results in an artificial estimate E. While unrelated to the actual user estimate for this particular job, this is expected to lead to the same general statistics of all the estimates taken together. The model also created the peak at 100% and the hump at low values. Finally, if E came out outrageous (because u happened to be very small), it was truncated to 24 hours. This was called the "ϕ-model" by Zhang et al. [27] (ϕ denoted the fraction of jobs in the 100% peak), who used it in various simulations.

The problem with this model is that it is missing a "hidden" factor which is often overlooked: that all production installations have a limit on the maximal allowed runtime. For example, on the SDSC SP2 machine this limit is 18 hours. Naturally, the limit also applies to estimates, as it is meaningless to estimate that a job will run for say 37 hours if all jobs are limited to 18 hours.

Consider Fig. 2 which displays the average accuracy of jobs grouped to 100 equally sized bins according to their runtime, for four different production traces. It has previously been conjectured that the apparent connection between longer runtimes and increased accuracy, is because the more a job progresses in its computation, the grater its chances become to reach successful completion [3]. However, this false hypothesis ignores the existence of a maximal allowed runtime,

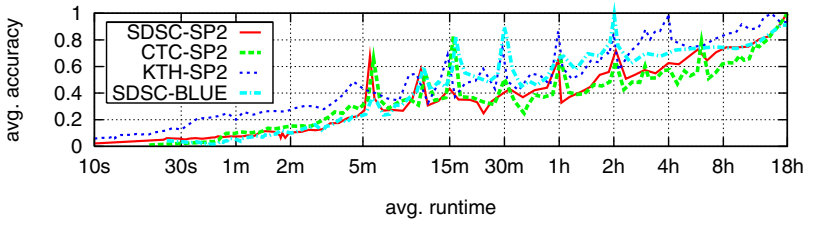


Fig. 2. Average accuracy as a function of jobs' (binned) average runtime

which suggests long jobs are guaranteed to have high accuracy. For example, if a job runs for 17 hours, its estimate must be in the range of 17 to 18 hours, so it's using at least 94.4% of its estimate. In other words, in contrast to the underlying assumption of the ϕ-model, the distribution of jobs in the accuracy histogram (Fig. 1) is not uniform. Rather, long jobs must be on the right, where accuracy is high, while short jobs tend to be on the left, at lower accuracies.

A fourth rather similar model was proposed by Cirne and Berman [3], which took the opposite direction in comparison to the previous model and chose to produce runtimes as multiples of estimates and accuracies, while generating direct models to the latter two. This decision was based on the argument that accuracies correlate with estimates less than they do with runtimes. In their model, accuracies were claimed to be well-modeled by a gamma distribution (this seems to be the result of trying to model the uniform part of the histogram along with the hump at low accuracies, by using one function for both). Estimates were successfully modeled by a log-uniform distribution. This methodology suffers from the same problem as the previous model, because accuracy is again independent of runtime. In addition, this model is not useful when attempting to add estimates to existing logs that lack them, or to workloads that are generated by other models which usually include runtimes and lack estimates [10, 6, 15, 20].

In addition to the per-model shortcomings mentioned above, there are two drawbacks from which all of them collectively suffer: The first is lack of repetitiveness: The work of users of parallel machines usually takes the form of bursts of very similar jobs, characterized as "sessions" [8, 28]. In the SDSC-SP2 log for example, the median value of the number of different estimates used by a user is only 3, which means most of the associated jobs look identical to the scheduler. It has been recently shown that such repetitiveness can have decisive effect on performance [26]. The second shortcoming is a direct result of the first: estimates form a modal distribution composed of very few values, a fact that is not reflected in any existing model. This is further discussed in the next section.

The conclusion from the above discussion is that all currently available models for generating user estimates are lacking in some respect. Consequently, using them in simulations leads to performance results that are generally unrealistically better than those obtained when real user estimates are used. Our goal in this paper is to capture the "badness" of real user estimates by finding a model that matches all known information about them: their distribution, their connection with each job's runtime, and their effect on scheduler performance.

2 Modality

We require a model capable of generating realistic user estimates. The usual manner in which such problems are tackled is by fitting observed data to well known distributions, later to be used for producing artificial data. To some extent, this methodology is applicable when modeling estimates, which appear to be well captured using the log-uniform distribution [3] as shown in Fig. 3.

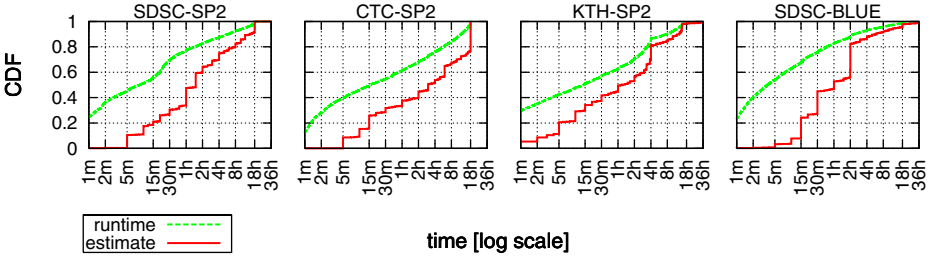


Fig. 3. Runtime and estimate CDFs (cumulative distribution functions) of the four workload traces. Runtime-curves are much higher than estimate-curves because runtimes are much shorter than estimates. For example, in CTC, 40% of the estimates are shorter than one hour (60% are longer), while for runtimes the situation is reversed (only 40% are longer than one hour).

The difficulty lies in that user estimates embody another important characteristic: unlike runtimes, they are inherently modal [21, 2, 17], because users tend to repeatedly use the same "round" values (e.g. five minutes, one hour, and so on). This is reflected in the staircase-like estimate curves of Fig. 3, in which each mode corresponds to a popular estimate value.

In particular, note the significant modes located at the maximal estimate of each trace, where the runtime and estimate curves finally meet (in Section 4 we will see that 4h and 2h effectively serve as the maximal estimate values of KTH-SP2 and SDSC-BLUE, respectively). Evidently, the maximal estimate is always a popular value. For example, this value is used by a remarkable 24% of CTC jobs. This phenomenon probably reflects users' lack of knowledge or inability to predict how long their jobs will run, along with their tendency to "play it safe" in the face of strict system policy to kill underestimated jobs.

In the context of job scheduling, this observation is quite significant, as maximal-estimate jobs are the "worst kind" of jobs in the eyes of a scheduler as they are too long to be backfilled. In fact, if all jobs chose their estimates to be the maximal value, all backfilling activity would stop completely.[2]

The observation about the maximal estimate mode may also be applied, to some extent, on other (shorter) modes: Consider the scenario in which an SJF

[2] Except for when using the "extra" nodes, see [21] for details.

scheduler must work with estimates that are highly inaccurate. If these estimates nevertheless result in a relatively correct ordering of waiting jobs, performance can be dramatically improved (up to an order of magnitude according to [1]). However, if estimates are modal, many jobs look the same in the eyes of the scheduler, which consequently fails to prioritize them correctly, and performance deteriorates. In general, if the estimate distribution is dominated by only a few large monolithic modes, performance is negatively effected, as less variance among jobs means less opportunities for the scheduler to perform backfilling.

Modality is absent from existing estimate models. An immediate heuristic that therefore comes to mind when trying to incorporate modality, is to "round" artificially generated estimates (e.g. by one of the models described above) to the nearest "canonical" value: values smaller than 1 hour are rounded to (say) the nearest multiple of 5 minutes, values smaller than 5 hours are rounded to the nearest hour, and so on. Experiments have shown that this heuristic fails in capturing the badness of user estimates, and performance results are similar to those obtained before this artificial modality was introduced. Additionally, arbitrary "rounding" fails to reproduce the various properties of the estimate distribution, as reported in the following sections.

The fact of the matter is that modes have a different (worse) nature than produced by the above. For example, when examining the number of jobs associated with the most popular estimates, we learn that these decay in an exponential manner e.g. half of the jobs use only 5 estimate values, 90% of the jobs use 20 estimates values etc. In contrast, the decay of less popular modes obeys a power law. In fact, almost every estimates-related aspect exhibit clear "model-able" (that can be modeled) characteristics.

3 Methodology

The modal nature of estimates motivates the following methodology. When examining a trace, we view its estimate distribution as a series of K modes given by $\{(t_i, p_i)\}_{i=1}^{K}$. Each pair (t_i, p_i) represents one mode, such that t_i is the estimate-value in seconds (t for time), and p_i is the percentage of jobs that use t_i as their estimate (p for percent or popularity). For example, the CTC mode series includes the pair $(18h, 23.8\%)$ because 23.8% of the jobs have used 18 hours as their estimate. Occasionally, we refer to modes as *bins* within the estimate histogram. Note that $\sum_{i=1}^{K} p_i = 100\%$ (we are considering all the jobs in the trace). The remainder of this section serves as a roadmap of this paper, describing step-by-step how the $\{(t_i, p_i)\}_{i=1}^{K}$ mode-series is constructed.

3.1 Roadmap of This Paper

Each of the following paragraphs correspond to a section or two in this paper, and may contain some associated definitions to be used later on.

Trace Files. We build our model carefully, one component at a time, in order to achieve the desired effect. Each step is based on analyzing user estimates in traces from various production machines, in an attempt to find invariants that are not unique to a single installation. The trace files we used and the manipulations we applied on them are discussed in Section 4.

Mass Disparity. Our first step is showing that the modes composing the mode-series naturally divide into two groups: About 20 "head" estimate values are used throughout the entire trace by about 90% of the jobs. The rest of the estimates are considered "tail" values. This subject is titled "mass disparity" and is discussed in Section 5. We will see that the two mode groups have distinctive characteristics and actually require a separate model. Naturally, the efforts we invest in modeling the two are proportional to the mass they entail.

Number of Estimates. We start the modeling in Section 6 by finding out how many different estimates there are, that is, modeling the value of K. Note that this mostly effects the tail as we already know the head size ($\sim$20).

Time Ranks. The next step is modeling the values themselves, that is, what exactly are the K time-values $\{t_i\}_{i=1}^{K}$. The indexing of this ascendingly sorted series is according to the values, with t_1 being the shortest and t_K being the maximal value allowed within the trace (also denoted T_{max}). The index i denotes the *time rank* of estimate t_i. This concept proved to be very helpful in our modeling efforts. We also define the *normalized time* of an estimate t_i to be t_i/T_{max} (a value between 0 and 1). Section 7 defines the function F_{tim} that gets i as input (time rank), and returns t_i (seconds).

Popularity Ranks. Likewise, we need to model the mode sizes / popularities / percentages: $\{p_j\}_{j=1}^{K}$. This series is sorted in order of decreasing popularity, so p_1 is the percentage of jobs associated with the most popular estimate. The index j denotes the *popularity rank* of the mode to which p_j belongs. For example, the popularity rank of 18h within CTC is 1 ($p_1 = 23.8\%$), as this is the most popular estimate. We also define the *normalized popularity rank* to be j/K (a value between 0 and 1). Section 8 defines the function F_{pop} that gets j as input (popularity rank), and returns p_j, the associated mode size.

Mapping. Given the above two series, we need to generate a mapping between them, namely, to determine the popularity p_j of any given estimate t_i, which are paired to form a mode. Section 9 defines the function F_{map} that gets i as input (time rank) and returns j as output (popularity rank). Using the two functions defined above, we can now associate each t_i with the appropriate p_j. This yields a complete description of the estimates distribution. The model is then briefly surveyed in Section 10.

Validation. Finally, the last part of this paper is validating that the resulting distribution resembles the reality. Additionally, we also verify through simulation that the "badness" of user estimates is successfully captured, by replacing the original estimates with those generated by our model. The replacement activity

mandates developing a method according to which estimates are assigned to jobs (recall that an estimate of a job must be bigger than or equal to its runtime). This is done in Section 11. The paper is concluded in Section 12.

3.2 Input, Output, and Availability

As we go along, the number of *model parameters* accumulates to around a dozen. Most are optional and are supplied with reasonable default values. The only mandatory parameters are the number of jobs N (the number of estimates to produce), and the maximal allowed estimate value T_{max}. Another important parameter is the percentage of jobs associated with T_{max}, as this popular mode exhibits great variance and has decisive effect on performance. The *output of the model* is the series of the modes: how many jobs use which estimate.

The model we develop is somewhat sophisticated and involves several technical issues with subtle nature. As it is our purpose to allow simulations that are more realistic, the C++ source code of the model is made available for download from the parallel workload archive [9]. Its interface is composed of two function: The first gets a structure containing all the model parameters (all but two are assigned default values), and returns an array of K modes. The second function gets the mode array and another array composed of job structures (which includes ID and runtime). It than associates each job with a suitable estimate.

4 The Trace Files

The analysis and simulations reported in this paper are based on four accounting logs from large-scale parallel machines that are listed in Table 1. These are all the logs from the parallel workload archive [9] that contain information about user estimates and were available at the time we began this research (the DAS2 log, which also contains this data, was added since). Since traces span the past decade, were generated at different sites, by machines with different sizes, and reflect different load conditions, we have reason to believe consistent results obtained in this paper are truly representative.

The data in Table 1 relates to the original traces, their recommended "cleaned" version (excludes various non-representative anomalies [9, 26]), and a "sane" version. The latter applies a filter on "cleaned" logs to remove jobs that cannot be used in simulations (unknown size, runtime, or submission time). As our goal is providing a model for the sake of performance analysis through simulation, our modeling activity targets only sane jobs. In particular, the K column in Table 1 is related to the sane versions, as is all the data presented in this paper.

During the study we found that two of the sane logs need to be further manipulated to be useful in this context. The first is the SDSC log: We say an estimate mode is "owned" by a user if this estimate was exclusively used by only that user within the log. It turns out that user 106 is uniquely creative in

Table 1. The trace files. The variables M, U, X, and K are months duration, number of users, maximal estimate value, and number of estimate bins, respectively. SC stands for "supercomputer". BLUE relates to San-Diego's Blue-Horizon machine. The others are SP2 machines. See [9] for more details.

Abbrev.	Site	Start	End	CPUs	Number of jobs (N)			M	U	X	K
					original	cleaned	sane	*mon*	*usr*	*max*	*est*
SDSC-106	San-Diego SC Ctr.	Apr 98	Apr 00	128	73,103	59,332	53,673	24	428	18h	339
CTC	Cornell Theory Ctr.	Jun 96	May 97	512	79,302	77,222	77,222	11	679	18h	265
KTH4H	Swedish Royal Instit.	Sep 96	Aug 97	100	23,070	23,070	23,070	11	209	4h	106
BLUE	San-Diego SC Ctr.	Apr 00	Jun 03	1,152	250,440	243,314	223,407	32	468	36h	525
SDSC	San-Diego SC Ctr.	Apr 98	Apr 00	128	73,496	59,725	54,053	24	428	18h	543
KTH	Swedish Royal Instit.	Sep 96	Aug 97	100	28,490	28,490	28,490	11	214	60h	271

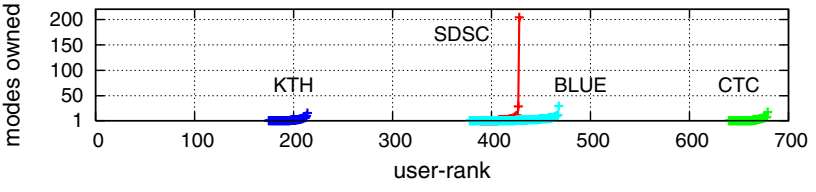


Fig. 4. Assume there are n users in a log. Users are associated with the number of modes they own m_i $(i = 1, ..., n)$ such that m_1 is the smallest and m_n is the biggest. The i index is defined to be the *user-rank* and serves as an X-value; m_i serves as the associated Y-value. Only positive m_i-s are displayed (users that own no modes are not shown). The SDSC outlier is associated with user 106 which is order of magnitude more "industrious" than other users, exclusively owning 38% of SDSC's modes.

comparison to others, owning 204 estimates of the 543 found in SDSC (38%). This is highly irregular[3] as shown in Fig. 4 which displays the number of modes owned by each user (only owners are shown). We therefore remove this unique activity from the log for the remainder of the discussion (regular activity of user 106, using estimates that are also used by others, is allowed to remain). The resulting log is called *SDSC-106*. This version is beneficial when modeling K in Section 6 (number of estimate modes) and F_{tim} in Section 7 (actual estimate time values). Other aspects of the model are not affected.

The other problematic workload was KTH: This log is actually a combination of three different modes of activity: running jobs of up to 4 hours on weekdays, running jobs of up to 15 hours on weeknights, and running jobs of up to 60 hours on weekends. We have found that in the context of user estimates modeling, considering these three domains in an aggregated manner is similar to, say, aggregating CTC and BLUE to be a single log. We therefore focused on only one of them — the daytime workload with the 4-hour limit, which is the largest component of the log. This will be denoted by *KTH4H*.

[3] In fact, as this activity is concentrated within about 2 months of the log, it actually constitutes a workload flurry [26].

Recall our claim that maximal estimate values are always popular (Fig. 3). We have argued that 4h and 2h are the effective maxima of KTH and BLUE, respectively. Obviously, this is the case for KTH (most of the time 4h is the maximum). As for BLUE, this machine had an "express" and "interactive" priority queues defined, with a limit of 2 hours on submitted jobs [9]. Indeed, the vast majority of 2-hours estimate jobs are from within these queues, which means here too users provided the maximal value available to them (while still allowing their jobs to be accepted to the higher priority queues).

5 Mass Disparity of Estimates

Examining the histogram of estimates immediately reveals that the distribution is highly modal (Fig. 3): A small number of values are used very many times, while many other values are only used a small number of times. In this section, we establish the mass disparity among estimate bins.

Human beings tend to estimate runtime with "round" or "canonical" numbers: 15 minutes, one hour etc. [21, 1, 17]. This has two consequences. One is that the number of bins in the histogram (K) is very small relative to the number of jobs in the trace (N). According to Table 1, N may be in the order of tens to hundreds of thousands, while K is invariably in the order of only a few hundreds.

The other consequence is that a small set of canonical bins dominates the set of values. Similar phenomena have been observed in many other types of workloads. They are called a "mass disparity", because the mass of the distribution is not spread out equally; rather, a small set of values gets a disproportionally large part of the mass [5].

The mass disparity of user runtime estimates is illustrated in Fig. 5. These are CDFs related to the bin size (the number of jobs composing a bin). In each graph, the top line is simply the distribution of bin sizes. This line grows sharply

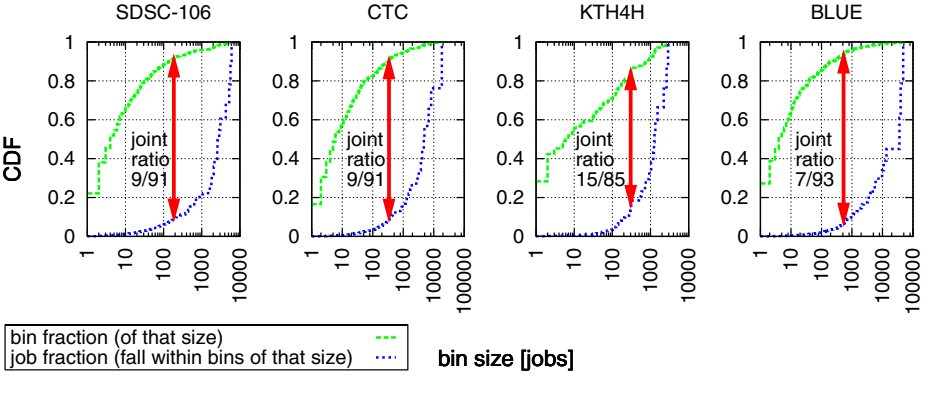


Fig. 5. Distributions of bins and of jobs, showing that a small fraction of the bins account for a large fraction of the jobs and vice versa. The actual fractions are indicated by the joint ratio, which is a generalization of the proverbial 10/90 rule.

Table 2. Mass disparity: per-log minimal number of estimate bins needed to cover the specified percent of the jobs

jobs	10%	50%	75%	90%	95%	98%	99%	100%
SDSC-106	1	6	12	22	39	77	116	339
CTC	1	4	10	22	36	62	89	265
KTH4H	1	6	12	21	28	36	43	106
BLUE	1	3	8	23	42	76	116	563
SDSC	1	6	12	23	43	91	156	543
KTH	1	8	21	41	60	89	122	270

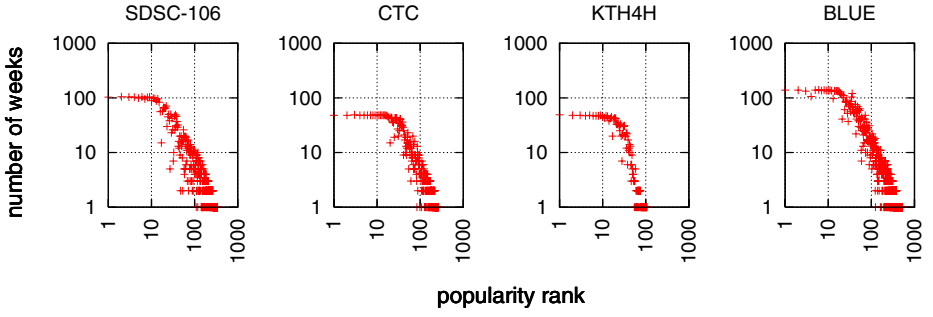


Fig. 6. Weeks in which an estimate appears, as a function of its popularity-rank. Recall that using popularity-ranks implies estimates are sorted on the X-axis from the most popular to the least. The top-20 most popular estimates appear throughout the logs.

at the beginning, indicating that there are very many small bins (i.e. values that are used by only a small number of jobs). The other line is the distribution of jobs, showing the fraction of jobs with estimates that fall into bins of the different sizes. This line starts out flat and only grows sharply at the end, indicating that most jobs belong to large bins (i.e. most estimate values are the popular values that are repeatedly used very many times).

The figure also shows the joint ratio for each case. This is a generalization of the well-know 10/90 rule. For example, the joint ratio of 9/91 for the CTC log means that 9% of the bins account for 91% of the jobs, and vice versa: the other 91% of the bins contain only 9% of the jobs. Further details about the shape of the distributions are given in Table 2. This shows the absolute number of bins involved, rather than their fraction; for example, the CTC row shows that a mere 4 bins cover 50% of the jobs, 10 bins cover 75% of the jobs, and 22 bins contain 90%. Indeed, a bit more than 20 head bins are enough to account for 90% of the jobs in all four logs.

"Head" bins dramatically vary in size: While the most popular is used by $10-25\%$ of the jobs, only $\approx 1\%$ use the 20-th most popular. Regardless, all head bins, whether large or small, have a common temporal quality: their use is not confined to a limited period of time. Rather, they are uniformly used throughout the entire log. This is shown in Fig. 6 that plots the number of

weeks in which estimates are used, as a function of their popularity ranks. The horizontal dot sequence associated with head bins indicates they are spread out evenly throughout the log. Further, the point of intersection between this sequence and the Y-axis is always the duration of the trace, e.g. for SDSC this is 2 years (a bit more than 100 weeks).

6 Number of Estimates

We have established that about 20 popular "head" bins represent about 90% of the jobs' estimate distribution mass. We are left with the question of modeling the number of the other "tail" bins used by the remaining 10%.

Examining the four traces of choice in Table 1, we see that K tends to grow with the size of the trace, where this "size" can be measured in various ways: as the number of jobs executed (N), as the duration of time spanned (M), as the maximal estimate (X), or as the number of different active users (U). Note that the U metric also measures size, as new users continue to appear throughout each log. This is relevant because after all, users are the ones generating the estimates. In fact, in each of the four traces of choice, about 40% of the estimate modes are exclusively owned (as defined above) by various users.[4]

We have experimented in modeling K as a function of the aspects mentioned above (individually or combined), and most attempts revealed some insightful observations. In fact, we are convinced K is the product of a combination of

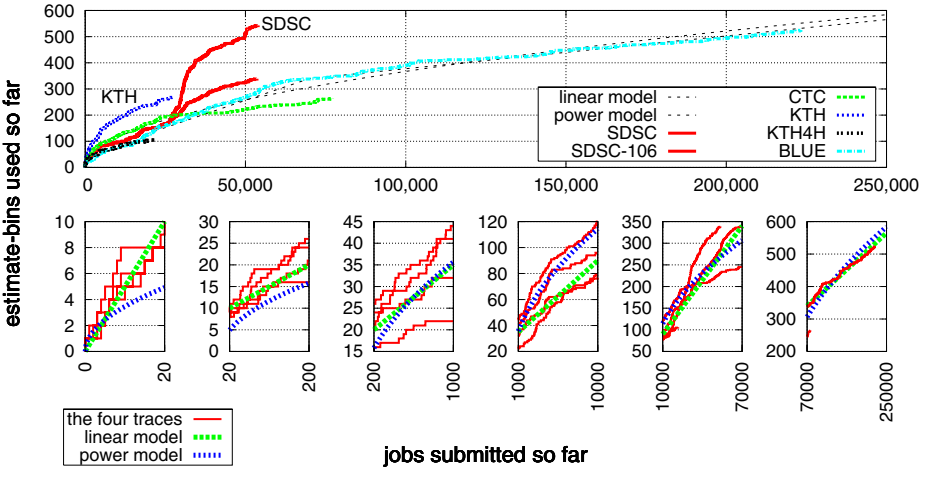


Fig. 7. Modeling K using a power model $K = \alpha N^{\beta}$ ($\alpha = 1.1$, $\beta = 0.5$) and a liner model which is defined by the points as specified in Table 3. In the top figure, curves associated with SDSC share the same texture (color), the higher is of SDSC-106.

[4] A surprising anecdote is that the actual number of bin-owners is also (exactly) 40, in three of the four traces.

Table 3. Points defining the linear model of K using N. The slope indicates the arrival rate of new estimates.

N (jobs)	0	20	200	1,000	10,0000	70,000	250,000
K (ests)	0	10	20	35	90	340	565
K/N (slope)		1/2	1/18	1/53	1/164	1/240	1/800

all factors, and that they all effect it to some degree. However, in the interest of being short while avoiding unwarranted complications (considering this only affects the tail of the distribution), we have chosen to model K as a function of N alone, which obtains tolerable results.

Fig. 7 plots K as a function of the number of jobs submitted so far (if n is an X value, its associated Y is the number of estimate bins in use, before the n-th job was submitted). Note how the vanilla version of KTH and SDSC stands out: the former due to the three estimate domains it contains, and the latter due to user 106. All curves can be rather successfully fitted with a power model on individual bases (we present one such power model that was simultaneously fitted against all four traces of choice). Accordingly, we allow the user of our model to supply the appropriate coefficients (as optional parameters). However, as this only effects tail bins, we set an ad-hoc linear model (defined by Table 3) as the default configuration. This provides a tolerable approximation of K for any given job number N.

7 Time Values of Estimates

Having computed a K approximation (order of a few hundreds), we know how many estimate bins should be produced by our model. Let us continue to generate these K values, namely manufacture the $\{t_i\}_{i=1}^{K}$ series. It has already been noted that users tend to give "round" estimates [21, 2, 17], but this loose specification is not enough. In this section we develop a simple method to generate K such appropriate values. We are currently not considering the most popular (20) estimates in a separate manner. These will be addressed in detail later on (Section 9), complementing the model we develop in this section.

Recall that the time-ranks of estimates are their associated *indexes*, when ascendingly numbered from shortest to longest. Evidently, this concept can be very helpful for our purposes. We define a function F_{tim} that upon a time-rank input i, return the associated time value t_i (seconds), such that $F_{tim}(i) = t_i$.

The top-left of Fig. 8 plots normalized estimate time (t_i/T_{max}, where T_{max} is the maximal estimate) as a function of its associated normalized time-rank (i/K), for all four traces. According to the top-right and bottom of Fig. 8, it turns out the resulting curves can be modeled with great success when using the fractional function $f(x) = \frac{(a-1)x}{a-x}$ for some $a > 1$ (x is normalized time-rank). Further, the actual values of a (Table 4) are correlated with K, in that bigger K implies smaller a.

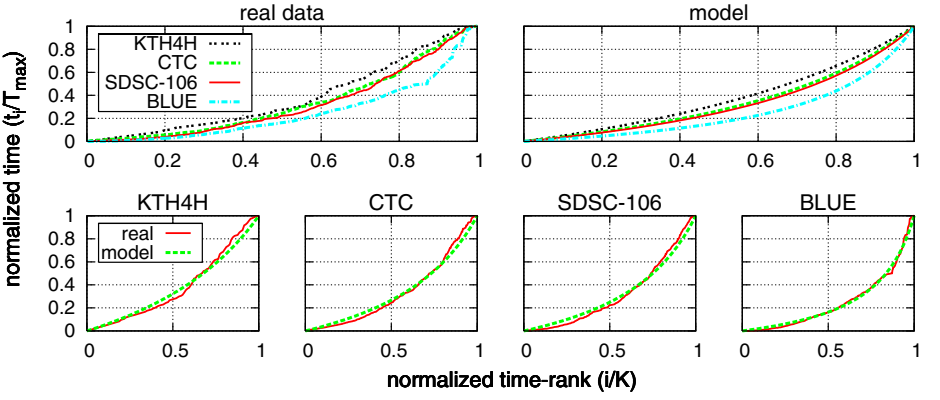


Fig. 8. Modeling estimate times using $f(x) = \frac{(a-1)x}{a-x}$

Table 4. The a parameter of the fractional fit presented in Fig. 8 is correlated with the number of different estimates (K)

trace	KTH4H		CTC		SDSC-106		BLUE
a	1.91	$>$	1.57	$>$	1.50	$>$	1.24
K	106	$<$	265	$<$	339	$<$	525

An obvious property of $f(x)$ in the relevant domain ($x \in [0,1]$) is that when a gets closer to 1, its numerator goes to zero and therefore the curve gets closer to the bottom and right axes. On the other hand, as a gets further from 1 (goes to infinity), its numerator and denominator get more and more similar, which means the function converges to $f(x) = x$ (the main diagonal). The practical meaning of this is that less estimate values (smaller K, bigger a) means estimates' temporal spread is more uniform. In contrast, more estimate values (bigger K, smaller a) means a tendency of estimates to concentrate at the beginning of the Y-axis, namely, be shorter.

In order to reduce the number of user-supplied parameters of our model, we can approximate a as a function of K (which we already know how to reasonably deduce from the number of jobs). The problem is that we only have four samples (Table 4), too few to produce a fit. One heuristic to overcome this problem is splitting the traces in two and computing K and a for each half. This enlarges our sample space by eight (two additional samples per trace) to a total of twelve. The results of fitting this data to the best model we could find are shown in Fig. 9 and indicate a moderate success.

We can now define the required F_{tim} to be

$$F_{tim}(i) = T_{max} \cdot f\left(i/K\right) = T_{max} \cdot \frac{(a-1)\frac{i}{K}}{a - \frac{i}{K}} \quad .$$

Generating the $\{t_i\}_{i=1}^{K}$ series of time values is done by simply assigning $1, 2, ..., K$ to the time-rank i in an iterative manner. Finally, as almost 100% of the estimates

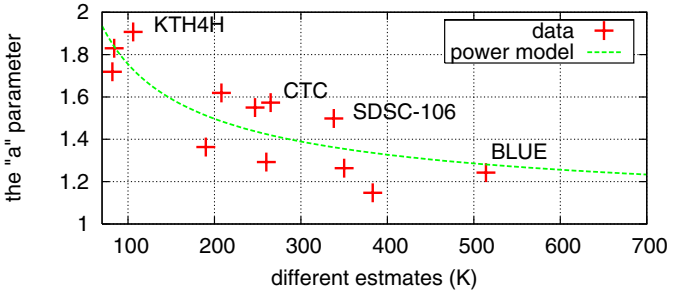


Fig. 9. Modeling a as a function of K using $1 + \alpha K^{\beta}$ (with $\alpha = 12.1$, $\beta = -0.6$). A bigger K results in an a-parameter that is closer (but never equal) to 1, as required.

are given in a minute resolution, the generated values are rounded to the nearest multiple of 60 (if not colliding with previously generated estimates).

8 Popularity of Estimates

In the previous section we have modeled the time values of estimates. Here we raise the question of how popular is each estimate, that is, how many jobs are actually using each estimate value? Answering this question implies modeling the $\{p_i\}_{i=1}^{K}$ percentage series. Once again, like in the previous section, ranking the estimates (this time based on popularity) proves to be highly beneficial. Recall that $\{p_i\}_{i=1}^{K}$ is descendingly sorted such that p_1 is the percentage of jobs using the most popular estimate value, p_i is the percentage of jobs using the i-most popular estimate value, and i serves as the associated popularity rank. We seek a function F_{pop} such that $F_{pop}(i) = p_i$. Note that the constraint of $\sum_{i=1}^{K} F_{pop}(i) = 100$ must hold.

Fig. 10 plots the size (percent) of each estimate bin, as a function of its popularity-rank. There's a clear distinction between the top-20 most popular estimates (distribution's head) and the others (tail), in that the sizes of head-bins decay exponentially, whereas the decay of the tail obeys some power law.

The suggested fits are indeed very successful ($R^2 > 0.95$ in both cases). However, when concentrating on the head (left or middle of Fig. 10), it is evident the exponential model is less successful for the first few estimates. For example, in CTC the most popular estimate is used by about 24% of the jobs, while in SDSC this is true for only 11%. In BLUE the situation is worse as the three top ranking estimates "break" the exponential curve. (Indeed, the exponential fit was produced after excluding these "abnormal" points.) Obviously, no model is perfect. But this seemingly minor deficiency (at the "head of the head") is actually quite significant, as a large part of the distribution mass lies within this part (differences in less popular estimates are far less important).

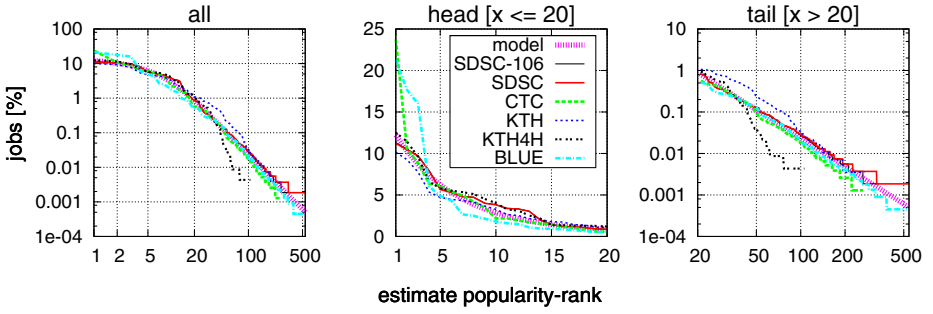


Fig. 10. Modeling percent of jobs associated with estimate bins, as a function of popularity rank. The head (middle) is modeled by the exponential function $\alpha e^{\beta x} + \gamma$ (with $\alpha = 14.05$, $\beta = -0.18$, and $\gamma = 0.46$). The tail (right) is modeled by the ωx^{ρ} power law (with $\omega = 795.6$ and $\rho = -2.27$). Note that the middle figure has linear axes, while the other two are log scaled. The left figure concatenates the head and tail models.

We note that the observed differences among the traces at the "head of the head" expose an inherent weakness in any estimate model one might suggest, because the effect of the variance among these 1-3 estimates is decisive. Consequently, our model will allow (though not mandate) the user to provide information regarding top-ranking estimates as model parameters (this will be further addressed in the next section). As for the default, recall that a job estimating to run for the maximal allowed value (T_{max}) is the worst kind of job in the eyes of a backfilling scheduler (Section 2). For this reason, we prefer the default model to follow the CTC example by making the (single) top ranking estimate "break" the exponential contiguity. This significant job percentage will later be associated with T_{max} to serve as a realistic worst case scenario. We therefore define F_{pop} as follows

$$F_{pop}(i) = \begin{cases} 89 - \sum_{j=2}^{20} \left(\alpha e^{\beta \cdot j} + \gamma\right) & i = 1 \\ \alpha e^{\beta \cdot i} + \gamma & i = 2, 3, ..., 20 \\ \omega \cdot i^{\rho} \cdot \frac{100-89}{\lambda} & i = 21, 22, ..., K \end{cases}$$

Starting with the (simplest) middle branch, F_{pop} is determined by the exponential model for all head popularity ranks but the first (the default values for the coefficients are specified in the caption of Fig. 10). The first branch is defined so as to preserve the invariant shown in Table 2 that the twenty top ranking estimates are enough to cover almost 90% of the jobs. Finally, the third branch determine sizes of tail estimates according to a power law (again, coefficient values are specified in Fig. 10). But to preserve the constraint that $\sum_{i=1}^{K} F_{pop}(i) = 100$, tail sizes are scaled by a factor of $\frac{100-89}{\lambda}$, where λ is the sum of the tail: $\sum_{i=21}^{K} \omega \cdot i^{\rho}$. The resulting default curve is almost identical to the one associated with the model as presented in Fig. 10, with a top rank of a bit more than 20% (to be associated with T_{max}).

9 Mapping Time to Popularity

The next step after separately generating the estimates' time $\{t_i\}_{i=1}^{K}$ and popularity $\{p_j\}_{j=1}^{K}$ is figuring out how two construct a bipartite matching between the two. We seek a function F_{map} such that $F_{map}(i) = j$, that is, we want to map each time-rank to a popularity-rank in a manner that yields an estimate distributions similar to those found in the original traces (Fig. 3).

9.1 Mapping of Tail Estimates

As a first step towards constructing F_{map}, let us examine this mapping as it appears in the four traces. Fig. 11 scatter plots normalized popularity-ranks vs. normalized time-ranks: one point per estimate.[5] The points appear to be more or less uniformly distributed, which means there is no apparent mapping rule.

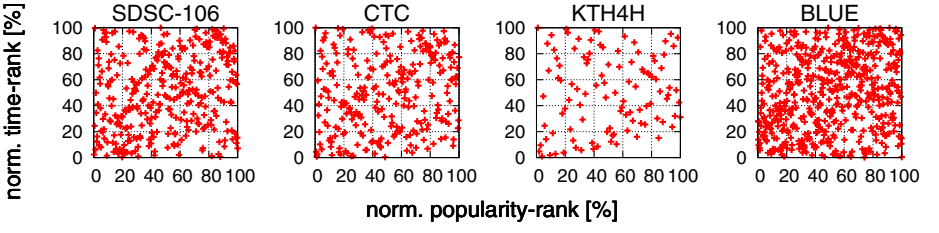


Fig. 11. Scatter plots of relative popularity-ranks vs. relative time-ranks appear to reveal a uniform distribution across all traces

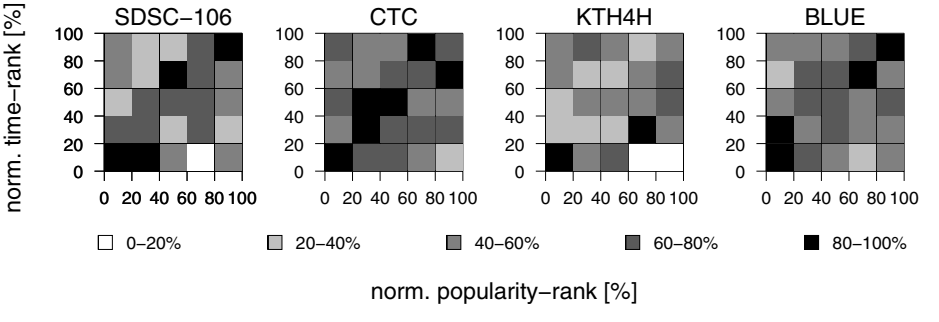


Fig. 12. Aggregating the data shown in Fig. 11 into a grid-based heat-map reveals no further insight, other than a consistent tendency of popular estimates to be short (bottom-left black cells)

In an effort to expose some trend possibly hidden within the "disorder" of the scatter plots, we counted the number of points in each grid-cell within Fig. 11.

[5] A scatter plot of actual values turns out to be meaningless.

We then generated an associated heat-map for each sub-figure by assigning a color based on the point-count of each cell: cells that are populated by 80-100% of the maximal (cell) point-count found within the sub-figure (denoted C), are assigned with black; cells populated by 0-20% of C are assigned with white; the remaining cells are assigned with a gray intensity that is linearly proportional to their point-count, batched in multiples of 20% of C.

The result, displayed in Fig. 12, appears to strengthen our initial hypothesis that the mapping between popularity-ranks and time-ranks is more or less uniformally random, as other than the bottom-left cell being consistently black (top-20 popular estimates show tendency of being shorter), there is no consistent pattern that emerges when comparing the different traces.

Our next step was therefore to randomly map between time and popularity ranks. Regrettably, this resulted in failure, as the generated CDFs were significantly different than those displayed in Fig. 3, because "big modes" fell in wrong places. The fact of the matter is that when (uniformly) randomly mapping between time and popularity ranks, there is a nonnegligible probability that the 4-5 most popular estimates are assigned to (say) times in the proximity of the maximal value, which means that the majority of the distribution mass is much too long. Alternatively, there is also a nonnegligible probability that the opposite will occur, namely, that none of the more popular estimates will be assigned to a time in the proximity of T_{max}, contrary to our previous findings.

We conclude that it is tail estimates (in terms of popularity) that are roughly randomly mapped to times in a uniform manner, forming the relatively balanced scatter plot observed in Fig. 11. This appearance is created due to the fact there are much more tail estimates (few hundreds) than head's (20). The head estimates minority, which nevertheless constitute 90% of the mass, distributes differently and requires a greater modeling effort.

9.2 Determining Head Times

We have reached the point where the effort to model user estimates is reduced to simply determining 20 actual time-values and mapping them correctly to the appropriate (head) sizes. In other words, our task is as simple as producing 20 (t_i, p_i) pairs. These are good news, as the number of samples is small enough to allow a thorough examination of the entire sample-space. The bad news is that unlike previous parts of the model that are actually relatively trivial, and in spite of considerable efforts we've made, we failed to produce a *simple* method to accomplish the task. In the interest of practicality and space, we do not describe our various unsuccessful attempts to produce a simple straightforward solution. Instead, we concentrate on describing the sophisticated algorithm we've developed that has finally managed to deliver satisfactory results.

Let us examine the relevant sample space. Table 5 lists the 20 most popular estimates in each trace, and their associated sizes (percent of jobs). Of the 36 values displayed, a remarkable 15 are *joint times* across all traces (we ignore KTH4H when deciding which values, bigger than 4h, are joint). The joint times are highlighted in bold font, and have values one would expect from humans

Table 5. The top-20 most popular modes in the four traces. Each column contains exactly 20 job percent values. Note that 15 of the top-20 estimates are joint across all traces (excluding KTH4H for estimates bigger than 4 hours). Joint estimates are highlighted in bold font. The parenthesized subscripts denote the associated popularity-ranks (e.g. in BLUE, 2h is the most popular value used by 21.3% of the jobs). Notice that the sum of each column is invariantly in the neighborhood of 89%, the value we used in Section 8 to define F_{pop}.

#	estimate *hh:mm*	SDSC-106	CTC	KTH4H	BLUE
1	00:01			6.6 (4)	
2	00:02			4.0 (10)	
3	00:03			2.2 (14)	
4	00:04			1.2 (20)	
5	**00:05**	11.3 (1)	8.8 (3)	11.5 (2)	2.7 (7)
6	**00:10**	7.9 (4)	6.4 (4)	9.6 (3)	4.3 (6)
7	00:12	1.2 (17)			
8	**00:15**	3.0 (13)	10.6 (2)	5.3 (7)	16.0 (3)
9	**00:20**	4.8 (7)	2.0 (12)	3.1 (12)	2.5 (8)
10	**00:30**	4.7 (8)	3.5 (9)	5.5 (6)	17.7 (2)
11	00:40			1.3 (19)	0.5 (19)
12	00:45	1.1 (18)			
13	00:50				0.5 (20)
14	**01:00**	10.5 (2)	4.2 (8)	5.8 (5)	4.9 (5)
15	01:30		0.8 (18)	1.3 (18)	1.5 (12)
16	01:40			1.4 (16)	
17	01:59				6.0 (4)
18	**02:00**	5.3 (6)	5.4 (6)	4.5 (9)	21.3 (1)
19	02:10			1.3 (17)	
20	02:30	1.2 (16)		1.4 (15)	
21	**03:00**	3.8 (10)	4.9 (7)	2.5 (13)	1.8 (10)
22	03:20			5.1 (8)	
23	03:50			3.3 (11)	
24	**04:00**	5.7 (5)	2.2 (11)	12.5 (1)	1.6 (11)
25	04:50		0.6 (20)		
26	**05:00**	1.4 (15)	1.1 (16)		0.9 (15)
27	**06:00**	2.0 (14)	6.1 (5)		1.0 (14)
28	07:00	0.9 (19)			
29	**08:00**	3.4 (11)	1.5 (14)		0.8 (17)
30	**10:00**	3.3 (12)	1.7 (13)		0.9 (16)
31	**12:00**	4.0 (9)	2.2 (10)		0.6 (18)
32	15:00	0.9 (20)	1.5 (15)		
33	16:00		1.0 (17)		
34	17:00		0.6 (19)		
35	**18:00**	9.8 (3)	23.8 (1)		2.1 (9)
36	36:00				1.1 (13)
sum (all)		86.4	88.9	89.3	88.7
sum (joint)		81.2	84.4	60.4	79.1

to ordinarily use. Note that this is regardless of the different per-trace maximal estimate limits. We conclude that joint times should be hard-coded in our model, as it is fairly reasonable to conjecture humans will always extensively use values like 15 minutes, 1 hour, etc. We therefore define the first head-mapping step — determining the 20 time values that are the most popular — as follows:

1. Choose T_{max}, the maximal estimate (which is a mandatory parameter of our model). As previously mentioned, this is always a top ranking value.
2. Choose all hard-coded joint times that are smaller than T_{max}.
3. Choose in descending order multiples of T_{round} (smaller than T_{max}), where T_{round} is 200h, then 100h, 50h, 10h, 5h, 2h, 1h, 20m, 10m, and 5m. We stop when the number of (different) chosen values reaches 20.

The role of the third item above is to add a *relative* aspect to the process of choosing popular estimates, which is largely hard-coded. As will later be shown, this manages to successfully capture KTH4H's condensed nature. At the other end, workloads with larger estimate domains, of jobs that span hundreds of hours, do in fact exist [2]. Regrettably, their owners refuse to share them with the community. Nevertheless, our algorithm generates longer times based on the modes they report (400h, 200h, 100h, and 50h in the NCSA O2K traces).

Finally, recall we have already generated K time values using F_{tim} defined in Section 7. Head times generated here, replace the 20 values generated by F_{tim} that are the closest to them (and so the structure reported in Fig. 8 is preserved).

9.3 Mapping of Head Estimates

Having both head times (seconds) and sizes (job percentages), we go on to map between them. As usual, mapping is made possible by using the associated ranks, rather than the actual values. For this purpose we need two new definitions:

First, we define a new type of time-rank, the *top-20 time rank* (or *ttr* for short), which is rather similar to the ordinary time-rank: All top-20 times, excluding T_{max}, are ascendingly sorted. The first is assigned a ttr=1, the second a ttr=2, and the last a ttr=19. For example, according to Table 5, in CTC, 00:05 has ttr=1, 00:10 has ttr=2, 01:30 has a ttr=7, and 17:00 has a ttr=19. T_{max} is always associated with ttr=0.

Second, for each trace-file *log*, we define a function F_{log} that maps ttr-s to the associated popularity ranks, within that log. For example, $F_{ctc}(0)$=1 as T_{max}=18h (associated with ttr=0) is its most popular estimate. Likewise, $F_{ctc}(1)$=3, as 5min is the smallest top-20 estimate (ttr=1) and is the third most popular estimate within CTC. Table 6 lists F_{log} of the four traces. Recall that 2h is the effective T_{max} of BLUE and therefore this is the estimate we choose to associate with ttr=0. Additionally, note the BLUE 01:59 mode near its T_{max}=2h (Table 5). This is probably due to users trying to enjoy both worlds: use the maximal value, while "tricking" the system to assign their jobs a higher priority as a result of being shorter. We are not interested (nor able) to model such phenomena. Therefore, in the generation of Table 6 and throughout the reminder

Table 6. The F_{log} functions of the four traces. The four most popular ranks in each trace are highlighted in bold font.

ttr	$F_{sdsc-106}$	F_{ctc}	F_{kth4h}	F_{blue}
0	**3**	**1**	**1**	**1**
1	**1**	**3**	**4**	6
2	**4**	**4**	10	5
3	17	**2**	14	**3**
4	13	12	20	7
5	7	9	**2**	**2**
6	8	8	**3**	18
7	18	18	7	19
8	**2**	6	12	**4**
9	6	7	6	11
10	16	11	19	20
11	10	20	5	9
12	5	16	18	10
13	15	5	16	14
14	14	14	9	13
15	19	13	17	16
16	11	10	15	15
17	12	15	13	17
18	9	17	8	8
19	20	19	11	12

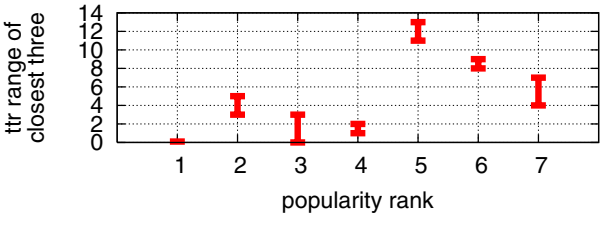


Fig. 13. There is only 0-3 difference between the closest three ttr-s that are associated with the more popular ranks (Table 6). For example, 3 of the ttr-s associated with popularity rank 2, are located in rows 3-5 in Table 6 (underlined and highlighted in a different color). In the above figure, this corresponds to range-bar associated with popularity rank 2 that stretches between lines 3-5.

of this paper, we aggregate the 01:59 mode with that of 2h and consider them a single 27.3% mode.

The F_{log} functions in Table 6 reflect reality, and are in fact the reason for the log-uniform CDFs observed in Fig. 3. We therefore seek an algorithm that can "learn" these functions and be able to imitate them. Given such an artificial F_{log}, we would finally be able to match head-sizes (produced in Section 8, their size defines their popularity rank) to head-times (produced in Section 9.2, their value defines their ttr-s) and complete our model.

At first glance, the four F_{log} functions appear to have little similarities (the correlation coefficient between the columns of Table 6 is only 0.1-0.3), seemingly deeming failure on the generalization attempt. However a closer inspection reveals some regularities. Consider for example the more popular (and therefore more important to model) ranks: at least three of four values of each such rank are clustering across neighboring lines (ttr-s). This is made clearer in Fig. 13.

Another observation is that when dividing popularity-ranks into two (1-10 vs. 11-20), around 75% of the more popular ranks are found in the top half of Table 6, which indicates a clear tendency of more popular ranks to be associated with smaller ttr-s. (This coincides with the log-uniformity of the original estimate distributions). It is our job to capture these regularities.

In the initialization part of our algorithm, which we call the *pool algorithm*, we associate ttr=0 (of T_{max}) with popularity rank=1, that is, the maximal estimate is also the most popular. The rationale of this decision is that

1. according to Table 6 this is usually the case in real traces,
2. as explained in Section 2, making T_{max} the most popular estimate constitutes a realistic worst case scenario, which is most appropriate to serve as the default setting, and
3. it is the "safest" decision due to the constraint that estimates must be longer than runtimes.

The last two items are the reason why we chose to follow the CTC example and enforce a sizable first rank on the construction of F_{pop} (end of Section 8) that "breaks" the exponential contiguity observed in Fig. 10. To complete the initialization part, we allocate an empty vector V_{pool} designated to hold popularity ranks. Any popularity rank may have up to four occurrences within V_{pool}.

The body of the pool-algorithm iterates through the rest of the ttr-s in ascending order ($J_{ttr} = 1, ..., 19$) and performs the following steps on each iteration:

1. For each trace file *log*, insert the popularity rank $F_{log}(J_{ttr})$ to V_{pool}, but only if this rank wasn't already mapped to some smaller ttr in previous iterations. (In other words, insert all the values from within the J_{ttr} line in Table 6, that weren't already chosen.)
2. If there exists popularity ranks that have four occurrences within V_{pool}, choose the smallest of these ranks R, map J_{ttr} to R, remove all occurrences of R from V_{pool}, and move on to the next iteration.
3. Otherwise, randomly choose two (not necessarily different) popularity ranks from within V_{pool}, map the smaller of these to J_{ttr}, and remove all its occurrences from V_{pool}.

A main principle of the algorithm is the gradual iteration over Table 6, such that no popularity-rank R is eligible for mapping to J_{ttr}, before we have actually witnessed at least one occasion in which R was mapped to a ttr that is smaller than or equal to J_{ttr}. This aims to imitate the original F_{log} functions, along with serving as the first safety-mechanism obstructing more popular ranks to be mapped to longer estimates (recall that estimate CDFs are log-uniform, which means most estimates are short).

Another important principle of the algorithm is that increased number of occurrences of the same R within V_{pool}, implies a greater chance of R to be randomly chosen. And so, an R that is mapped to a ttr $\leq J_{ttr}$ within two traces (two occurrences within V_{pool}), has double the chance of being chosen in comparison to a popularity rank for which this condition holds with respect to only one trace (one occurrence within V_{pool}). This aspect of the algorithm also aims to capture the commonality between the various traces.

Item number two in the algorithm tries to make sure an R will not be mapped to a ttr that is bigger than *all* the ttr-s to which it was mapped in the four traces. Like the first principle mentioned above, this item has the role of making sure the resulting mapping isn't too different than that of the original logs. It also serves as the second safety-mechanism limiting the probability of more popular ranks to be mapped to longer estimates.

The combination of the above "safety mechanisms" was usually enough to produce satisfactory results. However, on rare occasions, too many high popularity ranks have managed to nevertheless "escape" these mechanisms and be mapped to longer estimates. Adding a third safety-mechanism, in the form of using the minimum between two choices of popularity ranks (third item of the algorithm), has turned this probability negligible.

9.4 Embedding User-Supplied Estimates

While the estimate distributions of the traces bare remarkable resemblance, they are also very distinct within the "head of the head" (as discussed in Section 8), that is, the 1-3 most popular estimates. For example, considering Table 5, the difference between the percentage of SDSC and CTC jobs associated with 18h (10% vs. 24%) is enough to yield completely different distributions. Another example is BLUE's shift of the maximum from 36h to 2h, or its two huge modes in 15min and 30min; the fact that more than 60% of its jobs use one of these estimates (along with 01:59), cannot be captured by any general model. Yet another example is KTH4H's unique modes below 5min. This variance among the most important estimate bins, along with the fact users may be aware of special queues and other influential technicalities concerning their site, mandates a general model to allow its user to manually supply head estimates as parameters.

To this end, we allow the user to supply the model with a vector of up to 20 (t_i, p_i) pairs. The manner in which these pairs are embedded within our model is the following: The t_i values replace default-generated head times (Section 9.2) that are the closest to them, with the exception of T_{max} which is never replaced unless explicitly given by the user as one of the (t_i, p_i) pairs. (This is due to the reasons discussed in Section 9.3.) As an example, in order to effectively replace the maximal value of BLUE, the user must supply two pairs: $(36h, 1\%)$ to prevent the model from making the old maximum (36h) the most popular estimate, and $(2h, 27\%)$ to generate the new maximum.

Similarly to times, user supplied p_i sizes (job percents) replace default-generated sizes (Section 8) that are the closest to them. Once again, the biggest value (reserved for T_{max}) is not replaced if the user did not supply a pair

containing T_{max}. Additionally, the remaining non-user head-sizes are scaled such that the total mass of the head is still 89% (scaling however do applies to the largest non-user size). If scaling is not possible (sum of user sizes exceed 89%), non-user head-sizes are simply eliminated, and the tail sizes are scaled such that the sum of the entire distribution is 100%.

Finally, the pool algorithm is refined to skip ttr-s that are associated with user-supplied estimates and to avoid mapping their associated popularity ranks.

10 Overview of the Model

Now that all the different pieces are in place, let us briefly review the default operation of the estimates model we have developed:

1. Get input. The mandatory parameters are maximal estimate value T_{max}, and number of jobs N (which is the number of estimates the model must produce as output). A third, "semi mandatory", parameter is the percentage of jobs associated with T_{max}. While the model can arbitrarily decide this value by itself, its variation in reality is too big to be captured by a model, whereas its influence on performance results is too detrimental to be ignored (T_{max} jobs are the "worst kind" of jobs in the eyes of the scheduler; Section 2).
2. Compute the value of K (different estimate times) as defined in Section 6.
3. Generate K time-values using F_{tim} as defined in Section 7.
4. Generate 20 "head" time-values using the algorithm defined in Section 9.2 and combine them with the K time-values produced in the previous item. Non-head times are denoted "tail" times.
5. Generate K sizes (jobs percent) using F_{pop} as defined in Section 8. The largest 20 sizes are the head sizes. The rest are tail.
6. Map between time- and size-values using F_{map} as defined in Section 9, by
   - Randomly mapping between tail-times and tail-sizes in a uniform manner (Section 9.1).
   - Mapping head-times and head-sizes using the pool algorithm (Section 9.3).
7. If received user supplied estimate bins, embed them within the model as described in Section 9.4.

10.1 About the Complexity

The only part which is non-trivial in our model is the pool algorithm: Generating the estimate time values by themselves is a trivial operation. Generating sizes (percentages of jobs) is equally trivial. Mapping between these two value sets is also a relatively easy operation, as all but the 20 most popular sizes can be randomly mapped. All the complexity of the model concentrates in solving the problem of deciding how many jobs are associated with each "head" estimate, or in other words, where exactly to place the larger modes. The question of whether a simpler alternative than the one suggested here exists, is an open one, and it

is conceivable there's a positive answer. However, all the "immediate" heuristics we could think of in order to perform this task in a simpler manner have been checked and verified to be inadequate. In fact, it is these inadequacies that has lead us step by step in the development of the pool algorithm.

11 Validating the Model

Having implemented the estimate model, we now go on to validate its effectiveness. This is essentially composed of two parts. The first is obviously making sure that the resulting distribution is similar to that of the traces (Section 11.1). However, this is not enough by itself, as our ultimate goal is to allow realistic performance evaluation. The second part is therefore checking whether performance results obtained by using the original data are comparable to those produced when replacing original estimates with artificial values produced by the model (Section 11.3). The latter part mandates developing a method according to which artificial estimates are assigned to jobs (Section 11.2).

11.1 Validating the Distribution

Fig. 14 plots the original CDFs (solid line) against those generated by the "vanilla" model using various seeds. The only input parameters that are given to the model are those listed in Section 10, that is, the maximal estimate T_{max}, then number of jobs N, and the percentage of jobs associated with T_{max}. Recall that BLUE's maximum is considered to be 2 hours and that in order to reflect this we must explicitly supply the model with an additional pair (Section 9.4).

The results indicate the model has notable success in generating distributions that are remarkably similar to that of SDSC-106 and CTC; it is far less successful with respect to the other two traces. However, this should come as no surprise because, as mentioned earlier, the model has no pretense of reflecting abnormalities or features that are unique to individual traces. In the case of KTH4H, these are the large modes that are found below 5 minutes (Table 5). In fact, if aggregating these modes with that of 5 minutes, we get that a remarkable 25.5% of KTH4H's jobs have estimates that are 5 minutes or less, which is inherently different in comparison to the other traces. In the case of BLUE, its uniqueness takes the form of two exceptional modes located at 15 and 30 minutes. This distinctive quality is especially apparent in Fig. 10, where the three biggest modes "break" the log-uniform contiguity.

The practical question is therefore if the model can produce good results when provided with *minimal* additional information highlighting the trace-specific abnormalities. The amount of such information is inherently limited if we are to keep the model applicable and maintain its practical value. We therefore define the "improved" setting in which the KTH4H model is provided with the additional $(5min, 25\%)$ pair. The BLUE model is provided with two additional pairs associated with its two exceptional modes: $(15min, 16\%)$ and $(30min, 18\%)$.

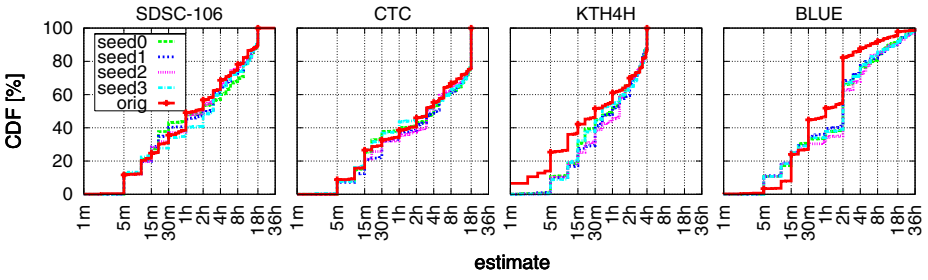


Fig. 14. The original estimate distribution of the traces (solid lines) vs. the output of the vanilla model, when used with four different seeds. Output is less successful for traces with unique features.

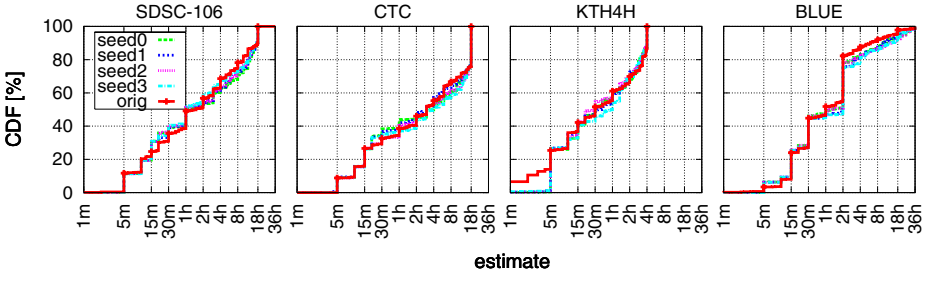


Fig. 15. Output of the model under the "improved" setting which provides minimal information identifying the unique features

The results of the improved setting are shown in Fig. 15 and indicate that this additional information was all that the model needed in order to produce satisfactory results (also) with respect to the two "unique" traces. To test the impact of additional information on situations where the vanilla model manages to produce reasonable results by itself, the improved setting supplied three additional pairs (of the most popular estimates) when modeling CTC and SDSC-106. It is not apparent whether the additional information made a qualitative difference.

The important conclusion that follows from the successful experiment we have conducted in this section, is that estimate distributions are indeed extremely similar: Most of their variance concentrates within the 1-3 most popular estimates, and once these are provided, the model produces very good results.

11.2 Assigning Estimates to Jobs

The next step in validating the model is putting it to use within a simulation. For this purpose we have decided to simulate the EASY scheduler and evaluate its performance under the four workloads. This can be done with original estimates or after replacing them with artificial values that were generated by our model. Similar performance results would indicate success.

The common practice when modeling a parallel workload is to define canonical random variables to represent the different attributes of the jobs, e.g. runtime, size, inter-arrival time etc. [6, 15, 20]. Generating a workload of N jobs is then performed by creating N samples of these random variables. Importantly, each sample is generated *independently* of other samples.

In this respect, assignment of artificial estimates to jobs is subtle, as this must be done under the constraint that estimates mustn't be smaller than the runtimes of the jobs to which they are assigned. Here, we can't just simply randomly choose a value. However, if independence between jobs is still assumed, we can easily overcome the problem by using the *random shuffle algorithm*. This algorithm gets two vectors $V_{estimate}$ and $V_{runtime}$ that hold N values as suggested by their names. The content of both vectors is generated as usual, according to the procedure described above (under the assumption of independence). Now all that is needed is a random permutation that maps between the two, such that every estimate is equal to or bigger than its associated runtime. The random shuffle algorithm finds such a permutation by iterating through $V_{runtime}$ and randomly pairing each runtime R with some estimate $E \in V_{estimate}$ for which $E \geq R$. After values are paired, they are removed from their respective vectors.

Note that we do not claim that the independence assumption underlying the random shuffle algorithm is correct. On the contrary. We only argue that this is the common practice. However, there is a way to transform the original data such that this assumption holds: The algorithm can be applied to the original data, that is, we can populate the $V_{estimate}$ vector with original trace estimates and reassign them to jobs using the shuffle algorithm. The outcome of doing this would be that the original estimates are "randomly shuffled" between jobs (which is the source of the algorithm's name). The result of such shuffling is to create independent "real" estimates. This is suitable as a basis for comparison with our model, as explained below.

11.3 Validating Performance Results

Several estimate-generation models have been evaluated and compared against the original data:

- The *X2*-model: simply doubles user estimates on the fly [16, 21].
- The *shfl*-model: the result of applying the random shuffle algorithm (defined above) to the original data. As noted, assuming independence in this context is correct.
- The f-model: upon receiving a job's runtime R, uniformly chooses an estimate from the closed range $[R, R \cdot (f+1)]$. In accordance with [21], six values of f were chosen: 0 (complete accuracy), 1, 3, 10, 100, and 300.
- The *feit*-model: targets accuracy (suggested by Mu'alem and Feitelson [21] and explained in the introduction).
- The *vanl*-model: the vanilla setting of the model developed in this paper (defined above).
- The *impr*-model: the improved setting of our model, supplying it with some additional information (defined above).

Notice *X2* and *shfl* aren't models per-se as both are based on real estimates. The competitors of our model are *f* and *feit* (producing estimates based on runtime).

Performance results are shown in Fig. 16 in the form of average wait time and bounded slowdown. The black dotted lines present the results of running the simulations using the original data. Therefore, models that are closer to this line are more realistic. Recall that our aim here is not to improve performance. Rather, it is to produce trustworthy results that are closest to reality. All the results associated with models that contain a random component (all but *X2* and *f0*) are the average of one hundred different simulation runs employing different seeds. The error-bars associated with these models display the absolute-deviation (average of absolute value of deviation from the average).

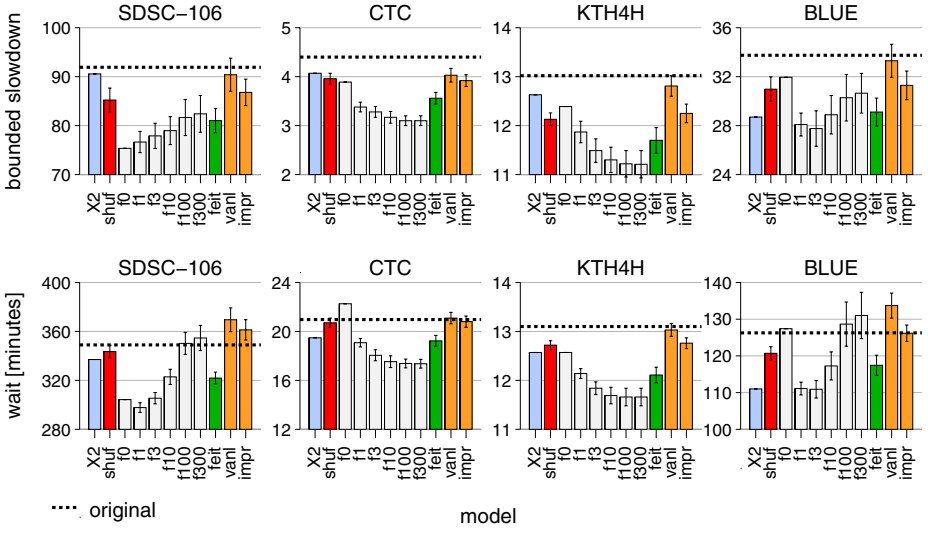


Fig. 16. Validating badness. The reason for the peculiar results associated with the average wait time of SDSC and BLUE, remain unknown.

When examining Fig. 16, it is clear the two variants of our algorithm are more realistic, in that they usually do a better job in capturing the "badness" of user estimates (compare with *f*-s and *feit*). Another observation is that using increased *f*-s (or *feit*) to model increased user inaccuracy (for the sake of realism) is erroneous, as *f0* usually produces results that are much closer to the truth. In fact, *f0* is usually comparable to the results obtained by our model with the exception of the SDSC trace. However, this is limited to the FCFS-based EASY scenario: if introducing a certain amount of limited SJF-ness to the scheduler (e.g. as in [25, 1]), *f0* yields considerably better performance results in comparison to the original, whereas our model stays relatively the same (figure not shown to conserve space). Another scenario in which *f0* can't be used is when evaluating system-generated runtime predictors that make use of estimates (along with other job characteristics) [14, 23, 18, 25]. Finally (returning to

the context of EASY), unlike *f0*, our model has room for improvement as will shortly be discussed, and we believe it has potential to "go the extra mile".

A key point in understanding the performance results is noticing that the vanilla setting of our algorithm is surprisingly more successful in being closer to the original than its improved counterpart. This is troublesome as our entire case is built on the argument that models that are more accurate would yield results that are closer to the truth. The answer to the riddle is revealed when examining the *shfl* model. The fact of the matter is that one cannot get more accurate than *shfl*, as it "generates" a distribution that is *identical* to that of the original. Yet it too seems to be inferior to our vanilla model. This exposes our independence assumption (the random shuffle algorithm) as the true guilty party which is responsible for the difference between *impr* and the original. The correct comparison between *impr* and *vanl* should actually be based on which is closer to *shfl*, not to the original, as only with *shfl* can independence be assumed. Based on this criterion, *impr* is consistently better than *vanl*.

Once this is understood, we can also explain why the performance of *impr* (in terms of wait and slowdown) is always better than that of *vanl*. Consider the difference between the two models: *impr* simply has much more accurate data regarding *shorter* jobs (e.g. KTH4H's 25% of 5 minutes jobs). As short jobs benefit the most from the backfilling optimization, *impr* consistently outperforms *vanl* (in absolute terms).

11.4 Repetitiveness Is Missing

We are not interested in artificially producing worse results by means of falsely boosting up estimates (as is done by *vanl* with respect to *impr*). This would be equivalent to, say, increasing the fraction of jobs that estimate to run T_{max}, which can arbitrarily worsen results. Our true goal is creating a reliable model. The above indicates that the problem lies in the assumption of independence, namely, the manner we assign estimates to jobs. While it is possible that this is partially because we neglected to enforce the accuracy to be as displayed in Fig. 1 (the accuracy histograms of even *shfl* are dissimilar to that of the original), we conjecture that the independence assumption is more acute.

It has been known for over a decade that the work generated by users is highly repetitive [12, 10]. Recent work [28, 24] suggests that the correct way to model a workload is by viewing it as a sequence of *user sessions*, that is, bursts of very similar jobs by the same user. This doctrine suggests that a correct model cannot just draw values from a given distribution while disregarding previous values as is done by most existing parallel workload models (e.g. [6, 15, 20, 4]). The rationale of this claim is that the repetitive nature of the sequence within the session may have a decisive effect on performance results.[6]

[6] A remarkable example stressing the importance of this phenomenon was recently published [26]: changing a runtime of only *one* job (within a log that spans two years) by a mere 30 seconds, resulted in a change of 8% in the average bounded slowdown of *all* the jobs; the reason was traced to be a certain user-session and its interaction with the scheduler.

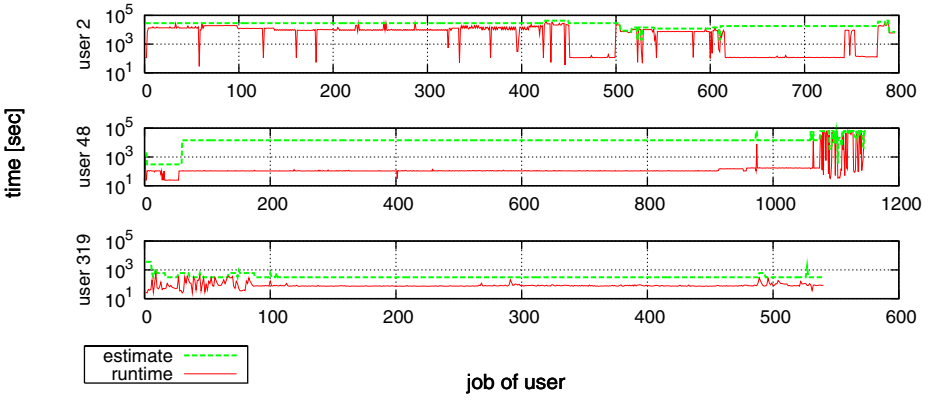


Fig. 17. Runtime and estimate of all the jobs submitted by three arbitrary users from the SDSC trace shows remarkable repetitiveness

Since users tend to submit bursts of jobs having the same estimate value (Fig. 17), the end result is somewhat similar to that of the existence of estimates modes, but in a more "temporal sense": At any time instance, jobs within the wait-queue tend to look the same to the scheduler, as jobs belonging to the same session usually share the same estimate value. Consequently, the scheduler has less flexibility in making backfilling decision and the performance is negatively effected. Our *shfl* algorithm, along with all the rest of the models, do not entail the concept of sessions and therefore result in superior performance in comparison to the original.

Accordingly, our future work includes developing an assignment mechanism that is session aware. This can be obtained if the procedure that pairs runtimes and estimates gets additional information associating jobs with users. User-based modeling [24] can supply this data.

12 Conclusions and Future Work

User runtime estimates significantly effect the performance of parallel systems [21, 1, 8]. As part of the effort to allow realistic and trustworthy performance analysis of such systems, there is a need for an estimates model that successfully captures their main characteristics.

A number of models have been suggested, but these are all lacking in some respect. Their shortcoming include implicitly revealing too much information about real runtimes, erroneously emulating the accuracy ratio of runtime to estimate, neglecting to take into consideration the fact that all production installations have a limit on the maximal allowed estimate, and that this value is always one of the more popular estimates. Importantly, two key ingredients are missing from existing models: the inherently modal nature of the estimates caused by users' tendency to supply "round" values [21, 2, 17], and the temporal repetitive nature of user estimates, assigning the same value to bursts of

jobs (sessions) [26, 28]. These have decisive effect on performance results, as low estimate-variance of waiting jobs reduces the effectiveness of backfilling.

Consequently, the outcome of using any of the existing models are simulation results that are unrealistically better than those obtained with real estimates. Thus, it is erroneous to use these models, and in particular, the popular "f-model" in which each job's estimate is randomly chosen from $[R, (f + 1)R]$, where R is the job's runtime and f is some positive constant.

Variants of the f-model are often used to investigate the impact of the inherent inaccuracy of user estimates, or to artificially generate estimates when those are missing from existing workloads (trace files, models) that are used to drive simulations [11, 29, 21, 1, 13]. When conducting performance evaluation, the common (false) justification for using the f-model is that "overestimation commonly occurs in practice and is beneficial for overall system performance" [13]. Indeed, overestimation is common. But the improved performance is simply an undesirable byproduct of the *artificial* manner in which overestimation is obtained; *real* user overestimation actually degrades performance significantly. In fact, using exact runtimes as estimates is actually more realistic than utilizing the f-model! While both approaches usually yield unrealistically improved performance (in comparison to those obtained with real estimates), perfect accuracy is almost always closer to the truth.

In this paper we produce a model that targets estimates modality. We view the estimates distribution as a sequence of modes, and investigate their main characteristics. Our findings include the invariant that 20 "head" estimates are used by about 90% of the jobs throughout the entire log. The popularity of head estimates (percentage of jobs using them) decreases exponentially, whereas the tail obeys a power-law. The few hundred values that are used as estimates, are well-fitted by a fractional model, while at the same time, 15 out of the 20 head estimates are identical across all the production logs we have examined. The major difficulty faced by this paper was determining how popular is each head estimate (how many jobs are associated with each). This was solved by the "pool algorithm", aimed to capture similarities between profiles of head-estimates within the analyzed production logs.

We found that all modeled aspects of the estimates distribution are almost identical across the logs, and therefore our model defines only two mandatory parameters: the number of jobs and the maximal allowed estimate (T_{max}). While considerable variance does in fact exist, it is mostly encapsulated within the percentage of jobs estimated to run for T_{max}. The remaining variance (if any) is attributed to another 1-2 very popular modes that sometimes exist, but are unique to individual logs. When provided this additional information, our model produces distributions that are remarkably similar to that of the original.

When put to use in simulation (by replacing real estimates with artificial ones), our model consistently yields performance results that are closer to the original than those obtained by other models. In fact, these results are almost identical to when real estimates are used and are randomly shuffled between jobs. This suggests that the temporal repetitiveness of per-user estimates may be the

final obstacle separating us from achieving truly realistic results. Consequently, our future work includes developing an improved assignment scheme of estimates to jobs that will preserve this feature.

Our estimates model is available to download from the parallel workload archive [9]. Its interface contains two functions: generating the distribution modes, and assigning estimates to jobs. The latter is essentially random shuffling of estimates between jobs, under the constraint that runtimes are smaller than estimates. Our future work includes refining this function such that the user-session quality takes effect.

Acknowledgment. This research was supported in part by the Israel Science Foundation (grant no. 167/03).

References

1. S-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, "*The impact of more accurate requested runtimes on production job scheduling performance*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 103–127, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
2. S-H. Chiang and M. K. Vernon, "*Characteristics of a large shared memory production workload*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 159–187, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
3. W. Cirne and F. Berman, "*A comprehensive model of the supercomputer workload*". In 4th *Workshop on Workload Characterization*, Dec 2001.
4. W. Cirne and F. Berman, "*A model for moldable supercomputer jobs*". In 15th *Intl. Parallel & Distributed Processing Symp.*, Apr 2001.
5. M. E. Crovella, "*Performance evaluation with heavy tailed distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–10, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
6. A. B. Downey, "*A parallel workload model and its implications for processor allocation*". In 6th *Intl. Symp. High Performance Distributed Comput.*, pp. 112–124, Aug 1997.
7. Y. Etsion and D. Tsafrir, *A Short Survey of Commercial Cluster Batch Schedulers*. Technical Report 2005-13, Hebrew University, May 2005.
8. D. G. Feitelson, "*Experimental analysis of the root causes of performance evaluation results: a backfilling case study*". *IEEE Trans. Parallel & Distributed Syst.* **16(2)**, pp. 175–182, Feb 2005.
9. D. G. Feitelson, "*Parallel workloads archive*". URL http://www.cs. huji.ac.il /labs /parallel/workload.
10. D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

11. D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.
12. D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
13. E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini, "*Parallel job scheduling under dynamic workloads*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 208–227, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.
14. R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
15. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "*Modeling of workload in MPPs*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 95–116, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
16. P. J. Keleher, D. Zotkin, and D. Perkovic, "*Attacking the bottlenecks of backfilling schedulers*". *Cluster Comput.* **3(4)**, pp. 255–263, 2000.
17. C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "*Are user runtime estimates inherently inaccurate?*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), Springer Verlag, Jun 2004. Lect. Notes Comput. Sci. vol. 3277.
18. H. Li, D. Groep, and J. T. L. Wolters, "*Predicting job start times on clusters*". In *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004.
19. D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
20. U. Lublin and D. G. Feitelson, "*The workload on parallel supercomputers: modeling the characteristics of rigid jobs*". *J. Parallel & Distributed Comput.* **63(11)**, pp. 1105–1122, Nov 2003.
21. A. W. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.
22. D. Perkovic and P. J. Keleher, "*Randomization, speculation, and adaptation in batch schedulers*". In *Supercomputing*, p. 7, Sep 2000.
23. W. Smith, I. Foster, and V. Taylor, "*Predicting application run times using historical information*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 122–142, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.
24. D. Talby, *User Modeling of Parallel Workloads*. PhD thesis, The Hebrew University of Jerusalem, Israel, 200? In preparation.
25. D. Tsafrir, Y. Etsion, and D. G. Feitelson, *Backfilling Using Runtime Predictions Rather Than User Estimates*. Technical Report 2005-5, Hebrew University, Feb 2005.
26. D. Tsafrir and D. G. Feitelson, *Workload Flurries*. Technical Report 2003-85, Hebrew University, Nov 2003.

27. Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "*An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 133–158, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
28. J. Zilber, O. Amit, and D. Talby, "*What is worth learning from parallel workloads? A user and session based analysis*". In *Intl. Conf. Supercomputing*, Jun 2005.
29. D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *Intl. Symp. High Performance Distributed Comput.*, Aug 1999.

Workload Analysis of a Cluster in a Grid Environment

Emmanuel Medernach

Laboratoire de Physique Corpusculaire, CNRS-IN2P3
Campus des Cézeaux,
63177 Aubière Cedex, France
medernac@clermont.in2p3.fr

Abstract. With Grids, we are able to share computing resources and to provide for scientific communities a global transparent access to local facilities. In such an environment the problems of fair resource sharing and best usage arise. In this paper, the analysis of the LPC cluster usage (Laboratoire de Physique Corpusculaire, Clermont-Ferrand, France) in the EGEE Grid environment is done, and from the results a model for job arrival is proposed.

1 Introduction

Analysis of a cluster workload is essential to understand better user behavior and how resources are used [1]. We are interested to model and simulate the usage of a Grid cluster node in order to compare different scheduling policies and to find the best suited one for our needs.

The Grid gives new ways to share resources between sites, both as computing and storage resources. Grid defines a global architecture for distributed scheduling and resource management [2] that enable resources scaling. We would like to understand better such a system so that a model can be defined. With such a model, simulation may be done and a quality of service and fairness could then be proposed to the different users and groups.

Briefly, we have some groups of users that each submit jobs to a group of clusters. These jobs are placed inside a waiting queue on some clusters before being scheduled and then processed. Each group of users have their own need and their own strategy to job submittal. We wish:

1. to have good metrics that describes the group and user usage of the site.
2. to model the global behavior (average job waiting time, average waiting queue length, system utilization, etc.) in order to know what is the influence of each parameter and to avoid site saturation.
3. to simulate jobs arrivals and characteristics to test and compare different scheduling strategies. The goal is to maximize system utilization and to provide fairness between site users to avoid job starvation.

As parallel scheduling for p machines is a hard problem [3] [4], heuristics are used [5]. Moreover we have no exact value about the duration of jobs, making

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 36–61, 2005.

the problem difficult. We need a good model to be able to compare different scheduling strategies. We believe that being able to characterize users and groups behavior we could better design scheduling strategies that promote fairness and maintain a good throughput. From this paper some metrics are revealed, from the job submittal protocol a detailed arrival model for single user and group is proposed and scheduling problems are discussed. We then suggest a new design based on our observation and show relationship between fairness issue and system utilization as a flow problem.

Our cluster usage in the EGEE (Enabling Grids for E-science in Europe) Grid is presented in section 2, the Grid middleware used is described. Corresponding scheduling scheme is shown in section 3. Then the workload of the LPC (Laboratoire de Physique Corpusculaire) computing resource, is presented (section 4) and the logs are analyzed statistically. A model is then proposed in section 5 that describes the job arrival rate to this cluster. Simulation and validation are done in section 6 with comparison with related works in section 7. Results are discussed in section 8. Section 9 concludes this paper.

2 Environment

2.1 Local Situation

The EGEE node at LPC Clermont-Ferrand is a Linux cluster made of 70 dual 3.0 GHz CPUs with 1 GB of RAM and managed by two servers with the LCG (LHC Computing Grid Project) middleware. Each server has the same queues with the same configuration, but different machines behind. One server has more machine behind it, because the other one runs other grid services. Users could run their jobs by sending them to the two servers transparently. The two servers are not coordinated. We are currently using MAUI as our cluster scheduler [6] [7]. It is shared with the regional Grid INSTRUIRE (`http://www.instruire.org`). Our LPC Cluster role in EGEE is to be used mostly by Biomedical users[1] located in Europe and by High Energy Physics Communities. Biomedical research is one core application of the EGEE project. The approach is to apply the computing methods and tools developed in high energy physics for biomedical applications. Our team has been involved in international research group focused on deploying biomedical applications in a Grid environment.

GATE is one pilot application. It is based on the Monte Carlo GEANT4 [8] toolkit developed by the high energy physics community. Radiotherapy and brachytherapy use ionizing radiations to treat cancer. Before each treatment, physicians and physicists plan the treatment using analytical treatment planning systems and medical images data of the tumor. By using the Grid environment provided by the EGEE project, we will be able to reduce the computing time of Monte Carlo simulations in order to provide a reasonable time consuming tool for specific cancer treatment requiring Monte-Carlo accuracy.

[1] Our cluster represented 75% of all the Biomed Virtual Organization (VO) jobs in 2004.

Another group is Dteam, this group is partly responsible of sending tests and monitoring jobs to our site. Total CPU time used by this group is small relatively to the other one, but the jobs sent are important for the site monitoring. There are also groups using the cluster from the LHC experiments at CERN (`http://www.cern.ch`). There are different kind of jobs for a given group. For example, Data Analysis requires a lot of I/O whereas Monte-Carlo Simulation needs few I/O.

2.2 EGEE Grid Technology

In Grid world, resources are controlled by their owners. For instance different kind of scheduling policies could be used for each site. A Grid resource center provides to the Grid computing and/or storage resources and also services that allow jobs to be submitted by guests users, security services, monitoring tools, storage facility and software management. The main issue of submitting a job to a remote site is to provide some warranty of security and correct execution. In fact the middleware automatically resubmits job when there is a problem with one site. Security and authentication are also provided as Grid services.

The Grid principle is to allow user a worldwide transparent access to computing and storage resources. In the case of EGEE, this access is aimed to be transparent by using LCG middleware built on top of the Globus Toolkit [9]. Middleware acts as a layer of software that provides homogeneous access to different Grid resource centers.

2.3 LCG Middleware

LCG is organized into Virtual Organizations (VOs): dynamic collections of individuals and institutions sharing resources in a flexible, secure and coordinated manner. Resource sharing is facilitated and controlled by a set of services that allow resources to be discovered, accessed, allocated, monitored and accounted for, regardless of their physical location. Since these services provide a layer between physical resources and applications, they are often referred to as Grid Middleware [10].

Bag of task applications are parallel applications composed of independent jobs. No communications are required between running jobs. Since jobs from a same task may execute on different sites communications between jobs are avoided. In this context, users submit their jobs to the Grid one by one through the middleware. Our cluster receives jobs only from the EGEE Grid, users could access it only by the LCG Middleware. Our current LCG Grid middleware forces users to divide their computations in jobs units composed of individual processes (with one CPU only), and submits them as such. This means that each job requests for one and only one processor. In our case we have dual processors machines, we allow 2 jobs per machines, one for each CPU on it. Users could directly specify the execution site or let a Grid service choose the best destination for them. Users give only a rough estimation of the maximum job running time,

mainly a wall clock time. In general this estimated time is overestimated and very imprecise [11]. Instead of speaking about an estimated time, it could be better to speak about an upper bound for job duration, so this value provided by users is more a precision value. The bigger the value is the more imprecise the value of the actual runtime could be.

Figure 1 shows the scenario of a job submittal. In this figure rounded boxes are grid services and ellipses are the different jobs states. As there is no communications between jobs, jobs could run independently on multiple clusters. Instead of communicating between job execution, jobs write output files to some Storage Elements (SE) of the Grid. Small output files could also be sent to the UI. Replica Location Service (RLS) is a Grid service that allow location of replicated data. Other jobs may read and work on the data generated, forming "pipelines" of jobs.

The users Grid entry point is called an User Interface (UI). This is the gateway to Grid services. From this machine, users are given the capability to submit jobs to the Grid and to follow their jobs status [12].

When they submit a job they usually don't specify where it will run, they even don't know about the queues where the job can run at the time of job submittal. In fact they could know about it by asking to the Information System of sites but they usually don't because the Resource Broker (RB) does that for them. When an user submits a job he submits it with a corresponding file describing the jobs attributes and requirements, that file is called a JDL file. In that file one could define the job runtime estimate but this is not mandatory.

A Computing Element (CE) is composed of Grid queues. A Computing Element is built on a homogeneous farm of computing nodes called Worker Nodes (WN) and on a node called a GateKeeper acting as a security front-end to the rest of the Grid. The RB submits the job to the best queue found and translates the JDL to talk to the Job Submission Service located on the site GateKeeper.

Users can query the Information System in order to know both the state of different grid nodes and where their jobs are able to run depending on job requirements. This match-making process has been packaged as a Grid service known as the Resource Broker (RB). Users could either submit their jobs directly to different sites or to a central Resource Broker which then dispatches their jobs to matching sites.

The services of the Workload Management System (WMS) are responsible for the acceptance of job submits and the dispatching of these jobs to the appropriate CEs, depending on job requirements and on available resources. The Resource Broker is the machine where the WMS services run, there is at least one RB for each VO. The duty of the RB is to find the best resource matching the requirements of a job (match-making process). Load balancing is done in EGEE globally with the use of the Resource Broker: User could specify a Rank formula in order to sort all the CE and to choose the best one. Different choosing strategy may be used: "Choose the site with the biggest number of free CPUs", "Choose

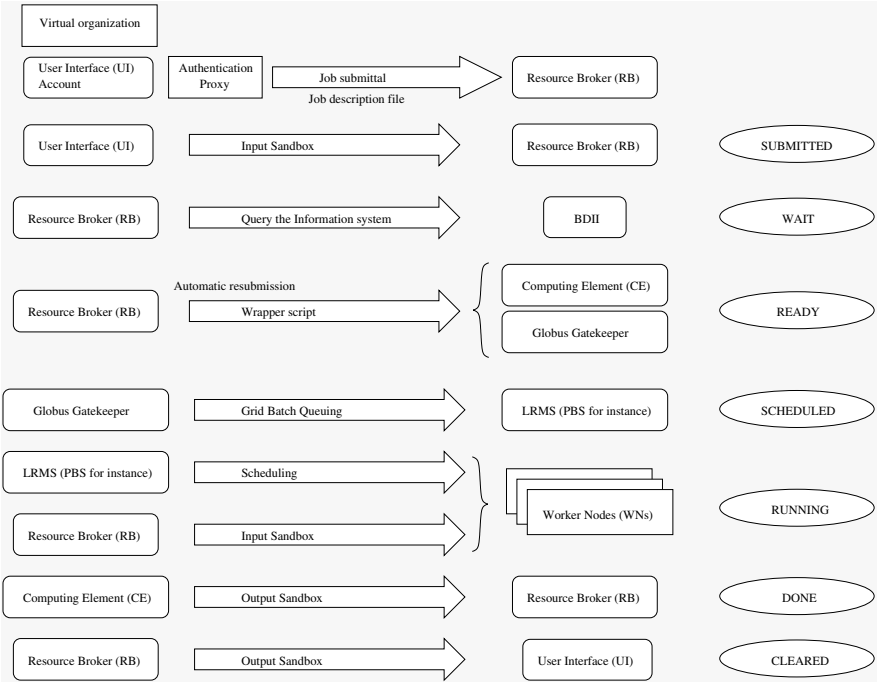


Fig. 1. Job submittal scenario

the site randomly", "Choose the site with the least mean waiting time", "Choose the site with the least number of waiting jobs", etc. (For more details read [13])

Users are then mapped to a local account on the chosen executing CE. When a CE receives a job, it enqueues it inside an appropriate batch queue, chosen depending on the job requirements, for instance depending on the maximum running time. A scheduler then proceeds all these queues to decide the execution of jobs. Users could question about status of their jobs during all the job lifetime.

3 Scheduling Scheme

The goal of the scheduler is first to enable execution of jobs, to maximize job throughput and to maintain a good equilibrium between users in their usage of the cluster [14]. At the same time scheduler has to avoid starvation, that is jobs, users or groups that access scarcely to available cluster resources compared to others.

Scheduling is done on-line, i.e the scheduler has no knowledge about all the job input requests but jobs are submitted to the cluster at arbitrary time. No preemption is done, the cluster uses a space-slicing mode for jobs: one job gets a dedicated CPU for its use and the batch system manage to kill job which exceeds their time limits. We have dual processors, we allow 2 jobs per machines, one

for each CPU on it. In a Grid environment long-time running jobs are common. The worst case is when the cluster is full of jobs running for days and at the same time receiving jobs blocked in the waiting queue.

Short jobs like monitoring jobs barely delay too much longer jobs. For example, a 1 day job could wait 15 minutes before starting, but it is unwise if a 5 minutes job has to wait the same 15 minutes. This results in production of algorithms classes that encourage the start of short jobs over longer jobs. (Short jobs have higher priority [15]) Some other solution proposed is to split the cluster in static sub-clusters but this is not compatible with a sharing vision like Grids. Ideal on-line scheduler will maximize cluster usage and fairness between groups and users. Of course a good trade-off has to be found between the two.

3.1 Local Situation

First our scheduling scheme is not LPC specific. It is common for most of the 150 sites part of the EGEE Grid because of the underlying middleware architecture constraints. They mostly differ from us only by the priorities granted to the different types of jobs, groups or users.

We are using two servers to manage our 140 CPUs, on each machine there are 5 queues where each group could send their jobs to. Each queue has its own limit in maximum CPU Time. A job in a given queue is killed if it exceeds its queue time limit. There are in fact two limits, one is the maximum CPU time, the other one is the maximum total time (or Wall time) a job could use. For each queue there is also a limit in the number of jobs than can run at a given time. This is done in order to avoid that the cluster is full with long running jobs and short jobs cannot run before days. Likely there is the same limit in number of running jobs for a given group.

Table 1. Queue configuration (maximum CPU time, Wall time, running jobs and priority)

Queue	Max CPU (H:M)	Max Wall (H:M)	Max Jobs	Priority
Test	00:05	00:15	130	1,000,000
Short	00:20	01:30	130	270,000
Long	08:00	24:00	130	10,000
Day	24:00	36:00	130	0
Infinite	48:00	72:00	130	-360,000

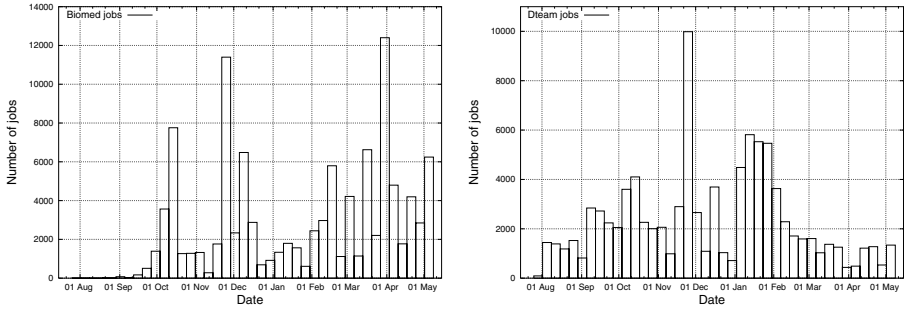
Maui Scheduler and the Portable Batch System (PBS) run on multiple hardware and operating systems. MAUI [7] is a scheduling policy engine that is used together with the PBS batch system. PBS manages the job reception in queues and execution on cluster nodes. MAUI is a priority based scheduler but it is unfortunately not event driven, it polls regularly the PBS queues to decide which jobs to run. MAUI allows to add a priority property for each queue. Our site

configuration is that the shorter the queue allows jobs to run, the more priority is given to that job. MAUI sees all queues as only one queue with priorities. Jobs are then selected to run depending on a priority based on the job attributes such as owner, group, queue, waiting time, etc. If a job violates a site policy it is placed temporary in a blocked state and temporarily not considered for scheduling. To avoid starvation, jobs get a priority bonus proportionnaly with their waiting time. The waiting time bonus is 10 for each waiting seconds. This allow old waiting jobs to jump to the top of the queue.

4 Workload Data Analysis

Workload analysis allows to obtain a model of the user behavior [16]. Such a model is essential for understanding how the different parameters change the resource center usage. Meta-computing workload [17] like Grid environments is composed of different site workloads. We are interested in modelling workload of our site which is part of the EGEE computational Grid. Our site receives only jobs coming from the EGEE Grid and each requests for only one CPU.

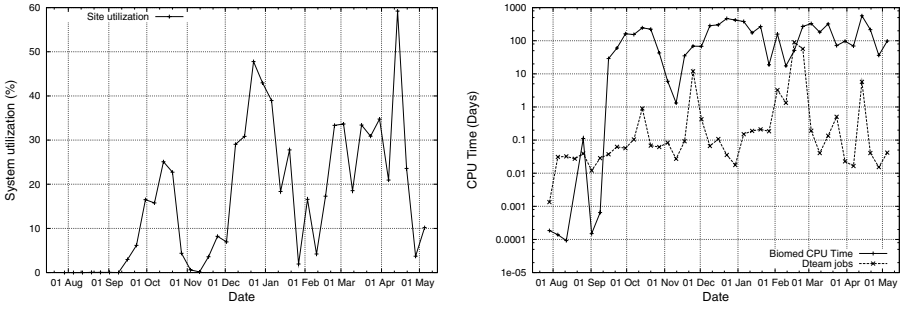
Traces of users activities are obtained from accountings on the server logs. Logs contain information about users, resources used, jobs arrival time and jobs completion time. It is possible to use directly these traces to obtain a static simulation or to use a dynamic model instead. Workload models are more flexible than logs, because they allow to generate traces with different parameters and better understand workload properties [1]. Workload analysis allows to obtain a model of users activity. Such a model is essential for understanding how the different parameters change the resource center usage. Our workload data has been converted to the Standard Workload Format (`http://www.cs.huji.ac.il/labs/parallel/workload/`) and made publicly available for further researches. A more detailed description of the scheduling scheme is available at the same place.



(a) Number of Biomed jobs received per weeks (from August 2004 to May 2005)

(b) Number of Dteam jobs received per weeks (from August 2004 to May 2005)

Fig. 2. Number of jobs received per VO and per week from August 2004 to May 2005



(a) System utilization per weeks (from August 2004 to May 2005)

(b) CPU consumed by Biomed and Dteam jobs per weeks (from August 2004 to May 2005)

Fig. 3. Cluster utilization as CPU consumed per VO and per week from August 2004 to May 2005

Workload is from August 1st 2004 to May 15th 2005. We moved to a cluster containing 140 CPUs since September 15th. This can be visible in the Fig. 2, 3(a) and 3(b), where we notice that the number of jobs sent increases. Statistics are obtained from the PBS log files. PBS log files are well structured for data analysis. An AWK script is used to extract information from PBS log files. AWK acts on lines matched by regular expressions. We do not have information about users *Login* time because users send jobs to our cluster from an User Interface (UI) of the EGEE Grid and not directly. There is no indication of which job is part of the same job and whether it has other parts on other machines from the site log files.

4.1 Running Time

During 280 days, our site received 230474 jobs from which 94474 Dteam jobs and 108197 Biomed jobs (table 2). For all these jobs there are 23208 jobs that failed and were dequeued. It appears that jobs are submitted irregularly and by bursts, that is lot of jobs submitted in a short period of time followed by a period of relative inactivity. From the logs, there are not much differences between CPU time and total time, so jobs sent to our cluster are really CPU intensive jobs and not I/O intensive. I/O is very important for us because there is a lot of file transfert before a job running on a node, they fetch their files from some Storage Element (SE) if needed before execution. We have pipelines of jobs, fetching and writing their data to SEs. Knowing that it does not take much time compared to the whole execution time is important.

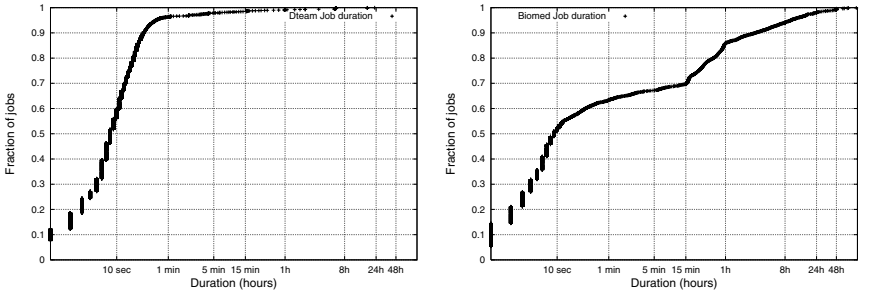
Dteam jobs are mainly short monitoring jobs but all Dteam jobs are not regularly sent jobs. We have 6784.6 days CPU time consumed by Biomed for 108197 jobs (Mean of one hour and half per jobs, table 2). Repartition of cumulative job duration distributions for Biomed VO is shown on Fig. 4. The duration of about 70% of Biomed jobs are less than 15 minutes and 50% under 10 seconds,

Table 2. Group running time in seconds, Total number of jobs submitted. Group mean waiting time in seconds, corresponding Standard Deviation and Coefficient of Variation.

Group	Number of jobs	Mean Runtime (s)	Standard Deviation	Mean Waiting time (s)	Standard Deviation	CV
Biomed	108197	5417	22942.2	781.5	16398.8	20.9
Dteam	94474	222	3673.6	1424.1	26104.5	18.3
LHCb	9709	2072	7783.4	217.7	2000.7	9.1
Atlas	7979	13071	28788.8	2332.8	13052.1	5.5
Dzero	1332	213	393.9	90.7	546.3	6.0

there are a dominant number of small running jobs but the distribution is very wide as shown by the high standard deviation compared to the mean in table 2.

Users submit their jobs with an estimated run length. For relationships between execution time and requested job duration and its accuracy see [18]. To sum up estimated jobs duration are essentially inaccurate. It is in fact an upper bound for job duration which could in reality take any value below it. Table 3 shows for each queue the mean running time, its standard deviation and coefficient of variation (CV) which is the ratio between standard deviation and the mean. CV decreases as the queue maximum runtime increase. This means that jobs in shorter queues vary a lot in their duration compared to longer jobs and we can expect that more the upper bound given is high the more confidence in using the queue mean runtime as a an estimation we could have.



(a) Dteam job runtime (b) Biomed job runtime

Fig. 4. Dteam and Biomed job runtime distributions (logscale on time axis)

A commonly used method for modelling duration distribution is to use log-uniform distribution. Figures 4(a) and 4(b) show the fraction of Dteam and Biomed jobs with duration less than some value. Job duration has been modelled with a multi-stage log-uniform model in [19] which is piecewise linear in log space. In this case Dteam and Biomed job duration could be approximated respectively with a 3 and a 6 stages log-uniform distribution.

Table 3. Queue number of job, mean running time in seconds, corresponding Standard Deviation and Coefficient of Variation, Queue mean waiting time in seconds, corresponding Standard Deviation, Coefficient of Variation

Queue	Number of jobs	Mean Running time (s)	Standard Deviation	CV	Mean Waiting time (s)	Standard Deviation	CV
Test	45760	31.0	373.6	12.0	33335.9	148326.4	4.4
Short	81963	149.5	1230.5	8.2	1249.7	27621.8	22.1
Long	32879	2943.2	11881.2	4.0	535.1	5338.8	9.9
Day	19275	6634.8	25489.2	3.8	466.8	8170.7	17.5
Infinite	49060	10062.2	30824.5	3.0	1753.9	24439.8	13.9

4.2 Waiting Time

Table 2 shows that jobs coming from the Dteam group are the more unfairly treaten. Dteam group sends short jobs very often, Dteam jobs are then all placed in queue waiting that long jobs from other groups finished. Dzero group sends short jobs more rarely and is also less penalized than Dteam because there are less Dzero jobs that are waiting together in queue before being treated. The best treated group is LHCb with not very long running jobs (average of about 34 minutes) and one job about every 41 minutes. The best behavior to reduce waiting time per jobs seems to send jobs that are not too short compared to the waiting factor, and send not too very often in order to avoid that they all wait together inside a queue. Very long jobs is not a good behavior too as the scheduler delay them to run shorter one if possible.

Table 3 shows the mean waiting time per jobs on a given queue. There is a problem with such a metric, for example: Consider one job arriving on a cluster with only one free CPU, it will run on it during a time T with no waiting time. Consider now that this job is splitted in N shorter jobs (numbered $0 \dots N-1$) with equal total duration T. Then the waiting time for the job number i will be iT/N, and the total waiting time $(N-1)T/2$. So the more a job is splitted the more it will wait in total. Another metric that does not depend on the number of jobs is the total waiting time divided by the number of jobs and by the total job duration. Let note $\widehat{WT}$ this normalized waiting time, We obtain:

$$\widehat{WT} = \frac{TotalWaitingTime}{NJobs * TotalDuration} = \frac{MeanWaitingTime}{NJobs * MeanDuration}. \quad (1)$$

With this metric, the Test queue is still the most unfairly treated and the Infinite queue has the more benefits compared to the other queues. Dteam group is again bad treated because their jobs are mainly sent to the Test queue. The more unfairly treated group is Dzero.

These results demonstrate that despite the fact that the middleware advises to set priorities to queues as a workaround, it does not prevent the fact that shorter jobs are unfairly treaten because there is no kind of preemption done. If the site is full with long jobs, job on the top of the waiting queue still has to wait until the first job finished before starting whatever priorities are in effect.

Table 4. Queue and Group normalized waiting time

Queue	$\widehat{WT}$	Group	$\widehat{WT}$
Test	2.35e-2	Biomed	1.58e-5
Short	1.02e-4	Dteam	6.79e-5
Long	5.53e-6	LHCb	1.08e-5
Day	3.65e-6	Atlas	2.23e-5
Infinite	3.55e-6	Dzero	31.9e-5

The intuition behind this measure is that the correct metric in a Grid environment has to be an *user-centric metric*, for instance a *computing flow rate* granted to that user as suggested below. What is really important for an user is the quantity of computing power granted to him. Grids are a way to grant computing power to different users, by having a way to control the part granted to everyone we could propose real fairness as a Quality of Services in Grids. But for the moment there is no warranty with the current middleware.

4.3 Arrival Time

Job arrival daily cycle is presented in Fig. 5. This figure shows the number of arrival depending on job arrival hours, with a 10 minutes sampling. Clearly users prefer to send their jobs at o'clock. In fact we receive regular monitoring jobs from the VO Dteam. The monitoring jobs are submitted every hour from `goc.grid-support.ac.uk`. Users are located in all Europe, so the effect of sending at working hours is summed over all users timezones. However the shape is similar compared to other daily cycle, during night (before 8am) less jobs are submitted and there is an activity peak around midday, 2pm and 4pm.

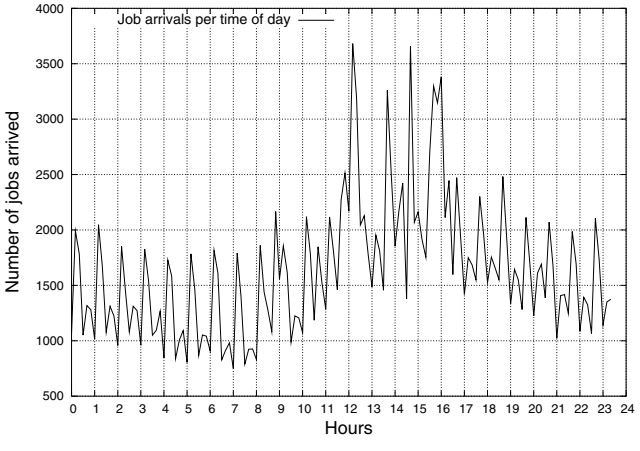


Fig. 5. Job arrival daily cycle

Table 5. Group interarrival time in seconds, corresponding Standard Deviation and Coefficient of Variation

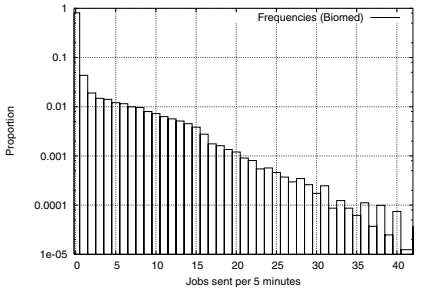
Group	Mean (seconds)	Standard Deviation	CV
Biomed	223.6	5194.5	23.22
Dteam	256.2	2385.4	9.31
LHCb	2474.6	39460.5	15.94
Atlas	2824.1	60789.4	21.52
Dzero	5018.7	50996.6	10.16

Table 5 shows the moments of interarrival time for each group. CV is much higher than 1, this means that arrivals are not Poisson processes and are very irregularly distributed. For instance we could receive 10 jobs in 10 minutes followed by nothing during the 50 next minutes. In this case we have a mean interarrival time of 6 minutes but in fact when jobs arrived they arrived every minutes.

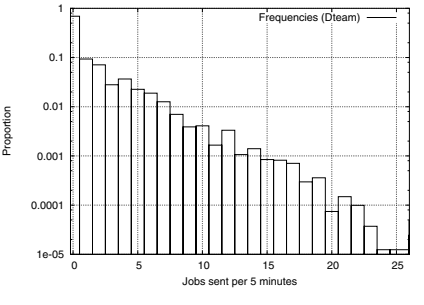
Figure 3(a) shows the system utilization of our cluster during each week. There are a maximum of 980 CPU days consumed each week for 140 CPUs. We have a highly varying cluster activity.

4.4 Frequency Analysis

Job arrival rate is a common measurement for a site usage in queuing theory. Figures 6(a) and 6(b) present the job arrival rate distribution. It is the number of time n jobs are submitted during interval length of 5 minutes. They show that most of the time the cluster does not receive jobs but jobs arrived grouped. Users actually submit groups of jobs and not stand-alone jobs. It is as a fact very common that users send thousands of jobs at once in a Grid environment



(a) Biomed job arrival rate each 5 minutes



(b) Dteam job arrival rate each 5 minutes

Fig. 6. Arrival frequencies for Biomed and Dteam VOs (Proportion of occurrences of n jobs received during an interval of 5 minutes)

because they tend to see the Grid as providing infinite resources. It explains the shape of the arrival rate: it fastly decreases but too slowly compared to a Poisson distribution. Poisson distribution is usually used for modelling the arrival process but evidences are against that fact [20].

Dteam monitoring jobs are short and regular jobs, there is no need of a special arrival model for such jobs. What we observe for other kind of jobs is that the job arrival law is not a Poisson Law (see table 5 where $CV \gg 1$) as for instance a web site traffic [21]. What really happens is that users come using the cluster from an User Interface during some time interval. During this time they send jobs to the cluster. Users log to an User Interface machine in order to send their jobs to a RB that dispatch them to some CEs. Note that one can send jobs to our cluster only from an User Interface, it means for instance that jobs running on a cluster cannot send secondary jobs. On a computing site we do not have this user login information, but only job arrival.

First we look at modelling user arrival and submission behavior. Secondly we show that the model proposed shows good results for a group behavior.

5 Model

5.1 Login Model

In this section we begin to model user *Login/Logout* behavior from the Grid job flow (Fig. 1). We neglect the case where an user has multiple login on different UI at the same time. We mean that a user is in the state *Login* if he is in the state of sending jobs from an UI to our cluster, else he is in the state *Logout*.

Markov chains are like automatons with for each state a probability of transition. One property of Markov chains is that future states depend only on the current state and not on the past history. So a Markov state must contains all the information needed for future states. We decided to model the *Login/Logout* behavior as a continuous Markov chain. During each dt, a *Logout* user has a probability during dt of λdt to login and a *Login* user has a probability during

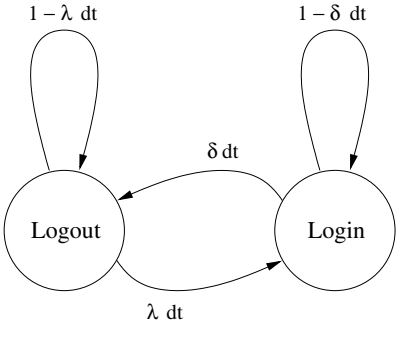


Fig. 7. *Login/Logout* cycle

dt of δdt to logout (see Fig. 7). λ is called the *Login* rate and δ is called the *Logout* rate.

All these parameters could vary over time as we see with the variation of the week job arrival (Fig. 3(a)) or during day time (Fig. 5) The Markov model proposed could be used more accurately with non-constant parameters at the expense of more calculation and more difficult fitting. For example, one could numerically use Fourier series for the *Login* rate or for the submittal rate to model this daily cycle. Another solution would be to take the parameters to be constants during the time interval $[h; h + t]$ for t small enough and study the frequencies with our model for that particular time interval. We use now constant parameters for calculation, looking for general properties.

We would like to have the probabilities during time that the user is logged or not logged. Let $\mathcal{P}_{Login}(t)$ and $\mathcal{P}_{Logout}(t)$ be respectively probability that the user is logged or not logged at time t. We have from the modelling:

$$\mathcal{P}_{Logout}(t + dt) = (1 - \lambda dt)\mathcal{P}_{Logout}(t) + \delta dt\mathcal{P}_{Login}(t) \tag{2}$$

$$\mathcal{P}_{Login}(t + dt) = (1 - \delta dt)\mathcal{P}_{Login}(t) + \lambda dt\mathcal{P}_{Logout}(t) \tag{3}$$

At equilibrium we have no variation so

$$\mathcal{P}_{Logout}(t + dt) = \mathcal{P}_{Logout}(t) = \mathcal{P}_{Logout} \tag{4}$$

$$\mathcal{P}_{Login}(t + dt) = \mathcal{P}_{Login}(t) = \mathcal{P}_{Login} \tag{5}$$

We obtain:

$$\mathcal{P}_{Logout} = \frac{\delta}{\lambda + \delta} \quad \mathcal{P}_{Login} = \frac{\lambda}{\lambda + \delta} \tag{6}$$

5.2 Job Submittal Model

During period when users are logged they could submit jobs. We model the job submittal rate for one user as follows: During dt when the user is logged he has a probability of μdt to submit a job. With $\delta = 0$ we have a delayed Poisson process, with $\mu = 0$ no jobs are submitted. The full model is shown at Fig. 8, it shows all the possible outcomes with corresponding probabilities from one of the possible state to the next after a small period dt. Numbers inside circles are the number of jobs submitted from the start. *Login* states are below and *Logout* states are at the top. We have:

- $\mathcal{P}_n(t)$ is the probability to be in the state "*User is not logged at time t and n jobs have been submitted between time 0 and t.*"
- $\mathcal{Q}_n(t)$ is the probability to be in the state "*User is logged at time t and n jobs have been submitted between time 0 and t.*"
- $\mathcal{R}_n(t)$ is the probability to be in the state "*n jobs have been submitted between time 0 and t.*" We have $\mathcal{R}_n = \mathcal{P}_n + \mathcal{Q}_n$.

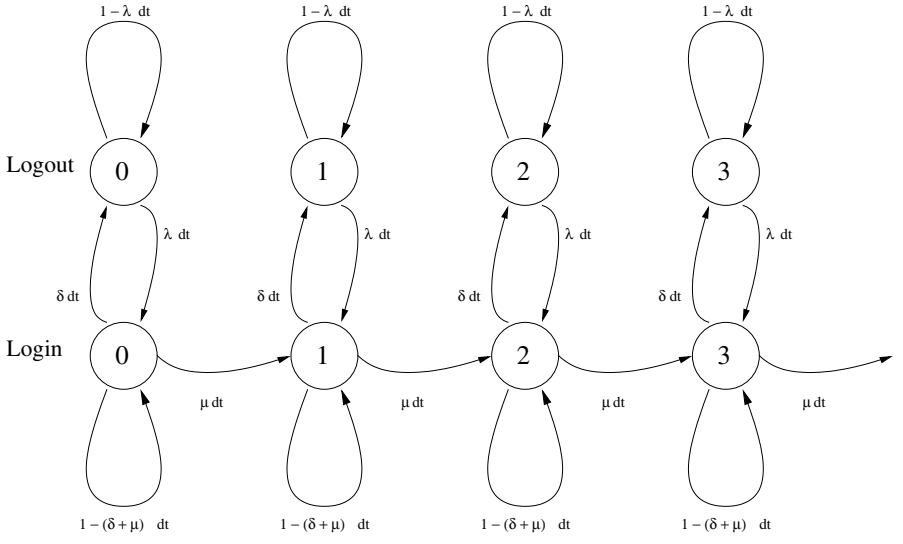


Fig. 8. Markov modelling of jobs submittal

From the model, we obtain with the same method as before this recursive differential equation:

$$\mathcal{M} = \begin{pmatrix} -\lambda & \delta \\ \lambda & -(\mu+\delta) \end{pmatrix} \tag{7}$$

$$\begin{pmatrix} \mathcal{P}_0 \\ \mathcal{Q}_0 \end{pmatrix}' = \mathcal{M} \begin{pmatrix} \mathcal{P}_0 \\ \mathcal{Q}_0 \end{pmatrix} \tag{8}$$

$$\begin{pmatrix} \mathcal{P}_n \\ \mathcal{Q}_n \end{pmatrix}' = \mathcal{M} \begin{pmatrix} \mathcal{P}_n \\ \mathcal{Q}_n \end{pmatrix} + \begin{pmatrix} 0 \\ \mu\mathcal{Q}_{n-1} \end{pmatrix} \tag{9}$$

This results to the following recursive equation (in case the parameters are constants, $\mathcal{M}$ is a constant)

$$\begin{pmatrix} \mathcal{P}_n \\ \mathcal{Q}_n \end{pmatrix} = e^{\mathcal{M}t} \int e^{-\mathcal{M}x} \begin{pmatrix} 0 \\ \mu\mathcal{Q}_{n-1} \end{pmatrix} dx. \tag{10}$$

We take a look at the probability of having no job arrival during an interval of time t which is $\mathcal{P}_0$ and $\mathcal{Q}_0$. $\mathcal{R}_0$ is the the probability that no jobs have been submitted between arbitrary time 0 and t. So from the above model, we have:

$$\begin{pmatrix} \mathcal{P}_0 \\ \mathcal{Q}_0 \end{pmatrix}' = \begin{pmatrix} -\lambda & \delta \\ \lambda & -(\mu+\delta) \end{pmatrix} \begin{pmatrix} \mathcal{P}_0 \\ \mathcal{Q}_0 \end{pmatrix} \tag{11}$$

At arbitrary time we could be in the state *Login* with probability $\lambda/(\lambda+\delta)$ and in the state *Logout* with the probability $\delta/(\lambda+\delta)$. We have from the results above:

$$\begin{pmatrix}\mathcal{P}_0(0)\\ \mathcal{Q}_0(0)\end{pmatrix} = \begin{pmatrix}\mathcal{P}_{Logout}\\ \mathcal{P}_{Login}\end{pmatrix} = \frac{1}{\lambda+\delta}\begin{pmatrix}\delta\\ \lambda\end{pmatrix} \tag{12}$$

$$\mathcal{R}_0 = \mathcal{P}_0 + \mathcal{Q}_0 \tag{13}$$

Finally we obtain the result.

$$\mathcal{R}_0(t) = m_0\frac{e^{-m_1 t} - e^{-m_2 t}}{m_1 - m_2} + \frac{m_1 e^{-m_2 t} - m_2 e^{-m_1 t}}{m_1 - m_2}. \tag{14}$$

Where

$$m_0 = \frac{\lambda\mu}{\lambda+\delta} = \mu\mathcal{P}_{Login} \tag{15}$$

$$m_1 + m_2 = \lambda + \delta + \mu \tag{16}$$

$$m_1 m_2 = \lambda\mu \tag{17}$$

$$\tag{18}$$

With $\lambda = 0$ or $\mu = 0$, we obtain that no jobs are submitted ($\mathcal{R}_0(t) = 1$). With $\delta = 0$, this is a Poisson process and $\mathcal{R}_0(t) = e^{-\mu t}$. Note that during a period of t there are in average $\mu\mathcal{P}_{Login}t$ jobs submitted, we have also for small period t,

$$\mathcal{R}_0(t) \approx 1 - \mu\mathcal{P}_{Login}t. \tag{19}$$

We have also

$$\mathcal{R}_0'(0) = -\mu\mathcal{P}_{Login} \tag{20}$$

$$\mathcal{R}_0'(0) = -\frac{\textit{Number of jobs submitted}}{\textit{Total duration}} \tag{21}$$

$\mathcal{R}_0(t)$ could be estimated by splitting the arrival processes in intervals of duration t and estimating the ratio of intervals with no arrival. The error of this estimation is linear with t. Another issue is that the logs precision is not below one second.

5.3 Model Characteristics

We have also these interesting properties:

$$\mathcal{R}_0'(0) = -\mu\mathcal{P}_{Login} \quad \frac{\mathcal{R}_0''(0)}{\mathcal{R}_0'(0)} = -\mu \tag{22}$$

Probability distribution of the duration between two jobs arrival is called an interarrival process. Interarrival process is a common metric in queuing theory. We have $\mathcal{A}(t) = \mathcal{P}_0(t) + \mathcal{Q}_0(t)$ with the initial condition that user just submits a job. This implies that user is logged.

$$\mathcal{P}_0(0) = 0.0, \quad \mathcal{Q}_0(0) = 1.0$$

$$\mathcal{A}(t) = \mu \frac{e^{-m_1 t} - e^{-m_2 t}}{m_1 - m_2} + \frac{m_1 e^{-m_2 t} - m_2 e^{-m_1 t}}{m_1 - m_2}. \tag{23}$$

$$p = \frac{\mu - m_2}{m_1 - m_2} \tag{24}$$

$$\mathcal{A}(t) = p e^{-m_1 t} + (1-p) e^{-m_2 t} \tag{25}$$

We have $\mu \in [m1; m2]$ because

$$\begin{aligned} (\mu - m_1)(\mu - m_2) &= \mu^2 - (\lambda + \delta + \mu)\mu + \delta\mu \\ (\mu - m_1)(\mu - m_2) &= -\delta\mu < 0 \end{aligned} \tag{26}$$

So $p \in [0; 1]$, and we have an hyper-exponential interarrival law of order 2 with parameters $p = (\mu - m_2)/(m_1 - m_2), m_1, m_2$. This result is coherent with other experimental fitting results [22] Moreover any hyper-exponential law of order 2 may be modelled with the Markov chain described in Fig. 8 with parameters $\mu = pm_1 + (1-p)m_2, \lambda = m_1 m_2/\mu, \delta = m_1 + m_2 - \mu - \lambda$.

Let calculate the mean interarrival time. Probability to have an interarrival time between θ and $\theta + d\theta$ is $\mathcal{A}(\theta) - \mathcal{A}(\theta + d\theta) = -\mathcal{A}'(\theta)d\theta$. The mean is

$$\tilde{\mathcal{A}} = \int_0^\infty -\theta \mathcal{A}'(\theta) d\theta = \int_0^\infty \mathcal{A}(\theta) d\theta. \tag{27}$$

$$\tilde{\mathcal{A}} = \frac{1}{\mu \mathcal{P}_{Login}} = \frac{\lambda + \delta}{\lambda \mu}. \tag{28}$$

Let compute the variance of interarrival distribution.

$$var = \int_0^\infty -(\theta - \tilde{\mathcal{A}})^2 \mathcal{A}'(\theta) d\theta \tag{29}$$

$$var = 2 \int_0^\infty \theta \mathcal{A}(\theta) d\theta - \tilde{\mathcal{A}}^2 \tag{30}$$

$$\frac{var}{\tilde{\mathcal{A}}^2} = CV^2 = 1 + 2\frac{\delta\mu}{(\lambda + \delta)^2} \tag{31}$$

$$CV^2 = 1 + 2\ \mathcal{P}_{Logout}^2 \frac{\mu}{\delta} \tag{32}$$

Another interesting property is the number of jobs submitted by this model during a *Login* period. Let P_n be the probability to receive n jobs during a *Login* period. We have:

$$P_n = \int_0^\infty \frac{(\mu t)^n}{n!} e^{-\mu t} \delta e^{-\delta t} dt = \frac{\delta}{\mu + \delta} (\frac{\mu}{\mu + \delta})^n. \tag{33}$$

This is a geometric law. The mean number of jobs submitted by *Login* period is μ/δ.

5.4 Group Model

Groups are composed of users, either regular users sending jobs at regular time or users with a *Login/Logout* like behavior. Metrics defined below as the mean number of jobs sent by *Login* state, the mean submittal rate and probability of *Login* could represent an user behavior.

Figure 9 shows the sorted distribution of users submittal rate ($\mu\mathcal{P}_{Login}$). Except for the highest values it is quite a straight line in logspace. This observation could be included in a group model.

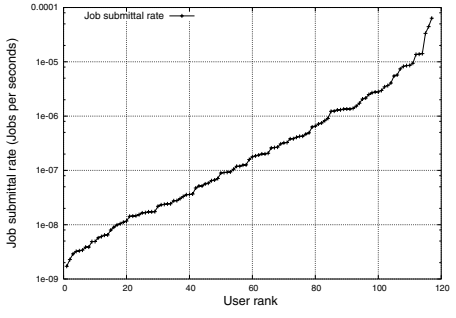


Fig. 9. Users job submittal rates during their period of activity

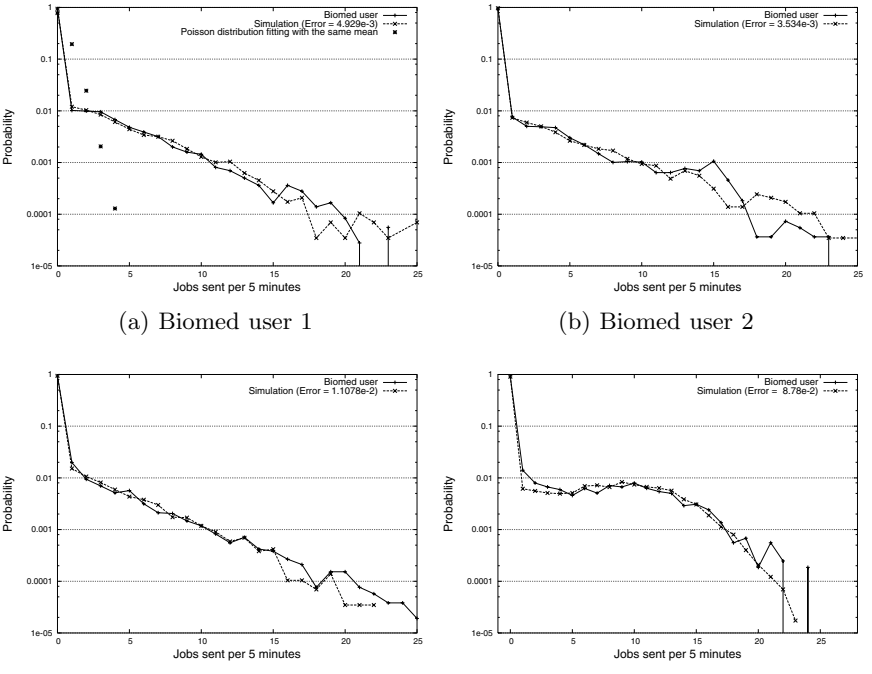
6 Simulation and Validation

We have done a simulation in Scheme [23] directly using the Markov model. We began by fitting users behavior from the logs with our model. Like the frequency obtained from the logs, the model shows a majority of intervals with no job arrival, possibly followed by a relatively flat area and a fast decreases. Some fitting results are presented in Fig. 6. Norm used to fit real data is the maximum difference between the two cumulative distributions. We fitted the frequency data for each user.

During a period of t there are in average $\mu\mathcal{P}_{Login}t$ jobs submitted. We evaluate the value of $\mu\mathcal{P}_{Login}$ which is the average number of jobs submitted by seconds. We use that value when doing a set of simulation in order to fit a known real user probability distribution. We have two free parameters, so we vary $\mathcal{P}_{Login}$ between 0.0 and 1.0 and *lambda* which the inverse is the average time an user is *Logout*. Some results obtained are shown in Fig. 6.

μ parameter decides of the frequency length of the curve. Without the *Login* behavior we would have obtained a classic Poisson curve of μ parameter. $1/\mu$ is the mean interarrival time during *Login* period. An idea to evaluate μ would be to evaluate the job arrival rate during *Login* periods, but we lack that *Login* information.

δ and λ are the *Logout* and *Login* parameters. What is really important is the ratio $\lambda/(\lambda+\delta)$ which is $\mathcal{P}_{Login}$. This is the ratio between time user is active



(a) Biomed user 1 (b) Biomed user 2

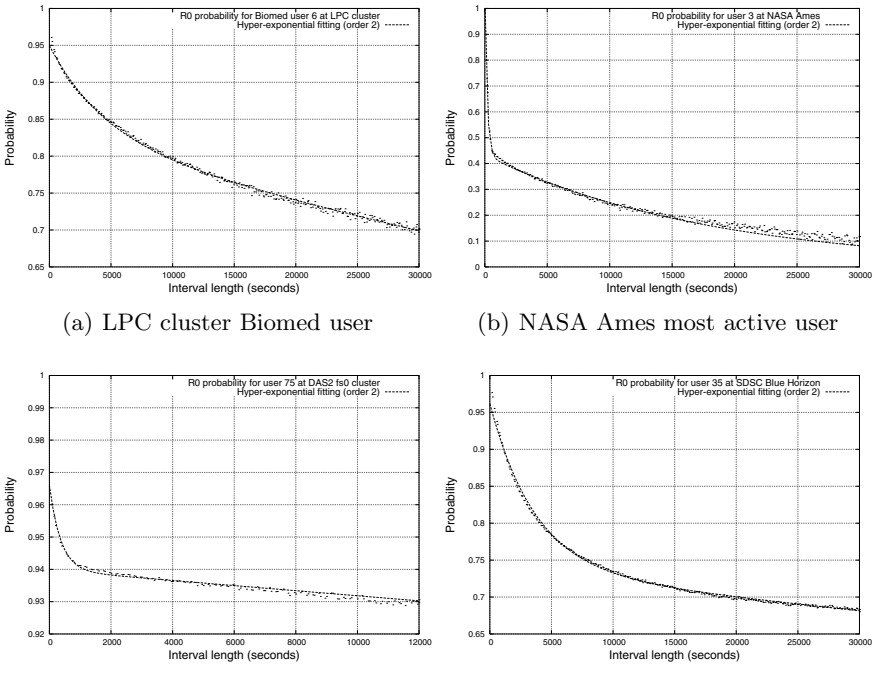
(c) Biomed user 3 (d) Biomed user 4

Name	μ	δ	λ	Error
Biomed user 1	0.0837	2.079e-2	2.1e-4	4.929e-3
Biomed user 2	0.0620	1.188e-2	1.2e-4	3.534e-3
Biomed user 3	0.0832	2.475e-2	2.5e-4	1.1078e-2
Biomed user 4	0.0365	1.428e-3	1.075e-4	8.78e-2

Fig. 10. Biomed simulation results

on the cluster and total time. δ and $\mathcal{P}_{Login}$ are measures of the deviation from a classic Poisson law. For instance, the mean number of jobs submitted by *Login* period is μ/δ and the mean job submittal rate is $\mu\mathcal{P}_{Login}$. For a same $\mathcal{P}_{Login}$ we could have very different scenarios. A user could be active for long time but rarely logged and another user could be active for short period with frequent login. $1/\delta$ is the mean *Login* time, $1/\lambda$ is the mean *Logout* time.

The $\mathcal{R}_0$ probability is essential for studying job arrival time. $1 - \mathcal{R}_0(t)$ is the probability that between time 0 and t we have received at least one job. It is easier to fit the $\mathcal{R}_0$ distribution for an user than the interarrival distribution because we have more points. Figure 11(a) shows a typical graph of $\mathcal{R}_0$ for a Biomed user. It shows for instance that for intervals of 10000 seconds, this Biomed user has a probability of about 0.2 to submits one or more jobs. We have fitted this probability with hyper-exponential curve, that is a summation of exponential curves. There was too much noises for high interval time to fit that curve. In fact errors on $\mathcal{R}_0$ are linear with t. So we have smoothed the curve



(a) LPC cluster Biomed user

(b) NASA Ames most active user

(c) DAS2 fs0 cluster most active user

(d) SDSC Blue Horizon most active user

Fig. 11. Hyper-Exponential fitting of $\mathcal{R}_0$ for a Biomed LPC user and for the most active users at NASA Ames, DAS2 and SDSC Blue Horizon clusters

before fitting by averaging near points. $\mathcal{R}_0$ for this user was fitted with a sum of two exponentials.

It seems that more than the *Login*/*Logout* behavior there is also a notion of user activity. For example during the preparation of jobs or analysis phase of the results an user does not use the Grid and consequently the cluster at all. More than the *Login* and *Logout* state an *Inactive* state could be added to the model if needed.

6.1 Other Workloads Comparison

User number 3 is the most active user from the NASA Ames iPSC/860 workload [2]. Figure 11(b) shows its $\mathcal{R}_0(t)$ probability, it is clearly hyper-exponential

[2] The workload log from the NASA Ames iPSC/860 was graciously provided by Bill Nitzberg. The workload logs from DAS2 were graciously provided by Hui Li, David Groep and Lex Wolters. The workload log from the SDSC Blue Horizon was graciously provided by Travis Earheart and Nancy Wilkins-Diehr. All are available at the Parallel Workload Archives `http://www.cs.huji.ac.il/labs/parallel/workload/`.

of order 2, as other users like number 22 and 23. Other users like number 12 and 15 are more classical Poissonian users.

DAS2 Clusters (see note 2) used also PBS and MAUI as their batch system and scheduler. The main difference we have with them is that they use Globus to co-allocate nodes on different clusters. We only have bag of tasks applications which interacts together in a pipeline way by files stored on SEs. Their fs0 144 CPUs cluster is quite similar with ours. Figure 11(c) shows the $\mathcal{R}_0(t)$ probability for their most active user and corresponding hyper-exponential fitting or order 2.

SDSC Blue Horizon cluster (see note 2) have a total of 144 nodes. The $\mathcal{R}_0(t)$ distribution probability of their most active user was fitted with a hyper-exponential of order 2 in Fig. 11(d).

7 Related Works

Our Grid environment is very particular and different from common cluster environment as parallelism involved requires no interaction between processes and degree of parallelism is one for all jobs.

To be able to completely simulate the node usage we need not only the jobs submittal process but also the job duration process. Our runtime model is similar with the Downey model [19] for runtime which is composed of linear pieces in logspace. There is a strong correlation between successive jobs running time but it seems unlikely that a general model for duration may be made because it depends highly on algorithms and data used by users.

Most other models use Poisson distribution for interarrival distribution. But evidences, like CV be much higher than one, demonstrate that exponential distributions does not fit well the real data [24][25]. The need of a detailed model was expressed in [26]. With constant parameters our model exhibits a hyper-exponential distribution for interarrival rate and justify such a distribution choice. One strong benefit of our model is that it is general and could be used numerically with non-constant parameters at the expense of difficult fitting.

8 Discussion

What could be stated is that job maximum run times provided by users are essentially inaccurate, some authors are even not using this information for scheduling [2]. Maybe a better concept is the relative urgency of a job. For example on a grid software managers are people responsible for installing software on cluster nodes by sending installation jobs. Software manager jobs may be regarded as more urgent than other jobs type. So sending jobs with an estimated runtime could be replaced by sending jobs with an urgency parameter. That urgency could be established in part as a site policy. Each site administrator could define some users classes for different kind of jobs and software used with different jobs priorities. For instance a site hosted in some laboratory might wish to promote its scientific domain more than other domain, or some specific applications might need quality of services like real time interaction.

Another idea for scheduling is to have some sort of risk assessment measured during the scheduler decision. This risk assessment may be based on blocking probabilities obtained either from the logs or from some user behavior models. For example, it could be wise to forbid that a group or an user takes all the cluster at a given time but instead to let some few percents of it open for short jobs or low CPU consuming jobs like monitoring. Because users are inclined to regard Grid as providing infinite resources they send thousands jobs at once. The risks of having the cluster taken by few users during a long amount of time is great, unfairly forcing other users/groups to wait.

Information System shows for a site the number of job currently running and waiting. But it is not really the relevant metric in an on-line environment. A better metric for a cluster is the computing flow rate input and the computing flow rate capacity. A cluster is able to treat some amount of computation per unit of time. So a cluster is contributing to the Grid with some computation flow rate (in GigaFLOPS or TeraFLOPS). As with classical queuing theory if the input rate is higher than the capacity, the site is overloaded and the global performances are low due to jobs waiting to be processed. What happens is that the site receive more jobs that is is able to treat in a given time. So the queues begin to grow and jobs have to wait more and more before being started, resulting in performance decay. Similarly when the computation submitted rate is lower than the site capacity the site is under-used. Job submittal have also to be fairly distributed according to the site capacity. For example, a site that is twice bigger than another site have to receive twice more computing request than the other site. But there is a problem to globally enforce this submittal scheme on all the Grid. This is why a local site migration policy may be better than a central migration policy done with the RB.

To be more precise there are two different kinds of cluster flow rate metrics, one is the local flow rate and the other one is the global flow rate. The local computing flow rate is the flow rate that one job sees when reaching the site. The global flow rate is the computing flow rate a group of jobs see when reaching the site. That global flow rate is also the main measurement for meta-scheduling between sites. These two metrics are different, for instance we could have a site with a lot of slow machines (low local flow rate and big global flow rate) and another site with only few supercomputers (big local flow rate and low global flow rate). But the most interesting metric for one job is the local flow rate. This means that if each job wants individually to be processed at the best local flow rate site, this site will saturate and be globally slow.

As far as all users and groups computation total flow rate is less than the site global flow rate or site capacity, there is no real fairness issue because there is no strife to access the site resources, there is enough for all. The problem comes when the sum of all computation flow rate is greater than the site capacity, firstly this globally reduces the site performance, secondly the scheduler must take decision to share fairly these resources. The Grid is an ideal tool that would allow to balance the load between sites by migrating jobs [2]. A site that share their resources and is not saturated could discharge another heavily loaded site. Some

kind of local site flow control could maintain a bounded input rate even with fluctuating jobs submittal. For instance fairness between groups and users could be maintained by decreasing the most demanding input rate and distributing it to other less saturated sites.

Another benefits is that applications computing flow rates may be partly expressed by users in their job requirements. Computing flow rate takes into account both the jobs sizes and their time limits. Fairness between users is then ensured if whatever may be flow values asked by each user, part granted to each penalizes no other one. Computing flow rate granted by a site to an application may depend on the applications degree of parallelism, that is for the moment the number of jobs. For instance it may be more difficult to serve an application composed of only one job asking for a lot of computing flow rate than to serve an application asking the same computing flow rate but composed of many jobs. Urgency is not totally measured by a computing flow rate. For example a critical medical application which is a matter of life or death arriving on a full site has to be treated in priority. Allocating flow rates between users and groups has to be right and to take under account priority or urgency issues.

To use a site wisely users have to bound their computational flow rate and to negotiate it with site managers. A computing model has to be defined and published. These remarks are important in the case of on-line computing like Grids where meta-scheduling strategy have to take a lot of parameters into account. General on-line load balancing and scheduling algorithms [27] [28] [29] [30] may be applied. The problem of finding the best suited scheduling policy is still an open problem. A better understanding of job running time is necessary to have a full model.

The LCG middleware allows users to send their jobs to different nodes. This is done by the way of a central element called a Resource Broker, that collects user's requests and distributes them to computing sites. The main purpose is to match the available resources and balance the load of job submittal requests. Jobs are better localized near the data they need to use.

Common algorithms for scheduling a site rely on such a central view of the site. But it is in fact unrealistic to claim to know the global state of the Grid at a given time in order to schedule jobs. This is why we would like to advise instead a peer to peer [31] view of the Grid over a centralized one. In this view computing sites themselves work together with other computing sites to balance the average workload. Not relying on dependent services greatly improves the reliability and adaptability of the whole systems. That kind of meta-scheduling have to be globally distributed as stated by Dmitry Zotkin and Peter J. Keleher [11]:

> *In a distributed system like Grid, the use of a central Grid scheduler*(like the Resource Broker used in LCG middleware) *may result in a performance bottleneck and lead to a failure of the whole system. It is therefore appropriate to use a decentralized scheduler architecture and distributed algorithm.* (Dmitry Zotkin and Peter J. Keleher)

gLite [32] is the next generation middleware for Grid computing. gLite will provide lightweight middleware for Grid computing. The gLite Grid services fol-

low a Service Oriented Architecture which will facilitate interoperability among Grid services. Architecture details of gLite could be viewed in [10]. The architecture constituted by this set of services is not bound to specific implementations of the services and although the services are expected to work together in a concerted way in order to achieve the goals of the end-user they can be deployed and used independently, allowing their exploitation in different contexts. The gLite service decomposition has been largely influenced by the work performed in the LCG project. Service implementations need to be inter-operable in such a way that a client may talk to different independent implementations of the same service. This can be achieved in developing lightweight services that only require minimal support from their deployment environment and defining standard and extensible communication protocols between Grid services.

9 Conclusion

So far we have analyzed the workload of a Grid enabled cluster and proposed an infinite Markov-based model that describes the process of jobs arrival. Then a numerical fitting has been done between the logs and the model. We find a very similar behavior compared to the logs, even bursts were observed during the simulation.

Acknowledgments

The cluster at LPC Clermont-Ferrand was funded by Conseil Régional d'Auvergne within the framework of the INSTRUIRE project (`http://www.instruire.org`).

References

1. Dror G. Feitelson. Workload modeling for performance evaluation. In Maria Carla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 114–141. Springer-Verlag, Sep 2002. Lect. Notes Comput. Sci. vol. 2459.
2. Darin England and Jon B. Weissman. Costs and benefits of load sharing in the computational grid. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 2004.
3. M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA., 1979.
4. Stephan Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, New York, 2004. Oxford University Press.
5. Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

6. David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
7. Brett Bode, David M. Halstead, Ricky Kendall, and Zhou Lei. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters, USENIX Association. *4th Annual Linux Showcase Conference*, 2000.
8. S. Agostinelli et al. Geant 4 (GEometry ANd Tracking): a Simulation toolkit. *Nuclear Instruments and Methods in Physics Research*, pages 250–303, 2003.
9. Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
10. EGEE Design Team. EGEE middleware architecture, EGEE-DJRA1.1-476451-v1.0, August 2004. Also available as `https://edms.cern.ch/document/476451/1.0`.
11. Dmitry Zotkin and Peter J. Keleher. Job-length estimation and performance in backfilling schedulers. In *HPDC*, 1999.
12. Antonio Delgado Peris, Patricia Méndez Lorenzo, Flavia Donno, Andrea Sciabà, Simone Campana, and Roberto Santinelli. LCG User guide, 2004.
13. G. Avellino, S. Beco, B. Cantalupo, A. Maraschini, F. Pacini, M. Sottilaro, A. Terracina, D. Colling, F. Giacomini, E. Ronchieri, A. Gianelle, R. Peluso, M. Sgaravatto, A. Guarise, R. Piro, A. Werbrouck, D. Kouřil, A. Křenek, L. Matyska, M. Mulač, J. Pospíšil, M. Ruda, Z. Salvet, J. Sitera, J. Škrabal, M. Vocù, M. Mezzadri, F. Prelz, S. Monforte, and M. Pappalardo. The datagrid workload management system: Challenges and results. *Kluwer Academic Publishers*, 2004.
14. Dror G. Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.
15. Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 103–127. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
16. Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8):1136–1150, 1993.
17. Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.
18. Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload, 2001.
19. Allen B. Downey and Dror G. Feitelson. The elusive goal of workload characterization. *Perf. Eval. Rev.*, 26(4):14–29, 1999.
20. Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 337–360. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
21. Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

22. Hui Li, David Groep, and Lex Wolters. Workload characteristics of a multi-cluster supercomputer. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*. Springer Verlag, 2004.
23. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
24. Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
25. Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riodan. Modeling of workload in MPPs. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 95–116. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
26. David Talby, Dror G. Feitelson, and Adi Raveh. Comparing logs and models of parallel workloads using the co-plot method. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 43–66. Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.
27. Yossi Azar, Bala Kalyansasundaram, Serge A. Plotkin, Kirk Pruhs, and Orli Waarts. On-line load balancing of temporary tasks. *J. Algorithms*, 22(1):93–110, 1997.
28. Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. On-line load balancing. *Theoretical Computer Science*, 130(1):73–84, 1994.
29. A. Bar-Noy, A. Freund, and J. Naor. New algorithms for related machines with temporary jobs. In E.K. Burke, editor, *Journal of Scheduling*, pages 259–272. Springer-Verlag, 2000.
30. Tak-Wah Lam, Hing-Fung Ting, Kar-Keung To, and Wai-Ha Wong. On-line load balancing of temporary tasks revisited. *Theoretical Computer Science*, 270(1–2):325–340, 2002.
31. Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.
32. EGEE Design Team. Design of the EGEE middleware grid services. *EGEE JRA1*, 2004. Also available as `https://edms.cern.ch/document/487871/1.0`.

ScoPred–Scalable User-Directed Performance Prediction Using Complexity Modeling and Historical Data

Benjamin J. Lafreniere and Angela C. Sodan

University of Windsor, Windsor ON N9B 3P4, Canada
{lafreni, acsodan}@uwindsor.ca

Abstract. Using historical information to predict future runs of parallel jobs has shown to be valuable in job scheduling. Trends toward more flexible job-scheduling techniques such as adaptive resource allocation, and toward the expansion of scheduling to grids, make runtime predictions even more important. We present a technique of employing both a user's knowledge of his/her parallel application and historical application-run data, synthesizing them to derive accurate and scalable predictions for future runs. These scalable predictions apply to runtime characteristics for different numbers of nodes (processor scalability) and different problem sizes (problem-size scalability). We employ multiple linear regression and show that for decently accurate complexity models, good prediction accuracy can be obtained.

1 Introduction

The typical approach in parallel job scheduling is that users provide estimates about the runtimes of their jobs with such estimates being, in the general case, much higher than the actual runtime (ranging from 20% [12] to 16 times [11] higher in different studies for different supercomputing centers). The availability of more accurate information about runtimes of parallel programs has been shown to be valuable to improve average response times. However, results also exist suggesting that the overestimation of runtimes provides some benefits by creating holes in the schedule that can be filled with short jobs [10]. Thus, the need for accurate estimates in standard job scheduling is not yet fully decided. However, in the context of grid scheduling for simultaneous execution of jobs on multiple sites, reservations of resources on remote sites is required and prediction is unequivocally relevant. The simplest approach to obtain accurate estimates is recording runtimes of previous job executions and using them to predict future runtimes with the same job configuration and the same number of resources. For grid computing, more detailed runtime information may be needed to estimate performance on different systems. Furthermore, grid middleware may be composed of different components, and a more detailed recording of these components (and their performance for certain applications and machines) is needed to support optimal configuration of grid jobs [16].

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 62–90, 2005.

To make matters more complicated, modern job scheduling employs advanced approaches such as flexible time sharing and adaptive resource allocation [6]. Flexible time sharing means relaxing global synchronous gang scheduling if jobs can be matched well [8][4] or could mean abandoning global control altogether [14]. In both cases the aim is improving resource utilization. Adaptive resource allocation means that the number of nodes allocated to a job changes during its runtime, typically driven by the changing workload of the machine [7][2]. Time sharing makes it necessary to know more detailed application characteristics such as the fractions of total runtime spent on communication and I/O, to allow us to find proper matches and estimate slowdowns. For adaptive resource allocation, it becomes relevant to estimate runtime on different numbers of resources; if we can predict resource times with different numbers of nodes, we can better predict the benefits of certain adaptation decisions. This leads to the challenge of generating scalable predictions, i.e. predicting resource times with allocations other than those previously measured. Such scalable prediction is also useful if the user moves to larger problem sizes for which no performance data is available yet.

Several approaches exist which utilize special compilers to provide performance models of applications. A difficulty of this approach is that abstract models may not always be extractable in a fully automated manner, particularly if the code behavior is complex. As well, runtime measurements or simulations are typically needed for quantification of the parameters. Our target is a standard job-execution environment running primarily MPI based applications, and not equipped with any such special compilers. However, we assume that users have some rough knowledge of their applications and know, for example, which parameters in their application determine runtime. Furthermore, users may be able to provide rough cost estimations (closely resembling the steps needed to derive complexity estimates) as suggested in [15]. Whether estimated formulas for performance models are provided by the user or extracted by a special compiler, the system must then perform quantification of coefficients within the formula, to turn the rough estimate into a concrete model for the system being used. This challenge motivates the following goals for our ScoPred performance predictor:

- Support prediction of overall execution times as well as prediction of resource times (computation, communication, I/O)
- Support such prediction on the same and on different numbers of nodes (scaling the prediction)
- Assume that a rough model is available as input such as via cost/complexity estimations from the user (user-directed) or a formula provided by the compiler (compiler-directed) and leave all quantification (determination of coefficients) to the system
- Provide a practically feasible approach
- Provide a mathematically sound approach.

In our ScoPred approach, we apply the following innovative solutions toward meeting these goals:

– Take the performance-determining parameters and a rough resource-usage model as input
– Employ multiple linear regression to determine the coefficients and provide mean values, confidence intervals, and prediction intervals.

2 Related Work

Performance data bases or repositories were first proposed in [9]. Due to a lack of models describing the application behavior, measurements such as runtime or memory usage were used to match the runtime of future runs with previous measurements, i.e. no scaling was applied. The approach is extended in [13] to consider a number of additional criteria such as the application parameters. The difficulty addressed by considering the different criteria is to associate the runtimes with a particular application and the correct instance of that application. This can become difficult as a single user may run the same application with different parameters and under different names [9][13].

Furthermore, to estimate the cost under varying resource allocation, a proper cost model needs to consider the fact that speedup curves are not linear but that the curve typically flattens (and finally declines) if more resources are allocated. In [5], a general statistical model is proposed which is useful for studying general adaptation benefits in simulation studies. This, however, does not help us to find an application's specific cost model. The Grads project [17] employs performance models created by the compiler to predict performance and monitors whether these predictions are met. The approach in [19] considers floating-point operations and accesses to the memory hierarchy to predict performance on different architectures, while focusing on the memory accesses. Memory accesses are extracted via a simulator, and application characteristics are convolved with machine characteristics. In [18], a prediction model for different architectures is described which extracts the program structure by static analysis of the program binary, after which execution profiling is used to obtain dependence on parameters such as execution frequency for vertexes in the program graph and reuse distances for memory locations. Linear regression is applied, but only used for the memory-access cost, and the approach currently only considers single-CPU performance. Scalability is not considered.

The most closely related approach is presented in [15]. This approach takes cost/complexity estimations as input, while differentiating explicitly different sources of overhead such as communication, load imbalance, or synchronization loss. Coefficients are then obtained from actual program runs.

Both [18] and [15], consider linear combinations of cost terms. In [15], either additive or multiplicative combinations are possible, the latter describing interaction between cost terms. Predictions are compared to actual runtimes, with the average relative error found for 2D FFT to be only 12.5%, whereas simple linear interpolation (without interactions) resulted in an error of 750%. The approach considers different parameters and employs a least-squares method but does not provide confidence or prediction intervals.

3 The Prediction System

3.1 Overall Framework

The ScoPred performance predictor is embedded into a job-scheduling and job-control framework as shown in Fig. 1. Several tools interact via an integrated design. The Dynamic Directory stores information about the current job execution, including user-provided estimates, and retrieves information about previous runs from a performance repository. The ScoPro monitor [1] provides information about the characteristics of jobs such as the fraction of computation, communication, and I/O time as well as slowdowns under coscheduling. Furthermore, it can monitor progress on heterogeneous resources. ScoPro employs dynamic instrumentation and can, for loosely synchronous applications, extract behavior from a few iterations. The adaptation controller interacts with the application, determining potentially new workload targets per node and initiating the adaptation. In [3], we have presented an approach to perform such adaptation via overpartitioning or partitioning from scratch in either the time or space dimension.

3.2 Application Model

We apply the view that application performance is dominated by certain data structures or computations that depend on a few critical parameters, such as

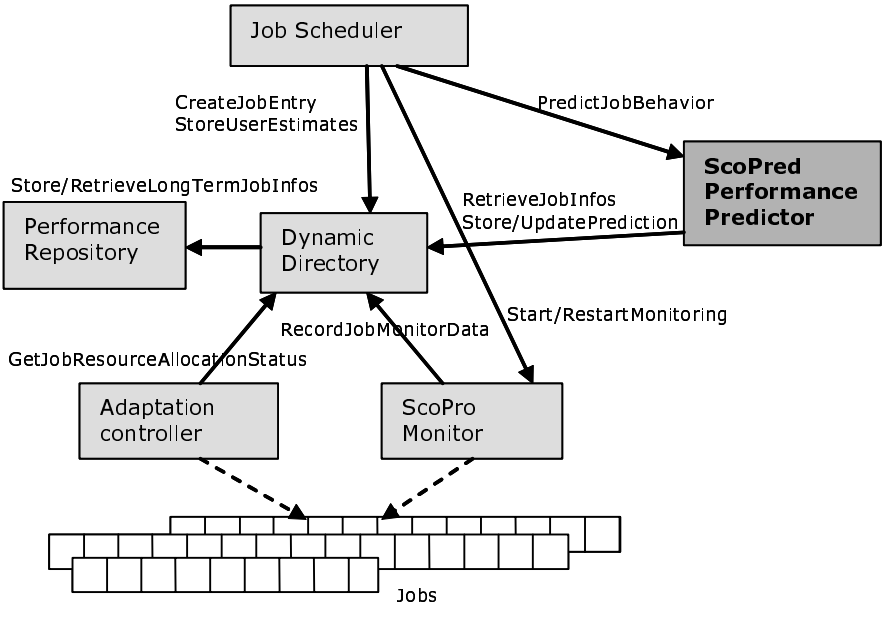


Fig. 1. Overall integrated framework for job scheduling and job control. Solid arrows indicate invocations and are labeled with the corresponding function.

the size of an array. Often these parameters determine the problem size of the application and changing the parameters changes the problem size. Then, it would be possible to model application performance by modeling the dependence of the cost on these parameters. In many cases, an application's user is aware of which application parameters are critical, as the user may explicitly want to change them to switch to a different problem size. In this case, the user should be able to specify which these parameters are.

Typically such parameters appear as input to the application but they may also be statically compiled into the code. We assume in the following that the values of the critical parameters are specified with the submission of the job (it would not be difficult to build a corresponding programming environment that makes such submissions straightforward). Furthermore, we assume that a unique identification is attached to the application (the name is not necessarily sufficient and may be different for program versions compiled for different numbers of nodes). We do not make any efforts to automatically match jobs as done in [9][13], but instead focus on the prediction aspect.

We assume that the user, in addition to the specification of the critical parameters, also provides a rough cost estimate describing the qualitative/structural relationship (without the quantification of system dependent coefficients). Such estimates could be close to the estimations represented by mathematically derived algorithmic complexity estimates. Importantly, complexity-oriented estimations would focus on the complexity of the model expressed in various terms such as $T = 2M^3 + log(N) + Mlog(N)$ with T being the runtime, M being the size of one dimension in a matrix, and N being the number of nodes. Let us assume that the first term specifies computation cost in terms of computation steps and the second and third term specify communication cost in terms of sizes and numbers of messages. Note that the coefficients from the replication in iteration steps as well as the computation time per step, message startup cost and transfer cost are omitted. These are the coefficients to be determined by the predictor. We assume that the estimate takes the form of a linear combination of different cost terms. However, as will be explained below, variables may be multiplied within the terms.

The idea is to specify a rough cost formula which only reflects critical, i.e. dominating performance influences. The system should provide the possibility to add an automatic correction to approximate missing cost terms which are not crucial but also not negligible if high prediction accuracy is to be obtained. What is exactly to be modeled depends on the application and on the desirable range of prediction. Thus, if a small problem size fits into the cache and a larger one does not, and this significantly influences performance, the memory access cost in dependence to the parameters should be modeled. Similarly, if load imbalance could become an issue, this should also be modeled, etc. Note that an alternative to the user providing specifications is the compiler providing them. In this case, the derived formulas may be more detailed though the compiler might require monitoring / runtime feedback or a simulator to prune irrelevant details and extract the relevant formulas.

Our current prediction system works for a particular machine, i.e. we do not include any machine model and prediction facilities across machines.

3.3 Architecture and Functionality

In the following, we describe the details of the predictor functionality. The ScoPred predictor takes as input:

- Specifications of the critical parameters and the estimation formulas depending on them
- The actual values of the critical parameters for the current job submission
- Information about previous runs of the same application (parameters values and performance of the job run).

Note that the number of nodes on which the application is run is typically one of the critical parameters (unless the application always runs with the same number of nodes).

The values predicted may be:

- Runtime of the whole job
- Differentiated time consumed on different resources such as computation time (CPU), communication time (network), and I/O time (disk).

The latter requires that the monitor provides the actual times for the different program execution components from previous runs to match them against the prediction. Providing cost estimations for different resources is typically not a problem for the user because these different aspects need to be considered (as far as applicable) to obtain an estimation of the runtime function for the whole job.

The goal of the estimation may be to predict

1. Future runtimes with the same parameter values
2. Future runtimes with different problem sizes (one or several of the parameters other than the number of nodes)
3. Future runtimes with a smaller or higher degree of parallelism (smaller or larger number of nodes)
4. Combinations of different problem sizes and a different degree of parallelism

Option 3 can be applied if resources are allocated adaptively (as for malleable or moldable applications), and is thus of particular interest. Option 4 is a typical case, as for production runs (runs other than tests for speedup graphs with varying numbers of nodes) there is typically a correlation between problem size and number of nodes, i.e. larger problem sizes require a larger number of nodes and vice versa. Note that this option is always at least a two-variable prediction, whereas the others potentially predict a single variable only. Options 2, 3, and 4 require the scalability features of our predictor.

The prediction system provides point estimates of the mean of values, and associated confidence and prediction intervals.

4 Multiple Linear Regression and Its Application in ScoPred

4.1 Overview of Multiple Linear Regression

Linear regression, also known as *linear least squares regression*, is a widely used mathematical modeling technique [20]. Given a data set and an appropriate function, least squares regression determines the values for coefficients within the function that produce an equation which best fits the data set. Simple linear regression applies to equations with a single dependent variable, and a single independent variable. Multiple linear regression applies to equations with a single dependent variable and multiple independent variables.

In order to be appropriate for linear regression, the supplied function must be *linear in the parameters* [21]. This is satisfied if and only if the function is of the form [20]:

$$f(\vec{x} : \vec{B}) = B_0 + B_1 x_1 + B_2 x_2 + \cdots + B_k x_k \tag{1}$$

where:

- Each independent variable $(x_1, x_2, \ldots, x_k)$ in the function is multiplied by an unknown coefficient $(B_1, B_2, \ldots, B_k)$.
- There is at most one unknown coefficient with no corresponding independent variable (B_0).
- The individual terms are summed to produce a final function value.

The independent variables may be the product of several application parameters, and particular application parameters may be components of more than one independent variable, though no two independent variable may be exactly the same.

For instance,

$$y_i = B_0 + B_1\sqrt{x} + B_2 x^2 + B_3 xz + B_4 z \tag{2}$$

is a valid function.

Given a function which satisfies the above conditions, we can relate the function to the data set by adding an error component, ϵ:

$$f(\vec{x} : \vec{B}) = B_0 + B_1 x_1 + B_2 x_2 + \cdots + B_k x_k + \epsilon \tag{3}$$

In all but perfect models, the value of ϵ will vary for each observation. The total set of ϵ values is referred to as the *error component*, or *set of residuals*. Three assumptions must hold regarding error component [22].

The set of values of ϵ must:

- Be independent, in the probabilistic sense (the value of a particular ϵ value should be unrelated to the other values of ϵ, except insofar as they satisfy these conditions)
- Have a mean of 0 and a common variance
- Have a normal probability distribution

Given a suitable function and data set, we can calculate the values for the coefficients such that the sum of the squares of the residual values is minimized. That is, we choose $B_0, B_1, \ldots, B_k$ such that

$$SSE = \sum_{i=1}^{k} (y_i - \hat{y}_i)^2 \tag{4}$$

is minimized, where y_i refers to the i^{th} observed response value, and $\hat{y}_i$ refers to the value of (1) with the i^{th} observations of all dependent variables.

To find values for the coefficients which minimize SSE, partial differentials of (1) are taken with respect to each unknown coefficient. The resulting set of partial differential equations is then solved as a system of linear equations. For further details, refer to [21][20].

4.2 Statistical Tests and Metrics

There are several ways of evaluating how well a calculated regression equation fits the data. Two commonly used measures of a fit's significance are the coefficient of determination, and the analysis of variance F-test [22].

The coefficient of determination (often called R^2 or *multiple* R^2) [22] expresses the proportion of the variance that is explained by the regression model. Informally, this is a measure of how much better the model is at expressing the relationship, as compared to simply using the mean of the response data. An R^2 value of 0 indicates that the mean of the response data is a better predictor of the response, whereas an R^2 value of 1.0 indicates that the response fits the curve perfectly, explaining 100% of the deviation from the mean in each response value. For convenience, we consider any value less than 0 to be equivalent to 0, and this allows us to interpret the coefficient of determination as a percentage. A value related to the coefficient of determination is the *adjusted* R^2, or *shrunken* R^2. As the number of independent variables in a regression formula rises, the R^2 value becomes artificially inflated. To compensate for this effect, the *adjusted* R^2 value takes into account the number of independent variables [22].

Another way of testing whether a regression model is significant is through a statistical test. For any given configuration of coefficient values we can test whether the data supports that configuration at a given confidence level. If the regression model is significant, then at least one of the independent variables is contributing significant information for the prediction of the response variable [22]. To prove this, we attempt to reject the contrapositive statement that no independent variable contributes any information for the prediction of the response. This is referred to as the null hypothesis, and we attempt to reject it with 95% confidence. If successful, we have shown with 95% confidence that at least one of the independent variables is contributing significant information for the prediction of the response variable. That is, the regression model is significant.

4.3 Using a Regression Model for Prediction

Once the regression model is generated and we have verified that the model effectively describes the relationship between the predictor variables and the

response, the model can be used to predict the response for given values of the independent variables. These predictions take the form of a point estimate which is a prediction of the mean function value for a certain parameter-value combination. In addition, the confidence interval (typically 95%) is provided, describing the probability with which the mean value of future observations for this parameter-value combination falls into the corresponding interval. Furthermore, a prediction interval is provided which describes the probability that future observed function values fall into the corresponding interval. Note that both the confidence and the prediction interval become wider as the parameter-value combinations are further away from the means of the observed values of the independent variables used to build the regression model, i.e. the prediction becomes less reliable.

4.4 Multiple Linear Regression in Parallel Performance Prediction

Given the above explanations, we can now explain how a scalable model of an application can be built through a straightforward application of linear regression. Given an application that we wish to model, we proceed as follows. To start, the static (static variables) or dynamic parameters (arguments of the program invocation) of an application, which are specified to influence the application's characteristics, are considered as components of the independent variables of our model. Similarly, application characteristics that we wish to construct a model of (runtime, I/O time, etc) are considered as the response variables. By storing the parameters with which application runs are submitted, and the corresponding observations of the characteristic under study, we build a data set of our independent and response variables.

At this point we have a data set, but we still need a function to fit to the data. As described above, the function is supplied by the user (or by a compiler) as an estimation formula (see Section 3.2) of the relation between the characteristic under study and the application parameters. However, it is still lacking a quantification of the coefficients. Coefficient variables (whose values are unknown at this point) are automatically added into the equation by our system in a manner that ensures the resulting equation is linear in the parameters. Once we have added the unknown coefficients, we have an equation of the form described in Section 4.1. We can now use linear regression to determine the quantification by calculating values for the coefficient variables that best fit the equation to the data, giving us a model of the application.

By substituting in the parameters, the model can be used to predict the performance of the characteristic under study for a future run of the application. We use statistical measures such as the coefficient of determination, R^2 and the analysis of variance F-test to determine how successful the regression model fits the data, and to calculate confidence intervals for our predictions.

Since fitting the function values into a regression model always involves some inaccuracy regarding the individual values for certain parameter-value combinations (points), employing the prediction from the model for points with available observations is not very meaningful. We can gain better predictions by only

considering the different historical values for exactly the corresponding point. The main benefit from the regression model comes from predicting performance for unobserved parameter-value combinations.

4.5 Implementation

We have used Java to write the overall interface for our performance predictor. This interface handles the interaction with the job scheduler and the dynamic directory in regards to storage of new or modified application profiles and retrieval of existing profiles. All statistical calculations are performed through calls to the Waterloo Maple symbolic algebra system [27] using custom functions programmed in Maple's native language. To allow the Java components access to the Maple components, we used the OpenMaple API for Java [28], which allows us to call code in a running Maple environment from external Java programs. The calls to the Maple environment to calculate a model, or make a prediction based on an existing model take in the order of a second, which is acceptable in a job-scheduling environment for parallel machines.

5 Experimental Evaluation

5.1 Experimental Setup

To evaluate our system, we have chosen the Linpack benchmark [29] and two applications from the NAS Parallel Benchmarks Version 2.4 [23]. The selected NAS benchmarks are the EP embarrassingly parallel benchmark, and the FT 3D Fast Fourier transformation benchmark.

Some of the tests were run on a local 16 node Debian GNU/Linux cluster running Linux kernel 2.6.6. Each node of this cluster is equipped with 2 Intel Xeon 2.0 Ghz processors, of which our tests only use one. Other tests requiring more than 16 nodes were run on 64 nodes of a Redhat GNU/Linux cluster running Linux kernel 2.6.8.1. Each node of this cluster is equipped with 2 Intel Opteron/244 1.8 Ghz processors, of which our tests only use one. Each cluster uses a Myrinet network for application-level communication, using MPICH version 1.2.6 over GM.

Each benchmark was run for varying problem sizes and different numbers of nodes (NAS provides a number of different categories: S, W, A, B, etc. which have increasing problem sizes, and Linpack allows problem sizes and node configurations to be specified). For each run, the runtime in seconds was recorded. Unless otherwise specified, each application was run 3 times in each configuration of problem size/nodes. In our case, the values for multiple runs were nearly identical, but we have used multiple observations to take into account the variation in runtime that could be caused by other applications running simultaneously on a time-shared system. In practice, if there is little variation between application runs with the same configuration for an application, we could model using only one run for each configuration. However, multiple linear regression does have the ability to model applications for which there is variation among runs with the same configuration.

For the subset of gathered data selected for each test, we feed all observed runs into our predictor, but for the sake of clarity only show the mean values in the tables of observations below. After providing a subset of the gathered data to our predictor, predictions are made for the excluded problem size/node configurations.

We test the following cases:

- Processor Scalability: Prediction towards larger number of nodes given data for a smaller number of nodes (malleability test or test for user choosing new number of nodes)
- Problem-size Scalability: Prediction towards larger problem sizes given data for smaller problem sizes (user switching to larger problem size)
- Problem-size/Processor Scalability: Prediction towards larger problem sizes and larger number of processors given data from situations with data for smaller numbers of nodes/problem size (user switching to larger problem size on larger number of nodes)

Whenever possible, we omit observations with runtimes of less than one second and in some of our experiments below this limits the number of experiments we can perform on the gathered data sets.

5.2 EP

The EP benchmark represents the simplest of parallel programs. Communication occurs only twice: once when the job begins, sending a segment of the total work to each of the nodes; and once right before the job terminates, collecting back the results of each node's calculations. The lack of communication time makes this benchmark very simple to model, and we use it as a demonstration of our technique.

We ran the EP benchmark on the local 16 node cluster, varying the number of nodes from 2 to 16 in increments of 2. The mean values of the three gathered observations for each configuration are shown in Table 1.

Before this data can be entered into our system, we must provide an estimate of how the benchmark's runtime varies with its parameters. To do so, we consider the benchmark's algorithm. The EP benchmark accepts two parameters, the number of nodes (P), and the problem size (N). Each node is assigned the task of generating (N/P) pairs of Gaussian random deviates according to a specific scheme, and tabulates the number of pairs in successive annuli [23]. Since the time to generate a pair of random numbers is relatively constant, we estimate that the runtime (T) of a particular job will be given by:

$$T = N/P \tag{5}$$

This estimate is entered into our system. As the first step toward turning this estimate into a detailed model, the system adds unknown coefficients into the equation, giving us:

$$T = B_1(N/P) + B_0 \tag{6}$$

where B_1 and B_0 represent the unknown constants added by our system.

Table 1. Mean values for EP benchmark observations

nodes \ class	**S = 2^{24}**	**W = 2^{25}**	**A = 2^{28}**	**B = 2^{30}**
2	2.10	4.19	33.60	138.24
4	1.09	2.10	16.81	67.28
6	0.72	1.46	11.26	44.82
8	0.54	1.09	8.40	34.86
10	0.43	0.87	6.74	29.03
12	0.36	0.71	5.62	22.47
14	0.31	0.62	5.17	20.11
16	0.28	0.56	4.36	17.38

In the first experiment, we test our system for processor scalability, using data from the A and B columns. Table 2 shows the results of using our system to model either the A or B column, i.e. the table shows the result of two separate sets of processor-scalability predictions. In each column, a subset of the observations is input into the system (shown in light grey). The cells highlighted in darker grey are the predictions of our system given the input data for that column. For each value we provide a point estimate, followed by the 95% confidence interval, and the 95% prediction interval (in parentheses). Below the point estimate and confidence/prediction intervals is the percentage that the estimate deviates from the mean of the observations.

For the A column test outlined in Table 2, the system assigns a value of $1/3996971.35$ for B_1 and a value of 0.025 for B_0, giving the model:

$$T = (N/3996971.35P) + 0.025 \tag{7}$$

For the B column, the values of B_0 and B_1 are similar, giving us the model:

$$T = (N/3906469.72P) + 0.085 \tag{8}$$

We observe in Table 2 that our predictions tend to be a bit low, though all predictions are well within a 10% deviation from the mean of the observations. While our predictions for Class A come very close, the mean of the observations does not fall within the 95% confidence or prediction intervals for the 14 and 16 node predictions. For Class B, the mean of the observations is within both the 95% confidence and prediction intervals for all predictions. This does not necessarily indicate that the predictions for Class B are better as we observe that the confidence and prediction intervals for Class B are considerably wider. This is likely due to greater variation observed between data points of the same configuration for Class B.

Table 2. Results of the processor-scalability predictions for 12, 14, and 16 node cases in the EP benchmark

nodes \ class	**A = 2^{28}**	**B = 2^{30}**
2	33.60	138.24
4	16.81	67.28
6	11.26	44.82
8	8.40	34.86
10	6.74	29.03
12	5.62 +/- 0.03 (0.08)	22.99 +/- 2.77 (8.09)
	- 0.06%	+ 2.31%
14	4.82 +/- 0.03 (0.08)	19.72 +/- 2.89 (8.13)
	- 6.71%	- 1.96%
16	4.22 +/- 0.03 (0.08)	17.26 +/- 2.98 (8.16)
	- 3.21%	- 0.73%

Table 3. Results of problem-size scalability prediction of the B problem size for the EP benchmark

nodes \ class	**S = 2^{24}**	**W = 2^{25}**	**A = 2^{28}**	**B = 2^{30}**
2	2.10	4.19	33.60	134.43 +/- 0.18 (0.19)
				- 2.76%
4	1.09	2.10	16.81	67.17 +/- 0.27 (0.29)
				- 0.17%
6	0.72	1.45	11.26	44.92 +/- 0.34 (0.36)
				+ 0.22%
8	0.55	1.09	8.40	33.49 +/- 0.28 (0.30)
				- 3.94%

In our second experiment, we test our system with the task of problem-size scalability, modeling the 2, 4, 6, and 8-processor rows respectively. We provide the data for problem sizes S, W, and A for each processor row, and use the generated model to predict problem size B for that row. The results are shown in Table 3.

We see that our predictions are very good, with all predictions less than 5% from the mean of the actual observations. Also, in all but the 2 processor case, the mean of the actual observation is within the 95% confidence and prediction intervals.

Table 4. Results of problem-size/processor scalability predictions for the EP benchmark

nodes \ class	**S = 2^{24}**	**W = 2^{25}**	**A = 2^{28}**	**B = 2^{30}**
2	2.10	4.19	33.60	
4	1.09	2.10	16.81	
6	0.72	1.45	11.26	
8	0.55	1.09	8.40	34.39 +/- 0.61 (3.95)
				- 1.36 %
10	0.43	0.87	6.74	27.51 +/- 0.59 (3.95)
				- 5.23 %
12			5.73 +/- 0.63 (3.95)	22.93 +/- 0.58 (3.94)
			+ 1.90 %	+ 2.05 %
14			4.91 +/- 0.64 (3.95)	19.65 +/- 0.58 (3.94)
			- 4.97 %	- 2.30 %
16			4.29 +/- 0.64 (3.95)	17.19 +/- 0.59 (3.94)
			- 1.61 %	- 1.13 %

In our final experiment for the EP benchmark, our system is tested with the task of modeling both problem size and number of processors simultaneously. We provide the system with columns S, W, and A and rows 2, 4, 6, 8, and 10, and use the generated model to predict A on 12, 14, and 16 processors, and B on 8, 10, 12, 14, and 16 processors. The results are shown in Table 4.

The multivariate model created by entering observations varying in both number of nodes and problem-size is almost as successful as the models of one or the other alone. All values are within 6% of the observed mean, and none of the observed means falls outside of the 95% prediction interval (only one falls outside the 95% confidence interval).

Finally, we want to note that the R^2 values for these predictions are close to 1.0 and that the F tests are passed at a greater than 95% confidence level in all cases.

5.3 FT

The FT benchmark solves a partial differential equation (PDE) using a 3-D Fast Fourier Transform (FFT) [23]. The 3-D FFT is solved using a standard transpose algorithm. Due to the nature of the algorithm, it can only be run on a power-of-two number of nodes. The FT benchmark is more complex to model than the EP benchmark, as there is extensive communication throughout the application run.

The algorithm employs a 3-dimensional array which determines the problem size. Though the size of the FFT array is set for each class of benchmark, the user may vary the number of *iterations* performed for each class. We have set all tests to 6 iterations to make the runs with different problem sizes comparable.

Table 5. Mean values for FT benchmark observations

nodes\ class	S = 64x64x64	W = 128x128x32	A = 256x256x128	B = 512x256x256	C = 512x512x512
2	0.21	0.48	9.91	43.94	N/A
4	0.11	0.25	5.30	23.23	101.69 *
8	0.06	0.14	2.68	12.16	N/A
16	0.03	0.08	1.51	6.69	29.36 *
32	0.02	0.04	0.87	3.69	16.57
64	0.01 *	0.02 *	0.47 *	1.95 *	8.63 *

In each iteration, a 3-D FFT is performed, and the resulting data set is evolved before being used as the input for the next iteration.

The FT benchmark was run for 2, 4, 8, 16, 32 and 64 nodes with problem sizes varying from Class S (64x64x64) to Class C (512x512x512). The mean values of the application runs are summarized in Table 5. We were unable to obtain any data points for two configurations, which are marked N/A. For most configurations, we gathered 6 data points. For the remaining configurations (marked with an *) we only gathered 3 data points.

The time complexity of computing a 3D FFT using the transpose algorithm is well studied and easily found in many textbooks. In [24], the following formula is given in an analysis of the 3-D FFT algorithm.

$$T = (N/P)log(N) + 2\sqrt{P-1} + 2(N/P) \tag{9}$$

where T is the runtime, N is the problem size, and P is the number of processors. However, an inspection of the code for the FT benchmark reveals that it utilizes all-to-all communications within processor subgroups rather than the point-to-point communications used by the algorithm analyzed in [24]. This motivates us to form an alternate estimate that assumes that scatter/gather is being used for the implementation of all-to-all. Based on the analysis of CPU time given in [24] and the complexity of scatter/gather operations given in [30], we get:

$$T = (N/P)log(N) + (N/P)log(N) \tag{10}$$

In this formula, the first $(N/P)log(N)$ represents the time spent on processing in the FFT algorithm. The second $(N/P)log(N)$ represents the time spent on communication.

We test the FT benchmark with three tests similar to those used for the EP benchmark: testing the processor scalability, problem-size scalability, and finally both together.

As with EP, we test processor scalability by providing a subset of the observations and predicting the observations not provided in that column. This time

Table 6. Results of processor scalability prediction and of the 64, 32-64, and 16-64 node cases of Class A of the FT benchmark

nodes \ class	**A=256x256x128**	**A=256x256x128**	**A=256x256x128**
2	9.91	9.91	9.91
4	5.30	5.30	5.30
8	2.68	2.68	2.68
16	1.51	1.51	1.57 +/- 0.1 (0.25)
			+ 4.20%
32	0.87	0.94 +/- 0.06 (0.21)	0.97 +/- 0.11 (0.25)
		+ 8.25%	+ 11.71%
64	0.61 +/- 0.05 (0.19)	0.64 +/- 0.07 (0.21)	0.67 +/- 0.11 (0.26)
	+29.79%	+ 36.17%	+ 42.55%

we perform three tests for each problem size. In the first test, we provide all but the 64 processor observations, in the second, we provide all but the 32 and 64 processor observations, and in the third we provide all but the 16, 32 and 64 processor observations. We perform these tests for Class A and Class B, as they are the only full columns which have a significant number of observations with runtime greater than one second. The results are summarized in Table 6 and Table 7.

As shown in Table 6, our scalability model yields fairly accurate point estimates for 16 and 32 processors, with all below 12% deviation from the mean of the observations.The point estimates for 64 processors seem quite poor, with deviations between 29% and 43% from the mean of the observations.This might

Table 7. Results of processor scalability prediction and of the 64, 32-64, and 16-64 node cases of Class B of the FT benchmark

nodes \ class	**B=512x256x256**	**B=512x256x256**	**B=512x256x256**
2	43.94	43.94	43.94
4	23.23	23.23	23.23
8	12.16	12.16	12.16
16	6.69	6.69	7.09 +/- 0.21 (0.56)
			+ 6.06%
32	3.69	4.22 +/- 0.17 (0.56)	4.45 +/- 0.23 (0.56)
		+ 14.52%	+ 20.76%
64	2.68 +/- 0.16 (0.64)	2.89 +/- 0.18 (0.56)	3.13 +/- 0.25 (0.57)
	+ 37.20%	+ 47.95%	+ 60.24%

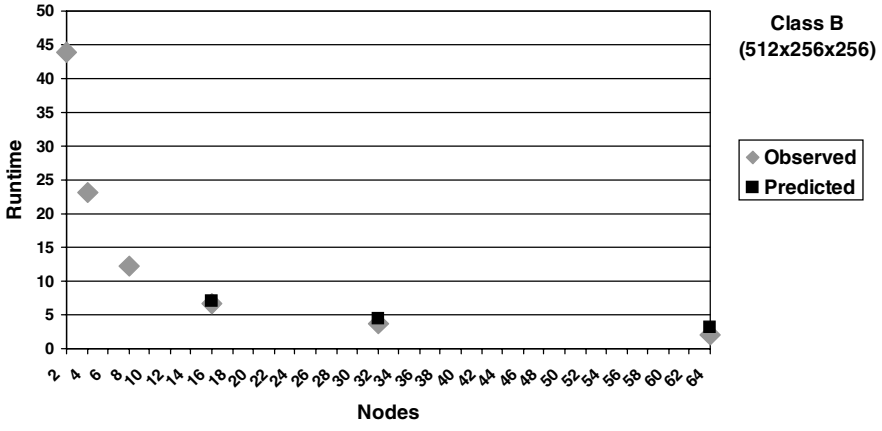


Fig. 2. A graph showing the results of the processor scalability prediction of 16-64 node configurations for Class B of the FT benchmark. The three observation points to the left (in gray) show the observations provided to the system, and the three prediction points (in black) represent the predictions made by the system. Where both observation and prediction points are shown, this is to illustrate the accuracy of the prediction vs. the mean of the actual observations.

be explained by the extremely small (less than a second) runtime values for the 64 node case. In all cases the mean of the observations falls within the 95% prediction interval, and two cases fall within the 95% confidence interval.

The scalability tests for Class B are slightly worse than Class A. However, the 16 processor case in the right column, and the 32 processor case in the center column are fairly accurate predictions, both with point estimates less than 15% from the mean of observations, and both with 95% prediction intervals that accurately predict the location of the mean of observations. As with Class A, some of the higher deviations of 64 processor predictions may be explained by the small size of the mean of observations. Fig. 2 shows a plot of the results for the 16-64 processor case, which appears to be the least accurate of the three Class B scalability tests. Even in this case, the 16 processor prediction is quite accurate (6% deviation from the mean of observations), and the all predictions are quite accurate when considered in an absolute sense, rather than as a percentage deviation from the mean of observations. Fig. 2 also shows how far away the 32 and 64 processor predictions are from the set of provided observations, and suggests that this may also be a factor affecting accuracy.

Table 8 shows the results of our problem-size scalability tests for the FT benchmark. Due to the small runtimes of all of the observations for Class S, these observations were excluded from the tests. The observations for Class W were included despite their small runtimes to provide us with enough data points to get meaningful predictions. We performed four tests, separately modeling 2, 16, 32, and 64 processors, and predicting Class C from the observations of Class W, A, and B. The point-estimates are very close to the mean of the observed

Table 8. Results of the problem-size scalability prediction of the C problem-size for the FT benchmark

nodes\ class	W = 128x128x32	A = 256x256x128	B = 512x256x256	C = 512x512x512
4	0.25	5.30	23.23	100.47 +/- 0.12 (0.13) - 1.20%
16	0.08	1.51	6.69	28.91 +/- 0.30 (0.33) - 1.54%
32	0.04	0.87	3.69	15.91 +/- 0.13 (0.14) - 4.00%
64	0.02	0.47	1.95	8.43 +/- 0.09 (0.09) -2.36%

values, all less than 4%. However, the means of the observed values do not fall into any of the 95% confidence or prediction intervals. This is likely due to the small number of configurations used to build the model. When very few observations are used to build a model, the model can easily pass very close to all provided observations, and because of this predicts tight confidence and prediction intervals.

As with the EP benchmark, we tested our system with the task of modeling both problem size and number of processors simultaneously for the FT benchmark. The very small runtimes of many of the observations for the FT benchmark made this difficult, and limited the number of useful observations that we could use. Including only observations greater than one second, we tested two configurations, with results shown in Table 9. In the first configuration, we provided observations from Class A and Class B for 2, 4, 8, and 16 processors, and 2, 4, 8, 16, and 32 processors respectively. We then predicted Class B with 64 processors, and Class C with 16, 32, and 64 processors. In the second configuration, we provided observations from Class A and Class B for 2, 4, 8, and 16 processors, and predicted Class B with 32 and 64 processors and Class C with 16, 32, and 64 processors. In both tests, the predictions for Class C deviate from the mean of observations by between 17% and 25%, with the 95% confidence and prediction intervals failing to capture the mean of observations. The predictions for Class B are better, with point estimates less than 11% from the mean of observations, and all of the prediction intervals (and most of the confidence intervals) accurately capturing the mean of observations. In the second test, the mean of observations for Class B with 32 processors falls outside of the 95% confidence interval, but only by 0.19 seconds. A possible explanation for the poorer predictions for Class C is the relative size of the Class C problem size as compared to the problem sizes of Class A and Class B. When making

Table 9. Results of problem-size/processor scalability predictions for the FT benchmark. In the upper table, the 64 processor case of Class B, and the 16-64 processor cases of Class C are predicted. In the lower table, the 32-64 processor cases of Class B and the 16-64 processor cases of Class C are predicted.

nodes \ class	**A=256x256x128**	**B=512x256x256**	**C=512x512x512**
2	9.91	43.94	
4	5.30	23.23	
8	2.68	12.16	
16	1.51	6.69	24.22 +/- 0.21 (1.13)
			- 17.52%
32		3.69	12.42 +/- 0.15 (1.12)
			- 25.06%
64		1.99 +/- 0.19 (1.13)	6.52 +/- 0.16 (1.12)
		+ 1.88 %	- 24.48%

nodes \ class	**A=256x256x128**	**B=512x256x256**	**C=512x512x512**
2	9.91	43.94	
4	5.30	23.23	
8	2.68	12.16	
16	1.51	6.69	24.20 +/- 0.22 (1.17)
			- 17.58 %
32		3.29 +/- 0.21 (1.17)	12.38 +/- 0.17 (1.17)
		- 10.72%	- 25.30%
64		1.92 +/- 0.22 (1.17)	6.46 +/- 0.19 (1.17)
		- 1.71%	- 25.17%

predictions for Class C, we are predicting for a problem size that is very far from the provided observations (4 times larger than Class B and 8 times larger than Class A).

Since the formula used for the FT benchmark provides us with a breakdown of the complexity into communication time and processing time, and the FT benchmark allows us to measure the time spent on various tasks, we can take a closer look at why some of our predictions for FT are inaccurate. Furthermore, this differentiation gives us a chance to demonstrate the capability of our system to predict different resource characteristics separately.

By taking a closer look at the algorithm, we find that the entire algorithm is made up of setup time and a number of iterations in which an FFT is performed and between which, the data is evolved. The FFT time consists of CPU ($FFTcpu$) and communication time ($FFTcomm$). We notice that two terms (setup time and evolve time) were not considered in the original model but have

significant effect on the total runtime. Motivated by this, we also model these additional terms ($SetupTime$ and $EvolveTime$).

$$T = SetupTime + EvolveTime + FFTcpu + FTcomm \tag{11}$$

Table 10 shows the breakdown of the runtimes for the B problem size, according to the four cost components described above. Furthermore, the percentage of communication in relation to the overall runtime is given. Unlike our other FT tests, in these tests we only use three observations for each configuration to calculate the mean of observations and to input into the system for prediction.

We model $SetupTime$ and $EvolveTime$ both as N/P (obvious from the observations). The original CPU-time model becomes $FFTcpu = (N/P)log(N)$. The model for communication remains the same and is now explicitly described in $FFTcomm = (N/P)log(N)$.

Table 11 shows two tests in which models are generated and predictions are made for all terms separately. In the first tests, we provide observations for 2, 4, 8, 16, and 32 processors and predict the 64 processor values. In the second test, we provide observations for 2, 4, 8, and 16 processors and predict the 32 and 64 processor cases. We see that $FFTcpu$ is modeled fairly accurately, with point predictions less than 11% from the mean of observations. However, $FFTcomm$ is quite poor, with deviations of 44% to 134% off the mean of observations. The Total Runtime prediction is simply the sum of the predictions of the individual components. These predictions are about the same as the predictions of the total runtime presented in Table 7. The results indicate that the inaccuracy of our predictions is due to an inaccurate communication model – which is critical, considering that the communication amounts to up to about 47%. It is well known that different algorithms may be chosen for the implementation of collective operations. For instance, the all-to-all which dominates in FT can be implemented using scatter/gather, pairwise-exchange, or a linear algorithm [31] [32]. Without knowing which algorithms are employed by the MPI library in use, the model

Table 10. Mean values for the FT benchmark observations, broken down into Setup, Evolve, FFTcpu, and FFTcomm time for Class B

nodes \ class	Total Runtime	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	3.72	0.08	0.13	1.96	1.48	42.98%
64	1.95	0.04	0.07	0.97	0.85	46.62%

Table 11. Results of the problem-size scalability prediction for Class B (512x256x256) of the FT benchmark, broken down into Setup, Evolve, FFTcpu, and FFTcomm time. In the upper table, we provide observations from 2-32 processors and predict the 64 processor case. In the lower table, we provide observations from 2-16 processors, and predict the 32 and 64 processor cases.

nodes\ class	Total Runtime	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	3.72	0.08	0.13	1.96	1.48	42.98%
64	2.71 +/- 0.34 (0.97)	0.04 +/- 0.00 (0.01)	0.07 +/- 0.01 (0.02)	0.88 +/- 0.06 (0.18)	1.72 +/- 0.27 (0.76)	46.62%
	+38.74%	0.00%	0.00%	- 9.59%	+102.35%	

nodes\ class	Total Runtime	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	4.30 +/- 0.35 (0.84)	0.08 +/- 0.00 (0.01)	0.15 +/- 0.01 (0.02)	1.93 +/- 0.09 (0.21)	2.14 +/- 0.25 (0.60)	42.98%
	+ 15.49%	0.00%	+ 15.38%	- 1.70%	+44.59%	
64	2.98 +/- 0.36 (0.84)	0.04 +/- 0.00 (0.01)	0.08 +/- 0.01 (0.02)	0.87 +/- 0.09 (0.21)	1.99 +/- 0.26 (0.60)	46.62%
	+ 52.56%	0.00%	+ 14.29%	- 10.62%	+134.12%	

cannot properly capture communication behavior. As a future extension, such information could be made available for all applications per collective operation in the dynamic directory. Another possible explanation is the small size of the communication observations for 32 and 64 processors. In a further test, we provided 2, 4, and 8 processors and predicted 16, 32, and 64 processors. The 16 processor prediction was fairly accurate (11.21% off the mean of observations).

5.4 Linpack

The Linpack benchmark is the standard test of a supercomputer's performance, and is used to establish the Top 500 list of supercomputers. The benchmark application generates and then solves a random dense linear system using LU

factorization [29]. We use the High-Performance Linpack (HPL) implementation of the Linpack benchmark provided in [29]. Unlike the FT benchmark, HPL can be run on any number of processors in a variety of configurations, and can be run with user-specified problem sizes. As well, HPL allows the user to specify several characteristics of the algorithm, such as whether the benchmark will use a binary exchange swapping algorithm, a spread-roll swapping algorithm, or a hybrid of the two. The user is also able to specify the block size that is used by the algorithm. The complexity of the benchmark is well studied and is provided in [32]. Dropping the constant terms which do not make a difference in our system, the formula is as shown below:

$$T = N^3/3PQ + N^2(3P + Q)/(2PQ) + Nlog(P) + NP \tag{12}$$

where T is the runtime, N is the size of one side of the square matrix constituting the problem size, and P and Q describe the arrangement of processors in a P by Q grid. Thus PQ gives the total number of processors used.

The benchmark was run on square configurations ($P = Q$) of nodes varying from 2x2 to 8x8, and with N varying from 8000 to 14000 in increments of 1000. Each configuration was run three times and in all cases we used the binary exchange swapping algorithm, and a block size of 64. The mean values of the observations are shown in Table 12.

As with the EP and FT benchmarks, we test processor-scalability, problem-size scalability, and both processor and problem-size scalability simultaneously.

To test processor-scalability, we ran seven tests, one for each problem size. In each test, the problem size was kept constant and the processor configuration was varied. The observations for 2x2, 3x3, 4x4, 5x5, 6x6, and 7x7 processor configurations was provided, and 8x8 was predicted. The results are shown in Table 13.

The quality of the predictions varies between the different problem sizes, with point estimates from 6.90% to 52.02% from the mean of the corresponding observations. A possible explanation for the poor predictions is the small number of provided observations for a relatively complex formula. Multiple linear regression works best with large numbers of observations, and relatively simple formulas, with few coefficients to calculate. In this case, we have a formula with five coefficients to calculate (one in front of each term in the formula, and one

Table 12. Mean values for the Linpack benchmark observations

node\ size	**8000^2**	**9000^2**	**10000^2**	**11000^2**	**12000^2**	**13000^2**	**14000^2**
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1066.88
3x3	69.12	104.54	148.43	201.67	264.84	344.20	436.65
4x4	32.37	49.69	74.84	105.76	140.54	183.68	233.86
5x5	20.38	29.72	42.49	59.68	81.12	111.18	145.49
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.51
7x7	11.50	15.10	20.76	29.13	38.36	46.98	59.28
8x8	8.63	11.75	15.76	20.50	27.16	34.47	43.87

Table 13. Processor-scalability tests for the Linpack benchmark

node \ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1066.88
3x3	69.12	104.54	148.43	201.67	264.84	344.20	436.65
4x4	32.37	49.69	74.84	105.76	140.54	183.68	233.86
5x5	20.38	29.72	42.49	59.68	81.12	111.18	145.49
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.51
7x7	11.50	15.10	20.76	29.13	38.36	46.98	59.28
8x8	7.20 +/- 2.19 (2.43)	8.57 +/- 1.75 (1.94)	19.49+/- 1.40 (1.55)	31.17 +/- 13.16 (14.55)	39.64 +/- 3.00 (3.33)	36.85 +/- 6.16 (6.82)	35.60 +/- 8.05 (8.91)
	-16.56%	-27.10%	+23.70%	+52.02%	+45.96%	+ 6.90%	-18.85%

Table 14. Problem-size scalability tests for the Linpack benchmark

node\ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	870.83 +/- 23.26 (24.66)	1130.51 +/- 65.55 (66.06)
						+ 1.51%	+5.96%
3x3	69.12	104.54	148.43	201.67	264.84	338.75 +/- 1.92 (2.04)	424.11 +/- 5.41 (5.46)
						-1.58%	-2.87%
4x4	32.37	49.69	74.84	105.76	140.54	177.14 +/- 2.58 (2.74)	213.59 +/- 7.28 (7.34)
						-3.56%	-8.67%
5x5	20.38	29.72	42.49	59.68	81.12	107.58 +/- 4.00 (4.24)	139.34 +/- 11.28 (11.36)
						-3.24%	-4.23%
6x6	14.30	20.48	27.31	38.11	51.30	69.2 +/- 2.86 (3.03)	92.20 +/- 8.05 (8.11)
						+1.13%	+1.87%
7x7	11.50	15.10	20.76	29.13	38.36	48.59 +/- 3.57 (3.78)	58.99 +/- 10.05 (10.13)
						+3.43%	-0.49%
8x8	8.63	11.75	15.76	20.50	27.16	35.83 +/- 2.13 (2.26)	47.18 +/- 6.00 (6.05)
						+3.96%	+7.56%

constant term at the end) and observations for six distinct processor configurations. To contrast, the EP and FT benchmarks only had two coefficients to calculate, though they also had fewer distinct configurations.

Table 15. Problem-size/processor scalability test for the Linpack benchmark

node\ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1084.47 +/- 3.61 (6.13)
							+1.65%
3x3	69.12	104.54	148.43	201.67	264.84	344.20	443.60 +/- 1.35 (5.14)
							+1.59%
4x4	32.37	49.69	74.84	105.76	140.54	183.68	231.75 +/- 1.36 (5.14)
							-0.90%
5x5	20.38	29.72	42.49	59.68	81.12	111.18	138.66 +/- 1.26 (5.11)
							-4.70%
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.12 +/- 1.29 (5.12)
							-0.42%
7x7	11.50	15.10	20.76	29.13	38.36	46.98	61.56 +/- 1.77 (5.26)
							+3.84%
8x8	6.88 +/- 1.44 (5.16)	9.77 +/- 1.49 (5.18)	13.68 +/- 1.62 (5.21)	18.78 +/- 1.81 (5.28)	25.25 +/- 2.06 (5.37)	33.28 +/- 2.36 (5.49)	43.03 +/- 2.69 (5.64)
	-20.36%	-16.89%	-13.22%	-8.41%	-7.01%	-3.46%	-1.90%

To test problem-size scalability we ran six separate tests. In each test, the processor configuration was kept constant, and the problem size was varied. The observations for 8000^2 to 12000^2 were provided, and 13000^2 and 14000^2 were predicted. The results are shown in Table 14. In all of the tests, the predictions are quite accurate, with all point estimates less than 9% from the mean of observations, and in all but two predictions, with the mean of observations falling within both the 95% confidence and prediction intervals.

Next, we test processor and problem-size scalability at the same time. In this test, we provide the observations for problem sizes 8000^2 to 13000^2, and processor configurations 2x2 to 7x7, and we predict the 14000^2 problem size for all processor configurations, and the 8x8 processor configuration for all problem sizes. The results are shown in Table 15. As with the separate processor and problem-size tests above, the model predicts problem-size more accurately than processor-size. However, it is interesting how the processor-size predictions are improved considerably when both are modeled simultaneously.

In our final set of tests, we test our system with the task of modeling three parameters (N, P, and Q) of Linpack simultaneously, demonstrating the capability of our multiple linear regression approach. That is, we are modeling all three parameters from the complexity estimate above. For these tests, we gathered observations on our local 16 node cluster for N ranging from 3000 to 9000,

Table 16. Mean values for observations of problem sizes 3000^2 to 9000^2 run on 16 nodes in various configurations

config \ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.66	25.22	36.22	50.15
2x8	2.49	5.22	9.45	15.37	23.42	33.59	46.71
4x4	2.50	5.14	9.26	14.96	22.74	32.87	45.57
8x2	2.95	5.95	10.42	16.60	24.84	35.48	48.81
16x1	4.23	8.09	13.68	21.08	31.01	43.51	58.97

Table 17. Problem-size/processor-configuration test for the Linpack benchmark, predicting 6000^2 problem size, and 4x4 processor-configuration

config\ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.7 +/- 0.15 (0.72)	25.22	36.22	50.15
				+ 0.21%			
2x8	2.49	5.22	9.45	14.96 +/- 0.11 (0.71)	23.42	33.59	46.71
				-2.63%			
4x4	2.62 +/- 0.13 (0.71)	4.97 +/- 0.14 (0.71)	8.84 +/- 0.15 (0.71)	14.57 +/- 0.15 (0.71)	22.51 +/- 0.16 (0.72)	33.01 +/- 0.19 (0.72)	46.42 +/- 0.27 (0.75)
	+5.09%	-3.22%	-4.53%	-2.63%	-1.01%	+ 0.43%	+ 1.87%
8x2	2.95	5.95	10.42	16.05 +/- 0.15 (0.71)	24.84	35.48	48.81
				-3.32%			
16x1	4.23	8.09	13.68	20.85 +/- 0.18 (0.72)	31.01	43.51	58.97
				-1.09%			

for all possible configuration of 16 processors (1x16, 2x8, 4x4, 8x2, and 16x1). The mean values for the observations are shown in Table 16.

As a first test, we provide all observations except those for the 6000^2 problem size and 4x4 processor configuration, and then predict the excluded row and column. The results are shown in Table 17. The results are very good, with all of the point estimates less than 6% from the mean of observations, and the 95% prediction interval accurately capturing the mean of observation.

Our second set of tests is somewhat more challenging, predicting the 16x1 processor configuration, and the 9000^2 problem size. The challenge comes from the 16x1 problem size, which shows significantly different runtimes from the other processor-configurations. The results are shown in Table 18. The predictions are quite accurate, this time with all point estimates less than 4% from the mean

Table 18. Problem-size/processor-configuration test for the Linpack benchmark, predicting the 9000^2 problem size, and 16x1 processor-configuration

config\ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.66	25.22	36.22	49.88 +/- 0.32 (0.65)
							-0.52%
2x8	2.49	5.22	9.45	15.37	23.42	33.59	46.88 +/- 0.44 (0.71)
							+ 0.36%
4x4	2.50	5.14	9.26	14.96	22.74	32.87	46.75 +/- 0.33 (0.66)
							+2.59%
8x2	2.95	5.95	10.42	16.60	24.84	35.48	50.18 +/- 0.27 (0.63)
							+ 2.81%
16x1	4.29 +/- 0.74 (0.93)	7.84 +/- 0.79 (0.97)	13.21 +/- 0.80 (0.98)	20.74 +/- 0.78 (0.96)	30.75 +/- 0.78 (0.97)	43.61 +/- 0.83 (1.01)	59.64 +/- 0.95 (1.10)
	+1.31%	-3.03%	-3.38%	-1.63%	-0.81%	+0.22%	+ 1.14%

of the observations, and nearly all of the means of observations within the 95% confidence and prediction intervals (the exceptions being 4x4 9000^2 and 8x2 9000^2).

6 Summary and Conclusions

We have presented an approach to employ both complexity estimates from the user and historical information from previous runs to make scalable predictions in the processor-number dimension (processor scalability), problem-size dimension (problem-size scalability), and processor-number/problem-size dimensions simultaneously. The solution applied is multiple linear regression, which not only provides predictions of mean values but also confidence and prediction intervals. The user provides rough complexity estimates and the coefficients are determined by the prediction system.

In our tests on the NAS EP and FT benchmarks, and the Linpack benchmark we have demonstrated that this approach is capable of making reliable predictions if the complexity estimate / model provided by the user are decently accurate. This requirement can create problems when communication libraries are utilized, as they may employ different algorithms, or even switch between algorithms depending on different parameters of the communication operations. The former problem can be addressed by documenting the complexity of the communication algorithms implemented in popular libraries, and possibly making this information available for automatic retrieval. Another difficulty can be

that problem-sizes and the number of nodes used often grow exponentially, as observed in the FT benchmark. This leads to the prediction of data points far away from the set of observations, which is more challenging.

We have also demonstrated that different characteristics such as computation and communication time can be considered and predicted separately, which is useful for coscheduling in a time-sharing environment. Another application would be prediction of an application's memory consumption using derived space-complexity estimates which are available for many algorithms.

Future work includes automatic checking of whether the assumptions regarding the error term hold and rejecting or modifying a model for which they are not met. A more advanced extension would be to experiment with automatically correcting inaccurate models by tear-down and build-up approaches of the function. That such approaches are feasible has been shown in [33]. How feasible it is for a user to provide the necessary models requires further exploration, as does the possibility of replacing or supplementing user estimates with compiler-derived models. The required models could also potentially be generated in a fully automated manner [34].

Acknowledgement

This research was supported by CFI via Grant No. 6191 and partially by NSERC. We thank SHARCNET for allowing us time on 64 nodes of their cluster at McMaster University, and in particular Mark Hahn for assisting us with running our tests.

References

[1] Angela C. Sodan and Lun Liu. *Dynamic Multi-Resource Monitoring for Predictive Job Scheduling with ScoPro.* Technical Report 04-002, U of W, CS Department, February 2005.

[2] Angela C. Sodan and Xuemin Huang. *Adaptive Time/Space Scheduling with SCOJO.* Int. Symp. on High-Performance Computing Systems (HPCS), Winnipeg/Manitoba, May 2004, pp. 165-178.

[3] Angela C. Sodan and Lin Han. *ATOP–Space and Time Adaptation for Parallel and Grid Applications via Flexible Data Partitioning.* 3rd ACM/IFIP/USENIX Workshop on Reflective and Adaptive Middleware, Toronto, Oct. 2004.

[4] Angela C. Sodan and Lei Lan. *LOMARC–Lookahead Matchmaking in Multi-Resource Coscheduling.* JSSPP (Workshop on Job Scheduling Strategies for Parallel Processing), New York / USA, June 2004, to appear in Springer.

[5] W. Cirne and F. Berman. *A Model for Moldable Supercomputer Jobs.* Proc. Internat. Parallel and Distributed Processing Symposium (IPDPS), April 2001.

[6] Angela C. Sodan. *Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling–A Survey.* Concurrency & Computation: Practice & Experience. Accepted for publication. (57 pages).

[7] V. K. Naik, S. K. Setia, and M. S. Squillante. *Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems.* J. of Parallel and Distributed Computing, Vol. 46, No. 1, 1997, pp. 28-47.

[8] Eitan Frachtenberg, Dror Feitelson, Fabrizio Petrini, and Juan Fernandez. *Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources.* Proc. Int. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.

[9] R.A. Gibbons. *Historical Application Profiler for Use by Parallel Schedulers.* Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), April 1997, Lecture Notes in Computer Science 1291, Springer Verlag.

[10] Mu'alem A and Feitelson D G. 2001. *Utilization, Predictability, Workloads and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling.* IEEE Transactions Parallel & Distributed Systems June 2001, 12(6).

[11] Perkovic D and Keleher P J. *Randomization, Speculation, and Adaptation in Batch Schedulers.* Proc. ACM/IEEE Supercomputing (SC), Dallas/TX, Nov. 2000.

[12] Chiang S-H and Vernon M K. *Characteristics of a Large Shared Memory Production Workload.* Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), June 2001, Lecture Notes in Computer Science 2221, Springer-Verlag, pp. 159-187.

[13] Smith W, Taylor V, and Foster I. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance.* Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 1999, Lecture Notes in Computer Science 1659, Springer Verlag.

[14] Arpaci-Dusseau A C, Culler D E, and Mainwaring A M. *Scheduling with Implicit Information in Distributed Systems.* Proc. SIGMETRICS'98/PERFORMANCE'98 Joint Conference on the Measurement and Modeling of Computer Systems, Madison/WI, USA, June 1998.

[15] M.E. Crovella and T.J. LeBlanc. *Parallel Performance Prediction Using Lost Cycles Analysis.* Proc. Supercomputing (SC), 1994.

[16] K. Keahey, P. Beckman, and J. Ahrens. *Ligature: Component Architecture for High Performance Applications.* The International Journal of High Performance Applications, 14(4):347-356, Winter 2000.

[17] Frederik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. *Performance Contracts: Predicting and Monitoring Grid Application Behavior.* Proc. 2nd Internat. Workshop on Grid Computing, Nov. 2001.

[18] G. Marin and J. Mellor-Crummey. *Cross-Architecture Predictions for Scientific Applications Using Parameterized Models.* Proc. Joint. Internat. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS), New York, NY, USA, June 2004.

[19] A. Snavely, L. Carrington, and N. Wolter. *Modeling Application Performance by Convolving Machine Signatures with Application Profiles.* Proc. IEEE 3th Annual Workshop on Workload Characterization, 2001.

[20] NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl .nist.gov /div898/handbook, retrieved October, 2004.

[21] J. Cohen, P. Cohen, S.G. West, and L.S. Alken. *Applied Multiple Regression/Correlation Analysis for the Behavioural Sciences, 3rd ed.* Mahwah, New Jersey, USA: Lawrence Erlbaum Associates, 2003.

[22] W. Mendenhall, R.J. Beaver, and B.M. Beaver. *Introduction to Probability and Statistics, 10th edition.* Pacific Grove, CA, USA: Brooks/Cole Publishing Company, 1999.

[23] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[24] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing, 2nd ed.* Addison Wesley, 2003.

[25] M. Yarrow, R.F. Van der Wijngaart. *Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes.* NAS Technical Report NAS-97-032, NASA Ames Research Center, Moffett Field, CA, 1997.

[26] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, S. Weeratunga. *Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors.* NAS Applied Research Branch Report RNR-94-007, NASA Ames Research Center, Moffet Field, CA, 1993.

[27] Maple 9.5–Advanced Mathematics Software for Engineers, Academics, Researchers, and Students, http://www.maplesoft.com /products /maple /index .aspx, retrieved December 2004.

[28] OpenMaple–An API into Maple, http://www.adaptscience .com/ products/ /maple/html/OpenMaple.html, retrieved December 2004.

[29] HPL–A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, http://www.netlib.org/benchmark/hpl/, retrieved June, 2005.

[30] Ian Foster. *Designing and Building Parallel Programs.* Reading, MA: Addison-Wesley, 1995.

[31] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. *Automatically Tuned Collective Communications.* IEEE/ACM Supercomputing, Nov. 2000.

[32] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. *Performance Analysis of MPI Collective Operations.* PMEO-PDS, Apr. 2005.

[33] Ljupčo Todorowski, Peter Ljubič, and Sašo Džeroski. *Inducing Polynomial Equations for Regression.* ECML, 2004.

[34] E. Schmidt, A. Schulz, L. Kruse, G. von Cölln, and W. Nebel. *Automatic Generation of Complexity Functions for High-Level Power Analysis.* PATMOS, 2001.

Open Job Management Architecture for the Blue Gene/L Supercomputer

Yariv Aridor[1], Tamar Domany[1], Oleg Goldshmidt[1], Yevgeny Kliteynik[1], Jose Moreira[2], and Edi Shmueli[1]

[1] IBM Haifa Research Labs, Haifa, Israel
{yariv, tamar, olegg, kliteyn, edi}@il.ibm.com
[2] IBM Systems and Technology Group, Rochester MN
jmoreira@us.ibm.com

Abstract. We describe an open job management architecture of the Blue Gene/L supercomputer. The architecture allows integration of virtually any job management system with Blue Gene/L with minimal effort. The architecture has several "openness" characteristics. First, any job management system runs outside the Blue Gene/L core (i.e. no part of the job management system runs on Blue Gene/L resources). Second, the logic of the scheduling cycle (i.e. when to match jobs with resources) can be retained without modifications. Third, job management systems can use different scheduling and resources allocation models and algorithms.

We describe the architecture, its main components, and its operation. We discuss in detail two job management systems, one based on LoadLeveler, the other — on SLURM, that have been successfully integrated with Blue Gene/L, independently of each other. Even though the two systems are very different, Blue Gene/L's open job management architecture naturally accommodated both.

1 Introduction

Blue Gene/L is a highly scalable parallel supercomputer developed by IBM Research for the Lawrence Livermore National Laboratory [1]. The supercomputer is intended to run highly parallel computational jobs developed using the popular MPI programming model [2, 3]. The computational core of the full Blue Gene/L consists of $64 \times 32 \times 32 = 65536 = 2^{16}$ nodes connected by a multitoroidal interconnect [5]. A Blue Gene/L prototype of a quarter of the full size ($16 \times 32 \times 32 = 16384 = 2^{14}$ nodes) is currently rated the fastest supercomputer in the world [4]. The computational core is augmented by an I/O subsystem that is comprised of additional nodes and an internal control subsystem.

An essential component of parallel computers is a job management system that schedules submitted jobs for execution, allocates computational and communications resources for each job, launches the jobs, tracks their progress, provides infrastructure for runtime job control (signaling, user interaction, debugging, etc.), and handles job termination and resource release. A typical job management system consists of a master scheduling daemon and slave daemons that

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 91–107, 2005.

launch, monitor, and control the running parallel jobs upon instructions from the master and periodically report the jobs' state. Quite a few such systems are available: the Portable Batch System (PBS, see [6]), LoadLeveler [7], Condor, [8], and SLURM [9], are but a few examples.

Job management systems are usually tightly integrated with the multicomputers they run on. Often the architectural details (such as the hardware and the OS, the interconnect type and topology, etc.) of the machine are exposed to — and hardwired into — the job management system. Even if this is not the case, there is still the problem that the slave daemons run on the same nodes that execute the user jobs. This means that the machine architecture and the requirements of the applications put restrictions on the job management system that can be used on the particular machine: a job management system that has not been ported to the particular architecture cannot be used. On the other hand, an application that needs, for example, a particular version of an operating system can only be run if there is a corresponding port of the slave job management daemons available.

This paper describes the *open job management architecture* we developed for Blue Gene/L. By "open" we mean that virtually any job management system, be it a third party product or an in-house development, can be integrated with Blue Gene/L without architectural changes or a major porting effort. In other words, our architecture decouples the job management system from Blue Gene/L's core, thus removing the restrictions mentioned above. The particular "openness characteristics" of Blue Gene/L's job management architecture are:

1. The whole job management system runs outside of Blue Gene/L's core, keeping Blue Gene/L clean of any external software and removing the traditional dependency of the job management system on the core architecture.
2. The job management system logic can be retained without modifications. For instance, while one job management system may search for available resources each time it schedules a job, another may partition the machine in advance and match the static partitions with submitted jobs. Our architecture allows both schemes, as well as many others.
3. The job management system can access and manipulate Blue Gene/L's resources in a well-defined manner that allows using different scheduling and resource allocation schemes. For instance, a job management system can use any scheduling policy (first come first served, backfilling, etc.),[1] and any algorithm for matching resources to the job (first fit, best fit, and so on), since Blue Gene/L presents raw information on its components without imposing a particular model of allocation of available resources.

In this paper we present the basic structure of Blue Gene/L's open job management architecture below, discuss its main features, and describe an implementation of a job management system based on IBM's LoadLeveler [7] for Blue

[1] At present, Blue Gene/L does not support checkpoint/restart, and therefore cannot preempt running jobs. This is not a limitation of our job management architecture: once the core support is added it will be simple to add the corresponding functionality to the job management system described here.

Gene/L. We also describe a very different job management system based on SLURM [9] that has recently been integrated successfully with Blue Gene/L. The sequence of job and resource management operations in the two systems differs very significantly. Nevertheless the architecture is capable of naturally accommodating both schemes.

The rest of the paper is organized as follows. Section 2 describes Blue Gene/L's open job management architecture in detail. Section 3 describes our implementation of LoadLeveler for Blue Gene/L. Section 4 shows how a very different SLURM-based job management system can be integrated into the same framework. Section 5 presents some experiments and performance measurements we conducted, and Section 6 concludes the paper.

2 The Open Job Management Architecture

The core of the job management infrastructure of Blue Gene/L consists of two main components. One is a portable API we developed to interact with the Blue Gene/L's internal control system. The control system is responsible for everything that happens inside the Blue Gene/L core, from boot to shutdown. In particular, it implements all the low level operations that are necessary for managing the Blue Gene/L's resources and the parallel jobs that run in the core. The API provides an abstraction for the job management functionality, namely access to information on the state of Blue Gene/L (which computational and communications resources are busy or free, which components are faulty or operational, what jobs are running, what resources are allocated to them, etc.) and a set of job and resource management primitives (such as allocation of resources, booting the nodes, launching, signaling, and terminating jobs, and so on), while hiding the internals of Blue Gene/L from the job management system. We called the API the *Job Management Bridge API*, or *Bridge API* for short, since it provides a "bridge" between the job management system and the internal Blue Gene/L control system.

The Bridge API is essential for providing Blue Gene/L with the openness characteristics 1 and 3 described above. Moreover, the implementation of the API does not impose any restrictions on the order in which the primitives can be invoked, which is what provides property 2 (see also "Order of operations" in Section 2.2 below).

The second component is a special program called `mpirun`, instances of which run on a cluster of designated machines outside of the Blue Gene/L core. Each instance of `mpirun` is a "proxy" of the real parallel job running on the Blue Gene/L core, and communicates with it using the Bridge API. The slave daemons of the job management system run on the same dedicated cluster and interact only with the `mpirun`s (see Figure 1). All the job control operations such as launching, signals, and termination are passed from a daemon to the corresponding `mpirun`, and from the `mpirun` to the real parallel job it controls. All the feedback from the running parallel job is delivered to the `mpirun` and passed to the job management system. Thus, the traditional job management

architecture is preserved: the job management system sees Blue Gene/L as a cluster running `mpirun`s. This provides a clean separation between the Blue Gene/L core and the job management system satisfying the openness property 1 above.

From the point of view of the slave daemons the `mpirun`s represent the real jobs. The `mpirun`s communicate with Blue Gene/L's internal control system via the Bridge API, which provides the necessary abstraction. The master daemon that manages a queue of submitted jobs[2], schedules jobs, and allocates resources, communicates with the slaves and uses the Bridge API to query the machine state and determine which resources are available (cf. Figure 1).

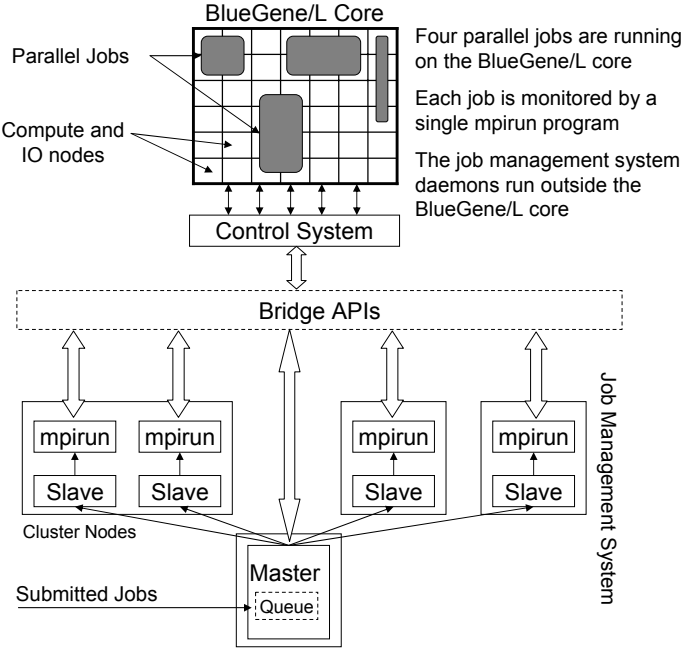


Fig. 1. Blue Gene/L's open job management architecture

2.1 Resource Management on Blue Gene/L

A central part of the job management system is allocating resources for each scheduled job. On Blue Gene/L each job requests — and is allocated — several classes of resources, namely computational nodes, interconnect resources such as network switches and links, and I/O resources. For reasons of protection and security Blue Gene/L does not allow messages belonging to one job pass through hardware allocated to another job (this also helps avoiding contention on network

[2] There may be more than one such queue, as is the case in Condor [8]. This is internal to the job management system and is transparently supported by our architecture.

resources and simplifies routing). Thus, all the resources allocated to a job are dedicated.

The set of resources that belong to a job is called a *partition*. A partition consists of a set of computational nodes[3] connected according to the job's requirements and the corresponding network and I/O resources. Before a job can run its partition must be "booted." By booting a partition we mean the process of booting and configuring all the nodes that belong to the partition, wiring all the network switches as required, and performing all the initialization required to start a job. This process is not instantaneous in general, and after issuing the appropriate command the system should monitor the partition's status until the partition is ready (cf. "Partition Management" in Section 2.2 below). Destroying a partition is the reverse process.

In what follows we will treat Blue Gene/L partition management as an integral part of the job management system.

2.2 The Bridge API

The Bridge API is logically divided into several major areas: the data retrieval part, the partition management part, and the job management part. The API can be called directly from C or C++ code. All the Bridge API functions return a status code that indicates success or failure of each operation.

2.2.1 Machine State Query

To allocate resources for a job, the scheduler must have access to the current state of Blue Gene/L. There is a number of accessor functions that will fetch the information on the machine as a whole or parts thereof from Blue Gene/L itself. Each accessor will allocate memory for the structure retrieved and fill it with data. This arrangement allows using the structures in name only, rather than in size, keeping the details hidden from the client. This means, however, that client code must free the memory when a data structure is no longer needed, and the API provides the corresponding functions.

The highest-level function that provides access to the machine state is `get_BGL()` that brings the full snapshot of the current state of all the Blue Gene/L resources into the scheduler. Once the snapshot is available, a generic data retrieval function, `get_data()` can be called to retrieve various components, properties, and fields: it accepts a pointer to the queried structure, a field specification, and a pointer to memory where the query result is written. This is the only mechanism to access the machine resources, and the implementation details are not exposed to the client.

[3] We assume below that a partition is a logical rectangle of nodes with dimensions that are specified by the job to which the partition is allocated. There may be other policies that specify the partitions' shapes: the jobs may specify the overall number of nodes only, letting the job management system determine the dimensions; the partition sizes may be predetermined and not related to job sizes at all (cf. Section 4 below); in general, partitions don't even have to be rectangular. All these options are supported by our architecture.

2.2.2 Partition Management

The partition management API facilitates adding and removing partitions, boot and shutdown of partition components, and queries of the partition state. The basic functions are:

1. `add_partition()` — aggregate some of the Blue Gene/L resources into a partition and add the partition to the system. This operation, as well as the complementary `remove_partition()` (cf. item 2 below) does not cause any physical side effects in the Blue Gene/L core but only creates a logical association of resources. Each resource, for instance a compute node or a network link, can belong to more than one partition as long as no more than one of the partitions is active (cf. item 3 below). Partitions may be added without reference to a particular job. This facilitates such operations as partition reservation. A job management system can limit the number of partitions that a resource can belong to, and, once `add_partition()` is called, consider the partition components "allocated" and unavailable until the partition is removed, but this is not mandatory.
2. `remove_partition()` — remove a partition from the system. This does not necessarily mean that the resources that belonged to that partition are free — some or all of them may belong to other partitions, one of which may be active. Just like `add_partition()`, `remove_partition()` does not have any physical consequences, it only removes a logical association of resources.
3. `create_partition()` — activate a partition, i.e. boot all the nodes that belong to it, connect all the switch ports according to the partition topology specification, and prepare the partition to run a job. This operation, together with the complementary `destroy_partition()` (cf. item 4 below) cause real changes in the Blue Gene/L core. In particular, any core resource, e.g. a node or a network link, can belong only to one active partition at a time, and no other partition that shares one or more resources with an active partition can be activated until the active partition is destroyed (cf. item 4 below). Executing `create_partition()` does not necessarily mean that the partition is allocated to a job — it is a policy decision left to the job management system. A partition that is already active but not allocated to a particular job can be used to run a job from the submit queue, if it fits the requirements.
4. `destroy_partition()` — deactivate, (i.e. shut down) a partition, usually after the job running in the partition terminates. This does not destroy logical associations between Blue Gene/L resources and partitions — the partition still exists and its resources remain appropriately marked.
5. `get_partition()` — retrieves the full information about a partition. This function is useful for various queries, the most important of which is checking the partition's state, for instance whether the partition is active or not.

2.2.3 Job Management

The job management API facilitates control of the jobs running on Blue Gene/L. The basic job management functions are:

1. `add_job()` — adds a job to Blue Gene/L. This is a purely logical operation that does not mean that the job starts to run, or has been scheduled, or has been allocated resources. It is, however, a necessary step before a job can run on Blue Gene/L.
2. `remove_job()` — removes a job from Blue Gene/L. This normally happens after the job terminates, whether normally or abnormally.
3. `start_job()` — launches the job. This does not necessarily mean that the job starts running immediately, nor is it implied that the job's partition is active: the job can remain "pending," i.e. waiting for its partition to boot, and it will start when the boot is completed.
4. `signal_job()` — send a signal to a job. The job does not have to be running. However, there is not much sense in sending a signal to a job that is not running: the signal will either be ignored or an error code will be returned that can be handled by the caller.
5. `cancel_job()` — cancel a job. This is a special case of `signal_job()` used to terminate a running job. Again, there is no sense is canceling a job that is not yet running — one should use `remove_job()` (item 2 above).
6. `get_job()` — retrieves the full information about the job. The most important use is to query the job's status, for instance whether or not it has terminated.

2.2.4 Order of Operations

It is important to note that very few restrictions are placed on the order of the above Bridge API functions. Consider the following scenarios that show how few dependencies there are:

- Partitions can be created in advance, i.e. not as a result of a request from a particular job. Thus, some or all of `add_partition()`, `create_partition()`, and `get_partition()` (items 1, 3, and 5 from "Partition Management" above) may precede the scheduling cycle that will consist of job management operations only. For instance, the SLURM-based job management system described in Section 4 below takes advantage of this.
- A job can be started before a partition is ready and will wait for the partition to boot, providing a "launch and forget" functionality. In principle there is nothing that prevents monitoring of the status of such "pending" job, or even terminating it before it starts executing — thus calls to `add_job()`, `start_job()`, and `get_job()` (items 1, 3, and 6 from "Job Management" above) can precede calls to functions `create_partition()` and `get_partition()` (items 3 and 5 from "Partition Management" above).
- In off-line (batch) scheduling systems, when the job list is known in advance, one can populate the system with all the needed partitions and jobs. The scheduling system will just need to start each job in its designated partition when the time comes. Thus `add_partition()` and `add_job()` can be called in the preparation phase and only `create_partition()` and `start_job()` will be called for each job.

There are some trivial exceptions, of course, e.g., a job must be added to the system with `add_job()`) before it can start via `start_job()`, a partition must be

added with `add_partition()` before it can be queried with `get_partition()`, and calling `destroy_partition()` will have no effect unless `create_partition()` has been called.

In general, there are purely logical operations, such as

- `add_partition()`,
- `remove_partition()`,
- `add_job()`,
- `remove_job()`,

physical operations like

- `create_partition()`,
- `destroy_partition()`,
- `start_job()`,
- `signal_job()`,
- `cancel_job()`,

and query operations like

- `get_partition()`,
- `get_job()`.

The logical operations have no real side effects (apart from storing or erasing information), and accordingly `add_partition()` and `add_job()` can be called virtually any time. The physical and query operations can be performed only on objects that logically exist, and one cannot destroy an inactive partition of signal or cancel a job that is not running.

These restrictions will be satisfied by any sane job management system. No other restrictions are imposed on the logic of the job management system, and any job cycle model can be used.

2.3 `mpirun`

A typical job management system consists of a master scheduling daemon running on the central management node and slave daemons that execute on the cluster or multicomputer nodes. The master daemon accepts the submitted jobs and places them in a queue. When appropriate, it chooses the next job to execute from that queue and the nodes where that job will execute, and instructs the slave daemon on that machine to launch the job. The slave daemon forks and executes the job, which can be serial or parallel. It continuously monitors the running job and periodically reports to the master that the job is alive. Eventually, when the job terminates, the slave reports the termination event to the master, signaling the completion of the job's lifecycle.

The slave daemons are not allowed to run inside the Blue Gene/L core, any action they perform has a corresponding Bridge API function that delegates the action to Blue Gene/L's internal control system. The slave daemons run on designated machines outside the Blue Gene/L core, and instead of forking the

real user job they execute `mpirun`, which starts the real job in Blue Gene/L's core via the Bridge API (as shown in Figure 1). A slave daemon needs monitor only the state of the corresponding `mpirun`, not the state of the real parallel job, while `mpirun` queries the state of the real job via the Bridge API and communicates the result to the slave daemon.

When `mpirun` detects that the job has terminated — normally or abnormally — it exits with the return code of the job. The slave reports the termination event and the exit code to the master, signaling the completion of that job's lifecycle.

On the other hand, a failure of `mpirun` is noticed by Blue Gene/L's internal control system via the usual socket control mechanisms. The parallel job the `mpirun` controlled is orphaned, loses its communication channel with the job management system, and, in general, dies. Thus, a parallel job and the corresponding `mpirun` do indeed form a single logical entity with the same lifespan.

The role of `mpirun` is not limited to just querying the state of the parallel job. It can actively perform other actions such as allocating and booting a partition for the job, as well as cleaning and halting the partition when the job terminates. Obviously, in this case the master daemon should not concern itself with these additional tasks.

This flexibility allows different job management system to optimize their performance by designating a different set of responsibilities to `mpirun`. For example, in our LoadLeveler port to Blue Gene/L (see Section 3 below), the booting of the partition is initiated by the LoadLeveler master daemon, but it is the `mpirun` that waits for the partition to boot and launches the job on it. This allows the single scheduling thread of LoadLeveler to consider the next jobs in the queue, even if the partition for the previous job is not yet ready.

Redirection of standard input and output to and from the parallel job is an additional important role of `mpirun`. Any input that is received on `mpirun`'s standard input, is forwarded to the parallel job running on the Blue Gene/L core. When the parallel job writes to standard output or error, its output is forwarded back to `mpirun`'s standard output or error respectively.

This redirection is important because it is similar to the way job management systems handle their jobs' I/O. When the slave daemons fork and execute jobs, they redirect the jobs standard input and output to files. For a Blue Gene/L job, this means that the files used for `mpirun`'s standard output and error will actually contain the parallel job's standard output and error, and the file used as an input for `mpirun` will be forwarded to the parallel job.

2.4 Related Work

There are other efforts to provide open architectures for resource and job management, notably in the realm of grid computing, e.g. GRAM [10], DRMAA [11]. The main focus in those efforts lies in management of heterogeneous resources and providing consistent, standardized APIs in heterogeneous systems. While there are similarities with our Bridge API, our focus is quite different. We have a homogeneous machine, and we are interested in allowing any job management

system to be integrated with it, while GRAM and DRMAA aim to allow a single job management system to operate on heterogeneous resources, or to facilitate co-operation between local schedulers and global meta-schedulers.

3 LoadLeveler for Blue Gene/L

To validate our design we implemented our own job management system on the basis of LoadLeveler, a job scheduling system developed by IBM [7], and integrated it with Blue Gene/L. LoadLeveler is a classical scheduling system that consists of a master scheduling daemon and slave daemons that launch and monitor jobs on cluster nodes. We successfully deployed LoadLeveler on a $16 \times 32 \times 32 = 2^{14}$ node Blue Gene/L prototype [4]. LoadLeveler launches and controls `mpirun`s on the job management cluster of 4 nodes, and the `mpirun`s, in turn, control the real parallel jobs running on the Blue Gene/L core.

Adapting LoadLeveler to work with Blue Gene/L included development of a partition allocator that used the Bridge API and was called by the LoadLeveler scheduler, and making the slave daemons call `mpirun` with the proper arguments. The difficulty of creating a partition allocator depends primarily on the sophistication of the associated algorithms, which may vary according to the needs of the customer. The actual integration with the Bridge API and `mpirun` is simple.

3.1 LoadLeveler Job Cycle Model

LoadLeveler does not make any assumptions regarding Blue Gene/L's workload. It is an on-line system where jobs of any size and priority can arrive at any time. The lifecycle of a job on Blue Gene/L starts when a user submits the job to the job management system. The job is placed on a queue, and the scheduler picks a job from the queue periodically and attempts to schedule it.

The scheduler's task is to allocate a partition to the chosen job and launch the job on the partition. The job carries a set of requirements that the partition must satisfy. In particular, the job may specify its total size or a particular shape (a three-dimensional rectangle of specified size $x \times y \times z$), and the required partition topology — a mesh or a torus (cf. [5]). It may happen that no partition that can accommodate the job can be found, then the scheduler may choose another job from the queue, according to some algorithm (e.g. backfilling, cf. [12]). The details of possible scheduling algorithms are beyond the scope of this paper — we focus on the general architecture of the job management system that can accommodate different schedulers.

In general, the input for the scheduler of the job management system consists of the job requirements and the current state of Blue Gene/L's resources. A partition that contains the necessary resources (computational nodes, communication, and I/O) is created, and the job is launched. For each terminating job the corresponding partition is destroyed, thus returning the resources to the system for further reuse.

Down to a finer level of details, the typical job lifecycle in Blue Gene/L looks as follows:

1. The scheduler obtains the full information on the machine state to make a decision whether to launch or defer the job and how to allocate resources for it. The information is obtained via the `get_BGL()` function of the Bridge API (cf. "Machine State Query" in Section 2.2).
2. If suitable resources cannot be found another job may be picked from the submissions queue. Once resources are found they are aggregated into a partition, and the Blue Gene/L's control system is informed via the `add_partition()` function of the Bridge API (item 1 in "Partition Management" in Section 2.2).
3. The new partition (i.e. all the computational and I/O nodes that belong to it) is booted next. This is accomplished via the `create_partition()` function of the Bridge API (item 3 in "Partition Management" in Section 2.2). From this moment on the resources included in the partition are considered effectively "allocated" by LoadLeveler.
4. The boot is not instantaneous, so the job management system will monitor the partition's state until the partition is ready to run the job. The probing functionality is provided by the `get_partition()` function of the Bridge API (item 5 in "Partition Management" in Section 2.2).
5. Once the partition is up and ready, the scheduled job is added to Blue Gene/L's control system. The `add_job()` of the Bridge API (item 1 in "Job Management" in Section 2.2) is used for this task.
6. The next logical task is launching the job on the prepared partition. This is achieved via the `start_job()` function of the Bridge API (item 3 in "Job Management" in Section 2.2).
7. The system then keeps monitoring the running job, checking its status periodically using the `get_job()` function (item 6 in "Job Management" in Section 2.2) until the job terminates, normally or abnormally.
8. Once the job terminates, it is removed from the system via the `remove_job()` (item 2 in "Job Management" in Section 2.2) and steps can be taken to release the resources.
9. The partition is shut down — a task performed using the `destroy_partition()` of the Bridge API (item 4 in "Partition Management" in Section 2.2).
10. The now idle partition can be removed — the `remove_partition()` (item 2 in "Partition Management" in Section 2.2) provides the necessary facilities — and the corresponding resources can be reused for other jobs from this moment on.

Note that if a job fails the system releases the resources in exactly the same way as in the case of normal termination, except that the failure is handled as appropriate by `mpirun`. Note also that Blue Gene/L kills a parallel job if any of its processes fails for any reason. This is a characteristic of the Blue Gene/L itself, and not a limitation of the job management architecture.

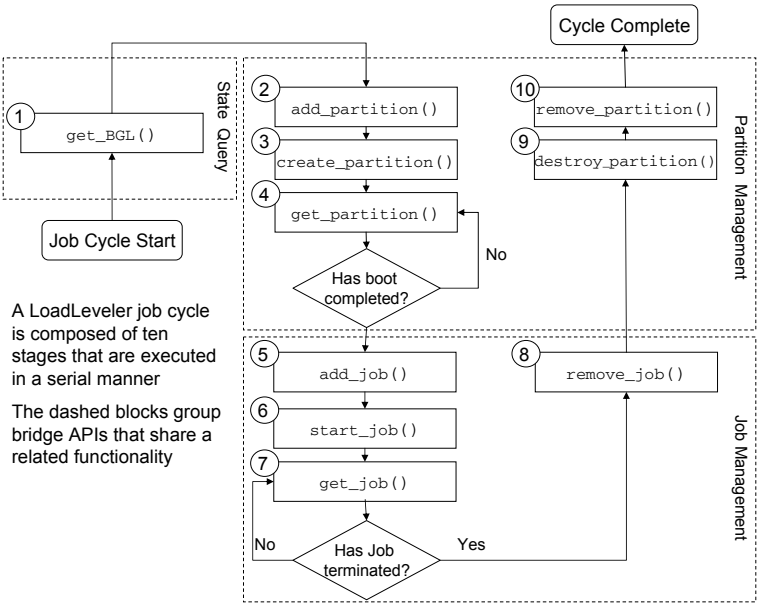


Fig. 2. LoadLeveler job cycle

The job cycle described above is depicted as a flow chart in Figure 2. The chart shows how the machine state query, the partition management, and the job management components of the Bridge API are used.

3.2 `mpirun` for LoadLeveler

In our architecture, the actual communication with Blue Gene/L's internal control system is a function of the `mpirun` proxy job. In our implementation `mpirun` handles stages 4 through 8 (cf. Section 3.1 above). In other words, LoadLeveler handles job scheduling and creation and destruction of partitions, while `mpirun` is responsible for monitoring the partitions' boot, adding, starting, monitoring jobs, and removing terminated jobs from the system.

Figure 3 shows the same job cycle flow chart as Figure 2, showing the separation of concerns between LoadLeveler and `mpirun`.

This division of labor between LoadLeveler and `mpirun` is by no means the only one possible. For instance an implementation of `mpirun` may be passed the job and partition structures (or references thereof) and handle all the stages starting from 2 to the final 10 (cf. Section 3.1). Alternatively, the role of `mpirun` may be reduced to stages 6 and 7 only: if the scheduler is invoked each time a job terminates it can handle stages 8 through 10 before picking another job from the queue.

Our LoadLeveler scheduler is single-threaded, so any additional tasks, e.g. synchronous waiting for a partition to boot (stage 4 in Section 3.1), will prevent it from scheduling another job from the queue while it is busy. A multi-threaded

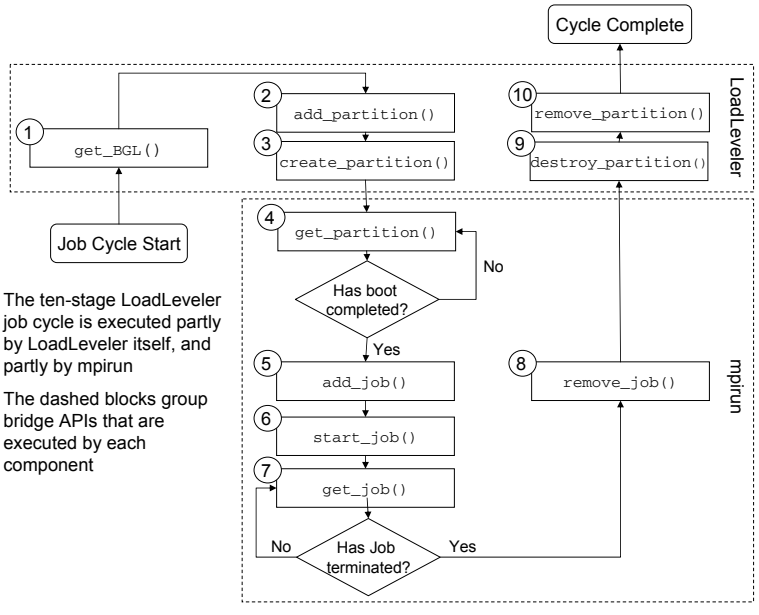


Fig. 3. `mpirun` for LoadLeveler

scheduler will be free of this disadvantage, at the expense of added complexity. Our design lets the scheduler process a job and allocate resources to it, and delegates all the tasks performed on a job and its allocated partition, including `get_partition()`, to `mpirun`.

4 SLURM and Blue Gene/L

In this section we describe another — very different — implementation of a job management system for Blue Gene/L based on SLURM (Simple Linux Utility for Resource Management, [9]). SLURM is an open source resource manager designed for Linux clusters. Like other classic job management systems, it consists of a master daemon on one central machine and slave daemons on all the cluster nodes. SLURM for Blue Gene/L was successfully deployed and tested on a 32K node Blue Gene/L machine [14]: like LoadLeveler, it launches and controls `mpirun` instances that in turn control parallel jobs running on Blue Gene/L. SLURM's job cycle, however, is very different from that of LoadLeveler (cf. Section 3.1).

4.1 Partitioning and Job Cycle in SLURM

SLURM operates under a number of assumptions regarding the expected workload that lead to a very different job cycle model and hence a different resource management scheme from LoadLeveler. In particular, SLURM assumes that

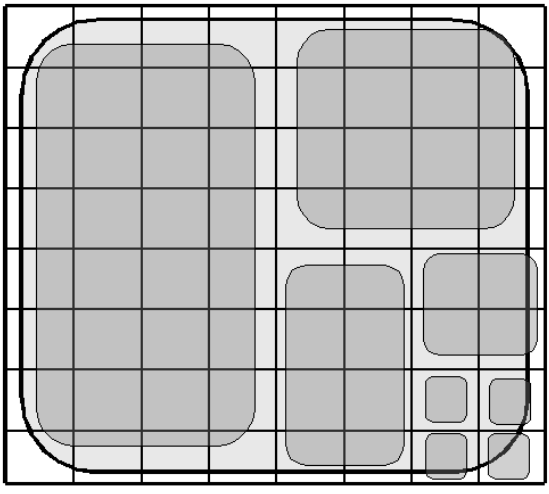


Fig. 4. Possible SLURM partitions

- most jobs are short enough for the partition boot time overhead to be substantial (see Section 5 for relevant performance measurements);
- all job sizes are powers of 2;
- jobs do not request a specific shape, specifying the total size only; the system is free to choose a partition of arbitrary shape and does so, picking a partition from the prepared set;
- jobs that require the full machine have low priority and thus can be delayed (e.g. until the next weekend).

SLURM takes advantage of the fact that the Bridge API allows resources belonging to different inactive partitions to overlap. Each resource can belong to a set of partitions. The system will not consider the resource allocated until a job starts in one of the partitions the resource belongs to. Accordingly, SLURM adds a set of partitions of various sizes to the system (using `add_partition()`, cf. "Partition Management" in Section 2.2) in advance. The partition set covers all resources and the partitions may overlap. For example, SLURM may divide the machine into partitions of size $1/2^k$, i.e. there will be partitions containing halves, quarters, eighths of the machine, etc., as well as a partition that contains the entire machine (cf. Figure 4).

SLURM maintains two job queues: one for jobs of size up to half the system and one for jobs larger than half the machine. The latter jobs are only run during specific time (e.g. on weekends). On job arrival, SLURM picks a partition for it from the existing set of partitions and boots it, if needed. Once booted, a partition will remain booted until its resources are required for another partition (i.e. the full machine partition). This scheme is designed to minimize the number of partition boots.

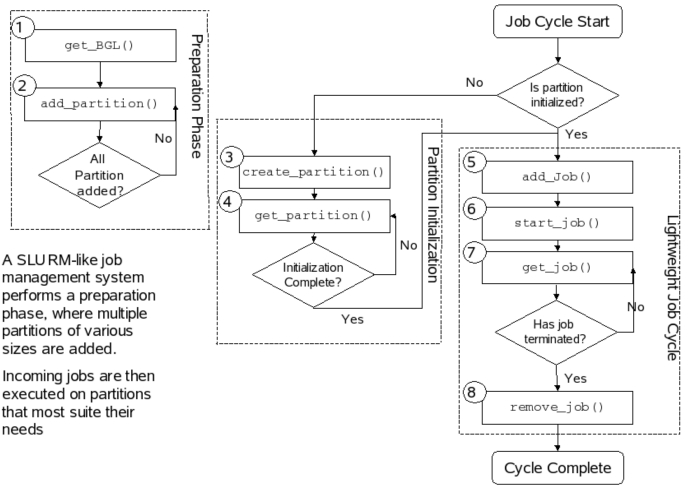


Fig. 5. Job cycle in SLURM

From a more general point of view we can say that the entire system is divided into a set of already booted partitions, and there is a job queue for each job size. Since the partitioning is done in advance the job cycle can be summarized as finding a suitable partition, starting the job, and waiting for the job to terminate, which corresponds to stages 5 through 8 in Section 3.1 above (cf. Figure 5). The lightweight job cycle is controlled by `mpirun`, while the preparation phase is performed by SLURM proper.

SLURM maintains the partition information and the job queues. For each job it chooses a partition from the existing set and starts `mpirun`. The `mpirun` in turn adds the job, starts it, monitors it, and removes it upon termination.

Note how different this picture is from the LoadLeveler model of Section 3. Note also that partition management is not a part of the normal job cycle. Only when jobs requiring the whole machine are run over weekends need partitions be rebooted. During normal operation they remain pre-allocated and pre-initialized.

Nonetheless, both SLURM and LoadLeveler are accommodated equally well by the open job management architecture of Blue Gene/L.

5 Performance

We performed some experiments on a 4096-node Blue Gene/L prototype in order to assess the performance of our architecture. We intentionally ran jobs that do nothing to isolate the overhead of adding, creating, and destroying partitions, and launching jobs. Some representative results are shown in Table 1 that lists times (in seconds) taken by different stages of the LoadLeveler job cycle (cf. Section 3.1). All the times are averages of several experiments.

These results are not final. The system is still under development, and the performance is continuously being improved.

Table 1. Times (in seconds) taken by different job cycle stages of Section 3.1

size (nodes)	stages $1-3$	stage 4	stages $5-8$	stages $9-10$	total (sec)
512	1	26	14	10	51
1024	1	36	15	11	63
2048	2	38	19	11	70

One can note that the dependency of partition boot times (stage 4 of Section 3.1) and the rest of the operations on the size of the partition is rather weak. This is expected since the boot process and the job loading are parallelized. Resource management proper (stages $1-3$ and $9-10$) takes even less time than booting a partition. Overall, we can conclude that for jobs longer than a few minutes the resource and job management overhead (of the order of a minute, according to Table 1) is small.

6 Conclusions

We presented Blue Gene/L's open job management architecture that allows integration of virtually any job management system with Blue Gene/L. The job management system runs outside of Blue Gene/L core, and the integration does not involve any architectural changes in the system, or affect its logic.

The two main components of the architecture are the Bridge API that provides an abstraction on top of Blue Gene/L's internal control system, and a proxy `mpirun` program that represents the real parallel job to the job management system. The Bridge API imposes only the most trivial restrictions on the job lifecycle model and logic used by the job management system, and the division of labor between `mpirun` and the job management system is also very flexible.

There are two implementations of job management system that have been successfully integrated with Blue Gene/L, one based on LoadLeveler, the other — on SLURM. Blue Gene/L's open job management architecture accommodates both system equally well, despite very significant differences. Work is now under way to integrate yet another job management system, PBS.

Acknowledgments

We are grateful to Morris Jette and Danny Auble for discussions of their work on SLURM for Blue Gene/L.

References

1. N. R. Adiga et al. "An Overview of the Blue Gene/L Supercomputer," Supercomputing 2002.
2. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`

3. W. Gropp, E. Lusk, & A. Skjellum. "Using MPI: Portable Parallel Programming with the Message Passing Interface," 2nd edition, MIT Press, Cambridge, MA, 1999.
4. Top 500, http://www.top500.org/lists/2004/11/
5. Y. Aridor et al. "Multi-Toroidal Interconnects: Using Additional Communication Links to Improve Utilization of Parallel Computers" In: 10th Workshop on Job Scheduling Strategies for Parallel Processing, New York, NY, 2004.
6. http://www.openpbs.org
7. A. Prenneis, Jr. "LoadLeveler: Workload Management for Parallel and Distributed Computing Environments." In Proceedings of Supercomputing Europe (SUPEUR), 1996.
8. http://www.cs.wisc.edu/condor/
9. M. A. Jette, A. B. Yoo, and M. Grondona "SLURM: Simple Linux Utility for Resource Management." In D. G. Feitelson and L. Rudolph, editors, Job Scheduling Strategies for Parallel Processing, pp. 37 51. Springer-Verlag, 2003.
10. http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/index.html
11. http://www.drmaa.org/
12. A. W. Mualem and D. G. Feitelson "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling." In IEEE Transactions on Parallel and Distributed Computing, v. 12, pp. 529–543, 2001.
13. G. Almasi et al. "System Management in the BlueGene/L Supercomputer," 3rd Workshop on Massively Parallel Processing, Nice, France, 2003.
14. M. Jette, D. Auble, private communication, 2005. See also "SLURM Blue Gene User and Administrator Guide", http://www.llnl.gov /linux /slurm /bluegene.html

AnthillSched: A Scheduling Strategy for Irregular and Iterative I/O-Intensive Parallel Jobs

Luís Fabrício Góes[1], Pedro Guerra[1], Bruno Coutinho[1], Leonardo Rocha[1], Wagner Meira[1], Renato Ferreira[1], Dorgival Guedes[1], and Walfredo Cirne[2]

[1] Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brasil
{lfwgoes, pcalais, coutinho, lcrocha, meira, renato, dorgival}@dcc.ufmg.br
[2] Universidade Federal de Campina Grande, Campina Grande, PB
walfredo@dsc.ufcg.edu.br

Abstract. Irregular and iterative I/O-intensive jobs need a different approach from parallel job schedulers. The focus in this case is not only the processing requirements anymore: memory, network and storage capacity must all be considered in making a scheduling decision. Job executions are irregular and data dependent, alternating between CPU-bound and I/O-bound phases. In this paper, we propose and implement a parallel job scheduling strategy for such jobs, called AnthillSched, based on a simple heuristic: we map the behavior of a parallel application with minimal resources as we vary its input parameters. From that mapping we infer the best scheduling for a certain set of input parameters given the available resources. To test and verify AnthillSched we used logs obtained from a real system executing data mining jobs. Our main contributions are the implementation of a parallel job scheduling strategy in a real system and the performance analysis of AnthillSched, which allowed us to discard some other scheduling alternatives considered previously.

1 Introduction

Increasing processing power, network bandwidth, main memory, and disk capacity have been enabling efficient and scalable parallelizations of a wide class of applications that include data mining [1, 2], scientific visualization [3, 4], and simulation [5]. These applications are not only demanding in terms of system resources, but also a parallelization challenge, since they are usually irregular, I/O-intensive, and iterative. We refer to them as 3I applications or jobs. As irregular jobs, their execution time is not really predictable, and pure analytical cost models are usually not accurate. The fact that they are I/O intensive make them even less predictable, since their performance is significantly affected by the system components and by the amount of overlap between computation and communication that is achieved during the job execution. Further, 3I jobs perform computations spanning several data domains, not only consuming data

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 108–122, 2005.

from those domains, but also generating new data dinamically, increasing the volume of information to be handled in real time. Finally, iterativeness raises two issues that affect the parallelization: locality of reference and degree of parallelism. The locality of reference is important because the access patterns vary over time with each iteration. The degree of parallelism is a function of the data dependencies among iterations. As a consequence of these characteristics, scheduling of 3I jobs is quite a challenge, and determining optimal scheduling for them is a very complex task, since it must consider locality, input size, data dependences, and parallelization opportunities.

Generally speaking, parallel job schedulers have been designed to deal with CPU-intensive jobs [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. Some researchers have proposed strategies to deal with I/O-intensive and irregular jobs [18, 19, 4, 5, 20, 21, 22], but not with 3I jobs.

In this paper we investigate the scheduling of 3I jobs, in particular filter-labeled stream programs executed in Anthill, our run-time system [23]. Filter stream programs are structured as pipelines of filters that communicate using streams, where parallelism is achieved by the instantiation of multiple copies of any given filter [24]. The labeled stream abstraction extends the model by guaranteeing consistent addressing among filter instances by the use of labels associated with any data that traverses a stream. These programs are fundamentally asynchronous and implemented using an event-based paradigm. In the scope of this work, a job is the execution of a program with specific input parameters on specific data using a number of instances for each filter. The main issue is that each filter demands a different amount of CPU and I/O and, in order to be efficient, there must be a continuous and balanced data flow between filters. Our premise is that the balance of the data flow between filters may be achieved by scheduling the proper number of filter copies or instances. In this paper we propose, implement, and evaluate a parallel job scheduling strategy called AnthillSched, which determines the number of filter instances according to each filter's CPU and I/O demands and schedules them. We evaluate AnthillSched using logs derived from actual workloads submitted to the Tamanduá[1] system, which is a data mining service that executes data mining 3I jobs on Anthill.

This paper is organized as follows. We present the related work in Section 2. The following section introduces the Anthill programming environment, and Section 4 describes our proposed scheduling strategy. We then present the workload, metrics, experimental setup, results and the performance analysis of AnthillSched in the following sections. Finally, we present our conclusions and discuss some future work.

2 Related Work

While we are not aware of works on scheduling 3I jobs, other researchers have addressed the issue of scheduling parallel I/O-intensive jobs. Wiseman *et al.* presented Paired-Gang Scheduling, in which I/O-intensive and CPU-intensive

[1] Tamanduá means anteater in Portuguese.

jobs share the same time slots [21]. Thus, when an I/O-intensive job waits for an I/O request, the CPU-intensive job uses the CPU, increasing utilization. This approach indicates that processor sharing is a good mechanism to increase performance in mixed loads.

Another work shows three versions of an I/O-Aware Gang Scheduling (IOGS) strategy [22]. The first one, for each job, looks for the row in the Ousterhout Matrix (time-space matrix) with the least number of free slots where job file nodes are available, considering a serverless file system. This approach is not efficient for workloads with lower I/O intensity. The second version, called Adaptive-IOGS, uses IOGS, but also tries the traditional gang scheduling approach. It fails to deal with high I/O-intensive workloads. The last version, called Migration-Adaptive IOGS, includes the migration of jobs to their associated file nodes during execution. This strategy outperformed all the other ones.

A job scheduling strategy for data mining applications in a cluster/grid was proposed by Silva and Hruschka [20]. It groups independent tasks that use the same data to form a bigger job and schedules it to the same group of processors. Thus, the amount of transferred data is reduced and the jobs performance is increased.

Storage Affinity is a job scheduling strategy that exploits temporal and spatial data locality for bag-of-tasks jobs. It schedules jobs close to their data according to the storage affinity metric it defines (distance from data) and also uses task replication when necessary. It has presented better performance than XSufferage (*a priori* informed) and WQR (non-informed) [5].

Finally, a very closely related work is LPSched, a job scheduling strategy that deals with asynchronous data flow I/O-intensive jobs using linear programming [4]. It assumes that information about job behavior is available *a priori* and it dynamically monitors cluster/grid resources at run time. It maximizes the data flow between tasks and minimizes the number of processors used per job. AnthillSched differs from LPSched in many points: it supports labeled streams and iterative data flow communication; it uses a simple heuristic and does not use run-time monitors.

3 The Anthill Programming Environment

A previous implementation of the Filter-Stream programming model is DataCutter, a middleware that enables efficient application execution on distributed heterogeneous environments [3]. DataCutter allows the instantiation of several (transparent) copies of each filter at runtime so that the application can balance the different computation demands of different filters as well as achieve high performance. The stream abstraction maintains the illusion of point-to-point communication between filters, and when a given copy outputs data to the stream, the middleware takes care of delivering the data to one of the transparent copies on the other end. Broadcast is possible, but selecting a particular copy to receive the data is tricky, since DataCutter implements automatic destination selection mechanisms based on round-robin or demand-driven models.

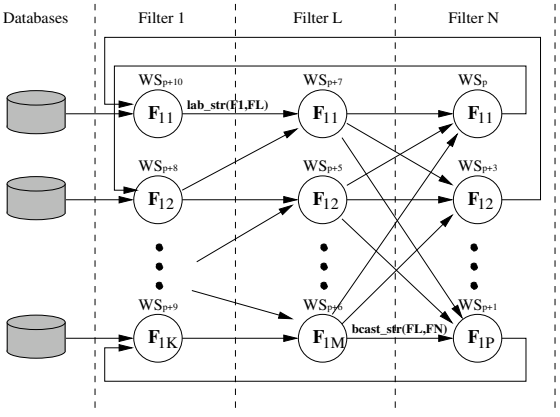


Fig. 1. Anthill programming model

We extend that programming model in the Anthill environment by providing a mechanism named labeled stream which allows the selection of a particular copy as destination based on some information related to the data (the labels). Such extension provides a richer programming environment, making it easier for transparent copies to partition global state [23]. Besides that, Anthill provides a task-oriented framework, in which the application execution can be modeled as a collection of tasks which represent iterations over the input data that may or may not be dependent on one another. In that way, Anthill explores parallelism in time and space, as well as it makes it easy to exploit asynchrony.

As we see in Figure 1, a job in Anthill explores time parallelism like a pipeline, since it is composed of N filters (processing phases or stages) connected by streams (communication channels). This job model explicitly forces the programmer to divide the job in well defined phases (filters), in which input data is transformed by each filter into another data domain that is required by the next filter.

The Anthill programming model also explores spatial parallelism, as each filter can have multiple copies, or instances, executing in different compute nodes. Each communication channel between filters can be defined as point-to-point, to direct each piece of data to a specific filter copy (either round-robin or by defining a labeled stream) or broadcast, where data is copied to all filter copies of the filter in next level. A consequence of spatial parallelism is data parallelism, because a dataset is automatically partitioned among filter copies. Together with streams, data parallelism provides an efficient mechanism to divide the I/O demands among filters, while labeling allows data delivery to remain consistent when necessary.

The task-oriented interface is what allows Anthill to efficiently exploit the asynchrony of the application. Each job is seen as a set of work slices (WS) to be executed which may represent iterations of an algorithm and may be created dynamically as input data is being processed (that is particularly useful for data-dependent, iterative applications). When a work slice WS_i is created,

its data dependencies to any previous slice WS_j are explicitly indicated. That gives Anthill information about all synchronization that is really required by the application, allowing it to exploit all asynchrony in slices that already had their dependencies met.

4 Anthill Scheduler

It should be noted that Anthill's programming model deals with only qualitative aspects of 3I jobs. As we presented, Anthill allows asynchrony, iterativeness, spatial and data parallelism, but it does not deal with quantitative aspects such as the number of filter copies, number of transmitted bytes during an iteration etc. Thus, to deal with quantitative aspects, we need a job scheduling strategy that can determine the number of filter copies considering filter execution times, filter I/O demands, data complexity, etc. It is important to notice that the overall application performance is highly dependent on such scheduling decisions.

To address that problem we propose AnthillSched, a parallel job scheduling strategy, implemented as Anthill's job scheduler. It focuses on the proper scheduling of a 3I job on a cluster, that is, the decision about the number of copies of each filter, based on the job input parameters. These parameters are specific for each algorithm being executed in a job; for a clustering algorithm, for example, it might be the number of clusters to be considered. Based on those parameters, AnthillSched must output the number of instances for each filter in the algorithm. There are two possible alternatives to implement that: use analytical modeling, which is very complex and can be infeasible to our problem, or a simpler solution, which is what we chose in the work.

Our approach is based on a simple experimental heuristic to solve a very complex problem in an efficient, although possibly not optimal, way. Our decision to use a heuristic was based on the fact that the 3I applications in which we are interested have very complex interactions, Since the processing and the applications themselves are iterative (users may run a same algorithm multiple times with different input parameters, trying to get a better result for their particular problems). Going for a full analytical model would most often be a very complex task. Although we may not be able to get to an optimal solution, with the simpler scheduling strategy, however, we still expect to eliminate possible bottlenecks and provide a continuous data flow with high asynchrony to 3I jobs.

Given a program that must be scheduled, the domain of its input parameters must be first identified and clearly mapped. AnthillSched requires q controlled executions, one for each possible permutation of the input parameters. For example, if we have input parameters A and B, and each parameter can assume 10 different values, we have 100 possible permutations. A controlled execution is the execution of a job with one copy of each filter (sequential pipeline) with certain combination of input parameters (say, combination i). For each job execution, we collect the number of input bytes β_{ij} and the execution time E_{ij} for each

filter j. During the execution of AnthillSched we use B_{ij} to represent the bytes actually received by a filter, since that may change during the computation. At first, $B_{ij} = \beta_{ij}$.

In Anthill, each job is executed according to a FCFS strategy with exclusive access to all processors in the cluster. When a new job arrives, Anthill executes AnthillSched with the job's set of input parameters (i) and the number of available processors (p) as input. The scheduler outputs the number of filter copies C_{ij} for each filter j (represented as a whole as C_i), after m iterations. First, for each iteration, the number of copies of each filter C_{ij} is calculated according to Fig. 2, where n is the number of filters in the pipeline for that application. In the first step, we normalize the number of input bytes B_{ij} and the execution time E_{ij} dividing them by the total sum of bytes and execution times, respectively. Then, we sum the normalized values and divide it by two, in order to obtain the relative resource requirements of each filter. For example, if we had a job with 3 filters, we might find that filter1, filter2 and filter3, respectively, utilize 0.6, 0.2 and 0.2 of the total of resources to execute the job. Finally, according to the number of available processors p, we calculate the number of copies C_{ij} proportionally to the relative requirements of each filter.

function AnthillSched (i, p : integer) : array of integer
  for 1 **to** m **do**
    for j = 1 **to** n **do**
      $C_{ij} = p \times \frac{\left(\frac{B_{ij}}{\sum_{k=1}^{n} B_{ik}} + \frac{E_{ij}}{\sum_{k=1}^{n} E_{ik}}\right)}{2}$
    endfor;

    for j = 1 **to** n **do**
      $q = (j+1) mod(n)$
      if $(broadcast(S_{ij}, S_{iq}))$
        $B_{iq} = \beta_{iq} \times C_{iq}$
    endfor;
  endfor;
  return C_i
end;

Fig. 2. AnthillSched's algorithm

The second step in Fig. 2 handles broadcast operations, since when a broadcast occurs between two filters, the number of input bytes of the destination filter will increase according to it's number of copies. For every filter j, we must consider its stream to the next filter q (where $q = (j+1) mod(n)$); that stream is identified as S_{jq}. If S_{jq} is a broadcast stream, the number of input bytes received by the destination filter during the controlled execution β_{iq} (which had a single copy of each filter) must be multiplied by the number of copies C_{iq}. Thus, AnthillSched must recalculate the number of input bytes B_{iq} for that filter and iterate again.

If we have a large number of possible input permutations, it is not feasible to run all controlled executions and store them. A solution in this case is to consider only a sampling of the space of possible permutations. When a new, or not yet considered, combination of input parameters of a job is found, an interpolation between the two nearest combinations can approximate the number of copies for each filter for that job.

For each new submitted job, Anthill calls AnthillSched with the job's permutation of input parameters. The scheduling process overhead is negligible, because the heuristic is very simple and can be solved in polynomial time as we see in Figure 2, since it defines a limit for the iterations, m.

During preliminary tests we verified that controlled executions that spent less than 5 seconds did not need to be parallelized. This threshold can vary according to the jobs and input data, but as a general rule, short sequential jobs do not need to be parallelized to improve performance. Thus, we created an optimized version of AnthillSched that determines if a certain job must execute in parallel (more than one copy per filter). Otherwise, it executes a sequential version of the job. We named this version Optimized AnthillSched (OAS).

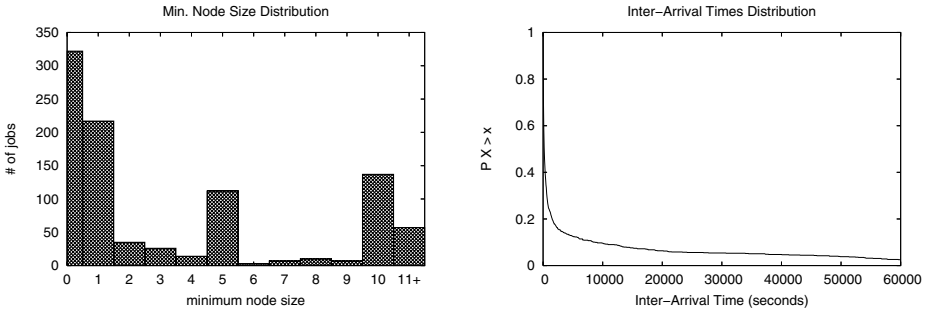
5 Results

In this section we evaluate our scheduling strategy by applying it to a data mining application: the ID3 algorithm for building decision trees. In particular, we want to investigate whether the number of filter copies C_i for a 3I job depends equally on the number of input bytes B_{ij} and execution time E_{ij} of each filter j. Thus, if the number of each filter's copies C_{ij} is uniformly distributed according to B_{ij} and E_{ij}, we eliminate possible bottlenecks and provide a continuous data flow with high asynchrony for a job.

To test and analyze our hypothesis, we compared two versions of AnthillSched (non-optimized and optimized) to other two job scheduling strategies: *Balanced Strategy* (BS) and *All-in-all Strategy*(AS). The proposed strategies use the maximum number of processors available. The BS tries to balance the number of processors assigned to each filter, assumint that all filters have equal loads. For example, if we have a job with 3 filters and a cluster of 15 processors, each filter will have 5 copies. In AS, every filter has one copy on every processor, executing concurrently.

5.1 Experimental Setup

For the workload we used real logs of data mining jobs executed in the Tamanduá platform by its users. As previously mentioned, Tamanduá is a scalable, service-oriented data mining platform executing on different clusters that uses efficient algorithms with large databases. The logs used are from clusters were Tamanduá is being used to mine government databases (one on public expenditures, another on public safety — 911 calls). Currently there are various data mining algorithms implemented in Anthill, such as *A priori*, K-Means, etc. In our experiments, we



(a) Minimum node size histogram

(b) Inter-arrival time inverse cumulative distribution

Fig. 3. Workload characterization

chose to focus on ID3 (a decision tree algorithm for classification) [1]. The main input parameter that influences ID3 is the minimum node size, which defines the minimum number of homogeneous points needed to create a node in the decision tree.

Based on real logs from Tamanduá, we characterized the inter-arrival time between jobs as shown in Fig. 3(b) and the minimum node size used as a parameter for each job in Fig. 3(a). As we see in Fig. 3(a), the majority of minimum node size values are concentrated between 0 and 10. In ID3, as an approximation, the minimum node size may be considered inversely proportional to the execution time, so it means that long-running jobs are predominant over short jobs.

Based on the characterization of Tamanduá logs, we created a workload model that uses the inter-arrival time pattern between jobs and the minimum node size pattern. We verified that the inter-arrival time (Fig. 3(b)) fits an exponential distribution with parameter $\lambda = 0.00015352$, with chi-square test value equal to 0. The minimum node size fits a Pareto distribution with parameters $\theta = 0.61815$ and $a = 0.00075019$, where θ is the continuous shape parameter and a is the continuous scale parameter (Fig. 3(a)). Using a workload generator, we generated 10 workloads, each one composed of 1000 jobs executing the ID3 algorithm with minimum node size and submission time derived from the distribution in Figure 3.

To test the scheduling strategies under different conditions, we varied the load (job arrival rate) between light, medium and heady. The light load considered the inter-arrival time between jobs based on all points shown in 3(b), so it has long periods of inactivity and a few peak periods, in which the inter-arrival time between jobs is small. To create the medium load workload, we used only a subset of the inter-arrival times from Fig 3(b) with the peak periods. Finally, the heavy load assumes that all jobs of the workload arrive at same time; in this case we just ignore the inter-arrival time between jobs.

To evaluate our proposal, we used 5 performance metrics: workload execution time (Eq. 1), workload idle time (Eq. 2), mean job response time (Eq. 4), mean job wait time (Eq. 3) and mean job slowdown (Eq. 5). As the parallel computer,

we used a Linux cluster composed of 16 nodes with 3.0 GHz Pentium 4 processors, 1 GB main memories and 120 GB secondary memories each, interconnected by a Fast Ethernet Switch.

$$WorkloadExecTime = \sum_{i=1}^{n} JobExecTime_i \quad (1)$$

$$WorkloadIdleTime = TotalTime - \sum_{i=1}^{n} JobExecTime_i \quad (2)$$

$$MeanJobWaitTime = \sum_{i=1}^{n} \frac{JobWaitTime_i}{NumberOfJobs} \quad (3)$$

$$MeanJobRespTime = \sum_{i=1}^{n} \frac{JobWaitTime_i + JobExecTime_i}{NumberOfJobs} \quad (4)$$

$$MeanJobSlowdown = \sum_{i=1}^{n} \frac{\frac{JobRespTime_i}{JobExecTime_i}}{NumberOfJobs} \quad (5)$$

5.2 Experimental Results

Using the workload derived from the previous characterization, we present some experimental results in order to evaluate the effectiveness of the scheduling strategies discussed. More specifically, we evaluate how well the scheduling strategies work when the system is submitted to varying workload and number of processors.

In order to evaluate the impact of the variability of the workload on the effectiveness of the strategies, we increased the load on each experiment to test which scheduling strategy presents a better performance to each situation and which strategies are impossible to use in practice. In the first three experiments (light, medium and heavy load), we used a cluster configuration composed of only 8 processors. With the heavy load, we saturated the system to test the alternatives. In our final experiment (scalability under heavy load), we compare the two best strategies with the same optimizations and analyze the scalability of the strategies for different cluster configurations (8, 12 and 16 processors). We used a 0.95 confidence level and approximate visual inspection to compare all alternatives. The confidence intervals are represented by c_1 (lower bound) and c_2 (upper bound).

This first experiment tests the scheduling strategies under light load for a cluster with 8 processors. As we see in Table 1(a), the mean execution time for all workloads and strategies was very close.

A light load implies large inter-arrival times; in this case, the intervals were often larger than the time necessary to execute a job. Thus, if a scheduling strategy spends more time executing jobs than another one, for a light load it does not matter. However, we observe in Table 1(b) that system using Optimized AnthillSched (OAS) spent more time idle than the other ones. This is a first indication that jobs executed with OAS strategy have a lower response time, as we confirm in Table 1(d). When a job spends less time executing, as the inter-arrival time is long, the system stays idle for more time, waiting for a new job submission, than a system in which a job spends more time executing. As can be seen on Table 1(c), the mean job wait time, and consequently the mean job response time for OAS, is really lower than the other strategies.

Table 1. Scheduling strategies performance for different workloads under light load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	2065488.41	2029272.33	2155312.68	36921.23	2042604.83	2088371.99
BS	2065463.77	2029279.84	2155222.41	36903.64	2042591.09	2088336.45
NOAS	2065464.68	2029245.12	2155317.20	36930.32	2042575.47	2088353.90
OAS	2065410.48	2029242.96	2155136.99	36896.76	2042542.06	2088278.89

(a) Execution time for each strategy under light load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	1189.08	61.86	3980.32	1309.36	377.54	2000.61
BS	1214.08	57.10	4070.59	1321.27	395.17	2033.00
NOAS	1209.70	90.26	3975.80	1304.82	400.98	2018.43
OAS	1263.55	90.19	4156.01	1328.39	440.22	2086.88

(b) Idle time for each strategy under light load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	9.61	7.07	11.82	1.49	8.69	10.53
BS	7.87	5.77	9.32	1.15	7.16	8.59
NOAS	8.13	6.01	10.24	1.33	7.30	8.95
OAS	4.84	3.64	5.89	067	4.43	5.25

(c) Mean job wait time for each strategy under light load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	189.28	186.76	191.46	1.50	188.35	190.21
BS	162.16	158.03	164.50	2.04	160.89	163.42
NOAS	168.28	166.28	170.67	1.36	167.44	169.44
OAS	107.82	101.68	110.65	2.63	106.45	109.19

(d) Mean job response time for each strategy under light load

As our first conclusions, this experiment shows that for a light load, independent of scheduling strategy, the inter-arrival time between jobs prevails over the workload execution time, because jobs are shorter than that. As we expected, a scheduling strategy that reduces the mean job wait time and consequently the response time, increases the idle time of the system.

Table 2 shows the results for the moderate load profile.

With medium load, the inter-arrival times are not always larger than response times. In Table 2(a), AS presented the worst execution time for all workloads. After that, BS and Non Optimized AnthillSched (NOAS) presented similar performance, with a little advantage for BS. Table 2(b) shows that on average, OAS achieved the higher idle time. As we confirm in Table 2(c,d), the mean job wait and response time are lower when the OAS is used, so the system have more idle time waiting for another job arrival.

Table 2. Scheduling strategies performance for different workloads under medium load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	179681.76	179401.58	179807.61	106.49	179615.75	179747.77
BS	154321.87	150995.24	155923.21	1394.01	153457.87	155185.87
NOAS	160181.04	159284.49	161230.22	548.70	159840.95	160521.12
OAS	103100.88	97447.90	106026.05	2413.90	101604.76	104597.01

(a) Execution time for each strategy under a medium load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	11.40	0.00	86.56	27.22	-5.48	28.27
BS	10.42	0.00	84.74	26.52	-6.02	26.86
NOAS	21.01	0.00	120.53	39.66	-3.57	45.60
OAS	32.59	0.00	119.77	48.37	2.61	62.57

(b) Idle time for each strategy under medium load

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	56660.14	53618.32	58737.13	1442.26	55766.23	57554.05
BS	44165.69	41342.79	46325.72	1612.33	43166.37	45165.00
NOAS	46849.72	43929.01	49249.10	1449.23	45951.49	47747.95
OAS	18721.51	15228.08	21785.22	2032.29	17461.90	19981.11

(c) Mean job wait time for each strategy under medium load

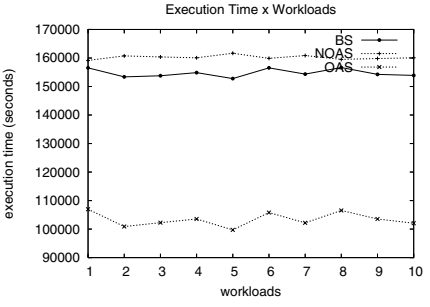
Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	56839.81	53798.01	58916.78	1442.28	55945.88	57733.73
BS	44319.97	41497.70	46481.64	1613.01	43320.23	45319.71
NOAS	47009.87	44089.37	49409.53	1449.21	46111.66	47908.09
OAS	18824.49	15325.52	21891.15	2033.92	17563.88	20085.10

(d) Mean job response time for each strategy under medium load

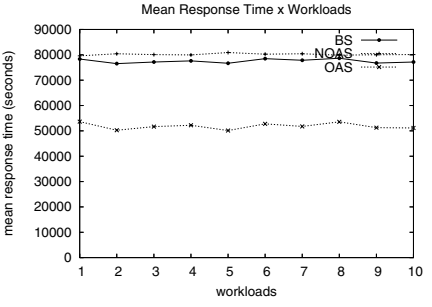
With this experiment, we observe that the All-in-all Strategy (AS) is not a viable strategy based to all evaluated metrics. In our context, we cannot assume that all filters are complementary (CPU-bound and I/O-bound), as AS does. However, for other type of jobs or maybe a subgroup of filters, resource sharing can be a good alternative [21]. Moreover, the Balanced Strategy (BS) and Non-Optimized AnthillSched (NOAS) presented similar performance, so we cannot discard both alternatives.

Based on our previous experiment, we do not consider AS an alternative from this point on. In the third experiment, we evaluate the scheduling strategies under heavy load. In this case we do not consider the system idle time, given that all jobs are submitted at the same time. The results are shown in Fig. 4.

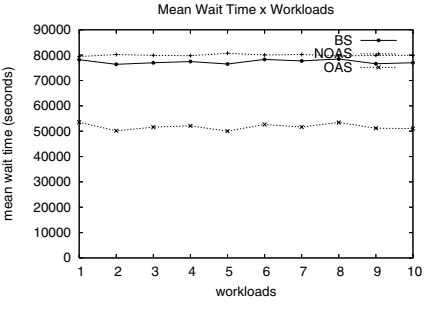
In this case, for all metrics, OAS was the best strategy, and NOAS the worst. NOAS and BS parallelize short jobs, creating unnecessary overhead and reducing performance. Not only that, but NOAS scheduling decisions are slightly worse



(a) Execution time per workload



(b) Mean job response time per workload



(c) Mean job wait time per workload

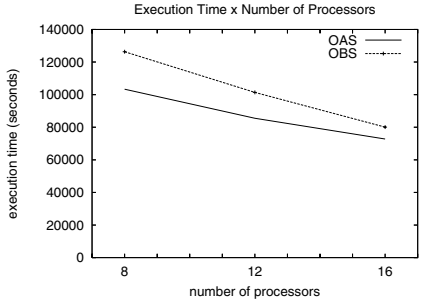
Fig. 4. Scheduling strategies performance for 10 different workloads

than BS. That happens because in our data mining jobs, the first filter tends to have much more work than the others, so NOAS assigns more processors to it and fewer processors to the other ones. However, the first filter reads data from secondary storage; its copies are only effective if they can be placed where data is stored. If there are more copies of it than nodes with input data, some copies will be idle, while other filters, that got fewer copies, become bottlenecks.

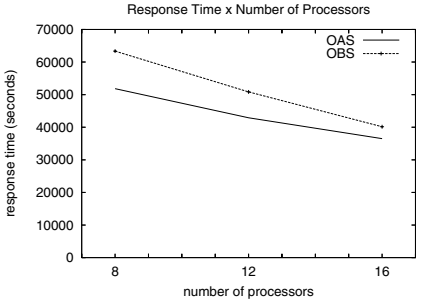
Based on the confidence intervals, our results show that NOAS is not a viable alternative, because for short jobs, the parallelization of jobs leads to a high response time, as we see in Fig. 4(b). Moreover, BS presented a lower performance than OAS. Despite of the low performance achieved with BS, we are not convinced yet that OAS is really better than BS. Because there is a considerable amount of short jobs in the workloads, the optimization in AnthillSched may be what gives it an advantage over BS. To check that, we included the same optimization in BS for the next experiment.

Our final experiment verifies whether OAS has a better performance than OBS (Optimized Balanced Scheduling) and if it scales up from 8 to 16 processors. We used the heavy load configuration and varied the number of processors from 8, 12 to 16 while considering four performance metrics (execution, response, wait and slowdown times, mean values for all workloads).

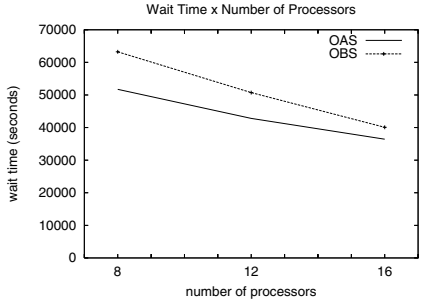
According to Fig. 5 and a visual inspection of the confidence intervals, all metrics show that even with the optimized version of BS, OAS has better per-



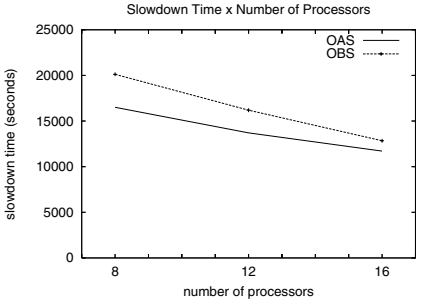
(a) Execution time per workload



(b) Mean job response time per workload



(c) Mean job wait time per workload



(d) Mean job slowdown per workload

Fig. 5. OBS and OAS performance for all workloads in a cluster with 8, 12 and 16 processors

formance. In Fig. 5(d), the large slowdown is due to the short jobs, which have low execution times, but high job wait times (Fig. 5(c)) under the heavy load.

Finally, this last experiment showed that OAS is more efficient than OBS. Moreover, OAS scaled up from 8 to 16 processors. Due to our limitations on computing resources, we could not vary the number of processors beyond 16. From the results, our main hypothesis that the number of filter copies C_i for 3I jobs depends equally on the number of input bytes B_i and the execution time E_i of each filter was verified. In preliminary tests, not shown in this paper, we observed that the use of different weights for CPU and I/O requirements in the AnthillSched algorithm (Fig. 2) did not seem to be a good alternative as the execution time of the controlled execution increased. However, as future work, these experiments can be more explored before we definitely discard this alternative.

6 Conclusion

In this work we have proposed, implemented (in a real system) and analyzed the performance of AnthillSched. Irregular and Iterative I/O-intensive jobs have some features that are not taken into account by parallel job schedulers. To deal with those features, we proposed a scheduling strategy based on simple heuristics that performs well.

Our experiments show that sharing all resources among all filters is not a viable alternative. They also show that a balanced distribution of filter copies among processors is not the best alternative either. Finally, we concluded that the use of a scheduling strategy which considers jobs input parameters and distributes the filter copies according to each job's CPU and I/O requirements is a good alternative. We named this strategy AnthillSched. It creates a continuous data flow among filters, avoiding bottlenecks and taking iterativeness into account. Our experiments show that AnthillSched is also a scalable alternative.

Our main contributions are the implementation of our proposed parallel job scheduling strategy in a real system and a performance analysis of AnthillSched, which discarded some other alternative solutions.

As future works we see, among others: the creation and validation of a mathematical model to evaluate the performance of parallel 3I jobs, the exploration of different weights for CPU and I/O requirements in AnthillSched, and its use with other applications.

References

1. Utgoff, P., Brodley, C.: An incremental method for finding multivariate splits for decision trees. In: Proceedings of the Seventh International Conference on Machine Learning, Morgan Kaufman (1990)
2. Veloso, A., Meira, W., Ferreira, R., Guedes, D., Parthasarathy, S.: Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In: Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery. (2004)
3. Beynon, C.M., Ferreira, R., Kurc, T., Sussmany, A., Saltz, J.: Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In: Proceedings of the IEEE Mass Storage Systems Symposium. (2000)
4. Nascimento, L.T., Ferreira, R.: LPSched — dataflow application scheduling in grids. Master's thesis, Federal University of Minas Gerais (2004) (in Portuguese).
5. Neto, E.S., Cirne, W., Brasileiro, F., Lima, A.: Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (2004)
6. Beaumont, O., Boudet, V., Robert, Y.: A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In: Proceedings of the IEEE Heterogeneous Computing Workshop. (2002)
7. Chapin, S.J., et al.: Benchmarks and standards for the evaluation of parallel job schedulers. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (1999) 67–90
8. Feitelson, D., Nitzberg, B.: Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (1995) 337–360
9. Feitelson, D., Rudolph, L.: Evaluation of design choices for gang scheduling using distributed hierarchical control. Journal of Parallel and Distributed Computing (1996) 18–34
10. Feitelson, D.: A survey of scheduling in multiprogrammed parallel systems research. Technical Report Report RC 19790, IBM T. J. Watson Research Center (1997)

11. Franke, H., Jann, J., Moreira, J., Pattnaik, P., Jette, M.: An evaluation of parallel job scheduling for ASCI Blue-Pacific. In: Proceedings of the ACM/IEEE Conference on Supercomputing. (1999)
12. Frachtenberg, E., Feitelson, D., Petrini, F., Fernandez, J.: Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium. (2003)
13. Góes, L.F.W., Martins, C.A.P.S.: Proposal and development of a reconfigurable parallel job scheduling algorithm. Master's thesis, Pontific Catholic University of Minas Gerais (2004) (in Portuguese).
14. Góes, L.F.W., Martins, C.A.P.S.: Reconfigurable gang scheduling algorithm. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (2004)
15. Streit, A.: A self-tuning job scheduler family with dynamic policy switching. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (2002) 1–23
16. Zhang, Y., Franke, H., Moreira, E.J., Sivasubramaniam, A.: Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (2000)
17. Zhou, B.B., Brent, R.P.: Gang scheduling with a queue for large jobs. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (2001)
18. Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P.: Ourgrid: An approach to easily assemble grids with equitable resource sharing. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (2003)
19. Batat, A., Feitelson, D.: Gang scheduling with memory considerations. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (2000) 109–114
20. Silva, F.A.B., Hruschka, S.C.E.R.: A scheduling algorithm for running bag-of-tasks data mining applications on the grid. In: Proceedings of the EuroPar. (2004)
21. Wiseman, Y., Feitelson, D.: Paired gang scheduling. IEEE Transactions Parallel and Distributed Systems (2003) 581–592
22. Zhang, Y., Yang, A., Sivasubramaniam, A., Moreira, E.J.: Gang scheduling extensions for I/O intensive workloads. In: Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop. (2003)
23. Fonseca, R., Meira, W., Guedes, D., Drummond, L.: Anthill: A scalable run-time environment for data mining applications. In: Proceedings of the 17th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), SBC (2005)
24. Acharya, A., Uysal, M., Saltz, J.: Active disks: Programming model, algorithms and evaluation. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII). (1998) 81–91

An Extended Evaluation of Two-Phase Scheduling Methods for Animation Rendering

Yunhong Zhou, Terence Kelly, Janet Wiener, and Eric Anderson

Hewlett-Packard Laboratories,
1501 Page Mill Rd,
Palo Alto, CA 94304
{yunhong.zhou, terence.p.kelly,
janet.wiener, eric.anderson4}@hp.com

Abstract. Recently HP Labs engaged in a joint project with DreamWorks Animation to develop a Utility Rendering Service that was used to render part of the computer-animated feature film *Shrek 2*. In a companion paper [2] we formalized the problem of scheduling animation rendering jobs and demonstrated that the general problem is computationally intractable, as are severely restricted special cases. We presented a novel and efficient two-phase scheduling method and evaluated it both theoretically and via simulation using large and detailed traces collected in DreamWorks Animation's production environment.

In this paper we describe the overall experience of the joint project and greatly expand our empirical evaluations of job scheduling strategies for improving scheduling performance. Our new results include a workload characterization of *Shrek 2* animation rendering jobs. We furthermore present parameter sensitivity analyses based on simulations using randomly generated synthetic workloads. Whereas our previous theoretical results imply that worst-case performance can be far from optimal for certain workloads, our current empirical results demonstrate that our scheduling method achieves performance quite close to optimal for both real and synthetic workloads. We furthermore offer advice for tuning a parameter associated with our method. Finally, we report a surprising performance anomaly involving a workload parameter that our previous theoretical analysis identified as crucial to performance. Our results also shed light on performance tradeoffs surrounding task parallelization.

1 Introduction

The problem of scheduling computational jobs onto processors arises in numerous scientifically and commercially important contexts. In this paper we focus on an interesting subclass with three special properties: jobs consist of nonpreemptible tasks; tasks must execute in stages; and jobs yield completion rewards if they finish by a deadline. In previous work we have formalized this problem as the *disconnected staged scheduling problem* (DSSP), described its computational complexity, proposed a novel scheduling method, and evaluated our solution

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 123–145, 2005.

theoretically and empirically using traces from an animation rendering application [2]. The companion paper also mentions a range of practical domains other than animation rendering in which DSSP arises.

This paper extends our previous work three ways. First, we provide a detailed description of the joint project between our company and DreamWorks Animation that first brought DSSP to our attention. We believe that this case study is interesting in its own right because it shows how modern parallel processing technologies are applied to a commercially important problem substantially different from traditional scientific applications of parallel computing. The joint project rendered parts of the animated feature film *Shrek 2* in a 1,000-CPU cluster on HP premises and thus illustrates the intersection of parallel processing and "utility computing." Our description of the project furthermore places scheduling and other parallel computing technologies in a broader context of business considerations. Finally, the project shows how new requirements driven by business needs led to a new and interesting formal problem, which in turn created research opportunities at the intersection of theoretical Computer Science, CS systems, and Operations Research.

Second, we provide a thorough and detailed workload characterization of traces we collected in the aforementioned cluster during the rendering of *Shrek 2*. These are the traces used in our previous empirical work, and in some of the extended empirical work presented in this paper.

Our third and major technical contribution in this paper is to greatly extend our empirical results and explore via simulation several open questions raised by our previous theoretical analysis. Whereas our previous empirical results were based exclusively on traces collected in DreamWorks Animation's production environment, in this paper we supplement the production traces with randomly-generated synthetic workloads. The latter transcend the domain-specific peculiarities of the former and thus allow us to evaluate the generality and robustness of our solution. Our new empirical work addresses three main issues:

1. We have proven theoretically that our solution architecture yields near-optimal results if jobs' critical paths are short; worst-case performance can be arbitrarily poor, however, if critical paths are long. How pessimistic are these theoretical results? Do real or random workloads lead to worst-case performance?
2. In some domains, including animation rendering, it is possible to shorten the critical paths of jobs by parallelizing tasks. What are the benefits of such parallelization? Is parallelization always beneficial?
3. Our solution architecture contains an adjustable parameter. Can we offer generic, domain-independent guidance on how to tune it?

In addition to addressing these issues we also report interesting and unanticipated relationships between problem parameters and performance.

The remainder of this paper is structured as follows: Section 2 defines DSSP and summarizes our previous results. Section 3 characterizes the DreamWorks Animation rendering workload, and Section 4 presents our extended empirical

results. Section 5 describes the joint HP/DreamWorks Animation project that motivated our interest in the DSSP, Section 6 reviews related work, and Section 7 concludes.

2 Background and Previous Work

In our previous work on DSSP [2] we formalized the abstract scheduling problem and described its computational complexity. We furthermore introduced a novel two-phase scheduling method better suited to the special features of DSSP than existing solutions. Finally, we evaluated our solution's performance through both theoretical analysis and trace-driven simulation using workload traces collected in a commercial production environment. This section reviews our previous work and describes open questions that we address in the present paper.

2.1 Problem

Informally, in DSSP we are given a set of independent computational *jobs*. Each job is labeled with a *completion reward* that accrues if and only if the job finishes by a given global deadline. Jobs consist of nonpreemptible computational *tasks*, and a job completes when all of its tasks have completed. Tasks must be performed in *stages*: no task in a later stage may start until all tasks in earlier stages have finished. Tasks within a stage may execute in parallel, but need not do so. We are also given a set of *processors*; at most one task may occupy a processor at a time. Our goal is to place tasks on processors to maximize the aggregate completion reward of jobs that complete by the global deadline.

Formally, we are given J jobs, indexed $j \in 1 \ldots J$. Job j contains G_j stages, indexed $g \in 1 \ldots G_j$. The set of tasks in stage g of job j is denoted S_{gj}. Stages encode precedence constraints among tasks within a job: no task in stage $g+1$ may begin until all tasks in stage g have completed; no precedence constraints exist among tasks in different jobs. The execution time (or "length") of task i is denoted L_i. The total processing demand of job j, denoted $T_1(j)$, equals $\sum_{g=1}^{G_j} \sum_{i \in S_{gj}} L_i$. The *critical path length* (CPL) of a job is the amount of time that is required to complete the job on an unlimited number of processors; it is denoted $T_\infty(j) \equiv \sum_{g=1}^{G_j} \max_{i \in S_{gj}} \{L_i\}$. Figure 1 illustrates the stage and task structure of two jobs, and their critical path lengths.

There are P identical processors. At most one task may occupy a processor at a time, and tasks may not be preempted, stopped/re-started, or migrated after they are placed on processors. Let C_j denote the completion time of job j in a schedule and let R_j denote its completion reward. D is the global deadline for all the jobs. Our goal is to schedule tasks onto processors to maximize the aggregate reward $R_\Sigma \equiv \sum_{j=1}^{J} U_D(C_j)$, where $U_D(C_j) = R_j$ if $C_j \leq D$ and $U_D(C_j) = 0$ otherwise. Our objective function is sometimes called "weighted unit penalty" [4].

The computational complexity of DSSP is formidable, even for very restricted special cases. In [2] we show that general DSSP is not merely NP-hard but also

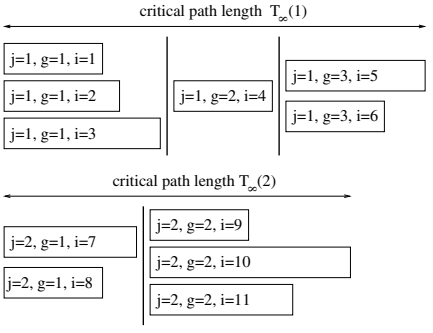


Fig. 1. Job (j), task (i), and stage (g) structure for two jobs

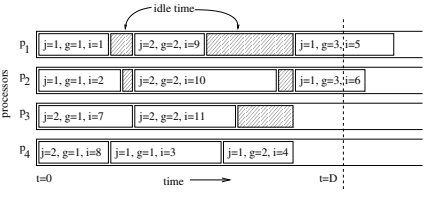


Fig. 2. A schedule for the jobs in Figure 1

NP-hard to approximate within any polynomial factor, assuming that P $\neq$ NP. Even for the special case where jobs have unit rewards and tasks have unit execution times, it is *strongly NP-complete* to solve DSSP optimally.

2.2 Two-Phase Scheduling Method

Before reviewing our approach to DSSP, we motivate the need for a specialized solution by considering shortcomings of existing alternatives. Scheduling in modern production environments almost always relies on priority schedulers such as the commercial LSF product [18] or an open-source counterpart like Condor [5]. Priority schedulers by themselves are inadequate to properly address DSSP. The fundamental problem is that ordinal priorities are semantically too weak to express completion rewards. Ordinal priorities can express, e.g., that "job A is more important than job B." However sums and ratios of priorities are not meaningful and therefore they cannot express, e.g., "jobs B and C together are 30% more valuable than A alone." When submitted workload exceeds available computational capacity, a priority scheduler cannot make principled decisions. In our example, it cannot know whether to run A alone or B and C together if those are the only combinations that can complete by the deadline. Priority schedulers make performance-critical *job selection* decisions as accidental by-products of *task dispatching* decisions.

Our solution, by contrast, decomposes DSSP into two conceptually simple and computationally feasible phases. First, an offline job selection phase chooses a reward-maximizing subset of jobs to execute such that it expects all of these jobs to complete by the deadline. An online task dispatching phase then places tasks from the selected jobs onto processors to complete as many as possible by the deadline. Completion rewards guide the first phase but not the second. Task dispatching can achieve better performance precisely because it can safely ignore completion rewards and consider only the *computational* properties of jobs but not their business value.

Job selection chooses a subset of jobs with maximal aggregate completion reward such that their total processing demand does not exceed available capacity.

Let binary decision variable $x_j = 1$ if job j is selected and $x_j = 0$ otherwise and let P denote the number of processors. Our selection problem is the following integer program:

$$\text{Maximize} \quad \sum_{j=1}^{J} x_j R_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{J} x_j T_1(j) \quad \leq \quad r \cdot PD \tag{2}$$

The summation in objective Equation 1 assumes that all selected jobs can be scheduled, regardless of their T_∞; jobs with $T_\infty(j) > D$ have $U_D(C_j) = 0$ and may be discarded before job selection. PD in the right-hand side of Equation 2 is the total amount of processor time available between $t = 0$ and $t = D$. By tuning selection parameter r we may select a set of jobs whose total processor demand is less than or greater than the capacity that is actually available. The final schedule after task dispatching typically achieves less than 100% utilization because precedence constraints force idleness as shown in Figure 2. Intuitively, r should therefore be set to slightly less than 1. Later we propose a way to compute good values of r.

The selection problem is a classic 0-1 knapsack problem, for which a wide range of solvers exist [11]. In this paper we solve the selection problem optimally using a simple knapsack algorithm, dynamic programming (DP) by profits. Our previous work explores a wider range of job selectors, including a sophisticated mixed-integer programming selector that can account for a wide range of side constraints [2].

Once a subset of jobs has been selected, a dispatcher places their tasks on processors. We employ a non-delay (or "work-conserving") dispatcher that places runnable jobs onto idle processors whenever one of each is available. The end result is a schedule that contains idle time due only to precedence constraints.

Given an idle processor and several runnable tasks, a *dispatcher policy* decides which task to run. In [2] we implemented and empirically evaluated over two dozen dispatcher policies. Our previous results show that our novel dispatcher policy LCPF outperforms a wide range of alternatives by several performance metrics. LCPF chooses a runnable task from the *job* whose critical path is longest. Intuitively, LCPF favors jobs at greatest risk of missing the deadline. To the best of our knowledge, LCPF represents the first attempt to tailor a dispatcher policy to the special features of DSSP, particularly its *disconnected* precedence constraint DAG. Our empirical results in this paper include two other dispatcher policies: STCPU chooses the runnable task whose parent job has the shortest total CPU time, and RANDOM chooses a runnable task at random. In the special case where each job contains exactly one task, LCPF coincides with the well-known *longest job first* policy, and STCPU reduces to *shortest job first*.

2.3 Previous Performance Evaluation

Our theoretical results on the computational complexity of DSSP show that this problem is hard to solve optimally, and it is hard even to approximate within a polynomial factor if job completion rewards and task execution times are unrestricted. However, we have also shown in [2] that in the unweighted case (i.e.,

uniform job completion rewards) a two-phase solution method using a greedy (possibly sub-optimal) selector and *any* non-delay dispatcher can achieve near-optimal performance if the maximum critical path length of any input job, denoted $T_\infty^{\max}$, is small relative to the global deadline D. Here we re-state this result:

Theorem 1. *The two-phase scheduling method with the selection parameter* $r = 1 - (1 - 1/P)(T_\infty^{\max}/D)$ *and any non-delay dispatcher completes at least* $(1 - \frac{T_\infty^{\max}}{D}(1 - \frac{1}{P}))OPT - 1$ *jobs before the deadline, where* OPT *is the maximum number of jobs that can be completed by any algorithm.*

Theorem 1 implies that any two-phase solution (with a proper selection parameter r) completes at least half as many jobs as an optimal algorithm if $T_\infty^{\max} \leq D/2$. As $T_\infty^{\max}/D$ goes to 0, its performance approaches that of the optimal algorithm.

The bound of Theorem 1 can be attained by two-phase algorithms that require remarkably little information about the computational demands of tasks. Specifically, it is necessary to know only the aggregate processing demand of *subsets of tasks* during selection. Knowledge of individual task execution times is *not* required.

Our previous work includes empirical evaluations of selectors, dispatchers, and combinations of the two. Briefly, we find that for the DreamWorks Animation production scheduling traces that we used:

1. Dispatcher policies differ dramatically in terms of job completion time distributions and other performance measures; our LCPF policy outperforms alternatives by several measures.
2. LCPF is relatively insensitive to a poorly-tuned selection parameter r.
3. An optimal selector with a well-tuned r and an LCPF dispatcher achieves 9%–32% higher aggregate value in the weighted case than a priority scheduler with no selection.

3 Workload Characterization

In this section, we describe a real production system where animation rendering jobs were run and then characterize the jobs and tasks in this workload.

3.1 Rendering Infrastructure

In early 2004, DreamWorks Animation began to supplement their in-house render farm with an extra cluster of 500 machines for production of the animated feature film *Shrek 2*. Each machine in this cluster is an HP ProLiant DL360 server with two 2.8-GHz Xeon processors, 4 GB of memory and two 36-GB 10k RPM SCSI disks. It contains a total of 1,000 CPUs and can serve up to 1,000 tasks simultaneously, because at most one task may occupy a processor at any time.

Our workload characterization is based on LSF [18] scheduler logs collected on this cluster during the period 15 February–10 April 2004. The logs associate tasks with their parent render jobs and we reconstructed their stage structure. We removed from consideration all jobs that did not complete successfully, e.g., because a user canceled them. Our final trace contains 56 nights, 2,388 jobs, and 280,011 tasks.

3.2 Jobs, Tasks, and Stages

Table 1 shows statistics about jobs, tasks, and stages in the eight week workload. While the number of tasks and jobs varied widely across all of the nights — there were a few weekend nights where very little rendering was done, as shown in Figure 3 — the medians shown in the fifth column represent the majority of nights. Most nights have a few tens of jobs and a few thousands of tasks. Task length varies widely; some tasks complete in less than a second while many tasks take hours. The median task length is 1.5 hours.

The middle section of Table 1 shows the percentage of jobs that were run twice during a single night. These jobs completed on the first run and produced the correct number of frames, but the frames were not satisfactory for some reason and the job was resubmitted. Note that in order for a job to run twice in one night, $2 \cdot T_\infty(j)$ must fit in the 13 hour time window.

Table 1. Task and job statistics

	Minimum	Maximum	Average	Std Dev	Median
Number of tasks/job	3	1328	117	114	84
Number of jobs/night	5	79	43	18	41
Number of tasks/night	880	8908	5000	1970	4976
Task length L_i	0	85686	8500	9899	5356
Maximum tasks in stage	1	700	93	93	91
Percent of jobs rerun/night	0%	26.4%	12.4%		
T_∞ in hours (all nights)	0.06	24.07	4.35	3.46	3.42
T_∞ in hours (average/night)	0.59	14.04	4.61	0.50	
T_1 in hours (all nights)	0.23	5026.36	276.77	420.08	134.72
T_1 in hours (average/night)	6.12	1849.48	301.22	396.49	

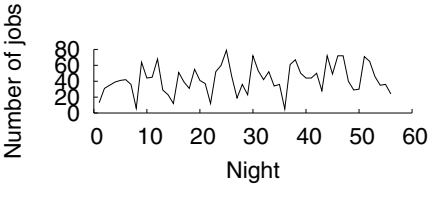


Fig. 3. Time series of the number of jobs each night. Low workloads correspond to weekends.

Table 2. Most stages of jobs have 1-3 tasks. Jobs have one or two stages with many tasks.

Number of stages per job with:	1 task	2 tasks	3 tasks	more than 3 tasks
Average (all jobs)	2.24	1.07	0.21	1.62

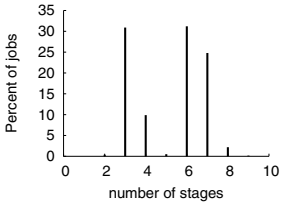


Fig. 4. Histogram of number of stages per job

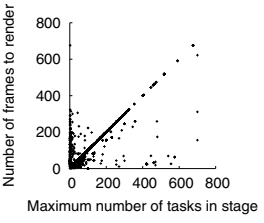


Fig. 5. The maximum number of tasks in a single stage is highly correlated with number of frames that the job renders

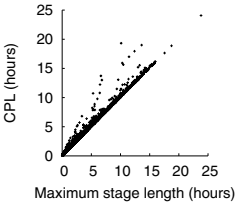


Fig. 6. The length of the longest single stage accounts for most of $T_\infty(j)$

The number of stages per job is shown in Figure 4. Most jobs have 3–7 stages, depending on whether all of physical simulation, model baking, and frame rendering need to be done (with some gluing stages in between). However, only 1 or 2 of the stages have more than 3 tasks, as shown in Table 2. Those stages usually compute something per frame (e.g., render the frame), which is why Figure 5 shows that the maximum number of tasks in a single stage is strongly correlated with the number of frames being rendered. For 82% of the jobs, the maximum number of tasks in a stage *equals* the number of frames. Figure 6 shows that the maximum length of that single stage is nearly equal to T_∞, i.e., in most cases a single stage accounts for most of a job's critical path length.

3.3 T_∞ and T_1 Distributions for Jobs

Figure 7 shows the relationship between T_∞ and T_1 for each job. T_1 is generally around 100 times as much as T_∞, because the longest stage has a median of 91 tasks. The median T_∞ of 3.4 hours shows that most jobs can complete in a short time, given enough resources. However, Figure 8 demonstrates that on most nights (75%), there is at least one job that cannot be completed within a 13 hour time window.

3.4 Predictions of T_∞ and T_1

The previous section analyzed the actual T_∞ and T_1 of jobs. In practice, the actual times are not known before the jobs execute. Scheduling decisions must be made based on predicted times. In this section, we compare the predictions that DreamWorks Animation artists supplied before a job was run with the actual run times.

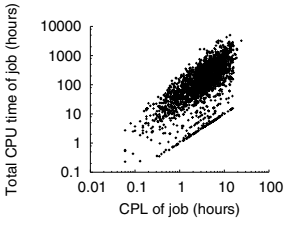


Fig. 7. Critical path lengths vs. total CPU times of jobs, logarithmic scales

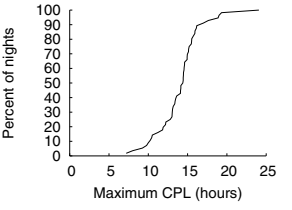


Fig. 8. CDF of $T_\infty^{\max}$ per night

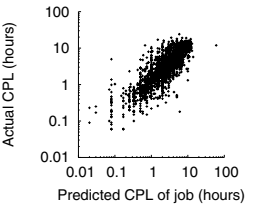


Fig. 9. Predicted and actual T_∞ appear correlated on log scales but often differ by 30–100%

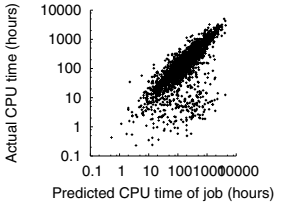


Fig. 10. Predicted and actual T_1 also appear correlated but often differ by a factor of 2–10

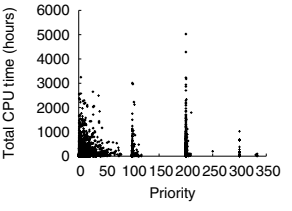


Fig. 11. Jobs' CPU demand are not correlated with priorities

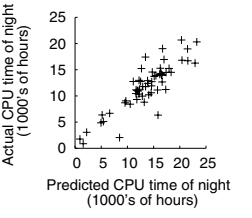


Fig. 12. Sums of predicted vs. actual CPU demand for all jobs in each night

Figure 9 shows a scatterplot of predicted versus actual T_∞ for jobs in our trace. Predictions are based on an estimate of the time required to render a single frame from the job. While Figure 9 suggests a strong correlation between the predicted and actual times, only 14% of the predictions are within 10% of the actual T_∞. Only 46% of the predictions are within 30% of the actual T_∞, and 21% of the predictions are wrong by a factor of 2 or more.

Figure 10 shows a similar scatterplot for T_1. These predictions are formed by multiplying the estimated time to render a single frame by the number of frames in the job, which leaves even more room for error, since not all frames take the same amount of time to render. Only 37% of the predictions are within 20%, 25% are wrong by a factor of 2 or more, and 7% are wrong by a factor of 10 or more.

However, while T_1 predictions are not very accurate for individual jobs (and we have no predictions of individual task execution times), our predictions are quite good when aggregated over all jobs for a given night. 32% of the predictions are within 10% and 80% are within 30% of the actual sums. Only 4% of the predictions are wrong by a factor of 2 or more. Figure 12 shows the sums of the predicted and actual T_1 for all jobs in each night, relative to each other. These sums are the *only* quantity for which we have good predictions. Fortunately, in job selection, we only need the sum of T_1 for a set of jobs rather than the individual jobs' requirements; that sum indicates whether the set can fit in the knapsack capacity of PD.

3.5 Job Priorities

Currently, DreamWorks Animation uses job priorities to decide which jobs to run first. Table 3 shows the priority categories that they use and the percentage of jobs assigned to each category. In Figure 11, we compare the CPU demand of jobs in different priority categories. Unsurprisingly, we find little correlation because job priorities are intended to reflect the relative importance of jobs, not their computational demands.

Table 3. Number and percent of jobs in each priority band

Priority	0-99	100-199	200-299	300-399
Meaning	Must do first	Must do tonight	Good to do	If there is time
Percent of jobs (all nights)	55.2%	8.1%	35.7%	1.0%
Number of jobs (all nights)	1318	193	853	24
Number of jobs (average/night)	24	4	15	0.4

4 Sensitivity Analysis

In this section, we evaluate how sensitive the performance of our scheduling method is to two variables: a parameter of our method and a property of our workload. We describe these variables first and then present the questions that the rest of this section tries to answer.

The scheduling method presented in Section 2.2 is a two-phase method. The first phase, job selection, uses a selection parameter r to decide the subset of jobs to run. As r approaches 1, the number of chosen jobs increases to fill all of the CPU time. The lower the value of r, the more idle time is allowed in the job schedule but the lower the possible reward if all jobs complete.

The workload itself contains many jobs, each of which has a critical path length T_∞, and the workload has a maximum CPL $T_\infty^{\max}$. Theorem 1 shows that our two-phase scheduling method achieves close-to-optimal performance if $T_\infty^{\max}/D$ is small and jobs have unit rewards. It leaves open the case where $T_\infty^{\max}/D$ is large, as it is in DreamWorks Animation's workload, and jobs have non-unit rewards. In addition, while Theorem 1 gives tight worst-case bounds for pathological examples, we wanted to explore average case performance. Our evaluation therefore aims to answer the following questions:

1. How should the selection parameter r be set? Does it depend on the dispatching policy used?
2. How sensitive is dispatcher performance to different maximum critical path lengths?
3. What happens to performance as $T_\infty^{\max} \approx D$? Is it as bad as the worst case given by Theorem 1?
4. Can we improve performance by breaking long tasks into small pieces (thereby shortening $T_\infty^{\max}$)?

In order to answer the latter three questions, we needed to generate workloads with varying values of $T_\infty^{\max}$. We therefore decided to generate synthetic workloads. We first describe our synthetic workload generation and then present our results.

4.1 Synthetic Workload Generation

In our standard synthetic workloads, $T_\infty^{\max}/D \approx 1$. We then transform these workloads to create new workloads with lower values of $T_\infty^{\max}$. We first describe how we generate a standard synthetic workload.

For all experiments in the next few subsections, the number of CPUs $P = 100$ and the global deadline $D = 13$ hours $= 4680 \times 10$ seconds. For job completion rewards, we use the size-dependent reward function $R_j = T_1(j)$. For each standard workload, we add jobs to the workload until the total CPU demand of the workload exceeds $2 \times P \times D$.

While the workload is not full, we create new jobs as follows: For each new job, we draw a random number of stages from $U[5, 10]$ where $U[a, b]$ denotes an integer drawn with uniform probability from the set $\{a, a + 1, \ldots, b\}$. For each stage, we draw a number of tasks from $U[1, 10]$. For each task, we draw a task length from $U[1, 600]$. After choosing task lengths for every task in a job, we then compute the job's critical path length T_∞. If $T_\infty \leq D$, we add the job to the workload; otherwise, we discard it.

To transform an already generated workload into a new one with a desired $T_\infty^{\max}$, we alter the number of stages that it has. A job with fewer stages will probably have a shorter T_∞, since it will have fewer dependencies between tasks. We therefore also generated workloads with a fixed number of stages. For each such workload, we first create a standard workload where the number of stages in each job is always 10, and the task lengths are drawn from $U[1, 600]$. We then alter the workload so that each job has a smaller (fixed) number of stages, say 6. For each job, we reassign all tasks that were previously in a stage > 6 to a stage drawn from $U[1, 6]$. The new workload therefore has the same number of jobs and the same total processing time as the original workload and, unlike the previous transformation, the same number of tasks. We call this transformation T1.

A different way to alter the T_∞ of a job is as follows: While the job's T_∞ exceeds the new $T_\infty^{\max}$, we find the task with the longest length and replace it with two tasks that are half as long (in the same stage as the original task). The CPU time of the job (and of each stage of each job) therefore remains constant, but the job now has more parallizable tasks. We call this transformation T2.

4.2 Scheduling Performance When $T_\infty^{\max} \approx D$

While Theorem 1 shows that the two-phase scheduling method achieves near-optimal performance if $(T_\infty^{\max}/D)$ is small and jobs have unit rewards, it does not apply to the more general case where $T_\infty^{\max}/D$ is large and jobs have non-unit completion rewards. Furthermore we do not know how pessimistic the bound of Theorem 1 is when critical paths are long: We know that the bound is tight

because we can construct pathological inputs that result in worst-case performance given by the theorem. However our theoretical results alone shed little light on whether such inputs are likely to arise in practice. How does the worst-case bound of Theorem 1 compare with average performance for workloads with $T_\infty^{\max} \approx D$?

In this section we answer this question by generating synthetic workloads with $T_\infty^{\max} \approx D$ and applying our two-phase scheduling method to them in simulation. We use DP by profits to solve the job selection problem optimally. Ideally, we would like to compare the performance achieved by our scheduler with the optimal solution value. However, as discussed in Section 2 and proven in [2], it is computationally infeasible to solve DSSP optimally. We therefore instead compute an *upper bound* equal to the aggregate value of jobs selected when selection parameter $r = 1$. This is the reward that would accrue if we select jobs to utilize available processor capacity as fully as possible, *and* all selected jobs complete by the deadline. It is clearly an upper bound for the optimal value of the overall scheduling problem (which may be lower if some jobs fail to complete on time). When evaluating our two-phase scheduler we use the term *performance ratio* to denote the ratio of its actual performance to the upper bound discussed above.

Section 4.1 describes how to generate synthetic workloads with $T_\infty^{\max}$ close to D. For each workload generated in this way, we run the two-phase scheduling method with varying values of selection parameter r during job selection. In this experiment we use three dispatcher policies during the task dispatching phase: LCPF, STCPU, and RANDOM, and we vary the selection parameter r from 0.7 to 1.0 in increments of 0.01. For each (r, dispatcher) pair we generate 20 different random workloads and report mean performance ratio. Figure 13 presents our results.

The figure shows that LCPF clearly dominates both STCPU and RANDOM for any fixed r value. This is not surprising because LCPF takes into account the critical paths of jobs while the other policies do not. For workloads with long critical paths, performance will suffer if the jobs with the longest critical paths are started too late to complete on time. Both STCPU and RANDOM ignore jobs' critical path lengths and therefore start many long jobs too late to complete by the deadline, while LCPF starts long jobs as early as possible.

Comparing STCPU with RANDOM, we see that STCPU is slightly better when $r \geq 0.92$ and RANDOM is slightly better when $r \leq 0.87$; their performance is similar when r is in the range [0.87, 0.92]. Note also that RANDOM becomes less effective when more jobs are selected whereas STCPU is relatively insensitive to the tuning of r. This is intuitive because RANDOM treats all *tasks* equally, and if too many *jobs* are selected it will spread available processor capacity among them roughly evenly, with the result that many fail to complete by the deadline. By contrast, STCPU imposes a near-total priority order on jobs, because it is rare for two jobs to have the same total processing demand. If r increases and more jobs are selected, some of the additional jobs have small processing demand and STCPU finishes them early during task dispatching. This has relatively little

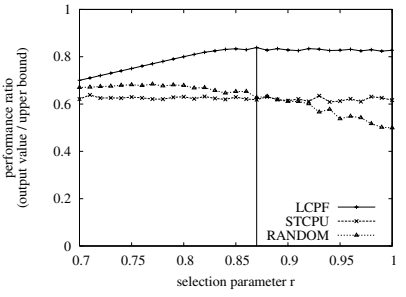


Fig. 13. Performance ratio vs. selection parameter r, for synthetic workloads with $T_\infty^{\max}/D \approx 1$

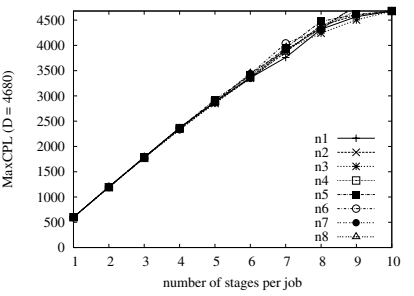


Fig. 14. FIGS / maxcpl . vs. s. eps MaxCPL vs. number of stages for each of eight synthetic workloads generated by transformation T1

impact on the remaining processing capacity available to larger jobs, and the overall result is that many jobs still finish by the deadline.

Figure 13 shows that the performance of LCPF reaches a maximum of roughly 0.84 when $r = 0.87$. In other words, LCPF achieves at least 84% of the optimal performance even though the maximum critical path lengths $T_\infty^{\max}$ in the workloads used are at least 97% of the deadline D. This result stands in stark contrast to the very weak performance bound that Theorem 1 would guarantee with similarly long critical paths: If $T_\infty^{\max}/D = 0.97$, our previous theoretical results state that a two-phase scheduler can achieve as little as 3% of optimal performance in the unit-reward case.

Finally, we observe in Figure 13 that LCPF's performance is relatively flat when r is in the range [0.85, 1]. When $r < 0.85$, all selected jobs finish by the deadline and selecting more jobs simply increases the number that finish. As the selection parameter increases beyond $r > 0.85$, performance does not improve because the additional jobs selected do not complete by the deadline; more processing capacity is used, but it is wasted on jobs that do not complete quickly enough to yield a reward. Because a job that fails to complete by the deadline simply wastes any capacity devoted to it, it might be best in practice to set r to a relatively low value rather than a higher value that achieves comparable reward (e.g., 0.85 as opposed to 1 in the figure). Our results suggest that for LCPF and the workload studied ($T_\infty^{\max} \approx D$) a value of r in the range [0.85, 9] is a good choice.

4.3 Performance vs. MaxCPL

In this section we empirically evaluate the performance of the two-phase scheduling method as the maximum critical path length $T_\infty^{\max}$ of the workload varies. Intuitively, jobs with relatively long critical paths face higher risk of finishing after deadline D and thus yielding no reward. Theorem 1 shows that the worst-case performance of two-phase scheduling for unit-reward DSSP is a strictly decreasing function of $(T_\infty^{\max}/D)$. However our theoretical worst-case bounds for

unit-reward DSSP do not necessarily predict *typical* performance in the weighted case. We therefore explore via simulation the relationship between maximum critical path length $T_\infty^{\max}$ and the performance ratio that our scheduling method achieves.

We describe two classes of simulation experiments to address this issue and compare their results qualitatively. Our first approach uses the workloads with a fixed number of stages and transform them into new workloads with fewer number of stages using transformation T1. While it directly alters only the number of stages per job, Figure 14 shows that the number of stages is closely related to $T_\infty^{\max}$ for the workload. $T_\infty^{\max}$ is almost a linear function of the number of stages for jobs in the synthetic workload.

Figures 15, 16, and 17 show the performance ratio of three dispatching policies as the number of stages per job varies. One feature of these figures is that RANDOM differs qualitatively from LCPF and STCPU: The latter two policies dominate RANDOM for most values of r and for most numbers of stages. Furthermore, for all r values except $r = 1.0$, the performance of RANDOM decreases slowly as the number of stages (and $T_\infty^{\max}$) increases, but suffers only slightly. Finally, the performance of RANDOM relies heavily on a well-tuned r. If r is close to 1 its performance degrades substantially. For the workload studied, r of 0.8 to 0.9 seems appropriate for RANDOM and yields far better performance than $r \approx 1$.

Figures 16 and 17 show that LCPF outperforms STCPU by a wide margin when the number of stages per job is large. Furthermore, contrary to our intuition that performance should decrease monotonically as the number of stages increases, we notice a local minimum of the performance ratios of both LCPF and STCPU at roughly 5 stages/job. Upon further investigation of Figure 14 we found that 5 stages corresponds to $T_\infty^{\max} \approx D/2$. We conjecture that LCPF and STCPU exhibit non-monotonicity while RANDOM does not for the following reason: Whereas RANDOM disperses processor capacity across tasks from *all* jobs, LCPF and STCPU take a different approach. They impose an order on jobs and try to finish some jobs (the ones with greatest CPL and smallest total CPU demand, respectively) before devoting any processing capacity to others. If $T_\infty^{\max} \approx 0.5 \times D$, STCPU and LCPF will process a set of jobs and finish them slightly after $t = D/2$. By the time they finish these jobs, there may not be time to start the remaining jobs and finish them by the deadline. The remaining jobs are started late and narrowly miss the deadline, thus contributing no value and wasting processing resources.

Our second set of experiments uses the workloads that altered $T_\infty^{\max}$ directly using transformation T2. We again present results for three dispatching policies: Figure 19 for LCPF, Figure 20 for STCPU, and Figure 18 for RANDOM.

The results for this set of experiments are similar to those for the first set: STCPU and LCPF resemble each other but not RANDOM; LCPF outperforms STCPU when MaxCPL is long; and RANDOM is more sensitive to selection parameter r than the other two. In both experiments, the results show minima for MaxCPL slightly larger than $D/2$. Furthermore these results shed additional light on the relationship between performance ratio and MaxCPL: Figures 19 and 20 show

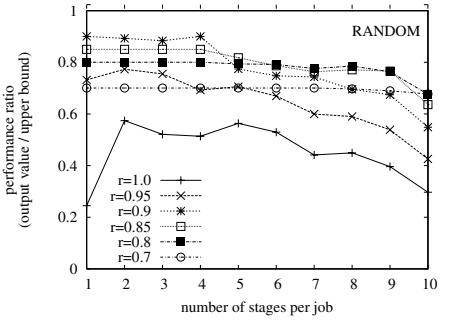


Fig. 15. Performance ratio vs. number of stages with dispatching policy RANDOM

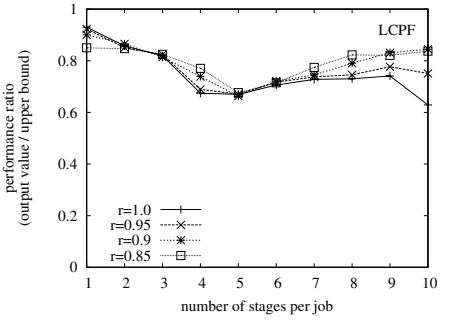


Fig. 16. Performance ratio vs. number of stages with dispatching policy LCPF

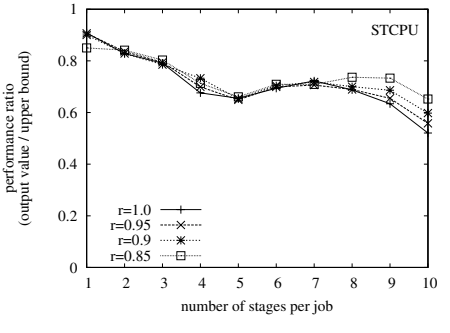


Fig. 17. Performance ratio vs number of stages with dispatching policy STCPU

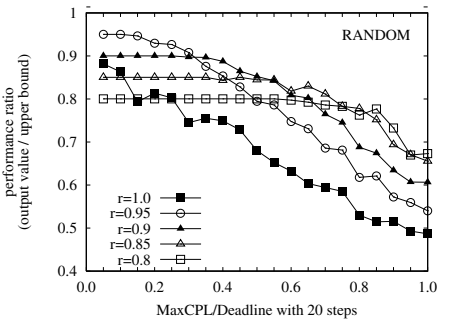


Fig. 18. Performance ratio vs MaxCPL with dispatching policy RANDOM

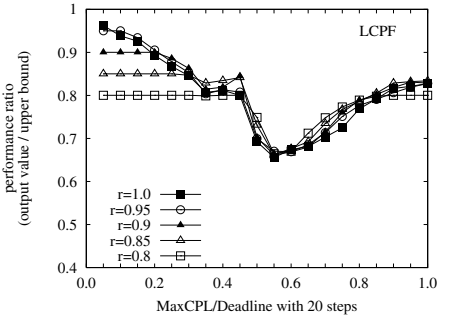


Fig. 19. Performance ratio vs. MaxCPL with dispatching policy LCPF

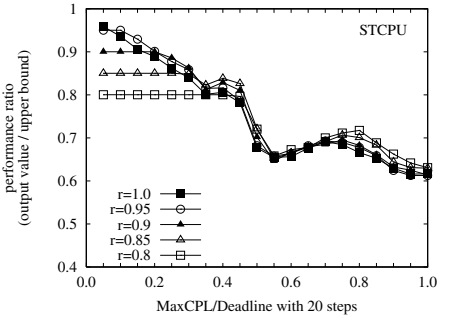


Fig. 20. Performance ratio vs. MaxCPL with dispatching policy STCPU

second local minima at around $T_\infty^{\max} = 0.35 \times D$, that is, at $T_\infty^{\max} \approx D/3$. It is possible for the performance of our scheduling method to have multiple local minima with respect to $T_\infty^{\max}$, and these local minima occur at approximately D/n from the right side, where $n = 2, 3, \ldots$ is an integer. We conjecture that LCPF and STCPU finish jobs in "rounds." If $T_\infty^{\max} \approx D/n$ these policies will finish the first $n-1$ rounds; jobs in the last round will narrowly miss the deadline, thereby wasting processor resources without contributing value.

4.4 Selection Parameter Tuning

DSSP is a deadline scheduling problem where a job's completion reward is zero if it completes after the global deadline D. If too many jobs are selected during the job selection phase, then during task dispatching these selected jobs will compete for limited processor capacity and each job has a higher risk of finishing too late. It is thus important to set the selection parameter r properly for each dispatching policy; in most cases it should be strictly less than 1. Let $r_0 \equiv 1-(1-1/P)(T_\infty^{\max}/D)$. Theorem 1 has proved that a general two-phase solution with $r = r_0$ completes at least $r_0 \cdot \mathrm{OPT} - 1$ jobs before the deadline, where OPT is the maximum number of jobs that can be completed by any scheduler. This seems to suggest a default selection parameter value. This section tries to find a reasonably good value for r for various dispatching policies.

We begin by briefly reviewing our simulation results from Section 4.3. Figure 15 shows that for dispatcher policy RANDOM, the *worst* strategy is to choose $r \geq 1$. Figure 15 suggests that the best choice of r is in the range $[0.85, 0.9]$. For any $r \in [0.85, 0.9]$, performance is strictly better than for $r = 1$.

For dispatching polices such as LCPF or STCPU, it is also true that r slightly less than 1 is better than $r = 1$. However the best selection parameter is no longer a fixed constant. Given two fixed selection parameters $r_1 < r_2$, we consider their corresponding performance ratio curves which we denote f_1, f_2 respectively. It is highly likely that when $T_\infty^{\max}$ is small, then $f_1 > f_2$. At some value ${T_\infty^{\max}}_0$ these two curves intersect. After that $T_\infty^{\max} > {T_\infty^{\max}}_0$, then the order is reversed $f_1 < f_2$.

Figures 19 and 20 suggest that the best value of r is correlated to the maximum critical path length of the workload. Inspired by the value of r_0 above, we suspect that it is possible to find a constant $\lambda \in [0, 1]$, such that $r = 1-\lambda(1-1/P)(T_\infty^{\max}/D)$ is a good choice for the selection parameter. Figure 21 shows the performance ratio of our algorithm using LCPF dispatching policy with three λ values: $\lambda = 0.0$, 0.2, and 1.0. $\lambda = 0.0$ corresponds to the selection parameter r= 1.0; $\lambda = 1.0$ corresponds to the selection parameter $r = r_0$; and $\lambda = 0.2$ corresponds to $r = 1 - 0.2(1-1/P)(T_\infty^{\max}/D)$.

Figure 21 shows that for dispatching policy LCPF, the curve corresponding to $\lambda = 0.2$ consistently performs better than the curve corresponding to $\lambda = 0.0$, i.e., $r = 1.0$. Furthermore, both $\lambda = 0.2$ and $\lambda = 0.0$ perform much better than $\lambda = 1.0$. This is because r_0 is too conservative; it is appropriate only for worst-case inputs. For the class of workloads studied and for our LCPF dispatcher policy, a good selection parameter is around $r = 1 - 0.2(1-1/P)(T_\infty^{\max}/D)$.

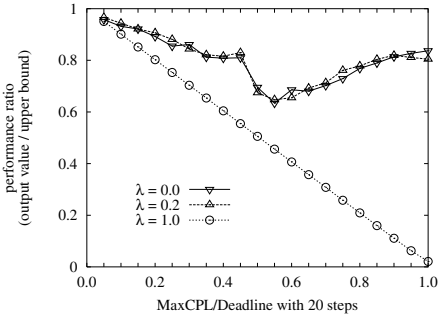


Fig. 21. Performance ratio vs. MaxCPL with LCPF and $r = 1 - \lambda(1 - 1/P)(T_\infty^{\max}/D)$, for three λ values

4.5 Tradeoff Between Parallelism and Performance

Parallelism is an important technique in scientific computing and parallel computing to improve system utilization and job throughput by breaking long sequential tasks into multiple parallel tasks. For a multi-processor system, if there is only one long-running task, then only one processor is fully occupied and all other processors are left idle. Breaking long tasks into multiple short tasks will definitely increase system utilization. Remarkably, however, for DSSP parallelization is not necessarily helpful even if it entails no overhead. In other words, if we have an opportunity to parallelize tasks in a particular instance of DSSP and thereby reduce critical path lengths, it is *not* always advantageous to do so.

Consider, for instance, the workloads with $T_\infty^{\max}/D \approx 0.8$ in Figures 19 and 20. If by parallelizing tasks we reduce $T_\infty^{\max}/D$ to roughly 0.55, performance will be *worse* for both LCPF and STCPU even though parallelization does not increase total processing demand. If LCPF is the task dispatching policy, then parallelism is beneficial only if the given workload has $T_\infty^{\max} < D/2$. Finally, if $T_\infty^{\max}/D$ is small, then the two-phase scheduling method produces a solution with near-optimal performance regardless of the dispatcher policy used; even if parallelism incurs no extra cost, it cannot yield large performance improvements.

5 DreamWorks Animation Engagement Experience

In August 2003, HP Labs embarked on a challenge to provide remote rendering services to DreamWorks Animation. In early February 2004, we went into full production for the movie *Shrek 2*. We learned that the utilification [22] process of bringing the customer's workload up on a remote facility was not straightforward, and had many unexpected challenges.

We went through four stages in utilification. First a feasibility stage where we determined whether or not a remote service was feasible. Second, an instantiation stage where we brought up the service. Third, a confidence-building stage where we demonstrated to the customer's satisfaction that the remote service could

correctly support their workload. Fourth, an ongoing maintenance stage where we optimize our delivery of the service, and keep the service up to date for the customer needs. In the following we describe these stages in detail.

5.1 Feasibility Stage

Our evaluation started in August 2003. We needed to determine whether we would be able to place a remote facility about 20 miles away from the primary site. We found that there were four feasibility questions:

1. Schedule - the proposed schedule said that we needed to be in production in only five months, by January 2004. Could we acquire all of the equipment and implement the system in that time?
2. Bandwidth and latency - will the 1 Gbit/s network connection between the sites be sufficient to support a cluster of 1,000 2.8 GHz CPUs?
3. Software and configuration - will we be able to install, configure, and adapt the existing software and configuration to work with a remote cluster?
4. Business Model - can the lawyers agree on an appropriate contract for the project?

While we expected (3) to provide the most difficulty, we in fact discovered that (4) presented the most substantial difficulties, in particular because the *Shrek 2* franchise had a large estimated monetary value, and DreamWorks Animation therefore had an understandable concern about exposing content outside of their company. Hence, we negotiated protections such as sanitization of gathered data for analysis, an isolated network to protect their content, a double-stage firewall to protect access from the HP network, and a camera to monitor the physical installation.

Performing the bandwidth and latency analysis was the hardest technical challenge we faced during the feasibility stage. The sustained and burst packet rates we needed to handle normally would require specialized network analyzer hardware, but we needed general purpose, full-packet capture for weeks of data. We developed a solution based on commodity hardware that used improved software for packet capture, buffering to local memory to handle bursts of data, parallel use of multiple disks spread across multiple trays, and opportunistic compression of data to increase the effective disk space and increase the time-periods for contiguous captures. The solution could handle traffic for hours at 30-50MB/s and bursts above 100MB/s with negligible drops on the tracing machine.

Our second challenge was to analyze the data. We have collected billions of NFS requests and replies, so putting this in a database or directly processing the raw traces would be either too slow or too expensive. Luckily we had previously developed a new, highly efficient trace storage format called "DataSeries" for handling block I/O traces and process traces. The format was general, and provided streaming access to database-like tables. We therefore developed a converter from the raw tcpdump traces into DataSeries, and built our analysis on

the converted files. This has provided us with a flexible, extensible data format and structure for our analysis. Multiple people have been able to add new analysis in to our existing structure within a few days.

We did not hit the original schedule because of the length of time it took to negotiate the legal agreements and the unexpected length of the confidence-building stage. Both of these parts were originally scheduled as taking at most a few weeks, and in fact both took months. Luckily, the need for the service was not excessively strong until early February 2004, so we were able to provide the service on an appropriate schedule, just one different than we expected.

5.2 Instantiation Stage

The primary difficulty that occurred during the instantiation stage was installing and configuring all 500 machines. When our racks of machines arrived, we discovered that some of them were configured both physically and logically wrong. The physical mistakes involved cabling errors, which were straightforward if tedious to repair. The logical mistakes were more difficult because they involved incorrect firmware versions. After a little study, we developed a tool for automatically updating the firmware and firmware configuration on all of our machines automatically. We wrote an extensive document about the problems with the order fulfillment process for rack systems which was presented back to the HP order fulfillment team. The key lesson was that many traditional tools that are used involve per-machine human effort. While those tools are acceptable if you have 1-10 machines, they become unusable at 500. We needed to automate many of these actions, and found moreover that it was important to write idempotent tools: any time we ran a task across 500 machines (even one as simple as removing a file), a few machines would fail to execute the task correctly. Our solution was to design our automation to take a machine to a particular state and report on changes it made, which meant that we could simply execute global tasks multiple times until we received a report of no changes.

5.3 Testing Stage

Once we had instantiated the rendering service, we then had to verify that it worked to the satisfaction of various people responsible for making the movie. Moreover, we wanted to perform these read-write tests with no risk to the production data. Our problem therefore was to clone an appropriate subset of the total 15-20 TB of data such that we could show that our cluster rendered frames correctly, and so that there would be minimal changes from testing to production.

While we solved this problem by using a DreamWorks Animation specific feature in their rendering system of having a few variables to change expected file locations, and setting the source file systems to read only, we found a better solution by having a write-redirector that snapshot the backend file systems to isolate us from underlying changes and store our writes in a second location. This solution enables us to test and verify our solution much faster. Then moving into production would merely have required removing the write redirector to send all of the accesses directly at the file systems.

5.4 Maintenance and Optimization

We moved into production in two stages, first on a movie that was not due to release until 2005, and then on *Shrek 2*. When we moved into production we identified some reporting and job submission issues that we had not addressed during the instantiation phase that we needed to solve. Once we entered into full production, our goal was really to optimize and maintain the cluster. We made many small, but cumulative improvements to our service: simple service-specific host monitoring to detect failed hosts, automatic job retry for failed jobs, farm usage analysis and reporting, tracing through render job executions, and NFS performance analysis. Specifically, we found an interesting aspect to explore scheduling improvements over this multi-processor rendering environment, and that motivated our current work.

6 Related Work

Scheduling is a basic research problem in both computer science and operations research. The space of problems is vast; [4, 17] provide good reviews. In this section we focus on non-preemptive multiprocessor scheduling without processor-sharing.

6.1 Minimizing Makespan or Mean Completion Time

Much scheduling research focuses on minimizing makespan for tasks with arbitrary precedence constraints. Variants of list scheduling heuristics and their associated dispatching policies are the main focus of both theoretical and empirical studies [8, 3, 9, 1]. See Kwok and Ahmad [14] for a recent survey of static scheduling algorithms and Sgall [19] for online scheduling algorithms.

Another important optimization metric for job scheduling research is to minimize mean task completion time. The classic *shortest job first* (SJF) heuristic is optimal in the offline case with no precedence constraints and each job consists of a single task; SJF works well in many online scheduling systems.

The large *queueing-theoretic* literature on processor scheduling typically assumes continuous online job arrivals and emphasizes mean response times and fairness, e.g., Wierman and Harchol-Balter [21]. Kumar and Shorey analyze mean response time for stochastic "fork-join" jobs, where fork-join jobs closely resemble the stage-structured jobs of DSSP [13]. Our work on DSSP differs because we are confronted with a fixed set of jobs rather than a continuous arrival process. *Deadline scheduling* is therefore a more appropriate goal for DSSP.

6.2 Grid and Resource Management

For heterogeneous distributed systems such as the Grid, job scheduling is a major component of resource management. See Feitelson *et al.* [7] for an overview of theory and practice in this space, and Krallmann *et al.* [12] for a general framework for the design and evaluation of scheduling algorithms. Most work

in this space empirically evaluates scheduling heuristics, such as backfilling [15], adaptive scheduling [10], and task grouping [20], to improve system utilization and throughput. Markov [16] described a two-stage scheduling strategy for Sun's Grid Engine that superficially resembles our two-phase decomposition approach. In fact there is no similarity: The first stage of Markov's approach assigns static priorities to jobs and the second stage assigns dynamic priorities to server resources. Most of the work in the Grid space does not emphasize precedence constraints among jobs/tasks.

6.3 Commercial Products

Open-source schedulers such as Condor manage resources, monitor jobs, and enforce precedence constraints [6]. Commercial products such as LSF additionally enforce fair-share constraints [18]. These priority schedulers have no explicit selection phase, so they must handle overload and enforce fair-share constraints through dispatching decisions. Our two-phase deadline scheduler for DSSP can employ a priority scheduler for task dispatching after an optimal solver has selected jobs. Selection can enforce a wide range of constraints, thereby allowing greater latitude for dispatching decisions. Furthermore, the completion rewards of our framework are more expressive than ordinal priorities and thus better suited to DSSP.

7 Concluding Remarks

In this paper we evaluated the two-phase scheduling method for DSSP through parameter tuning and sensitivity analysis. Contrary to our intuition that the performance of our scheduling method should decrease as the maximum critical path length of the input workload increases, our empirical results show that even though there is a close correlation between performance ratio and MaxCPL value, it is *not monotonic* for dispatching polices such as LCPF. More exploration is needed to determine why the performance ratio decreases significantly when $T_\infty^{\max} \approx D/2$. We tentatively conjecture that when deadline is an integral multiple of MaxCPL, dispatcher policies such as LCPF that associate task priorities with job properties suffer because many of the jobs narrowly fail to complete by the deadline, thus achieving no reward and wasting processor resources.

Furthermore, contrary to the worst-case performance bound in our previous work which is pessimistically bad if $T_\infty^{\max} \approx D$, our new empirical evaluation shows that our algorithm performs very well for this special case, with a performance ratio of more than 80%.

Based on these empirical evaluation results, we believe that MaxCPL alone is insufficient to describe the workload and predict the performance of our scheduling methods. It will be interesting to explore the *distribution* of critical path lengths for all the jobs in the workload and determine its impact on the performance of the two-phase scheduling method.

References

1. Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.
2. Eric Anderson, Dirk Beyer, Kamalika Chaudhuri, Terence Kelly, Norman Salazar, Cipriano Santos, Ram Swaminathan, Robert Tarjan, Janet Wiener, and Yunhong Zhou. Value-maximizing deadline scheduling and its application to animation rendering. In *Proceedings of 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Las Vegas, NV, July 2005. ACM press. A 2-page poster also appears in SIGMETRICS 2005.
3. Richard P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.
4. Peter Brucker. *Scheduling Algorithms*. Springer, 3rd edition, 2001.
5. The Condor Project. `http://www.cs.wisc.edu/condor/`.
6. Directed acyclic graph manager (DAGMan) for Condor scheduler. `http://www.cs.wisc.edu/condor/dagman/`.
7. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of JSSPP, LNCS 1291*, pages 1–34, 1997.
8. Ron Graham. Bounds for certain multiprocessor anomalies. *Bell Sys Tech J*, 45:1563–1581, 1966.
9. Ronald Graham. Bounds on multiprocessing time anomalies. *SIAM J Appl Math*, 17:263–269, 1969.
10. Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In Mark Baker Rajkumar Buyya, editor, *Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000), LNCS 1971*, pages 214–227. Springer, 2000.
11. Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.
12. Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the design and evaluation of job scheduling algorithms. In *Proceedings of JSSPP, LNCS 1659*, pages 17–42, 1999.
13. Anurag Kumar and Rajeev Shorey. Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system. *IEEE Trans Par Dist Sys*, 4(10), October 1993.
14. Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.
15. David A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of JSSPP, LNCS 949*, pages 295–303, 1995.
16. Lev Markov. Two stage optimization of job scheduling and assignment in heterogeneous compute farms. In *Proc. IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 119–124, Suzhou, China, May 2004.
17. Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 2nd edition, 2002.
18. Platform Computing. LSF Scheduler. `http://www.platform.com/products/LSFfamily/`.
19. Jiri Sgall. On-line scheduling—a survey. In A. Fiat and G.J. Woeginger, editors, *Online Algorithms: The State of the Art*, number 1442 in LNCS, pages 196–231. Springer-Verlag, 1998.

20. Ling Tan and Zahir Tari. Dynamic task assignment in server farms: Better performance by task grouping. In *Proc. of the Int. Symposium on Computers and Communications (ISCC)*, pages 175–180, July 2002.
21. Adam Wierman and Mor Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *SIGMETRICS*, pages 238–249, June 2003.
22. John Wilkes, Jeffrey Mogul, and Jaap Suermondt. Utilification. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

Co-scheduling with User-Settable Reservations

Kenneth Yoshimoto, Patricia Kovatch, and Phil Andrews

San Diego Supercomputer Center,
University of California, San Diego
{kenneth, pkovatch, andrews}@sdsc.edu

Abstract. As grid computing becomes more commonplace, so does the importance of coscheduling these geographically distributed resources. Negotiating resource management and scheduling decisions for these resources is similar to making travel arrangements: guesses are made and then remade or confirmed depending on the availability of resources. This "Travel Agent Method" serves as the basis for a production scheduler and metascheduler suitable for making travel arrangements for a grid. This strategy is more easily implemented than centralized metascheduler because arrangements can be made without requiring control over the individual schedulers for each resource: the reservations are set by users or automatically by negotiating with each local scheduler's user settable interface. The Generic Universal Remote is a working implementation of such a system and proves that a user-settable reservation facility on local schedulers in a grid is sufficient to enable automated metascheduling.

1 Introduction

A grid is a distributed computing and resource environment connected via software and hardware. The resources can be as diverse as electron microscopes or Terabyte-sized databases. Computational resources that are a component of a grid usually have many processors available for the use of scientists and researchers. These resources often have unique characteristics like a "large" amount of memory or access to "large" amounts of disk. Generally the compute resources have a local resource manager and scheduler to allow sharing of resources between the different users of the system.

In the case of this project, development was done within the context of a specific Grid, the TeraGrid [3], an NSF-funded project to create and link several supercomputer class systems across one of the world's fastest networks. The initial middleware suite used for coordination has been Globus [4,5], while recently there has been significant work done on the use of a Global File System [2], initially based on the General Purpose File System, GPFS [8], from IBM. A similar approach is being investigated by the European Grid community, DEISA [1].

A prior study by another group evaluated multi-site submission of single jobs [7]. This article describes a strategy for scheduling a single job over multiple sites.

A variety of applications can make use of these grid resources. Some applications use loosely-coupled communication resources while others require

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 146–156, 2005.

tightly-coupled communication resources. Some need to stage data from one location to another before it can compute and process additional data. Some applications can make use of distributed resources but synchronize some communications between the sites. Because of this synchronization, resources need to be available at multiple sites at the same time. Some applications want to run at the earliest time possible regardless of the machine type or geographical location of the resources.

Often the grid is compared to an electric utility: always available. But this does not necessarily apply to High Performance Computing (HPC) and grid computing, where demand exceeds supply. The electrical grid is planned to always have more electrical supply than demand. If demand ever exceeds supply, there are rolling brownouts. In this situation where supply exceeds demand, "on-demand" is easy: just plug into a wall outlet, because there's enough slack in the system to provide the power necessary to run a radio. This can be contrasted with HPC and grid computing, where demand often exceeds supply. There are always more applications, more compute, more data and other resources needed to further the never -ending stream of research. The continual demand for capability requires the commitment of resources for specific jobs. The electric grid/"on demand" analogy breaks down for HPC. The scheduling strategies for these two situations are entirely different.

A better analogy is an industry where complex workflows cause excess demand for limited resources: travel. Every day, travel agents create multi-resource itineraries. An itinerary with round-trip airplane flights, rental car and hotel reservations corresponds very well to that of a grid job that needs to do pipeline computation. The travel agent works with a number of different independent resource providers to generate a valid itinerary. One thing that's interesting about this model is that there is no requirement for global coordination. The airline does not need to know that there are rental cars available at the destination, for example. For the travel industry, there is a many-to-many supplier-to-consumer market. For the electric grid, there is one supplier through which all consumers in a region must go. The economics of the travel industry is a better model for HPC/grid computing.

The Generic Universal Remote (GUR) provides distributed resource management capabilities that make efficient use of HPC/grid computing resources based on a travel agent's methods. It provides automatic negotiation of coordinated cross-site (pipeline and co-scheduled) and first possible run-time (Disneyland or load balanced) reservations allowing these types of jobs to run efficiently. These types of jobs are scheduled in an automatic way by taking advantage of local scheduler's user-settable reservation capabilities. GUR requests and confirms reservations from independent local schedulers to generate a grid reservation. It uses a simple heuristic to generate a valid, but not necessarily optimal schedule. GUR does not require centralized control over all the grid resources, unlike other metascheduling solutions. GUR automatically reserves resources from a scientists' laptop that not only makes it easier for the scientist, but it reduces load on the remote "login" nodes. This decentralized local scheduler works like

a "universal remote" where it commands and coordinates access to multiple, heterogeneous distributed resources based on where you "point it", much like a universal electronics remote.

2 Grid Scheduling Strategies

Many different types of applications make use of grid resources. Some typical scenarios include communication-, compute-intensive and combination-type jobs. Communication-intensive applications launch jobs to run on a single remote cluster. Computation âĂŞintensive jobs run on geographically distributed clusters. Communication- and computation-intensive jobs run on a single remote cluster and then possibly transfer the data to another site for visualization.

These applications represent three types of job scheduling scenarios that are common to grid computing:

1. Run x job on n nodes, where all n nodes are located at any one of a set of sites, at the earliest possible time (Disneyland)
2. Run x job on n nodes, where n nodes may be distributed across multiple sites at the same time (co-scheduled)
3. Run x job at site A, then move output to site B for additional computation, visualization or database access (pipeline)

In the first scenario, a user wants to run as quickly as possible. For instance, the user has a communication-intensive type of job and doesn't care which single compute resource (out of a set of compute resources) the job runs on. For example, when you visit Disneyland with your friends, the object is to get on as many rides as possible. Since the rides are popular, there is a wait to get on the rides. Which means you and your friends will wait in line for a ride and then move to the next ride in sequential fashion. A more efficient way to get on more rides would be to split up with your friends and have each of you wait in line at different rides. Then, whoever gets to the front of the line first "wins" and you and your friends leave their different lines to join the "winning" person. This method allows you and your friends to ride as many rides as possible. Many scientific computation applications exhibit this behavior. A brain imaging application renders images from raw data is an example of a kind of application that uses this kind of scheduling scenario. It has been shown that this strategy does not necessarily reach the optimal solution, if the resources are heterogeneous [7]. We have also not studied the effect on global and local utilization with such a strategy. Intuitively, a "market" of resource consumers and resource providers might be expected to emerge. Information, in the form of submitted jobs, would be available to resource providers. We speculate that this dissemination of information might allow more efficient global utilization, without a centralized scheduling mechanism.

Some kinds of computation-intensive jobs make use of geographically distributed resources. These jobs run within each separate site and communicate between sites. For example, to develop a full weather model for the ocean's atmosphere, one part of the job computes the ocean's effects on the atmosphere at

one site and the other part of the model computes the effects of the atmosphere on the ocean. The distributed jobs then communicate at the end of the model to share the data to develop a complete model. This kind of job needs resources to be available simultaneously (co-scheduled).

Different sites offer different capabilities. Because of this, users want to compute at one site, move data to another site, visualize the data at another site and finally store the output at a last site. The storing of data can't start until the data is visualized. And the visualization can't start until the data is moved and so on. The space for the data needs to be available at the remote site. So a schedule of reservations is needed on resources to complete the job workflow. Each step in the process depends on the previous step. An example of this kind of job is an astronomy model that needs specific compute resources at one site and the data and visualization resources at another site. Another instance is a database server with a limited amount of space. In this limited amount of space, users can bring up their own database and stage data into it. A resource manager and scheduler for the database can grant space reservation requests to user. These reservations can be coordinated with regular compute reservations on a separate system. This staging of events is called a pipeline.

3 Grid User-Settable Reservations and Catalina Scheduling

A local scheduler called "Catalina," was developed to provide a user-settable reservation facility for IBM's LoadLeveler, Portable Batch System (PBS) or any local resource manager that has an interface for an external scheduler. Catalina is a reservations-based, single-queue scheduler, much like the Maui scheduler. It prioritize jobs, based on a number of different characteristics. It calculates the expansion factor for how long a job is waiting and adjusts the priority on the job so it won't starve. It also has backfill capabilities to keep the processors busy until the right number of nodes is available for a larger job with higher priority. When a system reservation is made, jobs that will complete before the system reservation starts are scheduled to run. It can schedule any kind of resource including data, database or compute. Catalina consists of 10,000 lines of Python with some functions written in C. The user-settable reservation facility consists of a command-line client run by an unprivileged user. Parameters provided to the user include:

1. Allocation charge account
2. Exact or maximum number of nodes requested
3. Duration of the reservation
4. Earliest time at which the reservation can start
5. Latest time at which reservation may end
6. Email address for failure notification

To keep users from severely disrupting the batch schedule, reservations are restricted by the following policies:

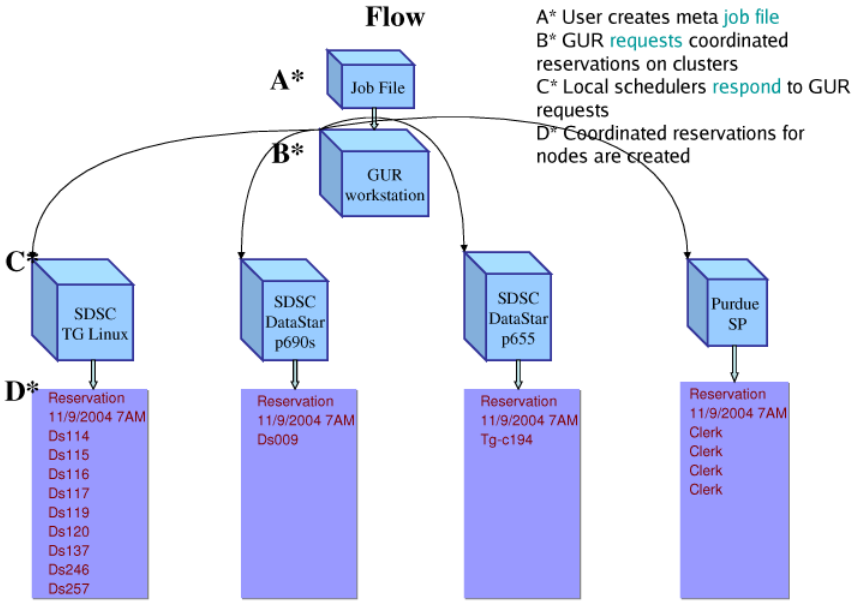


Fig. 1. Job Flow Schematic

1. User-settable reservations are made only after all currently queued jobs are scheduled
2. Number of reservations per account can be limited
3. Number of nodes and duration of each reservation can be limited
4. Global limit on the number of node-seconds devoted to user-settable reservations in a configurable time window

Catalina is the scheduler running on the production supercomputers at the San Diego Supercomputer Center. For instance, it's the scheduler for Blue Horizon, an 1100 processor IBM SP2. It interfaces with LoadLeveler, IBM's proprietary resource manager. It's been running successfully for over three years. In this time, none of the 2000+ user accounts on Blue Horizon interrupted, interfered or delayed the overall batch scheduling process with user-settable reservation capability. Catalina is consistent with the Global Grid Forum Advanced Reservations API. More information on Catalina can be found at http://www.sdsc.edu/catalina.

4 Travel Agent Method

When a travel agent makes reservations for a trip, the agent starts with specific dates and start/end point locations in mind. Then the agent makes a starting guess based on those dates and locations for a specific time for the reservation. The agent may check several airlines (resources) for availability that match the traveler's parameters. If the flight meets the traveler's needs, the reservation is made. If several

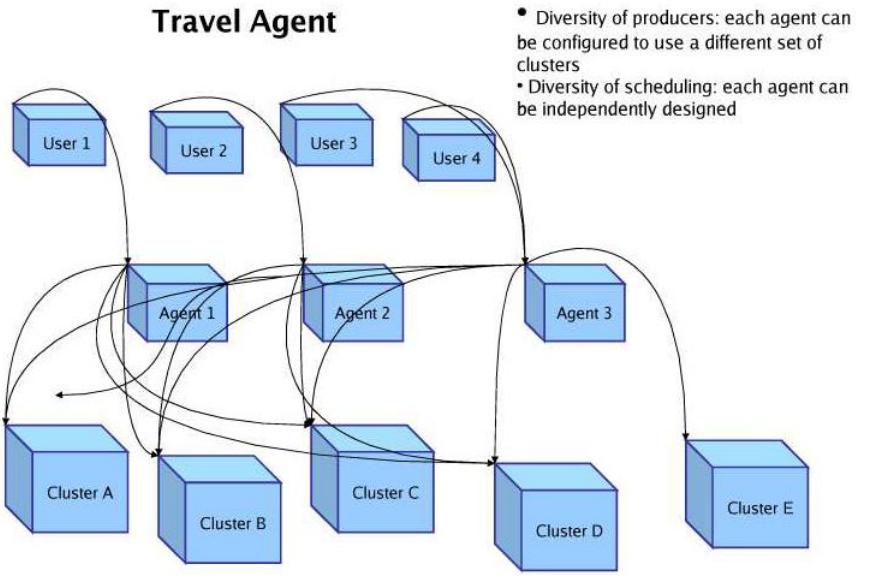


Fig. 2. Job Flow Schematic

flights meet the traveler's needs, the first available flight in those times is booked. If no flights meet the traveler's need, the agent makes another guess about flight times to match the traveler's next best time. If the traveler's needs are met, the reservation is made. If not, another guess is made, and so on. This trial and error strategy generates resource reservations in an acceptable amount of time since there are a finite number of resources available in a specific time period.

Once the airline is reserved, the agent will make the car reservation based on the airline arrival and departure times. And the hotel reservation is dependent on the airline and car rentals times and so on. A draft itinerary is created and checked with the traveler before being booked permanently.

Making reservations for distributed grid resources can be done in a similar way: resources can be scheduled sequentially for jobs requiring staged multiple resources or in parallel for co-scheduled resources. Both user-controlled (manual) and automatic reserved resource acquisitions make efficient use of grid resources since system administrator time is not required to check and make reservations at different sites. This strategy makes reservations for the three different types of jobs requests outlined in the Grid Usage Scenarios section.

5 Generic Universal Remote (GUR)

Traditionally, only system administrators make reservations for resources with local schedulers. With this approach, creating Disneyland, co-scheduled or pipeline reservations require communication with the local system administrators at each of the sites the user wants to run. Contacting individual system administrators

at each site is a time consuming process that undoubtedly requires multiple iterations Traditional approaches to metascheduling require a centralized scheduler that controls the schedulers at each of the sites. This is an impractical approach on the grid, where each site has it's own security and local scheduling policies.

When the user-settable reservation capability is available to users, then users can "self-schedule" and request resources when needed. Users act as their own travel agent and reserve co-scheduled jobs without manual system administrator intervention. This manual process is automated with GUR.

Using the Travel Agent Method, the GUR negotiates reservations with local schedulers. It probes the local schedulers to find potential suitable times and then makes a guess at possible times for the job to execute. Once a time is found, the reservation is made. In order for this to happen, user settable reservations need to be available on the local schedulers.

GUR is a metascheduler that was developed to automatically created coordinated reservations on local schedulers, using the user-settable reservation facility. A user submits a metajob to GUR providing information such as:

1. Total number of nodes needed
2. Minimum and maximum number of nodes needed for each local system
3. Job duration
4. Earliest start time
5. Latest end time
6. Usage scenario
   (a) Single system (Disneyland)
   (b) Multiple systems (co-scheduled)
   (c) Multiple systems (pipeline)

A GUR job request would look something like this:

$[metajob]$
$total\_nodes = 12$
$machine\_preference = datastar655, purduesp$
$machine\_preference\_reorder = yes$
$duration = 7200$
$earliest\_start = 07:00_1 1/09/2004$
$latest\_end = 09:30_1 1/09/2004$
$usage\_pattern = multiple$
$machines\_dict\_string = \{$
$'datastar655' : \{$
$'username\_string' :' kenneth',$
$'account\_string' :' sys200',$
$'email\_notify' :' kenneth@sdsc.edu',$
$'min\_int' : 1,$
$'max\_int' : 158$
$\},$

$$
\begin{aligned}
&'purduesp' : \{ \\
&'username\_string' :' kenneth', \\
&'account\_string' :' TG-STA040001N', \\
&'email\_notify' :' kenneth@sdsc.edu, \\
&'min\_int' : 1, \\
&'max\_int' : 2 \\
&\} \\
&\}
\end{aligned}
$$

For job submission, architecture-specific requirements are abstracted into a GUR configuration file. Jobs request required resource features by specifiying a 'machine', such as 'datastar655'. This would mean p655 nodes on DataStar. At least one node must be used on each cluster. No more than 158 can be used on 'datastar655', and no more than 2 can be used on 'purduesp'.

Then GUR probes each system to make a rough guess at each system's queue load. It does this by setting test reservations on each system and checking the delay for each reservation. This approach is not optimal for all possible queue states, but it does provide usable information on the status of each system.

In order of the least loaded system, GUR makes reservations consistent with the minimum and maximum nodes for each system and the total number of nodes requested. Nodes are preferentially distributed to the least loaded systems. If the initial distribution of nodes to systems is not possible in the requested time frame, a new distribution is generated, giving more nodes to the more heavily loaded clusters. As soon as a solution is found, GUR stops. GUR does not have a 24 hour hold on reservations or a two phase commit. It makes reservations and cancels them, if they are no longer required. While finding the initial optimal distribution of times and nodes, GUR makes a "sliding reservation window" time based on the requested running parameters of the earliest start time and latest end time. It binds the job to the reservation and vise versa.

In addition, GUR is "generic" and works with any local scheduler (with user-settable or manually-settable reservations) that schedules any kind of compute, data or instrumental resource. GUR can perform a series of scripted tasks. For instance, a job can automatically compile and execute based on GURs instructions. It can also stage data or an executable. This enables a "universal" grid roaming capability, where jobs can be launched from a laptop regardless of the operating system on the destination resource. It also reduces the load on "login" nodes because users don't need to login to submit jobs. If the user performs a grid-proxy-init and specifies a GSI-enabled SSH, then GUR will use that to contact the remote systems. If no GSI-enabled SSH is available, or the remote system's gatekeeper is down, GUR will use regular SSH, setting up an agent for the user.

If a reservation cannot be fulfilled, perhaps due to hardware failure or unplanned maintenance, then the local scheduler is expected to email notification to the user.

If any local scheduler is unable to provide the minimum number of nodes within the sliding reservation window, GUR informs the user that the reservation

is not possible. The user may then expand the time window, reduce the number of resources requested or both and resubmit.

GUR also gives your local computer the capacity to act like a television "remote" where it can control the geographically distributed resources on the grid behind it by providing a single, uniform, easy-to-use interface for the user.

GUR works with any local scheduler that takes user-settable reservations (such as Catalina) or has an interface for reservations. It can also work with the Maui Scheduler with manual system administrative assistance. GUR conforms to Global Grid Forum Advance Reservations API. GUR does not currently support pipeline jobs, but it would be easy to extend the multiple usage pattern to pipeline by providing a time offset to each resource so they happen in sequential rather than simultaneous order.

More information on GUR can be found at `http://www.sdsc.edu/~kenneth/gur.html`. GUR can be downloaded from `http://www.sdsc.edu/scheduler/gur.html`.

6 Real-Life Experiences with Metascheduling and GUR

Several previous experiences explored the use of coordinated reservations. At a previous SuperComputing conference, a user manually created reservations through the General-purpose Architecture for Reservation and Allocation (GARA) to the Portable Batch System Professional (PBSPro). The co-scheduled metajob ran across sites using MPICH-G.

In another demonstration, a user made reservations on an SDSC IA-32 Linux cluster running the Portable Batch System (PBS) resource manager and on SDSC's Blue Horizon, an 1100 processor SP, running the LoadLeveler resource manage r. The co-scheduled metajob ran between the machines using MPICH.

Co-scheduled reservations were made with the Silver metascheduler on Blue Horizon's LoadLeveler/Maui and an SP at Pacific Northwest Laboratories running LoadLeverl/Maui. ECCE/NWChem was the application. Silver is a centralized metascheduler that only works with the Maui scheduler.

Additionally, SP clusters at SDSC, University of Michigan âĂŞ Ann Arbor and University of Texas âĂŞ Austin were reserved and scheduled a centralized metascheduler during a separate SuperComputing demonstration. These clusters all shared the same UID/GID space. All the clusters used LoadLeveler and the Maui scheduler. Again, the centralized metascheduler made the reservations from a privileged account for a co-scheduled metajob.

In a new approach, GUR was used to automatically schedule jobs with various characteristics on several heterogeneous compute platforms. At SDSC, Disneyland and co-scheduled jobs were scheduled between Blue Horizon and a Linux cluster. Both of these resources are working, production supercomputers with real user jobs scheduled and running at the time of the tests.

User job requests were made from each compute platform to GUR. These jobs requested a various number of processors, run times and executables. Some jobs requested to run at the same time on both platforms. And other jobs requested to

run at the earliest possible time. All of these tests worked. This tests shows that grid metascheduling can be performed with local schedulers with user-settable reservations and a decentralized metascheduler.

In the latest demonstration of GUR capability, at SC2004, co-scheduled reservations were made across three platforms. These were SDSC DataStar (IBM SP with Catalina), SDSC Linux cluster (ia64/Myrinet with Catalina), and Purdue SP (IBM SP with PBSPro). The SDSC machines are running production workload, while the Purdue machine was a test system. GUR was able to successfully create a set of synchronized reservations across the three clusters. GUR reserved 10 nodes on SDSC DataStar, one node on SDSC Linux cluster, and one node on the Purdue SP. These reservations were all scheduled to start at 7am Nov 9, 2004 PST.

GUR is similar to Silver in that it depends on reservations created on the local schedulers. It differs in using user-settable reservations rather than privileged reservations. GUR resembles Condor-G, since GUR can submit jobs to diverse compute resources. GUR makes use of user-settable reservations to provide synchronization of job starts, which Condor-G [6] does not do.

7 Conclusion

Resources can be easily scheduled on a grid by deploying an automatic scheduler that mimics the human travel agent's process for making reservations. Reading the user's request for resources, making a guess at the best possible times and sliding the window of those times until a match is found reserves resources in a reasonable amount of time. Various types of typical HPC/grid computing jobs can be scheduled in this manner, including Disneyland, co-scheduled and pipeline jobs.

A working version of this metascheduler, GUR, along with a local scheduler with user-settable reservations, Catalina, proves that this approach is possible. The Catalina-GUR system demonstrates that a user-settable reservation facility is sufficient to enable automatic, coordinated metascheduling. User settable reservations are practical since they are controlled by policies that restrict users from interrupting the flow of the batch system. The ability to request any number of nodes up to a maximum makes it much easier to explore the node distribution space throughout the grid.

This architecture allows many alternate metaschedulers to participate in the grid. It also allows additional flexibility since it can be run from a laptop, giving unprecedented, ubiquitous access to users of the grid. In addition, it also removes the requirement for a centralized metascheduler, which is difficult to coordinate and make secure. The combination of user-settable reservations along with an automated, de-centralized metascheduler is an approach that allows many new types of HPC applications to run on a grid.

Acknowledgments

We wish to thank Wendy Lin of Purdue for development of code to enable user-settable reservations on the Purdue SP.

References

1. `www.deisa.org`.
2. Phil Andrews, Tom Sherwin, and Bryan Banister. A centralized data access model for grid computing. In *IEEE Symposium on Mass Storage Systems*, pages 280–289, April 2003.
3. Charlie Cattlett. The teragrid: A primer. `www.teragrid.org`, 2002.
4. Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
5. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.
6. James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *High Performance Distributed Computing (HPDC-10)*, August 2001.
7. Gerald Sabin, Rajkumar Kettimuthu, Arun Rajan, and Ponnuswamy Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–104. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.
8. Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In USENIX, editor, *Proceedings of the FAST '02 Conference on File and Storage Technologies: January 28-30, 2002, Monterey, California, USA*, Berkeley, CA, USA, 2002. USENIX.

Scheduling Moldable BSP Tasks*

Pierre-François Dutot[1], Marco A.S. Netto[2,**],
Alfredo Goldman[2], and Fabio Kon[2]

[1] Laboratoire ID-IMAG, 38330 Montbonnot, France
[2] Department of Computer Science, University of São Paulo, São Paulo, Brazil

Abstract. Our main goal in this paper is to study the scheduling of parallel BSP tasks on clusters of computers. We focus our attention on special characteristics of BSP tasks, which can use fewer processors than the original required, but with a particular cost model. We discuss the problem of scheduling a batch of BSP tasks on a fixed number of computers. The objective is to minimize the completion time of the last task (*makespan*). We show that the problem is difficult and present approximation algorithms and heuristics. We finish the paper presenting the results of extensive simulations under different workloads.

1 Introduction

With the growing popularity of Computational Grids [1] the model of environment in which parallel applications are executing is changing rapidly. In contrast to dedicated homogeneous clusters, where the number of processors and their characteristics are known *a priori*, Computational Grids are highly dynamic. In these new environments, the number of machines available for computation and their characteristics can change frequently. When we look at the case of Opportunistic Grid Computing, which uses the shared idle time of the existing computing infrastructure [2], the changes in machine availability occur even more rapidly. Thus, a model of parallel computation that does not allow variations in the number of processors available for computation would not fit well in this environment.

Moldable tasks are able to maximize the use of available resources in a dynamic Grid in the presence of fluctuations in machine availability. In this paper we extend the *Bulk Synchronous Parallel* (BSP) model [3] of computation to allow for the definition of moldable tasks that can be executed in a varying number of processors. As it will be described in detail later, a BSP application is a sequence of supersteps, composed of the execution of independent processes, separated by barrier synchronizations.

Due to complexity of the grid environment, we have first focused our work on the problem of scheduling tasks with a fixed set of available computers. However, we are currently investigating mechanisms to improve the scheduling by

* Research supported by a grant from CNPq, Brazil - grant number: 55.2028/02-9.
** PhD student supported by a fellowship from CAPES, Brazil.

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 157–172, 2005.

supporting preemption of BSP tasks so as to schedule malleable tasks. We are also studying mechanisms to propose a dynamic scheduling scheme. These improvements we allow us to develop sophisticated heuristics to schedule parallel applications on actual computational grids.

The remainder of this paper is organized as follows. At the end of this section we present our motivation and related work regarding the scheduling of moldable tasks. In Section 2 we describe the BSP model and we also discuss the moldability on BSP and the problem of scheduling moldable tasks. In Section 3 we propose an approximation algorithm and some heuristics providing complexity proofs. In Section 4 we show experimental results to evaluate the proposed algorithms. In Section 5, we close the paper with some final remarks and ideas for future works.

1.1 Motivation

Our group is developing a novel Grid middleware infrastructure called InteGrade [2]. The main principles of InteGrade are: modern object-oriented design, efficient communication based on CORBA, and native support for parallel computing. In the current version[1], the BSP model [4] for parallel computation is supported through an implementation of the BSPlib [5] library. In this paper, we propose new scheduling algorithms for batches of BSP tasks, which are being included into the InteGrade system.

Using only rigid BSP tasks, we could use classical results for scheduling tasks with different execution times and number of processors. However, in our grid environment we can easily reduce the number of processors of a BSP task, allocating two or more processes to the same processor. As our environment is based on CORBA, there are no differences between local and remote communications, this is transparent to the programmer.

Given a BSP task that requires execution time t on n processors, we can allocate it without effort, depending on the memory constraints, using fewer processors. The behavior of moldability can be approximated by a discrete function. If fewer than n processors are available, say n', the execution time can be estimated by $t\lceil \frac{n}{n'} \rceil$.

1.2 Related Work

Most existing works for scheduling moldable tasks are based on a two-phase approach introduced by Turek, Wolf, and Yu [6]. The basic idea is to select, in a first step, an allocation (the number of processors allocated to each task) and then solve the resulting non-moldable scheduling problem, which is a classical multiprocessor scheduling problem. As far as the makespan criterion is concerned, this problem is identical to a 2-dimensional strip-packing problem [7, 8]. It is clear that applying an approximation of guarantee λ for the non-moldable problem on the allocation of an optimal solution provides the same guarantee λ for the moldable problem. Ludwig [9] improved the complexity of the allocation

[1] Available for download at `http://gsd.ime.usp.br/integrade`

selection of the Turek's algorithm in the special case of monotonic tasks. Based on this result and on the 2-dimensional strip-packing algorithm of guarantee 2 proposed by Steinberg [10], he presented a 2-approximation algorithm for the moldable scheduling problem. These results however are designed for the general moldable tasks problem, where each task has a different execution time for each number of processors.

As we will see in the formal definition of BSP moldable tasks, the size of our instances is much smaller. This happens because we know the penalty incurred when the number of processors allocated to a task is different from the requested number of processors.

Mounié, Rapine and Trystram improved this 2-approximation result by concentrating more on the first phase (the allocation problem). More precisely, they proposed to select an allocation such that it is no longer needed to solve a general strip-packing instance, but a simpler one where better performance guarantees can be ensured. They published a $\sqrt{3}$-approximation algorithm [11] and later submitted a 3/2-approximation algorithm [12, 13]. However, these results are for a special case of moldable tasks where the execution time decreases when the number of processors allocated to the task increases and the workload (defined as *time*×*processors*) increases accordingly. We will see that this hypothesis is not verified here. To the best of our knowledge there is no other work on scheduling moldable BSP tasks.

2 The BSP Computing Model

The *Bulk Synchronous Parallel* model (BSP) [3] was introduced by Leslie Valiant as a bridging model, linking architecture and software. BSP offers both a powerful abstraction for computer architects and compiler writers and a concise model of parallel program execution, enabling accurate performance prediction for proactive application design.

A BSP abstract computer consists of a collection of virtual processors, each with local memory, connected by an interconnection network whose only properties of interest are the time to do a barrier synchronization and the rate at which continuous, randomly addressed data can be delivered. A BSP computation consists of a sequence of parallel supersteps, where each superstep is composed of computation and communication, followed by a barrier of synchronization.

The BSP model is compatible with conventional SPMD/MPMD (single/multiple program, multiple data), and is at least as flexible as MPI [14], having both remote memory (DRMA) and message-passing (BSMP) capabilities. The timing of communication operations, however, is different since the effects of BSP communication operations do not become effective until the next superstep.

The postponing of communications to the end of a superstep is the key idea for implementations of the BSP model. It removes the need to support non-barrier synchronizations among processes and guarantees that processes within a superstep are mutually independent. This makes BSP easier to implement on different architectures and makes BSP programs easier to write, to understand,

and to analyze mathematically. For example, since the timing of BSP communications makes circular data dependencies among BSP processes impossible, there is no risk of deadlocks or livelocks in a BSP program. Also, the separation of the computation, communication, and synchronization phases allows one to compute time bounds and predict performance using relatively simple mathematical equations [15].

An advantage of BSP over other approaches to architecture-independent programming, such as the PVM [16] and MPI [17] message passing libraries, lies in the simplicity of its interface, as there are only 20 basic functions. A piece of software written for an ordinary sequential machine can be transformed into a parallel application with the addition of only a few instructions.

Another advantage is performance predictability. The performance of a BSP computer is analyzed by assuming that, in one time unit, an operation can be computed by a processor on the data available in local memory and based on the following parameters:

1. P – the number of processors;
2. w_i^s – the time to compute the superstep s on processor i;
3. h_i^s – the number of bytes sent or received by processor i on superstep s;
4. g – the ratio of communication throughput to processor throughput;
5. l – the time required to barrier synchronize all processors.

To avoid congestion, for every processor on each superstep, h_i^s must be no greater than $\lceil \frac{l}{g} \rceil$.

Moreover, there are plenty of algorithms developed for CGM (Coarse Grained Multicomputer Model) [18], which has the same principles of BSP, and can be easily ported to BSP.

Several implementations of the BSP model have been developed since the initial proposal by Valiant. They provide to the users full control over communication and synchronization in their applications. The mapping of virtual BSP processors to physical processors is hidden from the user, no matter what the real machine architecture is. BSP implementations developed in the past include: Oxford's BSPlib [5] (1993), JBSP [19] (1999), a Java version, PUB [20] (1999) and BSP-G [21] (2003).

2.1 Moldability on BSP

Given a BSP task that requires n processors, it is composed of n different processes which communicate on the global synchronization points. When designing BSP algorithms, for example using CGM techniques, one of the goals can be to distribute the load across processes more or less as evenly as possible.

To model moldability we use the following fact. When embedding BSP processes into homogeneous processors, if a single processor receives two tasks, intuitively, it will have twice as much work as the other processors. To reach each global synchronization, this processor will have to execute two processes and to send and receive the data corresponding to these processes. However, to continue processing, all the other processors have to wait. Hence, the program

completion time on $n-1$ processors will be approximately two times the original expected time on n processors.

The same idea can be used when scheduling BSP tasks on fewer processors than the required. Each BSP process has to be scheduled to a processor and the expected completion time will be the original time multiplied by the maximum number of processes allocated to a processor. It is clear to observe that when processes are allocated to homogeneous processors, in order to minimize execution time the difference in the number of processes allocated to the most and to the least loaded processor should be at most one. This difference must be zero when the used number of processors exactly divides the number of processes.

For the scheduling algorithms used in this paper, given a BSP task composed of n processes and with processing time t, if $n' < n$ processors are used, the processing time will be $t\lceil \frac{n}{n'} \rceil$. So, if only $n-1$ processors are available, the execution time of these tasks will be the same whether using $n-1$, or $\lceil \frac{n}{2} \rceil$ processors. Obviously, in the last case, we will have a smaller work area (number of processors times execution time).

2.2 Notations and Properties

We are considering the problem of scheduling independent moldable BSP tasks on a cluster of m processors.

In the rest of the paper the number of processors requested by the BSP task i will be denoted req_i. The execution time of task i on a number p of processors will be $t_i(p)$. As we are dealing with BSP tasks, we can reduce the number of processors allocated to a task at the cost of a longer execution time. The relation between processor allocation and time is the following:

$$\forall q \forall p \in \left[\frac{req_i}{q+1}, \frac{req_i}{q}\right[,\ t_i(p) = (q+1)t_i(req_i)$$

where p and q are integers. In this work we do not consider a minimal number of processors for each task.

Table 1 shows an example with $req_i = 7$ and $t_i(req_i) = 1$, and the resulting workload which is defined as the product of processors allocated and execution times. We can see in this example that the workload is not monotonous in our case as in some other works on moldable tasks [11], but it is always larger than or equal to the workload with the required number of processors. Remark that for any task, on one processor the workload is equal to the minimum workload.

Table 1. A BSP task and its possible execution times and associated workloads

#procs.	7	6	5	4	3	2	1
time	1	2	2	2	3	4	7
work	7	12	10	8	9	8	7

2.3 NP-Hardness

The problem of scheduling independent moldable tasks is generally believed to be NP-hard, but this has never been formally proven. It contains as a special case the problem of scheduling independent sequential tasks (requiring only one processor), which is NP-hard [22]. However, the size of the moldable tasks problem is $O(n * m)$ since each task has to be defined with all its possible allocation, whereas the size of the sequential problem is $O(n + \ln(m))$ since we only need to know the number of available processors and the length of each task.

In the BSP moldable task problem, the problem size is hopefully much smaller, as we only need to know for each task the requested number of processors and the execution time for this required number of processors. The moldable behavior of the tasks is then deduced from the definition of BSP moldable tasks. Therefore the overall size of an instance is in $O(n * \ln(m))$ which is polynomial in both n and $\ln(m)$. The reduction from the multi-processor scheduling problem is then polynomial, which proves the NP-hardness of our problem.

3 Algorithms

To solve efficiently the problem of scheduling parallel BSP tasks, we have to design polynomial algorithm which provides on average a result close to the optimal. The first step is therefore to determine a good lower bound of the optimal value to be able to measure the performance of our algorithms. Two classic lower bounds for scheduling parallel tasks are the total workload divided by the number of available processors and the length of the longest task. With our previous notations, these two lower bounds are respectively $\sum_i t_i(req_i)/m$ and $\max_i t_i(req_i)$.

3.1 Guaranteed Algorithm

The best way to assess the quality of an algorithm is to mathematically prove that for any instance, the ratio between the makespan ω of the schedule produced by the algorithm and the optimal makespan ω^* is bounded by a constant factor ρ.

As we said in the introduction, the problem of scheduling independent moldable tasks has already been studied and some guaranteed algorithms have already been proposed for this problem. The best algorithm to date is a 3/2-approximation algorithm proposed by Mounié et al. [13], however this algorithm needs an additional monotonicity property for the tasks. This property states that the workload is non decreasing when the number of processors allocated to a task increases which is clearly not the case with our moldable BSP tasks. An older algorithm which does not require this monotonic property has been designed by Ludwig [9]. This algorithm has a performance ratio of 2 as does the one we are proposing below, however it is much more complicated to use since it involves a strip packing phase. This is why we decided to design a 2-approximation algorithm based on our knowledge of the BSP tasks.

The algorithm is based on the dual approximation scheme as defined by [23]. The dual approximation scheme is based on successive guess $\hat{\omega}$ of the optimal makespan, and for each guess runs a simple scheduler which either outputs a schedule of makespan lower or equal to $2\hat{\omega}$, or outputs that $\hat{\omega}$ is lower than the optimal. With this scheduler and a binary search, the value of $\hat{\omega}$ quickly converges toward a lower bound of the optimal makespan for which we can produce a schedule in no more than $2\hat{\omega}$ units of time.

The scheduler works as follows. Based on the guess $\hat{\omega}$, we determine for each task i the minimal allocation a_i (if it exists) such that $t_i(a_i) \leq 2\hat{\omega}$. If there is a task such that this a_i does not exists (i.e. $t_i(req_i) > 2\hat{\omega}$) the optimal makespan is larger than this particular $t_i(req_i)$ and therefore larger than $\hat{\omega}$. Given these a_i, we schedule all the tasks that require more than one processor ("large" tasks) on exactly a_i processors, and we schedule the remaining tasks ("small" tasks, requiring exactly one processor) on the q remaining processors with a largest processing time first order.

There are three cases in which this algorithm fails to produce a schedule in no more than $2\hat{\omega}$ units of time:

1. There are too many processors required by "large" tasks ($\sum_{a_i>1} a_i > m$).
2. There are no processors left for "small" tasks ($\sum_{a_i>1} a_i = m$ and $\sum_{a_i=1} a_i > 0$).
3. One of the sequential tasks is scheduled to complete after the $2\hat{\omega}$ deadline. As the first fit has a 2-approximation ratio, it means that there is too much workload for "small" tasks ($\sum_{a_i=1} t_i(1) > (m - \sum_{a_i>1} a_i)\hat{\omega}$).

For each case we will prove that if the schedule fails, the guess $\hat{\omega}$ is lower than the optimal makespan. Before going into details for each case, we need to prove the following lemma:

Lemma 1. *For all task i such that $a_i > 1$, we have $t_i(req_i)req_i \geq a_i\hat{\omega}$.*

The idea behind this lemma is that the a_i processors allocated to task i are used efficiently for a sufficient period of time.

Proof. For a_i equal to 2, we know that $t_i(a_i - 1) > 2\hat{\omega}$ as a_i is the minimal number of processors to have an execution time no more than $2\hat{\omega}$. As we noted in Section 2.2 the workload on one processor is equal to the minimal workload $req_i t_i(req_i)$, therefore we can write when $a_i = 2$ and $t_i(a_i - 1) = req_i t_i(req_i)$ that $t_i(req_i)req_i \geq a_i\hat{\omega}$.

For the other extremal case, when $a_i = req_i$, since $req_i \geq 2$ we have $req_i - 1 \geq req_i/2$ and then $t_i(req_i - 1) = 2t_i(req_i)$ by definition of the execution times (see Section 2.2). By definition of a_i, we then have $2t_i(req_i) > 2\hat{\omega}$ and then $req_i t_i(req_i) > a_i\hat{\omega}$.

For the general case where $2 < a_i < req_i$, by definition of $t_i(a_i)$, there exists an integer q such that $t_i(a_i) = (q+1)t_i(req_i)$. As a_i is minimum, $t_i(a_i - 1) > 2\hat{\omega}$

and there exists also an integer $s \geq 1$ such that $t_i(a_i - 1) = (q + s + 1)t_i(req_i)$. Therefore we have the following lower bound for $t_i(req_i)$:

$$t_i(req_i) > \frac{2\hat{\omega}}{q + s + 1} \tag{1}$$

By definition of the execution times, as $t_i(a_i - 1) = (q + s + 1)t_i(req_i)$, we have $a_i - 1 < req_i/(q + s)$ which can be rewritten as:

$$req_i \geq (q + s)(a_i - 1) + 1 \tag{2}$$

By combining inequalities 1 and 2, we have a lower bound for the left term of the lemma:

$$t_i(req_i)req_i > \frac{2((q + s)(a_i - 1) + 1)}{q + s + 1}\hat{\omega} \tag{3}$$

In order to conclude, we have to compare the values of a_i and $2((q + s)(a_i - 1) + 1)/(q + s + 1)$ which is done by comparing their difference:

$$\begin{aligned} 2((q + s)(a_i - 1) + 1) - a_i(q + s + 1) &= 2qa_i + 2sa_i - 2q - 2s + 2 - qa_i - sa_i - a_i \\ &= q(a_i - 2) + s(a_i - 2) - (a_i - 2) \\ &= (q + s - 1)(a_i - 2) \end{aligned}$$

This value being positive or equal to zero, $a_i\hat{\omega}$ is a lower bound of the right term of inequality 3, which concludes the proof of the lemma. □

Theorem 1. *When the schedule fails, the guess $\hat{\omega}$ is too small.*

Proof.

Case 1.

$$\sum_{a_i > 1} a_i > m$$

In this case the minimal total workload $\sum_i req_i t_i(req_i)$ can be bounded in the following way:

$$\begin{aligned} \sum_i req_i t_i(req_i) &\geq \sum_{a_i > 1} req_i t_i(req_i) \\ &\geq \sum_{a_i > 1} a_i\hat{\omega} \\ \sum_{a_i > 1} a_i\hat{\omega} &> m\hat{\omega} \end{aligned}$$

Therefore $\hat{\omega}$ is lower than the optimal makespan.

Case 2.

$$\sum_{a_i>1} a_i = m \text{ and } \sum_{a_i=1} a_i > 0$$

As previously, we can bound the minimal total workload but this time the strong inequality is the first one:

$$\sum_i req_i t_i(req_i) > \sum_{a_i>1} req_i t_i(req_i)$$
$$\sum_{a_i>1} req_i t_i(req_i) \geq \sum_{a_i>1} a_i \hat{\omega}$$
$$\sum_{a_i>1} a_i \hat{\omega} = m\hat{\omega}$$

Which again proves that the guess was too small.

Case 3.

$$\sum_{a_i=1} t_i(1) > \left(m - \sum_{a_i>1} a_i \right) \hat{\omega}$$

Finally in this case, the bounding is a little more subtle:

$$\begin{aligned} \sum_i req_i t_i(req_i) &= \sum_{a_i>1} req_i t_i(req_i) + \sum_{a_i=1} req_i t_i(req_i) \\ &\geq \sum_{a_i>1} a_i \hat{\omega} + \sum_{a_i=1} t_i(1) \\ &> \sum_{a_i>1} a_i \hat{\omega} + \left(m - \sum_{a_i>1} a_i \right) \hat{\omega} = m\hat{\omega} \end{aligned}$$

Therefore in all the cases where the schedule fails, the guess was lower than the optimal makespan. □

Corollary 1. *The proposed algorithm provides a 2-approximation for BSP moldable tasks.*

The sum of the sequential execution times of all the tasks is an upper bound of the optimal makespan, which is polynomial in the size of the instance. Starting from this guess, we can use the algorithm in a binary search of the lowest possible value $\hat{\omega}$ for which we can build a schedule in at most $2\hat{\omega}$. If $\epsilon/2$ is the size of the last step of the binary search, $\hat{\omega} - \epsilon/2$ is a lower bound of the optimal ω^*, and $2\hat{\omega} < 2\omega^* + \epsilon$ which means that the schedule produced in the last step is at most $2 + \epsilon$ times longer than the optimal.

3.2 Tested Heuristics

We have implemented four algorithms to schedule a set of BSP tasks, each task comprising a set of processes, on homogeneous processors.

The **first algorithm** $A1$ is the well-known Largest Task First list scheduling (where largest refers to *number of processors*$\times$*execution time* i.e. the workload) with a pre-processing stage. This pre-processing consists of modifying all tasks regarding the maximum number of processors *maxnprocs* each one will receive. The idea here is to reduce the size of the largest jobs in order to have less heterogeneity in the set of tasks.

When the original number of processors *reqnprocs* of a task is modified, the amount of time *reqtime* needed to execute it is also modified. The pseudo-code below is executed on each task before scheduling.

Algorithm 1. Pseudo-code for pre-processing each task to be scheduled in algorithm $A1$

```
if task.reqnprocs > maxnprocs then
  task.reqtime = ⌈(task.reqnprocs/maxnprocs)⌉ * task.reqtime
  task.reqnprocs = maxnprocs
end if
```

The main problem of this algorithm is that we must verify all possible *maxnprocs* values, from one to the number of processors available in the computing system so as to discover the most appropriated value. Doing this we noticed that the true LTF scheduling (i.e. when $maxnprocs = m$ tasks are not reduced) was usually far from the optimal makespan.

Once the tasks are reduced they are sorted according to their sizes in $O(n * \ln(n))$ steps, and then scheduled in n steps. The overall complexity of this algorithm is therefore $O(m * n * \ln(n))$.

The **second algorithm** $A2$ is based on the idea of reducing the idle time in the schedule by optimizing the placement of the different tasks (see Fig. 1). The algorithm comprises two steps:

1. Look for the *best* task such that, when scheduled, the idle time is reduced or remains the same. *Best* task means the smallest amount of idle time, the better the task. Note that in this step, the number of processors and time to execute the task can be modified. If a task is found, schedule it.
2. If Step 1 has failed, schedule the first largest task that was not scheduled yet.

As we have seen in the presentation of the BSP moldable model, for a given task there can be several allocations having the same execution time. For example, in Table 1 the allocations to 4, 5 and 6 processors all have an execution time of 2. We therefore will only consider here interesting allocations, for which there is no smaller allocation for the same execution time.

With this restriction the number of possible allocations goes down from *reqnprocs* to approximately $2\sqrt{reqnprocs}$. This greatly reduces the complexity of the algorithm, however the overall complexity is still greater than $O(n*\ln(m))$ which is the size of the instance.

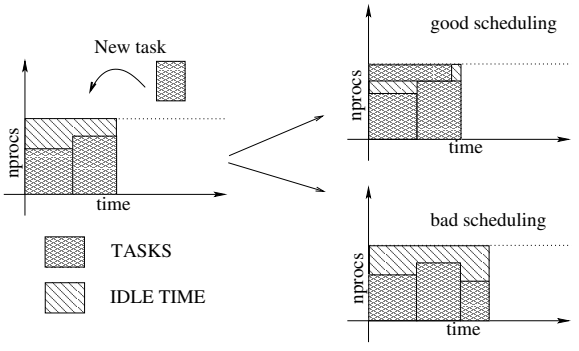


Fig. 1. Examples of schedulings to reduce the idle time

The **third algorithm** $A3$ is a derivation of the second one previously presented. It basically consists of scheduling tasks that generate the smallest idle time, even if the new idle time is greater than the original one. Thus, the first step presented in the previous algorithm is not limited to smaller idle times, and the second step is never executed.

The **fourth algorithm** $A4$ is the guaranteed algorithm presented in the previous section. It is the fastest algorithm, however we will see that its average behavior is far from the best solutions found.

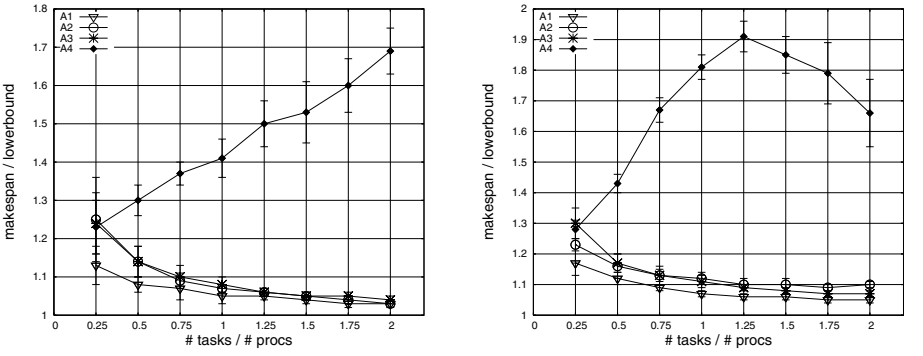
4 Experimental Results

In order to evaluate the algorithms, we developed a simulator that implements the presented algorithms and used both real and generated workloads. The real workloads[2] are from two IBM SP2 systems located at Cornell Theory Center (CTC) and San Diego Supercomputer Center (SDSC) [24], and the generated workloads were generated by a Gaussian distribution. Unlike the real workloads, the number of processors requested by the tasks in the generated instances are in most cases not powers of two [25], which are "bad" tasks for our algorithms as for example 32 is divisible by 16, 8, 4 and 2, and 33 is divisible only by 11 and 3. Note that although the real workloads are not from execution of parallel BSP tasks, the selected machines work with regular parallel applications, and to the best of our knowledge there should be no difference between workloads of MPI and BSP applications.

To perform the experiments we chose three different platforms: with respectively 64, 128 and 256 processors. We selected the SDSC workloads to evaluate the algorithms on 64 and 128 processors and the CTC workloads were used in the experiments with 256 processors. The generated workloads were used for all platforms.

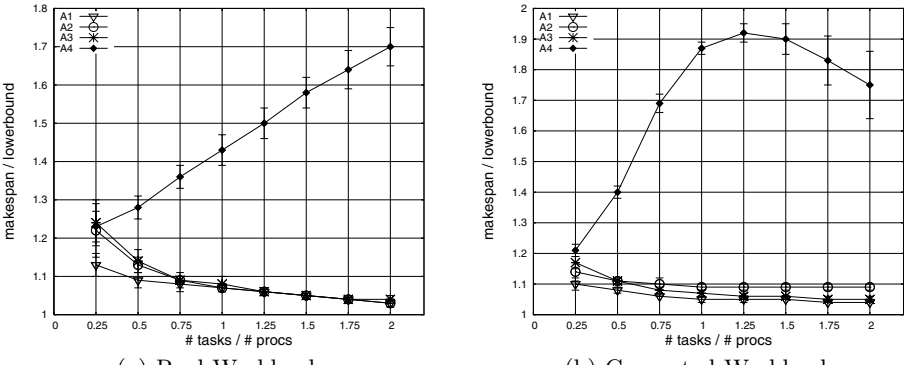
For each experiment we performed 40 executions with different workloads, and then we took out the five best and the five worst results to reduce the deviation.

[2] Available at: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html



(a) Real Workloads (b) Generated Workloads

Fig. 2. Evaluation of the scheduling algorithms on 64 processors

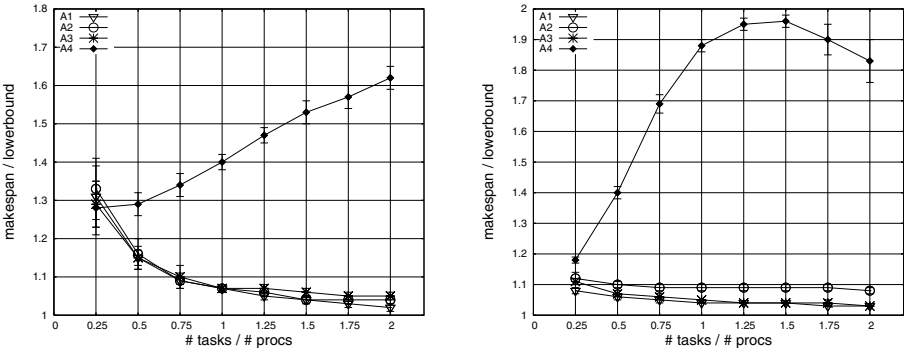


(a) Real Workloads (b) Generated Workloads

Fig. 3. Evaluation of the scheduling algorithms on 128 processors

The tasks in each real workload experiment were selected randomly from all the tasks in the corresponding logs. The graphics illustrated in Fig. 2, 3 and 4 depict the results obtained in our experiments. In these figures the x-axis is the ratio between the number of tasks scheduled and the number of processors of the computer, while the y-axis is the ratio between the schedule length and a lower bound for the considered instance. This lower bound is actually the maximum of the two classical lower bounds: the execution time of the longest task (when allocated to its required number of processors) and the minimal average workload per processor. The schedule produced by the fourth algorithm is always lower or equal to two times the average workload.

Based on the results we can observe that algorithm $A1$ generally produces the best schedules. The algorithms $A2$ and $A3$ have similar behaviors and are very close to $A1$. Finally, as expected the fourth algorithm has a ratio which is close to 2 in the unfavorable cases. Remark that for the generated workload, the worst results of $A4$ are for tasks/processors ratios close to 1. This result confirms the



(a) Real Workloads (b) Generated Workloads

Fig. 4. Evaluation of the scheduling algorithms on 256 processors

intuition [12] that for moldable task problems the difficult part is when there are approximately as many tasks as processors.

To illustrate the difference between the fourth algorithm and the three other algorithms, we included Fig. 5, 6, 7 and 8 that depict schedules for 25 tasks on 16 processors made with the four algorithms. On Fig. 8 it appears clearly that reducing all the tasks to the allocation which is the smallest below the $2\hat{\omega}$ limit tends to produce schedules close to twice the optimal, since most of the tasks are sequential.

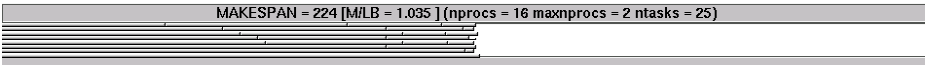


Fig. 5. A schedule of 25 tasks on 16 processors with algorithm A1

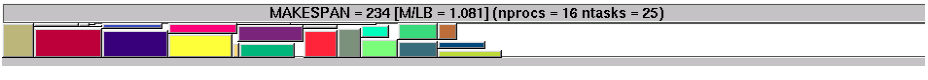


Fig. 6. A schedule of 25 tasks on 16 processors with algorithm A2

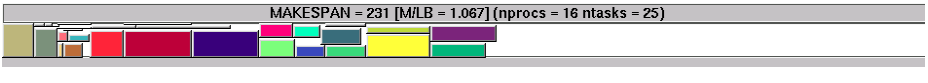


Fig. 7. A schedule of 25 tasks on 16 processors with algorithm A3

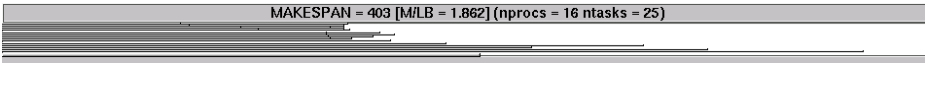


Fig. 8. A schedule of 25 tasks on 16 processors with algorithm A4

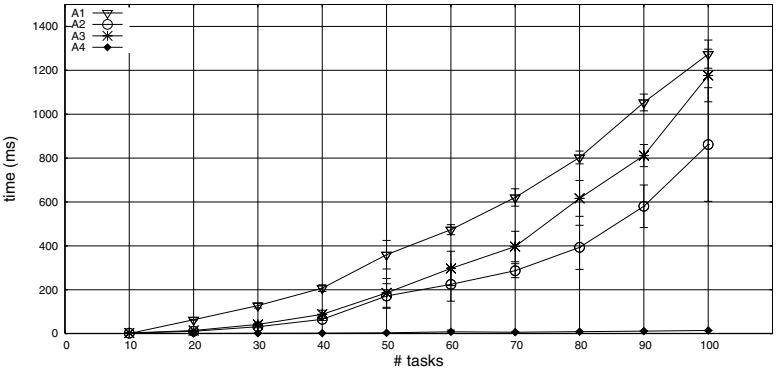


Fig. 9. Execution times for up to 100 tasks on 64 rocessors

As mentioned previously, the main problem of the algorithm $A1$ is that we need to schedule the tasks several times in order to discover the threshold, which is the maximum amount of processors the tasks should use. However, when there is a small number of processors in the computing environment, this algorithm is still usable in reasonable time. For larger numbers of processors, the algorithms $A2$ and $A3$ should be used, since even if they do not produce the best results, the difference is within reasonable bounds. As we could have guessed, the longer it takes to schedule the tasks, the better the results.

This is illustrated in Fig. 9, where the execution times of the four algorithms are compared on 64 processors for 10 to 100 tasks. As previously described, the fourth algorithm is much faster than the three others, and the slowest algorithm is the first one. The execution times on the time scale are in milliseconds. For larger instances (1024 tasks on 512 processors) we witnessed execution times of several minutes on a recent computer (Pentium III 800 MHz, 512MB RAM). We executed all the other experiments on the same computer.

Another important observation is that the results using real and generated workloads are similar for the algorithms $A1$, $A2$ and $A3$. Our main goal to make experiments with generated workloads is that the real workloads are mostly made of regulars tasks, as well as tasks requiring processors in powers of two. These characteristics are usually found only in dedicated computer systems, such as supercomputers and clusters. Thus, we have used workloads with other characteristics in order to verify the quality of the proposed algorithms on different environments.

5 Conclusion and Future Work

In this paper we studied the scheduling of moldable BSP parallel tasks. First we showed that the problem is NP-hard, and then we provided a 2-approximation algorithm and some good heuristics. On the algorithms, the number of processors given to a task with n processes can range from 1 to n. However, due mainly to memory limitations this may not be feasible in practice. Moreover, with few

processors the task can be delayed for long time. Thus, as future work we intend to limit the minimal number of processors for a task in order to limit the maximal number of processes in each processor.

This work has as its final goal an implementation to be used to schedule parallel applications on our grid environment, InteGrade. Also as future works we intend to explore the possibilities provided by our grid environment, processors heterogeneity, parallel tasks preemption, and machine unavailability. For the last two cases we will study in detail the effects of interrupting a parallel task and possibly continue to execute it on a different number of processors, which is possible with the BSP synchronizations and our already implemented checkpointing library [26].

References

1. Foster, I., Kesselman, C., eds.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, San Francisco (2003)
2. Goldchleger, A., Kon, F., Goldman, A., Finger, M., Bezerra, G.C.: InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. Concurrency and Computation: Practice and Experience **16** (2004) 449–459
3. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33** (1990) 103–111
4. Goldchleger, A., Queiroz, C.A., Kon, F., Goldman, A.: Running highly-coupled parallel applications in a computational grid. In: Proceedings of the 22th Brazilian Symposium on Computer Networks. (2004)
5. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPlib: The BSP programming library. Parallel Computing **24** (1998) 1947–1980
6. Turek, J., Wolf, J.L., Yu, P.S.: Approximate algorithms for scheduling parallelizable tasks. In: Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, California, SIGACT/SIGARCH (1992) 323–332
7. Baker, R., Coffman, E.G., Rivest, R.L.: Orthogonal packings in two dimensions. SIAM Journal on Computing **9** (1980) 846–855
8. Coffman, E.G., Garey, M.R., Johnson, D.S., Tarjan, R.E.: Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing **9** (1980) 808–826
9. Ludwig, W.T.: Algorithms for scheduling malleable and nonmalleable parallel tasks. PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences (1995)
10. Steinberg, A.: A strip-packing algorithm with absolute performance bound 2. SIAM Journal on Computing **26** (1997) 401–409
11. Mounie, G., Rapine, C., Trystram, D.: Efficient approximation algorithm for scheduling malleable tasks. In: Proceedings of the 11th ACM Symposium of Parallel Algorithms and Architecture. (1999) 23–32
12. Mounié, G.: Efficient scheduling of parallel application : the monotonic malleable tasks. PhD thesis, Institut National Polytechnique de Grenoble (2000) Available in french only.
13. Mounie, G., Rapine, C., Trystram, D.: A $\frac{3}{2}$-approximation algorithm for independent scheduling malleable tasks. (Submitted for publication 2001)

14. Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proceedings of Supercomputing '93, IEEE Computer Society Press (1993) 878–883
15. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. Journal of Scientific Programming **6** (1997) 249–274
16. Sunderam, V.S.: PVM: a framework for parallel distributed computing. Concurrency, Practice and Experience **2** (1990) 315–340
17. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
18. Dehne, F.: Coarse grained parallel algorithms. Algorithmica Special Issue on "Coarse grained parallel algorithms" **24** (1999) 173–176
19. Gu, Y., Lee, B.S., Cai, W.: JBSP: A BSP Programming Library in Java. Journal of Parallel and Distributed Computing **61** (2001) 1126–1142
20. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The paderborn university bsp (pub) library. Parallel Computing **29** (2003) 187–207
21. Tong, W., Ding, J., Cai, L.: A parallel programming environment on grid. In: Proceedings of the International Conference on Computational Science. Volume 2657 of Lecture Notes in Computer Science., Springer (2003) 225–234
22. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of NP-completeness. W. H. Freeman, New York (1979)
23. Hochbaum, D.S., Shmoys, D.B.: Using dual approximation algorithms for scheduling problems: theoretical and practical results. Journal of the ACM **34** (1987) 144–162
24. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Transactions Parallel & Distributed Systems **12** (2001) 529–543
25. Cirne, W., Berman, F.: A model for moldable supercomputer jobs. In: Proceedings of the 15th International Parallel & Distributed Processing Symposium. (2001)
26. de Camargo, R.Y., Goldchleger, A., Kon, F., Goldman, A.: Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In: Proceedings of the 2nd workshop on Middleware for grid computing, New York, NY, USA, ACM Press (2004) 35–40

Evolving Toward the Perfect Schedule: Co-scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm

Thomas Phan[1], Kavitha Ranganathan[2], and Radu Sion[3]

[1] IBM Almaden Research Center
phantom@us.ibm.com
[2] IBM T.J. Watson Research Center
kavithar@us.ibm.com
[3] Stony Brook University
sion@cs.stonybrook.edu

Abstract. Traditional job schedulers for grid or cluster systems are responsible for assigning incoming jobs to compute nodes in such a way that some evaluative condition is met. Such systems generally take into consideration the availability of compute cycles, queue lengths, and expected job execution times, but they typically do not account directly for data staging and thus miss significant associated opportunities for optimisation. Intuitively, a tighter integration of job scheduling and automated data replication can yield significant advantages due to the potential for optimised, faster access to data and decreased overall execution time. In this paper we consider data placement as a first-class citizen in scheduling and use an optimisation heuristic for generating schedules. We make the following two contributions. First, we identify the necessity for co-scheduling job dispatching and data replication assignments and posit that simultaneously scheduling both is critical for achieving good makespans. Second, we show that deploying a genetic search algorithm to solve the optimal allocation problem has the potential to achieve significant speed-up results versus traditional allocation mechanisms. Through simulation, we show that our algorithm provides on average an approximately 20-45% faster makespan than greedy schedulers.

1 Introduction

Traditional job schedulers for grid or cluster systems are responsible for assigning incoming jobs to compute nodes in such a way that some evaluative condition is met, such as the minimisation of the overall execution time of the jobs or the maximisation of throughput or utilisation. Such systems generally take into consideration the availability of compute cycles, job queue lengths, and expected job execution times, but they typically do not account directly for data staging and thus miss significant associated opportunities for optimisation. Indeed, the impact of data and replication management on job scheduling behaviour

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 173–193, 2005.

has largely remained unstudied. In this paper we investigate mechanisms that simultaneously schedule both job assignments and data replication and propose an optimised co-scheduling algorithm as a solution.

This problem is especially relevant in data-intensive grid and cluster systems where increasingly fast wide-area networks connect vast numbers of computation and storage resources. For example, the Grid Physics Network [10] and the Particle Physics Data Grid [18] require access to massive (on the scale of petabytes) amounts of data files for computational jobs. In addition to traditional files, we further anticipate more diverse and widespread utilisation of other types of data from a variety of sources; for example, grid applications may use Java objects from an RMI server, SOAP replies from a Web service, or aggregated SQL tuples from a DBMS.

Given that large-scale data access is an increasingly important part of grid applications, it follows that an intelligent job-dispatching scheduler must be aware of data transfer costs because jobs must have their requisite data sets in order to execute. In the absence of such awareness, data must be manually staged at compute nodes before jobs can be started (thereby inconveniencing the user) or replicated and transferred by the system but with the data costs neglected by the scheduler (thereby producing sub-optimal and inefficient schedules). Intuitively, a tighter integration of job scheduling and automated data replication potentially yields significant advantages due to the potential for optimised, faster access to data and decreased overall execution time. However, there are significant challenges to such an integration, including the minimisation of data transfers costs, the placement scheduling of jobs to compute nodes with respect to the data costs, and the performance of the scheduling algorithm itself. Overcoming these obstacles involves creating an optimised schedule that minimises the submitted jobs' time to completion (the "makespan") that should take into consideration both computation and data transfer times.

Previous efforts in job scheduling either do not consider data placement at all or often feature "last minute" sub-optimal approaches, in effect decoupling data replication from job dispatching. Traditional FIFO and backfilling parallel schedulers (surveyed in [8] and [9]) assume that data is already pre-staged and available to the application executables on the compute nodes, while workflow schedulers consider only the precedence relationship between the applications and the data and do not consider optimisation, e.g. [13]. Other recent approaches for co-scheduling provide greedy, sub-optimal solutions, e.g. [4] [19] [16].

This work includes the following two contributions. First, we identify the necessity for co-scheduling job dispatching and data replication and posit that simultaneously scheduling both is critical for achieving good makespans. We focus on a massively-parallel computation model that comprises a collection of heterogeneous independent jobs with no inter-job communication. Second, we show that deploying a genetic search algorithm to solve the optimal allocation problem has the potential to achieve significant speed-up results. In our work we observe that there are three important variables within a job scheduling system, namely the job order in the global scheduler queue, the assignment of jobs to

compute nodes, and the assignment of data replicas to local data stores. There exists an optimal solution that provides the best schedule with the minimal makespan, but the solution space is prohibitively large for exhaustive searches. To find the best combination of these three variables in the solution space, we provide an optimisation heuristic using a genetic algorithm. By representing the three variables in a "chromosome" and allowing them to compete and evolve, the algorithm naturally converges towards an optimal (or near-optimal) solution.

We use simulations to evaluate our genetic algorithm approach against traditional greedy algorithms. Our experiments find that our approach provides on average an approximately 20-45% faster makespan than greedy schedulers. Furthermore, our work provides an initial promising look at how fine-tuning the genetic algorithm can lead to better performance for co-scheduling.

This paper is organised in the following manner. In Section 2 we discuss related work. We describe our model and assumptions in Section 3, present our genetic algorithm methodology in Section 4 and present the results of our simulation experiments in Section 5. We conclude the paper in Section 6.

2 Related Work

The need for scheduling job assignment and data placement together arises from modern clustered deployments. The work in [24] suggests I/O communities can be formed from compute nodes clustered around a storage system. Other researchers have considered the high-level problem of precedence workflow scheduling to ensure that data has been automatically staged at a compute node before assigned jobs at that node begin computing [7] [13]. Such work assumes that once a workflow schedule has been planned, lower-level batch schedulers will execute the proper job assignments and data replication. Our work fits into this latter category of job and data schedulers.

Other researchers have looked into the problem of job and data co-scheduling, but none have considered an integrated approach or optimisation algorithms to improve scheduling performance. The XSufferage algorithm [4] includes network transmission delay during the scheduling of jobs to sites but only replicates data from the original source repository and not across sites. The work in [19] looks at a variety of techniques to intelligently replicate data across sites and assign jobs to sites; the best results come from a scheme where local monitors keep track of popular files and preemptively replicate them to other sites, thereby allowing a scheduler to assign jobs to those sites that already host needed data. However, this work only considers jobs that use a single input file and assumes homogeneous network conditions. The Close-to-Files algorithm [16] assumes that single-file input data has already been replicated across sites and then uses an exhaustive algorithm to search across all combinations of compute sites and data sites to find the combination with the minimum cost, including computation and transmission delay. The Storage Affinity algorithm [21] treats file systems at each site as a passive cache; an initial job executing at a site must pull in data to the site, and subsequent jobs are assigned to sites that have the most amount of

needed residual data from previous application runs. The work in [5] decouples jobs scheduling from data scheduling: at the end of periodic intervals when jobs are scheduled, the popularity of needed files is calculated and then used by the data scheduler to replicate data for the next set of jobs, which may or may not share the same data requirements as the previous set.

Although these previous efforts have identified and addressed the problem of job and data co-scheduling, the scheduling is generally based on decoupled algorithms that schedule jobs in reaction to prior data scheduling. Furthermore, all these previous algorithms perform FIFO scheduling for only one job at a time, resulting in typically locally-optimum schedules only. On the other hand, we suggest a methodology to provide simultaneous co-scheduling in an integrated manner using global optimisation heuristics. In our work we execute a genetic algorithm that converges to a schedule by looking at the jobs in the scheduler queue as well as replicated data objects at once. While other researchers have looked at global optimisation algorithms for job scheduling [3] [22], they do not consider job and data co-scheduling. In the future, we plan to use simulations to compare the performance and benefits of our genetic algorithm with the other scheduling approaches listed above.

3 Job and Data Co-scheduling Model

Consider the scenario illustrated in Figure 1 that depicts a typical distributed grid or cluster deployment. Jobs are submitted to a centralised scheduler that queues the jobs until they are dispatched to distributed compute nodes. This scheduler can potentially be a meta-scheduler that assigns jobs to other local schedulers (to improve scalability at the cost of increased administration), but

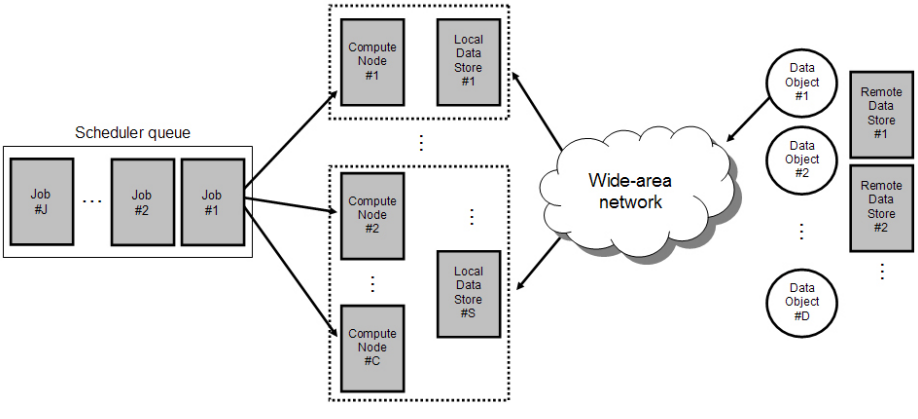


Fig. 1. A high-level overview of a job submission system in a generalised distributed grid. Note that although our work can be extended to multiple LANs containing clusters of compute nodes and local data stores (as is depicted here), for simplicity in this paper we consider only a single LAN.

in our work we consider only a single centralised scheduler responsible for assigning jobs; in future work we look to extend this model to a decentralised meta-scheduling system.

The compute nodes are supported by local data stores capable of caching read-only replicas of data downloaded from remote data stores. These local data stores, depending on the context of the applications, can range from web proxy caches to data warehouses. We assume that the compute nodes and the local data stores are connected on a high-speed LAN (e.g. Ethernet or Myrinet) and that data can be transferred across the stores. (The model can be extended to multiple LANs containing clusters of compute nodes and data stores, but for simplicity we assume a single LAN in this paper.) Data downloaded from the remote store must cross a wide-area network such as the Internet. In the remainder of this paper, we use the term "data object" [23] to encompass a variety of potential data manifestations, including Java objects and aggregated SQL tuples, although its meaning can be intuitively construed to be a file on a file system.

Our model relies on the following key assumptions on the class of jobs being scheduled and the facilities available to the scheduler:

- √ The jobs are from a collection of heterogeneous independent jobs with no inter-job communication. As such, we do not consider jobs with parallel tasks (e.g. MPI programs).
- √ Data retrieved from the remote data stores is read-only. We only consider the class of applications that do not write back to the remote data store; for these applications, computed output is typically directed to the local file system at the compute nodes, and such output is commonly much smaller and negligible compared to input data.
- √ The computation time required by a job is known to the scheduler. In practical terms, when jobs are submitted to a scheduler, the submitting user typically assigns an expected duration of usage to each job [17].
- √ The data objects required to be downloaded for a job are known to the scheduler and can be specified at the time of job submission.
- √ The local data stores are assumed to have enough secondary storage to hold all data objects. In a more realistic setting of limited storage, a policy like LRU could be implemented for storage management.
- √ The communication cost for acquiring this data can be calculated for each job. The only communication cost we consider is transmission delay, which can be computed by dividing a data object's size by the bottleneck bandwidth between a sender and receiver. As such, we do not consider queueing delay or propagation delay.
  - If the data object is a file, its size is typically known to the job's user and specified at submission time. On the other hand, if the object is produced dynamically by a remote server, we assume that there exists a remote API that can provide the approximate size of the object. For example, for data downloads from a web server, one can use HTTP's HEAD method to get the requested URI's size prior to actually downloading it.

- The bottleneck bandwidth between two network points can be ascertained using known techniques [12] [20] that typically trade off accuracy with convergence speed. We assume such information can be periodically updated by a background process and made available to the scheduler.

√ Finally, we do not include arbitrarily detailed delays and costs in our model (e.g. database access time, data marshalling, or disk rotational latency), as these are dominated by transmission delay and computation time.

Given these assumptions, the lifecycle of a submitted job proceeds as follows. When a job is submitted to the queue, the scheduler assigns it to a compute node (using a traditional load-balancing algorithm or the algorithm we discuss in this paper). Each compute node maintains its own queue from which jobs run in FIFO order. Each job requires data objects from remote data stores; these objects can be downloaded and replicated on-demand to one of the local data stores (again, using a traditional algorithm or the algorithm we discuss in this paper), thereby obviating the need for subsequent jobs to download the same objects from the remote data store. In our work we associate a job to its required data objects through a Zipf distribution. All required data must be downloaded before a job can begin, and objects are downloaded in parallel at the time that a job is run. (Although parallel downloads will almost certainly reduce the last hop's bandwidth, for simplicity we assume that the bottleneck bandwidth is a more significant concern.) A requested object will always be downloaded from a local data store, if it exists there, rather than from the remote store. If a job requires an object that is currently being downloaded by another job executing at a different compute node, the job either waits for that download to complete or instantiates its own, whichever is faster based on expected download time maintained by the scheduler.

Intuitively, it can be seen that if jobs are assigned to compute nodes first, the latency incurred from accessing data objects may vary drastically because the objects may or may not have been already cached at a close local data store. On the other hand, if data objects are replicated to local data stores first, then the subsequent job executions will be delayed due to these same variations in access costs. Furthermore, the ordering of the jobs in the queue can affect the performance. For example, if job A is waiting for job B (on a different compute node) to finish downloading an object, job A blocks any other jobs from executing on its compute node. Instead, if we rearrange the job queue such that other shorter jobs run before job A, then these shorter jobs can start and finish by the time job A is ready to run. (This approach is similar to backfilling algorithms [14] that schedule parallel jobs requiring multiple processors.) The resulting tradeoffs affect the makespan.

With this scenario as it is illustrated in Figure 1, it can be seen that there are three independent variables in the system, namely (1) the ordering of the jobs in the global scheduler's queue, which translates to the ordering in the individual queue at each compute node, (2) the assignment of jobs in the queue to the individual compute nodes; and (3) the assignment of the data object replicas to the local data stores. The number of combinations can be determined as follows:

- √ Suppose there are J jobs in the scheduler queue. There are then $J!$ ways to arrange the jobs.
- √ Suppose there are C compute nodes. There are then C^J ways to assign the J jobs to these C compute nodes.
- √ Suppose there are D data objects and S local data stores. There are then S^D ways to replicate the D objects onto the S stores.

There are thus $J! \cdot C^J \cdot S^D$ different combinations of these three assignments. Within this solution space there exists some tuple of {job ordering, job-to-compute node assignment, object-to-local data store assignment} that will produce the minimal makespan for the set of jobs. However, for any reasonable deployment instantiation (e.g. J=20 and C=10), the number of combinations becomes prohibitively large for an exhaustive search.

Existing work in job scheduling can be analysed in the context presented above. Prior work in schedulers that dispatch jobs in FIFO order eliminate all but one of the $J!$ job orderings possible. Schedulers that assume the data objects have been preemptively assigned to local data stores eliminate all but one of the S^D ways to replicate. Essentially all prior efforts have made assumptions that allow the scheduler to make decisions from a drastically reduced solution space that may or may not include the optimal schedule.

The relationship between these three variables is intertwined. Although they can be changed independently of one another, adjusting one variable will have an adverse or beneficial effect on the schedule's makespan that can be counterbalanced by adjusting another variable. We analyse this interplay in Section 5 on results.

4 Methodology: A Genetic Algorithm

With a solution space size of $J! \cdot C^J \cdot S^D$, the goal is to find the schedule in this space that produces the shortest makespan. To achieve this goal, we use a genetic algorithm [2] as a search heuristic. While other approaches exist, each has its limitations. For example, an exhaustive search, as mentioned, would be pointless given the potentially huge size of the solution space. An iterated hill-climbing search samples local regions but may get stuck at a local optima. Simulated annealing can break out of local optima, but the mapping of this approach's parameters, such as the temperature, to a given problem domain is not always clear.

4.1 Overview

A genetic algorithm (GA) simulates the behaviour of Darwinian natural selection and converges toward an optimal solution through successive generations of recombination, mutation, and selection, as shown in the pseudocode of Figure 2 (adapted from [15]). A potential solution in the problem space is represented as a chromosome. In the context of our problem, one chromosome is a schedule that consists of string representations of a tuple of {queue order, job assignments, object assignments}.

```
Procedure genetic algorithm
{
  t = 0;
  initialise P(t);
  evaluate P(t);
  while (! done)
  {
        alter P(t);
        t = t + 1;
        select P(t) from P(t - 1);
        evaluate P(t);
  }
}
```

Fig. 2. Pseudocode for a genetic search algorithm. In this code, the variable t represents the current generation and P(t) represents the population at that generation.

Initially a random set of chromosomes is instantiated as the population. The chromosomes in the population are evaluated (hashed) to some metric, and the best ones are chosen to be parents. In our context, the evaluation produces the makespan that results from executing the schedule of a particular chromosome. The parents recombine to produce children, simulating sexual crossover, and occasionally a mutation may arise which produces new characteristics that were not present in either parent; for simplification, in this work we did not implement the optional mutation. The best subset of the children is chosen, based on an evaluation function, to be the parents of the next generation. We further implemented elitism, where the best chromosome is guaranteed to be included in each generation in order to accelerate the convergence to an optimum, if it is found. The generational loop ends when some criteria is met; in our implementation we terminate after 100 generations (this value is an arbitrary number, as we had observed that it is large enough to allow the GA to converge). At the end, a global optimum or near-optimum is found. Note that finding the global optimum is not guaranteed because the recombination has probabilistic characteristics.

Using a GA is naturally suited in our context. The job queue, job assignments, and object assignments can be intuitively represented as character strings, which allows us to leverage prior genetic algorithm research in how to effectively recombine string representations of chromosomes (e.g. [6]).

It is important to note that a GA is most effective when it operates upon a large collection of possible solutions. In our context, the GA should look at a large window of jobs at once in order to achieve the tightest packing of jobs into a schedule. In contrast, traditional FIFO schedulers consider only the front job in the queue. The optimising scheduler in [22] uses dynamic programming and considers a large group of jobs called a "lookahead," on the order of 10-50 jobs. In our work we call the collection of jobs a snapshot window. The scheduler takes this snapshot of queued jobs and feeds it into the scheduling algorithm.

Our simulation thus only models one static batch of jobs in the job queue. In the future, we will look at a more dynamic situation where jobs are arriving

even as the current batch of jobs is being evaluated and dispatched by the GA. In such an approach, there will be two queues, namely one to hold incoming jobs and another to hold the latest snapshot of jobs that had been taken from the first queue. Furthermore, note that taking the snapshot can vary in two ways, namely by the frequency of taking the snapshot (e.g. at periodic wallclock intervals or when a particular queue size is reached) or by the size of the snapshot window (e.g. the entire queue or a portion of the queue starting from the front).

4.2 Workflow

The objective of the genetic algorithm is to find a combination of the three variables that minimises the makespan for the jobs. The resulting schedule that corresponds to the minimum makespan will be carried out, with jobs being executed on compute nodes and data objects being replicated to data stores in order to be accessed by the executing jobs. At a high level, the workflow proceeds as follows:

i. Jobs requests enter the system and are queued by the job scheduler.
ii. The scheduler takes a snapshot of the jobs in the queue and gives it to the scheduling algorithm.
iii. Given a snapshot, the genetic algorithm executes. The objective of the algorithm is to find the minimal makespan. The evaluation function, described in subsection 4.5, takes the current instance of the three variables as input and returns the resulting makespan. As the genetic algorithm executes, it will converge to the schedule with the minimum makespan.
iv. Given the genetic algorithm's output of an optimal schedule consisting of the job order, job assignments, and object assignments, the schedule is executed. Jobs are dispatched and executed on the compute nodes, and the data objects are replicated on-demand to the data stores so they can be accessed by the jobs.

4.3 Chromosomes

As mentioned previously, each chromosome consists of three strings, corresponding to the job ordering, the assignment of jobs to compute nodes, and the assignment of data objects to local data stores. We can represent each one as an array of integers. For each type of chromosome, recombination and mutation can only occur between strings representing the same characteristic. The initial state of the GA is a set of randomly initialised chromosomes.

Job ordering. The job ordering for a particular snapshot window can be represented as a queue (vector) of job unique identifiers. Note that the jobs can have their own range of identifiers, but once they are in the queue, they can be represented by a simpler range of identifiers going from job 0 to J-1 for a snapshot of J jobs. The representation is simply a vector of these identifiers. An example queue is shown in Figure 3.

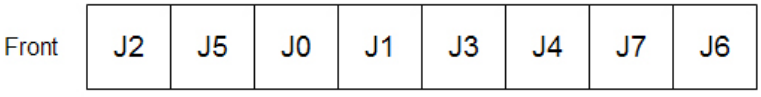


Fig. 3. An example queue of 8 jobs

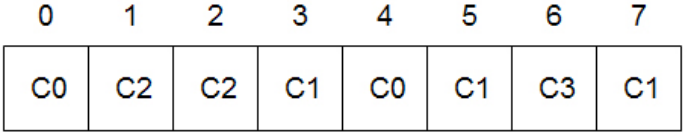


Fig. 4. An example mapping of 8 jobs to 4 compute nodes

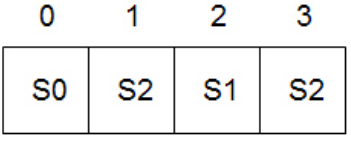


Fig. 5. An example assignment of 4 data objects to 3 local data stores

Assignment of jobs to compute nodes. The assignments can be represented as an array of size J, and each cell in the array takes on a value between 0 and C-1 for C compute nodes. The i^{th} element of the array contains an identifier for the compute node to which job i has been assigned. An example assignment is shown in Figure 4.

Assignment of data object replicas to local data store. Similarly, these assignments can be represented as an array of size D for D objects, and each cell can take on a value between 0 and S-1 for S local data stores. The i^{th} element contains an integer identifier of the local data store to which object i has been assigned. An example assignment is shown in Figure 5.

4.4 Recombination and Mutation

Recombination is applied only to strings of the same type to produce a new child chromosome. In a two-parent recombination scheme for arrays of unique elements, we can use a 2-point crossover scheme where a randomly-chosen contiguous subsection of the first parent is copied to the child, and then all remaining items in the second parent (that have *not* already been taken from the first parent's subsection) are then copied to the child in order [6]. In a uni-parent mutation scheme, we can choose two items at random from an array and reverse the elements between them, inclusive. Note that in our experiments, we did not implement the optional mutation scheme, as we wanted to keep our GA as simple as possible in order to identify trends resulting from recombination. In the future we will explore ways of using mutation to increase the probability of finding global optima. Other recombination and mutation schemes are possible (as well as different chromosome representations) and will be explored in future work.

4.5 Evaluation Function

A key component of the genetic algorithm is the evaluation function. Given a particular job ordering, set of job assignments to compute nodes, and set of object assignments to local data stores, the evaluation function returns the makespan calculated deterministically from the algorithm described below. The rules use the lookup tables in Table 1. We note that the evaluation function is easily replaceable: if one were to decide upon a different model of job execution (with different ways of managing object downloads and executing jobs) or a different evaluation metric (such as response time or system saturation), a new evaluation function could just as easily be plugged into the GA as long as the same function is executed for all the chromosomes in the population.

At any given iteration of the genetic algorithm, the evaluation function executes to find the makespan of the jobs in the current queue snapshot. The pseudocode of the evaluation function is shown in Figure 6. We provide an overview of this function here.

The evaluation function considers all jobs in the queue over the loop spanning lines 6 to 37. As part of the randomisation performed by the genetic algorithm at a given iteration, the order of the jobs in the queue will be set, allowing the jobs to be dispatched in that order.

Table 1. Lookup tables used in the GA's evaluation function

Lookup table	Comment
REQUIRES (Job J_i, DataObject O_i)	1 if Job J_i requires/accesses Object O_i.
COMPUTE (Job J_i, ComputeNode C_i)	The time for Job J_i to execute on compute node C_i.
BANDWIDTH (Site a, Site b)	The bottleneck bandwidth between two sites. The sites can be data stores or compute nodes.
SIZE (DataObject O_i)	The size of object O_i (e.g. in bytes).
NACT (ComputeNode C_i)	Next Available Compute Time: the next available time that a job can start on compute node C_i.
NAOT (Object O_i)	Next Available Object Time: the next available time that an object O_i can be downloaded.

In the loop spanning lines 11 to 29, the function looks at all objects required by the currently considered job and finds the maximum transmission delay incurred by the objects. Data objects required by the job must be downloaded to the compute node prior to the job's execution either from the data object's source data store or from a local data store. Since the assignment of data object to local data store is known during a given iteration of the GA, we can calculate the transmission delay of moving the object from the source data store to the assigned local data store (line 17) and then update the NAOT table entry corresponding to this object (lines 18-22). Note that the NAOT is the next available

```
01: int evaluate(Queue, ComputeNodeAssignments, DataStoreAssignments)
02: {
03:    makespan = 0;
04:    clock = getcurrenttime();
05:
06:    foreach job J in Queue
07:    {
08:        // This job J is assigned to compute node C.
09:
10:        maxTD = 0; // maximum transmission delay across all objects
11:        foreach object Oi required by this job
12:        {
13:            // This data object O resides originally in Ssource and is
14:            //   assigned to Sassigned.
15:
16:            // calculate the transmission delay for this object
17:            TD = SIZE(Oi) / BANDWIDTH(Ssource, Sassigned);
18:            if ((clock+TD) < NAOT(Oi))
19:            {
20:               NAOT(Oi) = clock + TD;
21:               // file transfer from Ssource to Sassigned would occur
                  here
22:            }
23:            finalHopDelay = SIZE(Oi) / BANDWIDTH(Sassigned, C); //
               optional
24:
25:            // keep track of the maximum transmission delay
26:            maxTD = MAX(maxTD, NAOT(Oi) + finalHopDelay);
27:
28:            // file transfer from Sassigned to compute node C would
               occur here
29:        }
30:
31:        startComputeTime = NACT(C)+ maxTD;
32:        completionTime = startComputeTime + COMPUTE(J, C);
33:        NACT(C) = MAX(NACT(C), completionTime);
34:
35:        // keep track of the largest makespan across all jobs
36:        makespan = MAX(makespan, completionTime);
37:    }
38:    return makespan;
39: }
```

Fig. 6. Evaluation function for the genetic algorithm

time that the object is available for a final-hop transfer to the compute node regardless of the local data store. The object may have already been transferred to a different store, but if the current job can transfer it faster to its assigned store, then it will do so (lines 18-22). Also note that if the object is assigned to

a local data store that is on the compute nodes' LAN, then the object must still be transferred across one more hop to the compute node (see line 23 and 26).

Lines 31 and 32 compute the start and end computation time for the job at the compute node. Line 36 keeps track of the largest completion time seen so far for all the jobs. Line 38 returns the resulting makespan, i.e. the longest completion time for the current set of jobs.

5 Experiments and Results

To show the effectiveness of the GA in improving the scheduling, we simulated our GA and a number of traditional greedy FIFO scheduler algorithms that dispatch jobs (to random or to least-loaded compute nodes) and replicate data objects (no replication or to random local data stores). We used a simulation program developed in-house that maintains a queue for the scheduler, queues for individual compute nodes, and simulation clocks that updates the simulation time as the experiments progressed. We ran the simulations on a Fedora Linux box running at 1 Ghz with 256 MB of RAM.

5.1 Experimental Setup

Our aim is to compare the performance of different algorithms to schedule jobs. Since all the algorithms use some randomisation in their execution, it was important to normalise the experiments to achieve results that could be compared across different schemes. We thus configured the algorithm simulations to initially read in startup parameters from a file (e.g. the jobs in the queue, the job assignments, the object assignments, etc.) that were all randomly determined beforehand. All experiments were performed with three different initialisation sets with ten runs each and averaged; the graphs represent this final average for any particular experiment. The experimental parameters were set according to values shown in Table 2.

Table 2. Experimental parameters

Experimental parameter	Comment
Queue size	Varies by experiment; 40-160
Number of compute nodes	Varies; 5-20
Number of local data stores	Varies; 5-20
Number of remote data stores	20
Number of data objects	50
Data object popularity	Based on Zipf distribution
Average object size	Uniformly distributed, 50-1500 MB
Average remote-to-local store bandwidth	Uniformly distributed, 700-1300 kbps
Average local store-to-compute node bandwidth	Uniformly distributed, 7000-13000 kbps
GA: number of parents	Varies; typically 10
GA: number of children	Varies; typically 50
GA: number of generations	100

The simulations use a synthetic benchmark based on CMS experiments [11] that are representative of the heterogeneous independent tasks programming model. Jobs download a number of data objects, perform execution, and terminate. Data objects are chosen based on a Zipf distribution [1]. The computation time for each job is kD seconds, where k is a unitless coefficient and D is the total size of the data objects downloaded in GBytes; in our experiments k is typically 300 (although in subsection 5.2 this value is varied).

5.2 Results

We first wanted to compare the GA against several greedy FIFO scheduling algorithms. In the experiments the naming of the algorithms is as follows:

√ Genetic algorithms (2 variations):
- all varying: the genetic algorithm with all three variables allowed to evolve
- rep-none: the genetic algorithm with the job queue and the job assignments allowed to evolve, but the objects are not replicated (a job must always download the data object from the remote data store)

√ Greedy algorithms (2x2 = 4 variations):
Job dispatching strategies
- jobs-LL: jobs are dispatched in FIFO order to the compute node with the shortest time until next availability
- jobs-rand: jobs are dispatched in FIFO order to a random compute node

Data replication strategies
- rep-none: objects are not replicated (a job must always download the data object from the remote data store)
- rep-rand: objects are replicated to random local data stores

Makespans for Various Algorithms. In this experiment, we ran the six algorithms with 20 compute nodes, 20 local data stores, and 100 jobs in the queue. Two results, as shown in Figure 7, can be seen. First, as expected, data placement/replication has a strong impact on the resulting makespan. Comparing the three pairs of experiments that vary by having replication activated or deactivated, namely (1) GA all varying and GA rep-none, (2) Greedy, jobs-LL, rep-none and Greedy, jobs-LL, rep-rand, and (3) Greedy, jobs-rand, rep-none and Greedy, jobs-rand, rep-rand, we can see that in the absence of an object replication strategy, the makespan suffers. Adding a replication strategy improves the makespan because object requests can be fulfilled by the local data store instead of by the remote data store, thereby reducing access latency as well as actual bandwidth utilisation (this latter reduction is potentially important when bandwidth consumption is metered).

The second result from this experiment is that the GA with all varying parameters provides the best performance of all the algorithms. Its resulting makespan is 22% faster than the best greedy algorithm (Greedy, jobs-LL, rep-rand) and 47% faster than the worst greedy algorithm (Greedy, jobs-rand, rep-none). To

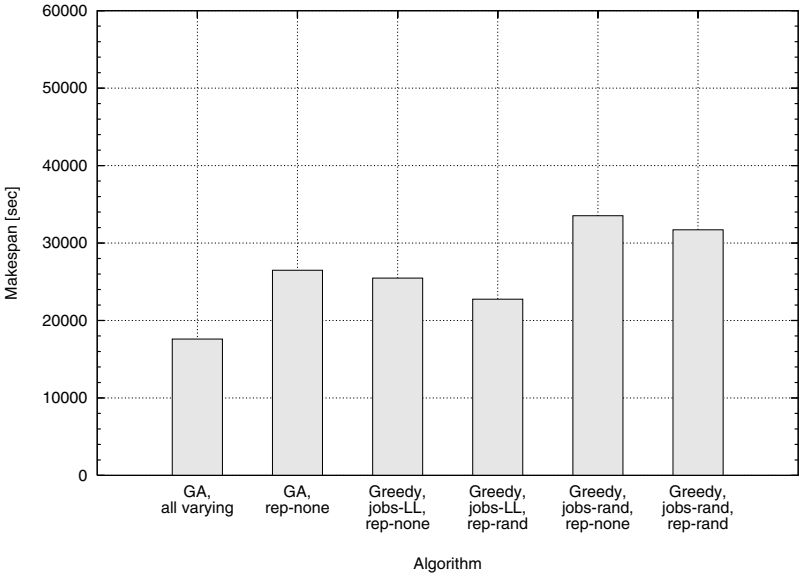


Fig. 7. Makespans for various algorithms using 20 compute nodes and 20 local data stores

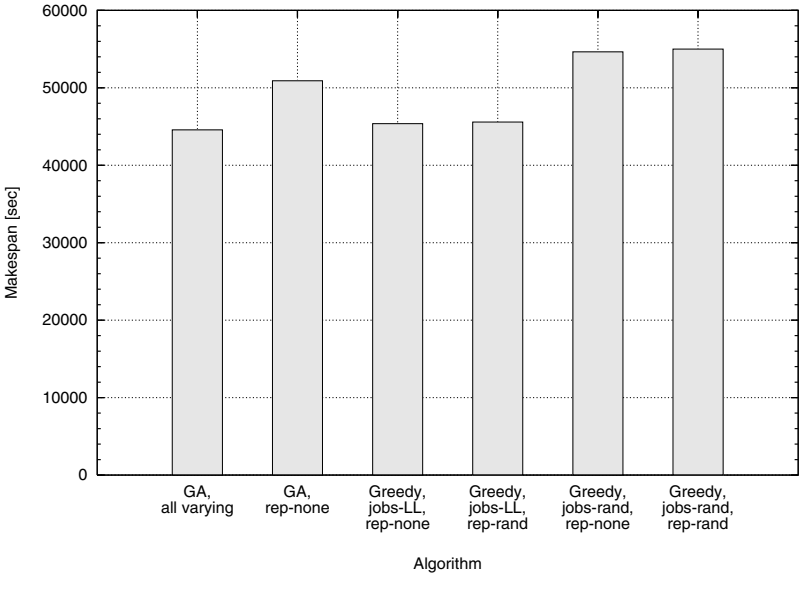


Fig. 8. Makespans for various algorithms using 5 compute nodes and 5 local data stores

better explain the result of why the GA is faster than the greedy algorithm, we ran another experiment with 5 compute nodes and 5 local data stores, as shown in Figure 8.

As can be seen, the performance of the GA is comparable to that of the greedy algorithms. This result is due to the fact that with the reduced number of compute nodes and local data stores, the solution space becomes smaller, and both types of algorithms become more equally likely to come across an optimum solution. If we restrict our attention to just the assignment of the 100 jobs in the queue, in the previous experiment with 20 compute nodes there are 20^{100} possible assignments, whereas with 5 compute nodes there are only 5^{100} possible assignments, a difference in the order of 10^{60}. With the larger solution space in the previous experiment, the variance of makespans will be larger, thus allowing the GA to potentially find a much better solution. It can be seen that in these scenarios where the deployment configuration of the grid system contains a large number of compute nodes and local data stores, a GA approach tends to compute better schedules.

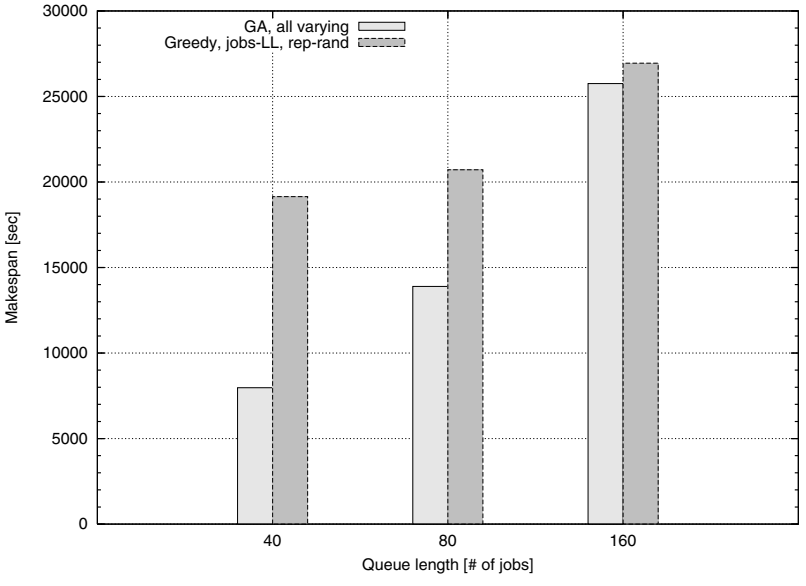


Fig. 9. Makespans for different queue lengths

Effect of Queue Length. In this experiment we ran the same application but with varying numbers of jobs in the queue and with 20 compute nodes and 20 local data stores; Figure 9 shows the results. For conciseness, we show only the best GA (GA all varying) and the best greedy algorithm (Greedy, jobs-LL, rep-rand). It can be seen that the GA performs consistently better than the greedy algorithm, although with an increasing number of jobs in the queue, the difference between the two algorithms decreases. We suspect that as more jobs are involved, the number of permutations increases dramatically (from 40! to 160!), thereby producing too large of a solution space for the GA to explore in 100 generations. Although in the previous subsection we observed that increasing the solution space provides a more likely chance of finding better solutions,

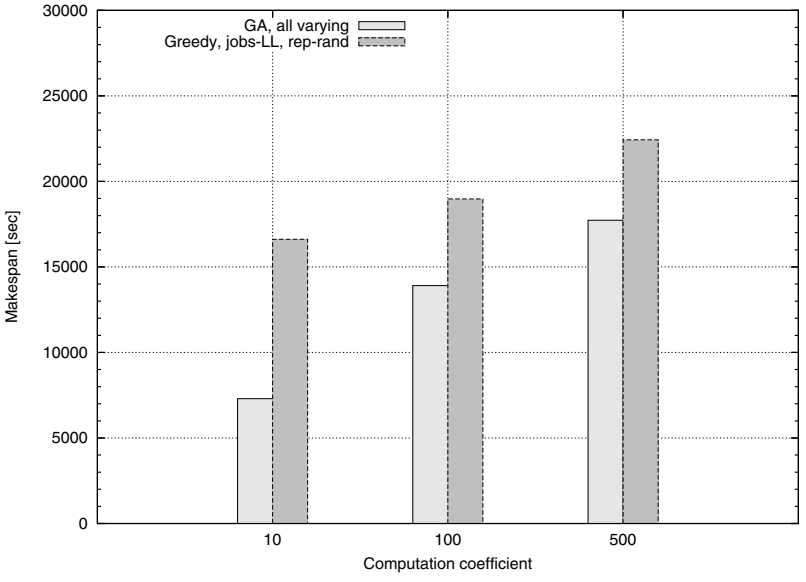


Fig. 10. Makespans for different computation coefficients

we conjecture that there is a trade-off point somewhere; we are continuing to investigate this issue.

Effect of Computation Ratio Coefficients. In previous experiments we set the computation coefficient to be 300 as mentioned in subsection 5.1. In Figure 10 we show the effect of changing this value. With a smaller coefficient, jobs contain less computation with the same amount of communication delay, and with a larger coefficient, jobs contain more computation. As can be seen, as the coefficient increases, the difference between the GA and the greedy algorithms decreases. This result stems from the fact that when there are more jobs with smaller running times (which includes both computation and communication), the effect of permuting the job queue is essentially tantamount to that of backfilling in a parallel scheduler: when a job is delayed waiting, other smaller jobs with less computation can be run before the long job, thereby reducing the overall makespan.

Effect of Population Size. In Figure 11 we show the effect of population size on the makespan produced by the GA. In all previous experiments, we had been running with a population comprising 10 parents spawning 50 children per generation. We can change the population characteristics by varying two parameters: the number of children selected to be parents per generation and the ratio of parents to children produced. The trend shown in the figure is that as the population size increases, there are more chromosomes from which to choose, thereby increasing the probability that one of them may contain the optimum

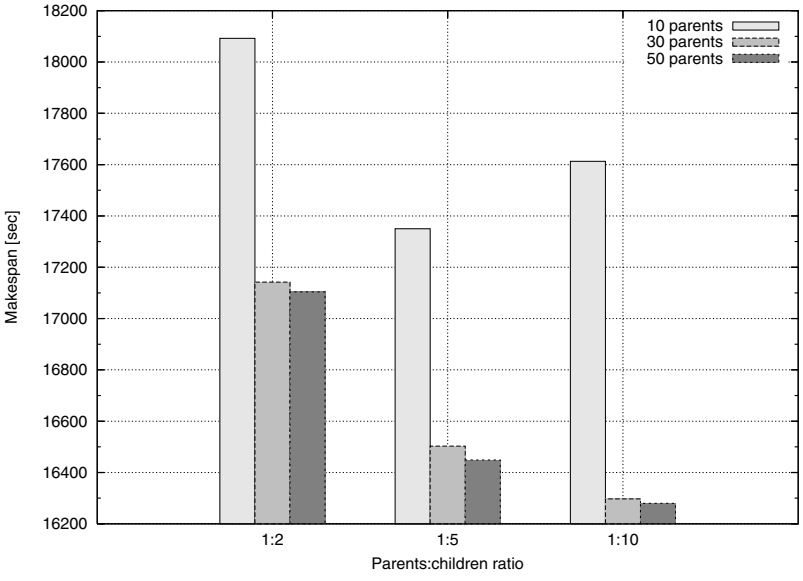


Fig. 11. Makespans for different GA ratios of parents to children

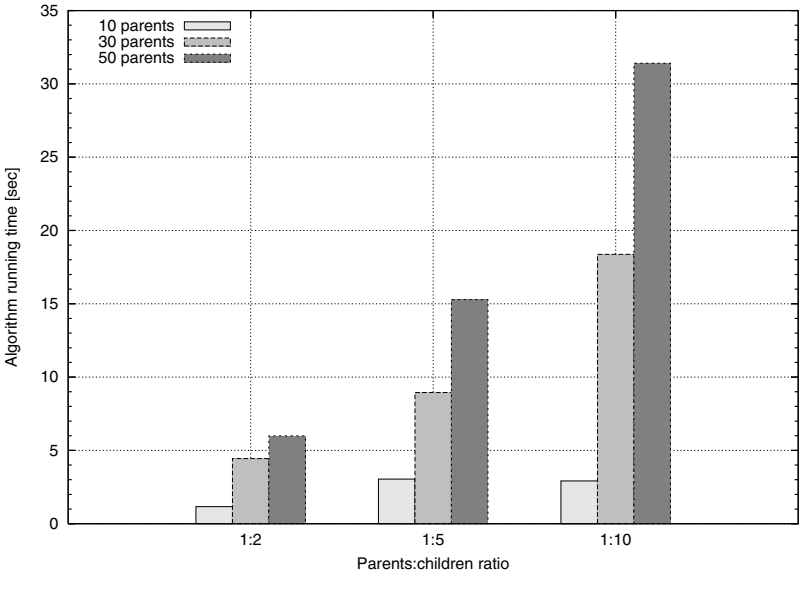


Fig. 12. GA running times for different ratios of parents to children

solution. As expected, the best makespan results from the largest configuration in the experiment, 50 parents and a ratio of 1 parent to 50 children.

However, this accuracy comes at the cost of increased running time of the algorithm. As the population size increases, the time to execute the evaluation

function on all members increases as well. As can be seen in Figure 12, the running time accordingly increases with the population size. This tradeoff of running time against the desire to find the optimal solution can be made by the scheduler's administrator. For completeness, we note that the greedy algorithms typically executed in under 1 second. While this performance is faster than that of the GA, this distinction is dwarfed by the difference between the makespans produced by greedy algorithms and the GA; as was shown in Figure 4 for this benchmark, the makespan difference can be on the order of thousands of seconds.

6 Conclusion and Future Work

In this paper we looked at the problem of co-scheduling job dispatching and data replication in wide-area distributed systems in an integrated manner. In our model, the system contains three variables, namely the order of the jobs in the global scheduler queue, the assignment of jobs to the individual compute nodes, and the assignment of the data objects to the local data stores. The solution space is enormous, making an exhaustive search to find the optimal tuple of these three variables prohibitively costly. In our work we showed that a genetic algorithm is a viable approach to finding the optimal solution. Our simulations show our implementation of a GA produces a makespan that is 20-45% faster than traditionally-used greedy algorithms.

For future work, we plan to do the following:

- √ More comprehensive comparisons. We look to simulate other approaches that can be used to perform co-scheduling, including those found in the related work section as well as other well-known scheduling algorithms, such as traditional backfilling, shortest-job-first, and priority-based scheduling.
- √ Handling inaccurate estimates. Our evaluation function used in the GA relies on the accuracy of the estimates for the data object size, bottleneck bandwidth, and job computation time. However, these estimates may be extremely inaccurate, leading the GA to produce inefficient schedules. In the future we will look into implementing a fallback scheduling algorithm, such as those in the related work, when the scheduler detects widely fluctuating or inaccurate estimates. Additionally, we will research different evaluation functions and metrics that may not be dependent on such estimates.
- √ Improved simulation. We plan to run a more detailed simulation with real-world constraints in our model. For example, we are looking at nodal topologies, more accurate bandwidth estimates, and more detailed evaluation functions that consider finer-grained costs and different models of job execution.
- √ More robust GA. Alternative genetic algorithm methodologies will also be explored, such as different representations, evaluation functions, alterations, and selections. Furthermore, we conjecture that since all three variables in the chromosome were independently evolved, there may be conflicting interplay between them. For instance, as the job queue permutations evolves to an optimum, the job assignments may have evolved in the opposite direction; the latter situation might occur because the job queue evolution has

a greater impact on the evaluation function. In the future we will look into ways of hashing all three variables into a single string for the chromosome so that there will be reduced interplay.

Acknowledgments

We would like to thank the paper reviewers for their invaluable comments and insight.

References

1. L. Adamic. "Zipf, Power-laws, and Pareto – a ranking tutorial," `www.hpl.hp.com/research/idl/papers/ranking/ranking.html`
2. T. Baeck, D. Fogel, and Z. Michalewicz (eds). "Evolutionary Computation 1: Basic Algorithms and Operators," Institute of Physics Publishing, 2000.
3. T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hengsen, and R. Freund. "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," Journal of Parallel and Distributed Computing, vol. 61, no. 6, June 2001.
4. H. Casanova, A. Legrand, D. Zagorodnov, F. Berman. "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," In *Proceedings of the 9th Heterogeneous Computing Workshop*, May 2000.
5. A. Chakrabarti, D. R. A., and S. Sengupta. "Integration of Scheduling and Replication in Data Grids," In *Proceedings of the International Conference on High Performance Computing*, 2004.
6. L. Davis. "Job Shop Scheduling with Genetic Algorithms," In *Proceedings of the International Conference on High Performance Computing*, 1985.
7. E. Deelman, T. Kosar, C. Kesselman, and M. Livny. "What Makes Workflows Work in an Opportunistic Environment?" Concurrency and Computation: Practice and Experience, 2004.
8. D. Feitelson. "A Survey of Scheduling in Multiprogrammed Parallel Systems," *IBM Research Report RC 19790 (87657)*, 1994.
9. D. Feitelson, L. Rudolph, and U. Schwiegelshohn. "Parallel Job Scheduling – A Status Report," In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.
10. The Grid Physics Project. `www.griphyn.org`
11. K. Holtman. "CMS Requirements for the Grid," In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, 2001.
12. N. Hu, L. Li, Z. Mao, P. Steenkiste, and J. Wang. "Locating Internet Bottlenecks: Algorithms, Measurements, and Implications," In *Proceedings of SIGCOMM*, 2004.
13. T. Kosar and M. Livny. "Stork: Making Data Placement a First Class Citizen in the Grid," In *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2004.
14. D. Lifka. "The ANL/IBM SP Scheduling System," In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes on Compute Science, Springer-Verlag 1995.
15. Z. Michaelewicz and D. Fogel. How to Solve It: Modern Heuristics, Springer-Verlag, 2000.

16. H. Mohamed and D. Epema. "An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters," In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.
17. A. Mu'alem and D. Feitelson. "Utilization, Predictability, Workloads,and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," IEEE Transactions on Parallel and Distributed Systems, June 2001.
18. The Particle Physics Data Grid, `www.ppdg.net`
19. K. Ranganathan and I. Foster. "Computation Scheduling and Data Replication Algorithms for Data Grids," Grid Resource Management: State of the Art and Future Trends, J. Nabrzyski, J. Schopf, and J. Weglarz, eds. Kluwer Academic Publishers, 2003.
20. V. Ribeiro, R. Riedi, and R. Baraniuk. "Locating Available Bandwidth Bottlenecks," *IEEE Internet Computing*, September-October 2004.
21. E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. "Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids," In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.
22. E. Schmueli and D. Feitelson. "Backfilling with Lookahead to Optimize the Packing of Parallel Jobs," Springer-Verlag Lecture Notes in Computer Science, vol. 2862, 2003.
23. H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. "File and Object Replication in Data Grids," In *Proceedings of the 10th International Symposium on High Performance Distributed Computing*, 2001.
24. D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. "Gathering at the Well: Creating Communities for Grid I/O", In Proceedings of Supercomputing, 2001.

Wave Scheduler: Scheduling for Faster Turnaround Time in Peer-Based Desktop Grid Systems

Dayi Zhou and Virginia Lo

University of Oregon, Eugene OR 97402, USA

Abstract. The recent success of Internet-based computing projects, coupled with rapid developments in peer-to-peer systems, has stimulated interest in the notion of harvesting idle cycles under a peer-to-peer model. The problem we address in this paper is the development of scheduling strategies to achieve faster turnaround time in an open peer-based desktop grid system. The challenges for this problem are two-fold: How does the scheduler quickly discover idle cycles in the absence of global information about host availability? And how can faster turnaround time be achieved within the opportunistic scheduling environment offered by volunteer hosts? We propose a novel peer-based scheduling method, *Wave Scheduler*, which allows peers to self organize into a timezone-aware overlay network using structured overlay network. The Wave Scheduler then exploits large blocks of idle night-time cycles by migrating jobs to hosts located in night-time zones around the globe, which are discovered by scalable resource discovery methods.

Simulation results show that the slowdown factors of all migration schemes are consistently lower than the slowdown factors of the non-migration schemes. Compared to traditional migration strategies we tested, the Wave Scheduler performs best. However under heavy load conditions, there is contention for those night-time hosts. Therefore, we propose an adaptive migration strategy for Wave Scheduler to further improve performance.

1 Introduction

It is widely acknowledged that a vast amount of idle cycles lie scattered throughout the Internet. The recent success of Internet-based computing projects such as SETI@home [1] and the Stanford Folding Project [2], coupled with rapid developments in peer-to-peer systems, has stimulated interest in the notion of harvesting idle cycles for desktop machines under a peer-to-peer model.

A peer-based desktop grid system allows cycle donors to organize themselves into an overlay network. Each peer is a potential donor of idle cycles as well as a potential source of jobs for automatic scheduling in the virtual resource pool. Many current research projects are exploring Internet-wide cycle-sharing using a peer-based model [3–6].

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 194–218, 2005.

The major benefits of peer-based desktop grid systems are that they are scalable and lightweight, compared with institutional-based Grid systems [7–9], load sharing systems in local networks [10, 11, 12, 13] and Internet-based global computing projects [1, 2, 14, 15, 16]. The latter systems do not scale well because they depend on centralized servers for scheduling and resource management and often incur overhead for negotiation and administration.[1]

However, to design scheduling methods satisfying jobs with fast turnaround requirements is a big challenge in dynamic, opportunistic peer-based desktop grid systems, which faces a number of unique challenges inherent to the nature of the peer-to-peer environment.

The challenge comes from the opportunistic and volatile nature of the peer-based desktop grid systems. Peer-based desktop grid systems use non-dedicated machines in which local jobs have much higher priority than foreign jobs. Therefore, compared to running on dedicated machines, the foreign job will make slower progress since it can only access a fraction of the host's CPU availability. The resources are highly volatile in peer-based desktop grid system. Nodes may leave and join the systems at any time, and resource owners may withdraw their resources at any time. Therefore, the foreign jobs may experience frequent failures due to the volatility of the resources.

Another challenge for design of an efficient scheduling system for peer-based desktop grid systems comes from the difficulties in collecting global and accurate resource information. It is unscalable to collect resource information on all the nodes in a large scale peer-based desktop grid system. Also users, especially home machine cycle donors, may find it is intrusive to report their CPU usage periodically to some remote clients. Therefore, scheduling in large scale cycle sharing systems are usually best effort scheduling based on limited resource information.

The problem we address in this paper is the development of scheduling strategies that achieve fast turnaround time and are deployable within a peer-based model. To our best knowledge, we are the first to study this scheduling problem in a fully distributed peer-based environment using Internet-wide cycle donors.

We propose *Wave Scheduler*, a novel scalable scheduler for peer-based desktop grid systems. Wave scheduler has two major components: a self-organized, timezone-aware overlay network and an efficient scheduling and migration strategy.

Self-organized Timezone-Aware Overlay Network. The Wave Scheduler allows hosts to organize themselves by timezone to indicate when they have large blocks of idle time. The Wave Scheduler uses a timezone-aware overlay network built on a structured overlay network such as CAN [17], Chord [18] or Pastry [19]. For example, a host in Pacific Time timezone can join the corresponding area in the overlay network to indicate that with high probability his machine will be idle from 8:00-14:00 GMT when he sleeps.

[1] A range of research issues faced by desktop grid systems are beyond the scope of this paper including incentives and fairness, security (malicious hosts, protecting the local host), etc.

Efficient Scheduling and Migration. Under the Wave Scheduler, a client initially schedules its job on a host in the current nighttime zone. When the host machine is no longer idle, the job is migrated to a new nighttime zone. Thus, jobs ride a wave of idle cycles around the world to reduce turnaround time.

A class of applications suitable for scheduling and migration in Wave Scheduler are long running *workpile* jobs. Workpile jobs, also known as *bag-of-tasks*, are CPU intensive and embarassingly parallel. For workpile jobs which run in the order of hours to days, the overheads of migration costs are negligible. Examples of such workpile applications include state-space search algorithms, ray-tracing programs, and long-running simulations.

The contributions of this paper include the following:

- Creation of an innovative scheduling method, the Wave scheduler, which exploits large chunk of idle cycles such as the nighttime cycles for fast turnaround.
- Analysis of a range of migration strategies, including migration under Wave Scheduler, with respect to turnaround time, success rate, and overhead.

2 Problem Description

The problem we address in this paper is the design of scheduling strategies for faster turnaround time for bag-of-tasks applications in large, open peer-based desktop grid systems. In this section, we discuss the key dimensions of the problem including the open peer-based cycle sharing infrastructure, the host availability model, and the characteristics of the applications supported by our scheduling strategies.

2.1 Open Peer-Based Desktop Grid System

The peer-based desktop grid system we are studying is open, symmetric, and fully distributed. In peer-based desktop grid systems, hosts join a community-based overlay network depending on their interests. Any client peer can submit applications; the application scheduler on the client will select a group of hosts whose resources match requirements of the application.

In a large, open peer-based desktop grid system, global resource discovery and scheduling of idle hosts using centralized servers is not feasible. Several research projects have addressed resource discovery for peer-to-peer cycle sharing [20, 4, 21]. One approach builds an auxiliary hierarchy among the peers such that a dynamic group of super-peers (similar to Gnutella's ultrapeers [22]) are designated to collect resource information and conduct the resource discovery on behalf of the other peers. Alternatively, each client peer uses a distributed scalable algorithm to discover available resources. These protocols are either probing based, such as expanding ring and random walk, or gossip based such as exchanging and caching resource advertisements.

On receiving a request for computational cycles from some client, the host returns resource information including CPU power, memory size, disk size, etc.

It also returns information regarding the current status of the machine regarding its availability to accept a foreign job. The client then chooses a host to schedule the job using its host selection criteria and waits for acknowledgment. The client then ships the job to the selected host. The job can be migrated to a new host if the current host becomes unavailable.

2.2 Host Availability Model

In open cycle sharing systems, users can make strict policies to decide when the host is available, which will limit the amount of cycles available to foreign jobs. A lesson learned from previous cycle sharing systems is that inconvenienced users will quickly withdraw from the system. To keep the users in the system, a cycle sharing system must be designed to preserve user control over her idle cycles and to cause minimal disturbance to the host machine.

The legendary Condor load sharing project [10, 8], developed at the University of Wisconsin in the 1980s for distributed systems, allows users to specify that foreign jobs should be preempted whenever mouse or keyboard events occur. Condor also supports strict owner policies in its *classified advertisement* of resource information [23]: users can rank foreign applications, specify a minimum CPU load threshold for cycle sharing, or specify specific time slots when foreign jobs are allowed to that host.

Another example of strict user-defined policies is the popular SETI@home project [1]. Users donate cycles by downloading a screensaver program from a central SETI@home server. The screensaver works like standard screensaver programs: it runs when no mouse or keyboard activities have been detected for a pre-configured time; otherwise it sleeps. SETI@home can also be configured to run in background mode, enabling it to continue computing all the time. However, screensaver mode is the default mode suggested by the SETI@home group and is the dominant mode employed by the SETI@home volunteer community.

As peer-based cycle sharing becomes more widespread, the majority of the users will most likely be more conservative than current users when donating

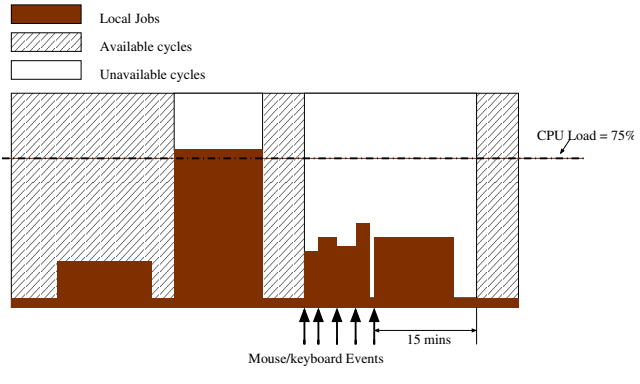


Fig. 1. A sample host profile of available idle cycles

cycles to anonymous clients in an open cycle sharing environment. The cycle sharing pattern will be cautiously generous: users are willing to join the system only if they know their own work will not be disturbed by foreign jobs. CPU cycle sharing will be likely limited to the time when owners are away from their machines and the CPU load from local applications is light. Figure 1 illustrates a sample host profile of available idle cycles under a strict user local policy: Host is available only when CPU load is less than 75% and there is no mouse or keyboard activity for 15 minutes.

2.3 Application Characteristic

The type of applications we are looking at in this study are large *Workpile (Bag-of-tasks)* jobs, requiring large amounts of CPU cycles but little if any data communication. Examples of workpile applications include state-space search algorithms, ray-tracing programs, gene sequencing and long-running simulations. Often these applications have higher performance requirements such as faster turnaround time and higher throughput. Some of the above applications may have real time constraints, such as weather forecasting, or deadlines such as scientific simulations that must be completed in time for a conference submission (such as JSSPP).

The migration cost is higher in global peer-based cycle sharing systems than in local area network as the codes and data are transferred on the Internet. If a short job that runs for a few minutes, is migrated many times in its life span, the accumulated migration cost may well counter the migration benefit. Long jobs which run for hours or even for months receive maximal benefit from migration schemes. For such jobs, the cost of migration, which includes resource discovery overhead to find a migration target, checkpointing, and cost to transfer the code and data is negligible compared to the total runtime of the job.

The size of many long running applications is small and these applications require minimal data communication. For example, the average data moved per CPU hour by users of SETI@home is only 21.25 KB, which is feasible even for users with slow dial-up connections. With respect to program size, Stanford Folding is only about 371KB, and SETI@home is around 791KB (because it includes the graphical interface for the screensaver). These applications run for a long time. Using the same SETI@home example, the average computation time of each job is over 6 hours (the major type of CPU in SETI@home is Intel x86).

3 Wave Scheduling for Workpile Applications

Wave scheduling is an idea that springs naturally from the observation that millions of machines are idle for large chunks of time. For example, most home machines and office machines lie idle at night. It is influenced by the notion of prime time v. non-prime time scheduling regimes enforced by parallel job schedulers [24], which schedules long jobs at night to improve turnaround time.

There are many motivations for the design of Wave Scheduler.

- First, resource information such as when the host will be idle and how long the host will continue to be idle with high probability will help the scheduler make much better decisions. Wave scheduler builds this information into the overlay network by having hosts organize themselves into the overlay network according to their timezone information.
- Second, efficient use of large and relatively stable chunks of continuing idle cycles provides the best performance improvement, while performance improvement by using sporadic and volatile small pieces of idle cycles in seconds or minutes is marginal and may be countered by high resource discovery and scheduling overhead. Therefore, the Wave scheduler proposes to use long idle night-time cycles.
- Third, the cycle donors are geographically distributed nodes, so that their idle times are well dispersed on the human time scale. Machines enter night-time in the order of the time zones around the world. In such a system, migration is an efficient scheme for faster turnaround.
- Fourth, the scheduler should be easy to install and not intrusive to users' privacy. The particular wave scheduler studied in this paper only needs minimal user input such as time zone information.

Wave Scheduler builds a timezone-aware, structured overlay network and it migrates jobs from busy hosts to idle hosts. Wave scheduler can utilize any structured overlay network such as CAN [17], Pastry [19], and Chord [18]. The algorithm we present here uses a CAN overlay [17] to organize nodes located in different timezones. In this section, we introduce the structured overlay network, and then we describe Wave Scheduler.

3.1 Structured Overlay Network

Structured overlay networks take advantage of the power of regular topologies: symmetric and balanced topologies, explicit label-based or Cartesian distance based routing and theoretical-bounded virtual routing latency and neighbor table size. In this section, we will describe the original CAN (Content Addressable Network) protocol, which is used by our Wave Scheduler.

The CAN structured overlay [17] uses a Cartesian coordinate space. The entire space is partitioned among all the physical nodes in the system at any time, and each physical node owns a distinct subspace in the overall space. Any coordinate that lies within its subspace can be used as an address to the physical node which owns the subspace. For example, in Figure 2, the whole Cartesian space is partitioned among 7 nodes, A,B,C,D,E,F and G. The neighbors of a given node in the CAN overlay are those nodes who are adjacent along $d-1$ dimensions and abut along one dimension. The neighbors of node G are nodes F, E, B, and C.

In Figure 2, new node N joins the CAN space by picking a coordinate in the Cartesian space and sending a message into the CAN destined for that coordinate. (The method for picking the coordinate is application-specific.) There is a bootstrap mechanism for injecting the message into the CAN space at a starting

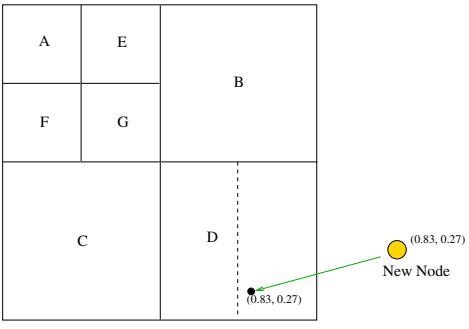


Fig. 2. Node join in CAN

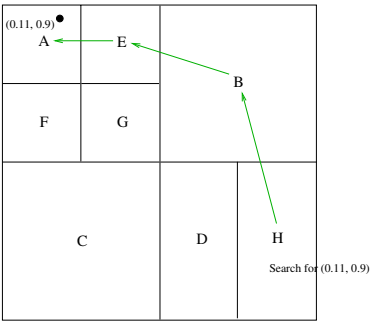


Fig. 3. Routing in CAN

node. The message is then routed from node to node through the CAN space until it reaches node D who owns the subspace containing N's coordinate. Each node along the way forwards the message to the neighbor that is closest in the Cartesian space to the destination. When the message reaches D, it then splits the CAN space with the new node N and adjusts the neighbor tables accordingly. Latency for the join operation is bounded by $O(n^{(1/d)})$ in which n is the number of peers in the overlay network and d is the number of dimensions of the CAN overlay network.

Figure 3 illustrates the coordinate-based routing from a source node to a destination node which uses the same hop by hop forwarding mechanism and is also bounded by $O(n^{(1/d)})$.

In this study, we utilize CAN's label-based routing for host discovery in all of our scheduling strategies. The CAN protocol is fully described by Ratnasamy et.al [17].

3.2 Wave Scheduling

In this section, we will present the Wave Scheduler, which takes advantage of the idle night cycles. Our wave scheduling protocol functions as follows (see Figure 4).

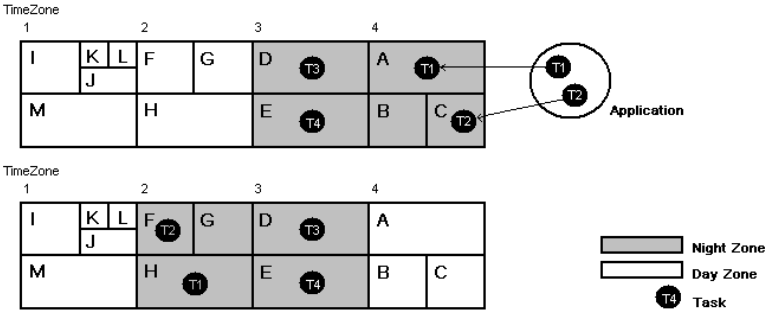


Fig. 4. Job initiation and migration in wave scheduling

- **Wavezones in the CAN overlay.** We divide the CAN virtual overlay space into several *wavezones*. Each *wavezone* represents several geographical timezones. A straightforward way to divide the CAN space is to select one dimension of the d-dimensional Cartesian space used by CAN and divide the space into several wavezones along that dimension. For example, a 1 x 24 CAN space could be divided into 4 wavezones each containing 6 continuous timezones.
- **Host nodes join the overlay.** A host node that wishes to offer its night-time cycles knows which timezone it occupies, say timezone 8. It randomly selects a node label in wavezone 2 containing timezone 8 such as (0.37, 7.12) and sends a join message to that node. According to the CAN protocol, the message will reach the physical node in charge of CAN node (0.37, 7.12) who will split the portion of the CAN space it owns, giving part of it to the new host node.
- **Client selects initial nightzone.** The scheduler for a workpile application knows which timezones are currently nightzones. It select one of these nightzones (based on some nightzone selection criteria) and decides on the number h of hosts it would like to target.
- **Selects set of target hosts.** The scheduler randomly generates a collection of h node labels in the wavezone containing the target nightzone and sends a request message to each target node label using CAN routing which finds the physical host which owns that node label. Or it does an expanding ring search starting from a random point in the target wavezone. After negotiations, the application scheduler selects a subset of those nodes to ship jobs to.
- **Migration to next timezone.** When morning comes to a host node and the host is no longer available, it selects a new target nightzone, randomly selects a host node in that nightzone for migration, and after negotiating with that host, migrates the unfinished job to the new host.

3.3 Extensions to Wave Scheduler

The *night-time* concept can be extended to any long interval of available time. The overlay does not necessary need to be timezone based but can be organized based

on available time intervals. For example, a user in Pacific Time timezone can register her home machine in a wavezone containing hosts idle from 0:00-10:00 GMT, when she knows she is at work during the daytime. Wave scheduler can even accept more complicated user profiles, which indicates users' daily schedules.

Wave Scheduling can also be easily leveraged to handle heterogeneity of the hosts in the system. Information such as operating systems, memory and machine types can be represented by further dividing the node label space or adding separate dimensions. For example, a third dimension is added to represent the operating system and it is divided into subspaces that represent each type of operating systems. When a client needs extra cycles, it can generate a node label. The first two dimensions are generated in the way described above and the third dimension has a value indicating the hosts need to be running a specific operating system. The third dimension can be empty, indicating the client does not care about the operating system type. In heterogeneous peer-based desktop grid system, the hosts have different CPU speeds. The difference between CPU speeds can be solved by normalizing the length of the idle time by the CPU speed and organize hosts according to this normalized profile information.

4 Peer-Based Scheduling Strategies That Utilize Migration

In this section, we describe the scheduling/migration strategies we evaluated in this paper. First we describe the key components that serve as building blocks for our scheduling strategies. Then we describe the peer-based migration strategies studied in this paper. In this study, we assume that all the migration schemes are built on a structured overlay network.

Our scheduling model is composed of the following four key components: host selection criteria, host discovery strategy, local scheduling policy, and migration scheme. When a client wants to schedule a job, the scheduler chooses the candidate host(s) satisfying the host selection criteria via host discovery. Then it schedules the job on the candidate host. If there are multiple candidate hosts, it will select one to schedule the job on. A migration scheme decides when and where to migrate the job. It uses host discovery and host selection strategy to decide where to migrate the host. Other modules including checkpointing, result verification and monitoring are out of the scope of this discussion.

Host Selection. A client uses its host selection criteria to decide whether the host can be a candidate, and it selects one of them on which to schedule a job if there are multiple candidates. We also assume that each host can host at most one foreign job at a time.

The following terms define the criteria we use in this study, motivated by our earlier discussion of strict user control. *Unclaimed* means that there is no foreign job on that host. *Available* means that there is no foreign job on that host and the host is idle. Local user policy can be made to decide whether the

host is idle based on criteria such as if the CPU load is below some threshold, or if there are no recent mouse/keyboard activities. The user policy can even blackout arbitrary time slots.

Different scheduling methods use different host selection criteria. Simple scheduling methods relax their hosts selection criteria to use any *unclaimed* hosts, while fast turnaround scheduling methods try to schedule foreign job on *available* hosts for instant execution.

In this study, we use a low-complexity host selection strategy when there are multiple candidates: a client selects the first discovered host that satisfies the particular host selection criteria used by the scheduling method.

Host Discovery. The purpose of the host discovery scheme is to discovery candidate hosts to accept the foreign job. The scheme needs to take into account the tradeoff between the message overhead and the success rate of the operation [20, 21]. Two schemes are used in this study.

Label-based random discovery. When the client needs extra cycles, the client randomly chooses a point in the CAN coordinate space and sends a request to that point. The peer owning that point will receive this request and return the resource information about whether it has already accepted a foreign job (*claimed*) and whether it is *available*. If the host does not satisfy the host selection criteria, the client can repeatedly generate another random point and contact another host. The maximum number of queries the client can issue is a configurable parameter.

Expanding Ring Search. When the client needs extra cycles, the client sends out a request with the host selection criteria to its direct neighbors. On receiving such request, if the criteria can be satisfied, the neighbor acknowledges the client. If the request is not satisfied, the client increases the search scope and forwards the request to its neighbors one-hop farther away. This procedure is repeated until the request is satisfied or the searching scope limit is reached. The client can choose to initiate the search in its own neighborhood, or it can choose a random point in the system (by generating a random node label and asking the owner of that random label to start the search). The benefit of the latter approach is to create a balanced load in cases of a skewed client request distribution in the CAN space.

Local Scheduling. The local scheduling policy on a host determines the type of service a host gives to a foreign job that it has accepted. Two common policies are: *screensaver* and *background*. With screensaver, foreign jobs can only run when there is no recent mouse/keyboard activity. With background, foreign jobs continue running as background processes with low priority even when users are present at their machines. We only consider the screensaver option in this study to reflect a conservative policy most likely in open peer-to-peer cycle sharing systems.

Note that under screensaver mode, the availability of a host to run the foreign job does not mean that the job receives 100% of the machine's CPU cycles. Instead the job concurrently shares cycles with other local jobs.

Migration. Migration was originally designed for load sharing in distributed computing to move active processes from a heavily loaded machine to a lightly loaded machine. Theoretical and experimental studies have shown that migration can be used to improve turnaround time [25, 26].

There are two important issues for migration schemes: *when* to migrate the jobs and *where* to migrate the jobs. Traditional load sharing systems used central servers or high overhead information exchange to collect resource information about hosts in the system to determine when and where to migrate jobs [25, 26]. New scalable strategies are needed to guide migration decisions in a peer-to-peer system.

The optimal solution of *when* to migration the job requires accurate predication of future resource availability on all the hosts. Many researchers have addressed the CPU availability prediction problem for the Grid or for load sharing systems [27, 28, 29], but they all require a central location to collect and process the resource information. In our study, we assume there is no resource availability prediction and that migration is a simple best effort decision based primarily on local information, e.g. when the host becomes unavailable due to user activity.

The same resource availability issue exists for *where* to migrate the job. But the issue of *where* to migrate the job is also related to the scalable host discovery which we have discussed. The scheduler needs the host discovery to discover candidate hosts which are suitable to migrate the job to.

We compare several migration schemes that differ regarding when to migrate and where to migrate.

The options for when to migrate include:

- *Immediate migration.* Once the host is no longer available, the foreign jobs are immediately migrated to another available host.
- *Linger migration.* Linger migration allows foreign jobs to linger on the host for a random amount of time after the host becomes unavailable. After lingering, if the host becomes available again, the foreign job can continue execution on that host. Linger migration avoids unnecessary migration as the host might only be temporarily unavailable. Linger migration can also be used to avoid network congestion or contention for available hosts when a large number of jobs need to be migrated at the same time.

There are also two options for where to migrate the jobs:

- *Random.* The new host is selected in a random area in the overlay network. There is no targeted area for the search; the new host is a random host found in the area where the resource discovery scheme is launched.
- *Night-time machines.* The night-time machines are assumed to be idle for a large chunk of time. The Wave Scheduler uses the geographic CAN overlay to select a host in the night-time zone.

4.1 Scheduling Strategies

The scheduling strategies we study are built on the above components. They all use the same host discovery schemes but different host selection criteria and different migration schemes.

Each strategy has two distinct steps: initial scheduling and later migration. In initial scheduling, the initiator of the job uses host discovery to discover hosts satisfying the host selection criteria and schedules job on the chosen host. The migration schemes also use host discovery to discover candidate hosts, and they use different options discussed above to decide when and where to migrate the job. Table 1 summarizes the difference between different migration schemes.

Table 1. Different migration strategies

	When to migrate	
Where to migrate	*Immediate Migration*	*Linger*
Random Host	Migration-immediate	Migration-linger
Host in night-zone	Wave-immediate	Wave-linger

The non-migration strategy follows the SETI@home model. It uses the more relaxed host selection criteria: any *unclaimed* host can be a candidate.

- **No-migration.** With no-migration, a client initially schedules the task on an unclaimed host, and the task never migrates during its lifetime. The task runs in screensaver mode when the user is not using the machine, and sleeps when the machine is unavailable.

The following are all migration schemes. The first four migration schemes try to only use *available* hosts for fast execution. When it fails to find *available hosts* for migration, the host will inform the initiator of the job and let the initiator reschedule the job.

- **Migration-immediate.** With migration-immediate, the client initially schedules the task on an available host. When the host becomes unavailable migration-immediate immediately migrates the job to a *random* available host. In the best case, the task begins running immediately, migrates as soon as the current host in unavailable, and continues to run right away on a new available host.
- **Wave-immediate.** Wave-immediate works the same as migration-immediate except the job is migrated to a host in the night-time zone.
- **Migration-linger.** With migration-linger, a client initially schedules the task on an available host. When the host becomes unavailable, migration-linger allows the task to linger on the host for a random amount of time. If the host is still unavailable after the lingering time is up, it then migrates.
- **Wave-linger.** Wave-linger works the same as migration-linger except that the job is allowed to linger before migrating to a host in the night-time zone.

The migration schemes described above put minimal burden on the hosts. A host only needs to try to migrate the task once when it becomes unavailable, i.e. there is no backoff and retry. Instead, if the host fails to find an idle host, it notifies the original client node who initially scheduled the job on this host, letting the client reschedule the job.

The last two migration strategies are more persistent in their efforts to find an available host. They are adaptive strategies in that they adjust to the conditions on the local host, and on their ability to find a migration target. These adaptive strategies put a bigger burden on the host since it must retry several times on behalf of the foreign task.

- **Migration-adaptive.** For initial scheduling, migration-adaptive tries to find a host that is available. If it cannot, migration-adaptive schedules the task on an unclaimed host where the task will sleep for a random amount of time.

  When the host becomes unavailable, migration-adaptive will try to migrate the task to a *random* new host that is available. If it cannot find such a host, it allows the job to linger on the current host for a random amount of time and try again later. A cycle of attempted migration and lingering is repeated until the job finishes.
- **Wave-adaptive.** Wave-adaptive is the same as migration-adaptive except that it migrates to a host in the night-time wave zone.

5 Simulation

We conducted simulations to investigate the performance of the migration strategies described above and their effectiveness at reducing turnaround time, relative to a no-migration policy similar to SETI@home. We also evaluated the performance of the Wave Scheduler to see what gains are achievable through our strategy of exploiting knowledge of available idle blocks of time at night.

5.1 Simulation Configuration

We use a 5000 node structured overlay in the simulation. Nodes join the overlay following the CAN protocol (or timezone-aware CAN protocol in the case of the Wave scheduler). The simulation is built with ns, a widely used network simulation tool [30].

Profile of Available Cycles on Hosts. To evaluate the performance of different scheduling methods, a coarse-grain hourly synthetic profile is generated for each machine as follows: During the night-time (from 12pm to 6 am), the host is available with a very low CPU utilization level, from 0% to 10%. During the daytime, for each one hour slot it is randomly decided whether the machine is available or not. Finally, the local CPU load in a free daytime slot is generated from a uniform distribution ranging from 0% to 30%. We assume that when a host is running a foreign job, it can still initiate resource discovery for migration

and relay messages for other hosts. The percentage of available time during the day varies from 5% to 95%. For simplicity, we assume all the hosts have the same computational power.

Job Workload. During the simulation, a given peer can be both a client and a host. A random group of peers (10% to 90%) are chosen as clients. Each client submits a job to the system at a random point during the day. The job *runtime* is defined as the time needed for the job to run to completion on a dedicated machine. Job runtime is randomly distributed from 12 hours to 24 hours.

Host Discovery Parameters. We set the parameters of the resource discovery schemes to be scalable and to have low latency. The maximum number of scheduling attempts for random node label-based resource discovery is 5 times and the search scope for expanding ring search is 2 hops.

Migration Parameters. The lingering time for the linger-migration models is randomly chosen in the range 1 hour to 3 hours. In the adaptive model, the linger time of a foreign job when it cannot find a better host to migrate to is also randomly chosen in the range 1 hour to 3 hours.

Wave Scheduler. For the Wave scheduler, a 1x 24 CAN space is divided into 6 wavezones, each containing 4 time zones based on its second dimension. We studied the performance of a variety of strategies for selecting the initial wavezone and the wavezone to migrate to. The variations included (a) migrate the job to the wavezone whose earliest timezone just entered night-time, (b) migrate the job to a random night-time zone, and (c) migrate the job to a wavezone that currently contains the most night-time zones. The first option performs better than the others, since it provides the maximal length of night-time cycles. The simulation results presented in this paper use this option. However, it may be better to randomly select a nightzone to balance network traffic if many jobs simultaneously require wave scheduling.

5.2 Simulation Metrics

Our evaluation of different scheduling strategies is focused on the turnaround time of a job, the time from when it first began execution to when it completes execution in the system.

In our study, a job is considered to have failed if the client fails to find a host satisfying host selection criteria, either when it initially tries to schedule the job, or when it later tries to migrate. Most of the performance metrics are measured only for those jobs that successfully complete execution. In this study, we do not model rescheduling, as we are interested in the success rate of the first scheduling attempt which includes the initial scheduling and the following successful migrations. The job completes in the shortest time if the client only needs to schedule the job once, so the slowdown factor measured this way show the peak performance of each migration scheme. Also it is interesting to see what percentage of jobs needs to be rescheduled under different migration models.

The metrics used in the study are the followings:

- **% of jobs that fail to complete (job failure rate).** the number of failed jobs divided by the total number of jobs submitted to the system.
- **Average slowdown factor.** The slowdown of a job is its turnaround time (time to complete execution in the peer-to-peer cycle sharing system) divided by the job runtime (time to complete execution on a dedicated machine). We average the slowdown over all jobs that successfully complete execution.
- **Average number of migrations per job.** the number of times a job migrates during its lifetime in the system, averaged over all jobs that successfully complete execution.

We do not include migration and resource discovery overhead when plotting the average slowdown factor. The migration and resource discovery overhead do not make a visible difference in the results when migrations and resource discoveries are infrequent and the jobs run for a long time. We will analyze migration overhead, which dominates the computation overhead in the discussion of number of migrations.

5.3 Simulation Results

In this section, the legends in each graphs are ordered from top to bottom to match the relative position of the corresponding curves. Each data point is the average over 15 simulation runs.

No-migration vs. Migration. We first compare no-migration with the two basic migration schemes: migration-immediate and migration-linger. We measure the performance of these scheduling strategies as a function of percentage of free time on the hosts. When the percentage of free time on hosts increases on the x-axis, the load of the system decreases. We also examine the impact of the resource discovery scheme employed.

(a) The impact of migration on job turnaround times
Figure 5 shows the average slowdown factor for successfully completed jobs as a function of free time on the hosts during the daytime hours. As expected, jobs progress faster with more available time on hosts during daytime. The performance of the no-migration strategy is clearly the worst since there is no effort made to avoid long waiting times when the host is not free. Note that the slowdown of migration-immediate (for both expanding ring and random host discovery) is always 1, since only jobs that successfully run to completion are considered. The success rate of different scheduling schemes will be discussed in section 5.3(b).

The performance of migration-linger is better than the no-migration strategy but worse than the others with its wasted lingering time associated with each migration. The performance of the adaptive models is the closest to the idealized migration-immediate schemes since it adaptively invokes the migration-immediate scheme whenever possible.

The slowdown factor is mainly influenced by the migration model used. However, for the linger and adaptive strategies, the resource discovery protocol also plays a role when the free time on the host is limited (e.g. when the percentage of hosts free time during daytime is less than 65%). We noticed that for the linger strategy, random performs better with respect to slowdown, but for the adaptive strategy, expanding ring performs better. This can be explained as follows: For comparable search times, expanding ring contacts more hosts than the random node label-based search, and therefore yields a higher successful migration rate. However, since with migration-linger, every successful migration implies (wasted) lingering time, ultimately random has lower slowdown with its lower migration success rate. This observation is supported by Figure 9 which shows the average number of migrations for each strategy. For the adaptive strategy, expanding ring has lower slowdown than random as expected.

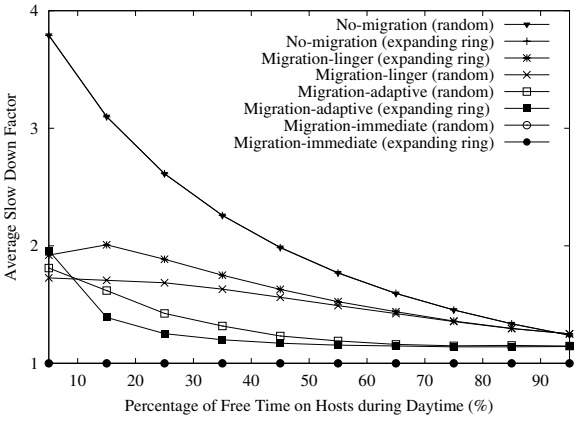


Fig. 5. Average slowdown factor for no-migration vs. migration (The percentage of clients in the system is 20%)

Figure 5 also shows that the slowdown factor for the no-migration strategy and for migration-immediate is insensitive to the resource discovery schemes.

Figure 6 further confirms the improvement of turnaround time when using a migration model under heavy load. The majority of jobs scheduled by no-migration scheduling experienced a slowdown greater than 2 and in the extreme case, jobs may experience slowdown greater than 5. The majority of jobs scheduled by migration-adaptive and migration-immediate have small slowdown or no slowdown at all.

(b) The impact of migration on successful job completion

The above results regarding slowdown factor cannot be considered in isolation. In particular, it is necessary to also consider the job failure rates, i.e. percentage of jobs for which a host satisfying host selection criteria cannot be found either in initial scheduling or in migration.

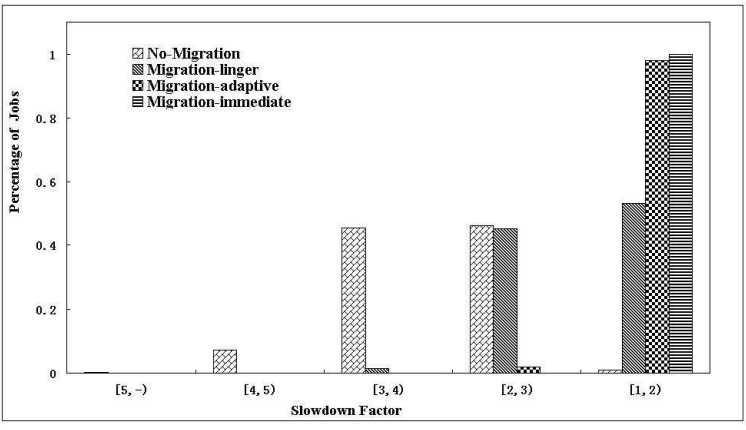


Fig. 6. Histogram of slowdown factors of successfully finished jobs (The percentage of clients is 20% and the percentage of free time on hosts is 15%)

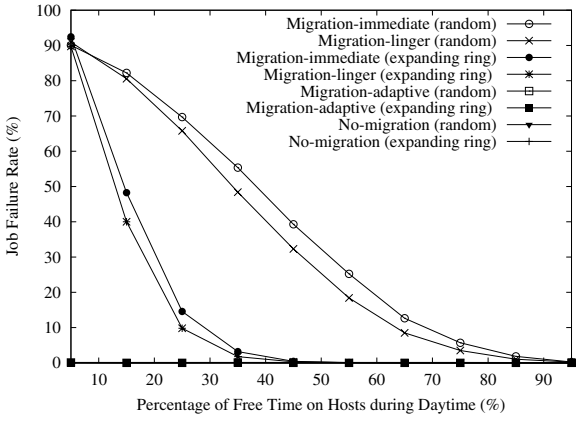


Fig. 7. % of jobs that fail to complete(The percentage of clients in the system is 20%)

Figure 7 shows the percentage of jobs that failed to finish assuming the same simulation configuration as in Figure 5. For the two strict migration models (migration-immediate and migration-linger) which only use local availability information, the failure rate is extremely high. The adaptive models, which use a small amount of global information at migration time, have dramatically fewer failed jobs – close to zero.

Clearly, there is a tradeoff between the job turnaround time and percentage of jobs that successfully complete. The strict models have the lowest turnaround time, but extremely high failure rates when free time on the hosts is limited. The adaptive model performs best because it achieves low turnaround time with most of the jobs successfully completed.

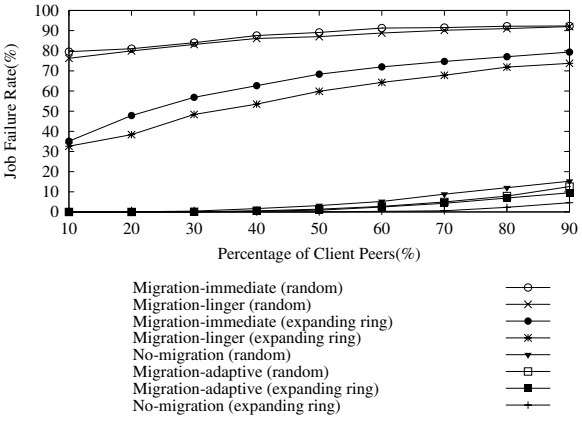


Fig. 8. % of jobs that fail to complete (The percentage of free time on the hosts during the day is 15%)

When the number of client requests increase, there will be intense competition for free hosts. When the free time on these hosts is small, the situation is even worse. Figure 8 shows that the failure rate of all scheduling strategies increases with the increasing number of client requests. The persistently high failure rate of migration-immediate makes it impractical for real applications when the available resources are very limited.

The simulation results show that with abundant host free time, the failure rate of migration-adaptive using an expanding ring search is even slightly lower than the no-migration scheme.

(c) Number of migrations during job lifetime

Figure 9 shows that the average number of migrations varies with the different migration scheduling strategies. The graph shows that when the percentage of host free time increases, the number of migrations increases first and then decreases. The reason is that there are two factors that influence the number of migrations: the number of currently available hosts and the length of free time slots on the hosts. With more available hosts, there is higher chance of migration success and therefore a larger number of migrations. With longer free time slots, the need for the jobs to migrate is reduced. With higher percentage of free time, the amount of currently available hosts increases and the length of free time slots also increases.

We can demonstrate that migration overhead is low using the same graph. In early morning or late night, the network traffic in the Internet is usually light. Therefore the network connection from the end-host to the Internet is usually the bottleneck link when downloading or uploading data. In the following computation, we assume the upload bandwidth of the host is 256kb, which is the bandwidth of slow-end DSL users and download bandwidth is higher than upload bandwidth with DSL. If the amount of data to be transmitted during the migration is 1MB, the slowdown factor of migration schemes will increase

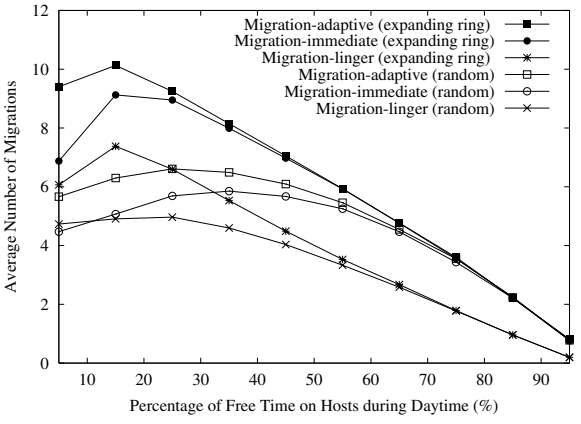


Fig. 9. Average number of migrations for successfully finished jobs (The percentage of clients in the system is 20%)

by at most 0.005 when the running time of the job is longer than 12 hours. Even when the amount of data to transmit is 20MB, which is quite large for a scientific computation, the influence is at most 0.1. The time overhead of resource discoveries is much smaller and negligible compared with that of migration.

Performance of Wave Scheduler. This section presents the evaluation of the Wave Scheduler. In order to focus on the difference between migration strategies, we only describe results with the resource discovery method set as expanding ring search. (Simulations with random node label-based discovery show the same relative performance.)

(a) Impact of Wave scheduler on turnaround time
Figure 10 shows that the turnaround time of jobs with Wave Schedulers is lower than other migration schedulers. Jobs progress faster with the Wave Scheduler because it can identify hosts with potentially large chunks of available times. The turnaround time of wave-adaptive is consistently low, while the turnaround time of migration-adaptive is significantly higher when the amount of free time is small. When the percentage of free time on hosts is 15%, the turnaround time of jobs under wave-adaptive is about 75% of that under migration-adaptive.

(b) Impact of Wave scheduler on successful job completion
The percentage of jobs that fails to complete using the Wave scheduler is influenced by two factors. Wave identifies available hosts with large chunks of future free time. However, if the ratio of requests to the number of such hosts is limited, there will be scheduling conflicts.

When the free time on hosts is limited, the Wave scheduler does better than other migration schemes since it was designed for this exact situation. (see Figure 12). The intensive contention for night-time hosts is relieved by wave-adaptive, which adapts to the amount of free time by continuing to stay on the

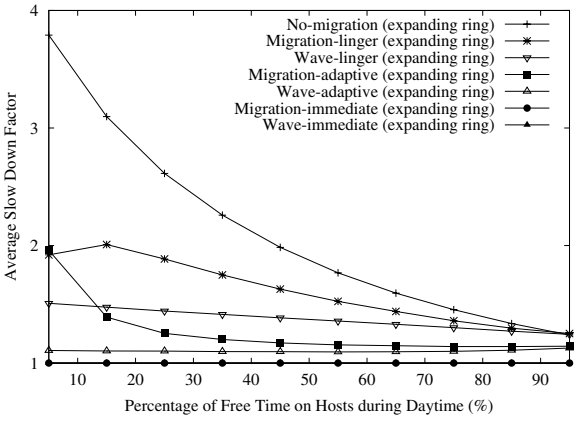


Fig. 10. Wave Scheduler: Average slow down factor(The percentage of clientd request is 20%)

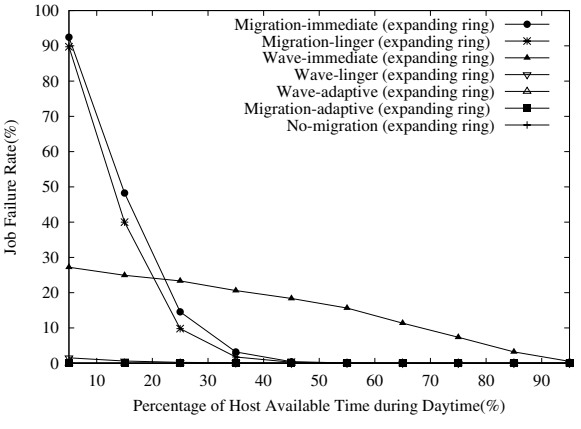


Fig. 11. Wave scheduler: % of job that fail to complete (The percentage of clients in the system is 20%)

hosts in case of contention. The job failure rate of wave-adaptive is competitive with the no-migration model and slightly lower than with migration-adaptive.

Figure 11 shows the percentage of jobs that failed to finish under the same simulation configuration as in Figure 10. The job failure rate of the Wave scheduler is relatively higher than others when the percentage of free time on hosts increases, as wave-immediate uses strict rules about using night-time hosts and this cause contention. The other two wave scheduler strategies perform as well as migration-adaptive and the no-migration strategy.

(c) Number of migrations during job lifetime with Wave scheduler
Figure 13 compares the average number of migrations of successfully finished jobs with the wave migration strategies versus the standard migration. As we

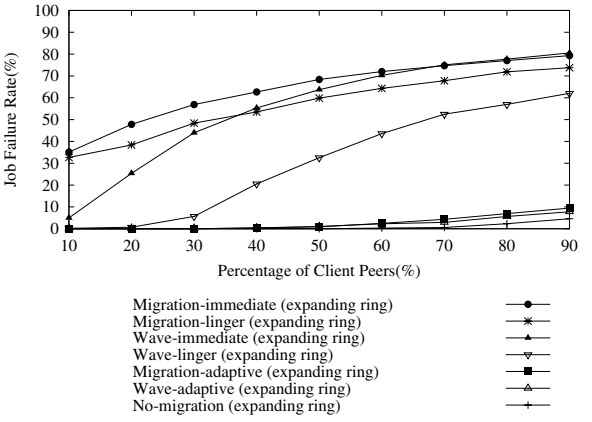


Fig. 12. Wave scheduler: % of job that fail to complete (The percentage of available time on the hosts during the day is 15%)

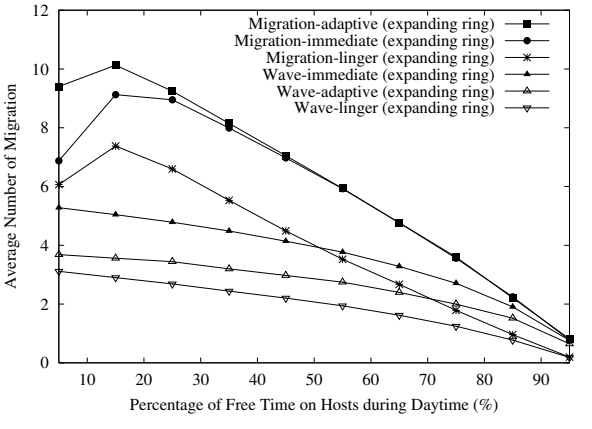


Fig. 13. Wave scheduler: Average number of migrations (The percentage of client in the system is 20%)

expected, jobs scheduled with the Wave scheduler finished with fewer migrations, because it exploits the long available intervals at night while the others may end up using short, dispersed time slots during the day. As in the discussion about migration overhead in section 5.3(c), the migration overhead of Wave Scheduler is even smaller compared with the standard migration schemes and therefore it is acceptable for jobs with long running time.

6 Conclusion and Future Work

We studied job scheduling strategies with the goal of faster turnaround time in open peer-based cycle sharing systems. The fundamental lesson to be learned

is that under strong owner policies on host availability for cycle sharing, the ability to utilize free cycles wherever and whenever they are available is critical. Migration achieves this goal by moving jobs away from busy hosts and to idle hosts. The best migration strategies that we studied, wave and adaptive, were able to use fairly straightforward mechanisms to make better decisions about when and where to migrate.

We also observed that careful design of the infrastructure of a peer-to-peer cycle sharing system impacts the performance of its schedulers. For example, the use of a structured overlay network that supports timezone-aware overlay was essential for the functioning of our wave scheduler. To recap:

- Compared with no-migration schemes, migration significantly reduces turnaround time.
- The adaptive strategies perform best overall with respect to both low turnaround time and low job failure rate.
- Wave scheduler performs better than the other migration strategies when the free time during the day is limited.
- Wave-adaptive improves upon wave because it reduces collisions on night-time hosts. It performs best among all the scheduling strategies.

To further improve the performance of Wave Scheduler and conduct a more realistic and comprehensive evaluation of Wave Scheduler and other peer-based scheduling using migration schemes, our ongoing and future work include the following tasks:

1) Improving Wave Scheduler by using the schemes discussed in section 3 including expanding the night-time cycles to any idle cycles and overlay construction including information other than just CPU cycles.
2) Evaluating Wave Scheduler and other migration schemes using a retry model in which clients retry after a random amount of time if the job fails to be scheduled on host in initial scheduling or migration. Alternatively, a client can make an intelligent decision about the how long it needs to wait before retry based on estimation of current load of the system.
3) Evaluating Wave Scheduler using workload trace such as Condor trace and desktop grid trace used by [31].
4) Studying the characteristics of bag-of-tasks scientific computation to understand what impact Wave Scheduler can make for real applications.
5) Collecting activity based resource profiles and generating CPU usage pattern from such profiles as a supporting study to our work.
6) Studying the migration cost on an Internet test-bed such as PlanetLab [32].

6.1 Related Work

The related work can be divided into two categories: peer-to-peer networks and cycle sharing systems.

Peer-to-peer networks emerged with the popular file sharing systems. The first generation peer-to-peer protocols [22] were extended to efficient, scalable

and robust structured peer-to-peer overlay networks [17, 19, 18]. Structured peer-to-peer overlay networks are motivated by distributed hash table algorithms which use consistent hash function to hash a key onto a physical node in the overlay network. A wealth of peer-to-peer applications including file sharing, storage sharing, and web caching are built atop structured overlay networks. The popularity of peer-to-peer file sharing techniques naturally stimulated the development of peer-based cycle sharing systems.

The second research area is cycle sharing systems which can be divided into three categories: Internet-based computing infrastructures, institutional-based cycle sharing systems and desktop grid systems.

Internet-based computing infrastructures [1, 14, 15] use a client-server model to distribute tasks to volunteers scattered in the Internet. The hosts actively download tasks and data from a central server. A foreign task then stays on the same host during their entire life spans. The hosts report the results to the central server when the computation is done. Because Internet-based computing projects require manual coordination from central servers, they may experience downtime due to surges in hosts requests.

Institutional-based cycle sharing systems promote resource sharing within one or a few institutions. In order to access resources in institutional-based cycle sharing systems, the user first needs to acquire an account from the system administrator. The most notable practical institutional-based cycle sharing system is Condor [10, 8], which was first installed as a production system 15 years ago. The work continued to evolve from a load sharing system within one institution to a load sharing system within several institutions. Condor-G uses Globus [7] for inter-domain resource management. The strict membership requirement and heavyweight resource management and scheduling methods used by this type of system make it hard for average users to install and maintain their own cycle sharing systems.

The new desktop grid systems [33, 34] harness idle cycles on desktop machines. Our work belongs to one type of desktop grid systems, the peer-based desktop grid systems [3, 4, 5], which harness idle cycles on desktop machines under a peer-based model. Each node in these systems can be either a single machine or an institution joining the peer-to-peer overlay network. Each peer can be both a cycle donor and a cycle consumer. Peer-based cycle sharing systems is a new research field, which charts many challenging research problems including scalable resource discovery and management, incentives for node to join the system, trust and security schemes. OurGrid [3] proposed an accounting scheme to aid equitable resource sharing in order to attract nodes to join the system. Flock of Condor [4] proposed to organize the nodes in a Pastry [19] overlay network. Nodes broadcast resource information in a limited scope for resource discovery. Our CCOF model [5] proposed a generic scalable modular peer-to-peer cycle sharing architecture which supports automatic scheduling for arbitrary client applications.

To our best knowledge, none of the previous work has addressed the fast turnaround scheduling problem in a scalable peer-based cycle sharing system.

A recent paper [31] describes scheduling for rapid application turnaround on enterprise desktop grids. The computation infrastructure used a client-server model, in which a server stores the input data and schedules tasks to a host. When the host becomes available, it sends a request to the server. The server keeps a queue of available hosts and chooses the best hosts based on resource criteria such as clock rate and number of cycles delivered in the past. The work did not considering migration schemes. Moreover, this work is limited to scheduling within one institution and it uses a client-server infrastructure, while scheduling in a large scale fully distributed peer-based cycle sharing system is much more complicated and challenging.

References

1. D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An experiment in public-resource computing," *Communications of the ACM*, vol. 45, 2002.
2. "Folding@home http://folding.stanford.edu/."
3. N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "OurGrid: An approach to easily assemble grids with equitable resource sharing," in *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.
4. A. Butt, R. Zhang, and Y. Hu, "A self-organizing flock of condors," in *Proceedings of SC2003*, 2003.
5. V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, "Cluster Computing on the Fly: P2P scheduling of idle cycles in the Internet," in *IPTPS*, 2004.
6. R. Gupta and A. Somani, "Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes," in *Proceedings of second Workshop on the Economics of peer-to-peer systems*, 2004.
7. I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, 1997.
8. D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computng: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., 2002.
9. "Legion http://www.cs.virginia.edu/ legion/."
10. M. Litzkow, M. Livny, and M. Mutka, "Condor -a hunter of idle workstations," in *the 8th International Conference on Distributed Computing Systems*, 1988.
11. S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software - Practice and Experience*, vol. 23, no. 12, 1993.
12. D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson, "GLUnix:a global layer unix of a network of workstations," *Software-Practice and Experience*, vol. 28, no. 9, 1998.
13. "LSF load sharing facility, http://accl.grc.nasa.gov/lsf/aboutlsf.html."
14. "BOINC: Berkeley open infrastructure for network computing, http://boinc.berkeley.edu/."
15. N. Camiel, S. London, N. Nisan, and O. Regev, "The popcorn project: Distributed computation over the Internet in java," in *Proceedings of The 6th International World Wide Web Conference*, 1997.

16. B. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the web," in *Proceedings of the 9th International Conference on Paralel and Distributed Computing Systems*, 1996.
17. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. ACM SIGCOMM*, 2001.
18. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM*, 2001.
19. A. Rowstron and P. Druschel, "Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, 2001.
20. A. Iamnitchi and I. Foster, "A peer-to-peer approach to resource location in grid environments," in *Grid Resource Management*, J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski, Eds., 2003.
21. D. Zhou and V. Lo, "Cluster Computing on the Fly: resource discovery in a cycle sharing peer-to-peer system," in *Proceedings of the 4th International Workshop on Global and P2P Computing (GP2PC'04)*, 2004.
22. "The Gnutella protocol specification (version 0.6)."
23. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, 1998.
24. V. Lo and J. Mache, "Job scheduling for prime time vs. non-prime time," in *Proc 4th IEEE International Conference on Cluster Computing (CLUSTER 2002)*, 2002.
25. D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. on Software Engineering*, vol. 12(5), 1986.
26. N. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *Computer*, vol. 25(12), 1992.
27. J. Brevik and D. Nurmi, "Quantifying machine availability in networked and desktop grid systems," UCSB, Tech. Rep. CS2003-37.
28. L. Yang, I. Foster, and J. Schopf, "Homeostatic and tendency-based CPU load predictions," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.
29. P. Dinda, "The statistical properties of host load," *Scientific Programming*, vol. 7, no. 3–4, 1999.
30. "ns, http://www.isi.edu/nsnam/ns/."
31. D. Kondo, A. Chien, and H. Casanova, "Resource management for rapid application turnaround on enterprise desktop grids," in *Proceedings of SC2004*, 2004.
32. "Planetlab, http://www.planet-lab.org/."
33. O. Lodygensky, G. Fedak, V. Neri, F. Cappello, D. Thain, and M. Livny, "Xtremweb & condor: sharing resources between internet connected condor pool," in *Proceedings of GP2PC2003(Global and Peer-to-Peer computing on large scale distributed systems)*, 2003.
34. "Entropia, inc. http://www.entropia.com."

Enhancing Security of Real-Time Applications on Grids Through Dynamic Scheduling

Tao Xie and Xiao Qin

Department of Computer Science,
New Mexico Institute of Mining and Technology,
801 Leroy Place, Socorro, New Mexico 87801-4796
{xietao, xqin}@cs.nmt.edu

Abstract. Real-time applications with security requirements are emerging in various areas including government, education, and business. However, conventional real-time scheduling algorithms failed to fulfill the security requirements of real-time applications. In this paper we propose a dynamic real-time scheduling algorithm, or SAREG, which is capable of enhancing quality of security for real-time applications running on Grids. In addition, we present a mathematical model to formally describe a scheduling framework, security-sensitive real-time applications, and security overheads. We leverage the model to measure security overheads incurred by security services, including encryption, authentication, integrity check, etc. The SAREG algorithm seamlessly integrates security requirements into real-time scheduling by employing the security overhead model. To evaluate the effectiveness of SAREG, we conducted extensive simulations using a real world trace from a supercomputing center. Experimental results show that SAREG significantly improves system performance in terms of quality of security and schedulability over three existing scheduling algorithms.

1 Introduction

A computational grid is a collection of geographically dispersed computing resources, providing a large virtual computing system to users. With rapid advances in processing power, network bandwidth, and storage capacity, Grids are emerging as next generation computing platforms for large-scale computation and data intensive problems in industry, academic, and government organizations. As typical scientific simulation and computation require a large amount of compute power, it becomes crucial to take advantage of application scheduling to enable the sharing and aggregation of geographically distributed resources to meet the needs of highly complex scientific problems [32].

Recently there have been some efforts devoted to the development of real-time applications on Grids [9]. Real-time applications depend not only on results of computation, but also on time instants at which these results become available [13]. The consequences of missing deadlines of hard real-time systems may be catastrophic, whereas such consequences for soft real-time systems are relatively less damaging. Examples of hard real-time applications include distributed defense and surveillance applications [38], and distributed medical data systems [20]. On-line transaction processing systems are examples of soft real-time applications [26].

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 219 – 237, 2005.

There exist a growing number of systems that have real time and security considerations [36], because sensitive data and processing require special safeguard and protection against unauthorized access. In particular, a variety of motivating real-time applications running on Grids require security protections to completely fulfill their security-critical needs. Unfortunately, conventional wisdom on real time systems is inadequate for applications with real-time and security requirements. This is mainly because traditional real-time systems are developed to guarantee timing constraints while possibly posing unacceptable security risks.

To tackle the aforementioned problem, we proposed a real-time scheduling algorithm (referred to as SAREG) with security awareness, which is intended to seamlessly integrate security into real-time scheduling for applications running on Grids. In this paper we use trace-driven simulation to compare the performance of the SAREG algorithm against three baseline scheduling algorithms for Grids. Our simulator combines performance and security overhead estimates using a security overhead model based on three most commonly used security services. We have used a real world trace from a supercomputing centre to drive our simulations. Also, we concentrate on three security services, namely, authentication, integrity, and encryption. Our empirical results demonstrate that the SAREG scheduling algorithm is able to achieve high quality of security while guaranteeing timing constraints posed by real-time applications.

To put our work in a large perspective, in the next section we summarize related work in the areas of computer security and real-time systems. Section 3 describes the preliminary system architecture and task model. In Section 4 we propose a real-time scheduling algorithm for parallel applications on Grids. We present in Section 5 the experimental results based on real world traces from a supercomputing centre. We also provide insight into system parameters that ultimately affect performance potential of SAREG. Finally, Section 6 concludes the paper with summary and futuredirections.

The rest of the paper is organized in the following way. Section 2 includes a summary of related work in this area. Section 3 discusses the system architecture and task model with security requirements. Section 4 proposes a security overhead model. Section 5 presents the security-aware real-time scheduling strategy. Performance analysis of the SAREC-EDF algorithm is explained in Section 6. Section7 concludes the paper with summary and future research directions.

2 Related Work

Scheduling algorithms for Grids have been extensively studied in the past both experimentally and theoretically. Wu and Sun considered memory availability as a performance factor and introduced memory-aware scheduling in Grid environments [42]. Li compared the performance of a variety of scheduling and processor allocation algorithms for grid computing [21]. In et al. proposed a framework for policy based scheduling in resource allocation of grid computing [18]. However, these scheduling algorithms are not suitable for real-time applications, because there is no guarantee to finish real-time tasks in specified time intervals.

The issue of scheduling for real-time applications was previously reported in the literature. Conventional real-time scheduling algorithms such as Rate Monotonic

(RM) algorithm [23], Earliest Deadline First (EDF) [37], and Spring scheduling algorithm [33] were successfully applied in real-time systems. We proposed both static [30] and dynamic [31] scheduling schemes for real-time systems. Unfortunately, none of the above real-time scheduling algorithms can be directly applied to the Grid environments.

Real-time scheduling is a key factor in obtaining high performance and predictability for Grids. Various aspects of complicated real-time scheduling problems in the context of Grids were addressed in the literature. He et al. proposed a dynamic scheduling for parallel jobs with soft-deadlines in Grids [16]. Caron et al. developed an algorithm that considers both priority and deadlines of tasks on Grids [6]. However, most of existing real-time scheduling algorithms perform poorly for security-sensitive real-time applications on Grids due to the ignorance of security requirements imposed by the applications.

Recently increasing attention has been drawn toward security-awareness in Grids, because efficient and flexible security has become a baseline requirement. Humphrey et al. examined the current state of the art in securing a group of activities and introduced new technologies that promise to meet the security requirements of Grids [17]. Azzedin and Maheswaran integrated the notion of "trust" into resource management of a grid computing systems [3]. Wright et al. proposed a security architecture for a network of computers bound together by an overlying framework used to provide users a powerful virtual heterogeneous machine [41]. Connelly and Chien proposed an approach to protecting tightly coupled, high-performance component communication [7]. However, the above security techniques are not appropriate for real-time applications due to the lack of ability to express and handle timing constraints.

Some work has been done to incorporate security into a variety of real-time applications. George and Haritsa proposed concurrency control protocols to support applications with real-time and security requirements [12]. Ahmed and Vrbsky developed a secure optimistic concurrency control protocol that can make trade-offs between security and real-time requirements [2]. Son et al. proposed an approach to trading off quality of security to achieve required real-time performance [35]. In [36], a new scheme was developed to improve timeliness by allowing partial violations of security. Our work is fundamentally different from the above approaches because they are focused on concurrency control protocols whereas ours is intended to develop a security-aware real-time scheduling algorithm, which can meet security constraints in addition to real-time requirements of tasks running on Grids.

Most recently, we proposed a dynamic security-aware scheduling algorithm for a single machine [43] and clusters [44][45]. We conducted simulations to show that compared with three heuristic algorithms, the proposed algorithm can consistently improve overall system performance in terms of quality of security and system schedulability under a wide range of workload conditions.

3 Mathematical Model

A mathematical model of security-aware real-time scheduling for Grids is presented in this section. This model, which describes a scheduling framework, security-sensitive real-time jobs, and security overheads, allows the SAREG algorithm to be formally presented in Section 4.

3.1 Scheduling Framework

A Grid can be specified as $G = \{M_1, M_2, \ldots, M_n\}$, where M_i, $1 \le i \le n$, is a site or cluster [27]. The n sites are connected by wide-area networks (See Figure 1). The scheduling framework is general in the sense that it can be applied to small-scale grids where computational sites are connected by LAN or MAN. Each site M_i is represented as a vector, e.g., $M_i = (P_i, N_i, T_i, Q_i)$, where P_i is the peek computational power measured by an overall CPU capacity (e.g., BIPS), N_i is the number of machines in the site, T_i is a set of accepted jobs running on M_i, and Q_i is a scheduler. Note that there exists a scheduler in each site, and we advocate the use of a distributed scheduling framework rather than a centralized one. This is mainly because a centralized scheduler in a large-scale grid inevitably becomes a severe performance bottleneck, resulting to a significant performance drop when workload is high. The distributed scheduling infrastructure makes a system portable, secure, and capable of distributing scheduling workload among an array of computational sites in the system [28][29].

Each site continues to receive reasonably up-to-date global load information by monitoring resource utilization of the Grid, and periodically broadcasts it local load information to other sites of the Grid. When a real-time job is submitted by a user to a local site, the corresponding scheduler assigns the job to a group of local machines or migrate the job to a remote site within in the Grid. The scheduler consists of a s*chedule queue* used to accommodate incoming real-time jobs. The scheduler queue is maintained by an admission controller. If the incoming real-time jobs can be scheduled, the admission controller will place the tasks in the *accepted queue* for further processing. In case no site can guarantee the deadline of the submitted real-time job, it will be dropped into a local *rejected list*. The scheduler processes all the accepted tasks by its scheduling policy before transmits them into the *dispatch queue*, where the quality of security of accepted jobs are maximized. After the quality of security is enhanced, the real-time job is dispatched to one of the designated site $M_i \in G$. The machines in site M_i can execute a number of real-time tasks in parallel.

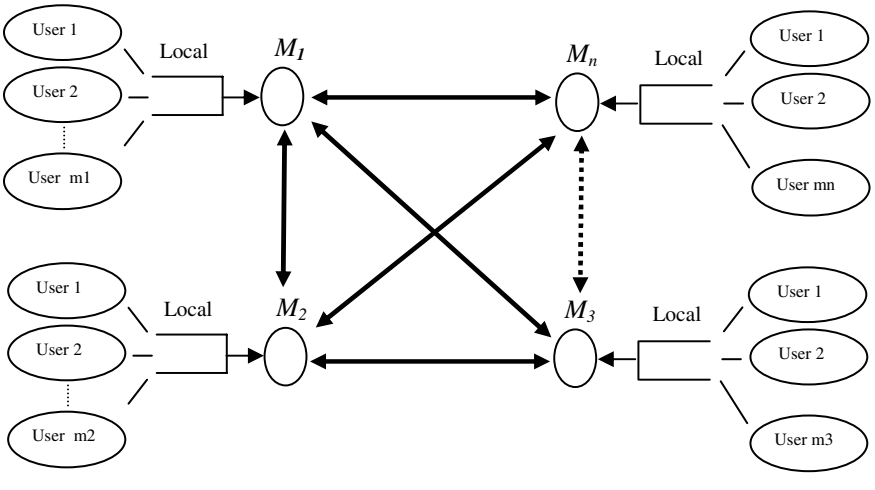


Fig. 1. The scheduling framework for SAREG in a computational Grid

3.2 Security-Sensitive Real-Time Jobs

A real-time job is specified as a set of rational parameters, e.g., $J_i = (e_i, p_i, d_i, l_i, S_i)$, where e_i is the execute time, p_i is the number of machines required to execute J_i, d_i is the deadline, and l_i denotes the amount of data (measured in KB) to be protected. e_i can be estimated by code profiling and statistical prediction [5]. A collection of security services required by J_i is specified as $S_i = (S_i^1, S_i^2, \ldots, S_i^q)$, where S_i^j denotes the security level range of the jth security service. Our security-aware scheduler is intended to determine the most appropriate point s_i in space S_i, e.g., $s_i = (s_i^1, s_i^2, \ldots, s_i^q)$, where $s_i^j \in S_i^j$, $1 \le j \le q$.

A schedule of a job J_i is formally denoted as the following expression:

$$x_i = \left((\sigma_{i,1}, s_{i,1}), (\sigma_{i,2}, s_{i,2}), \ldots (\sigma_{i,pi}, s_{i,pi})\right) . \tag{1}$$

where J_i is divided into p_i tasks, $\sigma_{i,j}$ and $s_{i,j}$ are the start time and the security level of the jth task.

The SAREG algorithm is able to measure the security benefits gained by each admitted job. To implement this basic and valuable functionality, we quantitatively model the security benefit of the jth task of job J_i as a security level function denoted by SL: $S_i \rightarrow \Re$, where $\Re$ is the set of positive real numbers:

$$SL(s_{i,j}) = \sum_{k=1}^{q} w_i^k s_{i,j}^k \ . \tag{2}$$

where $s_{i,j} = \left(s_{i,j}^1, s_{i,j}^2, \ldots, s_{i,j}^q\right), 0 \le w_i^j \le 1, \sum_{j=1}^{q} w_i^j = 1$.

Note that w_i^j is the weight of the jth security service for the task. Users specify in their requests the weights to reflect relative priorities given to the required security services. The security benefit of job J_i is computed as the summation of the security levels of all its tasks. Thus, we have the following equation:

$$SL(s_i) = \sum_{j=1}^{p_i} SL(s_{i,j}) \ . \tag{3}$$

where $s_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,pi})$.

The scheduling decision of the job J_i is feasible if (1) all its tasks can be completed before the deadline d_i, and (2) the corresponding security requirements are satisfied. Specifically, given a real-time job J_i that consists of p_i tasks, we can obtain the following non-linear optimization problem formulation to compute the optimal security benefit of J_i:

$$\text{maximize } SL(s_i) = \sum_{j=1}^{p_i} \sum_{k=1}^{q} w_i^k s_{i,j}^k \ . \tag{4}$$

subject to $\min(S_i^k) \le s_{i,j}^k \le \max(S_i^k)$, $f_{i,j} \le d_i$, where $f_{i,j}$ is the finish time of the jth task of J_i, and $\min(S_i^j)$ and $\max(S_i^j)$ are the minimum and maximum security requirements of J_i.

The SAREG scheduling algorithm to be presented in the next section strives to enhance quality of security, which is defined by the sum of the security levels of all the admitted jobs. Thus, the following security value function needs to be optimized, subjecting to certain timing and security constraints:

$$\text{maximize } SV = \sum_{j=1}^{n} \sum_{J_i \in T_j} y_{i,j} SL(s_i) \quad \cdot \tag{5}$$

where T_j is a set of accepted jobs running on site M_j, $y_{i,j}$ is set to 1 if job J_i is accepted by the jth site, and is set to 0 otherwise. Substituting Equation (4) into (5) yields the following security value objective function. Thus, our job scheduling problem for Grid environments can be formally defined as follows: given a Grid $G = \{M_1, M_2, \ldots, M_n\}$ and a list of jobs submitted to the Grid, find a schedule $x_i = \left((\sigma_{i,1}, s_{i,1}), (\sigma_{i,2}, s_{i,2}), \ldots (\sigma_{i,pi}, s_{i,pi})\right)$ for each job J_i, such that the total security level of jobs on G (Equation 6) is maximized.

$$SV = \sum_{i=1}^{n} \sum_{J_j \in T_i} y_{i,j} \sum_{k=1}^{p_j} \sum_{l=1}^{q} w_j^k s_{j,k}^l \quad \cdot \tag{6}$$

3.3 Security Overhead

Since security is achieved at the cost of performance degradation, it is critical and fundamental to quantitatively measure overhead caused by various security services. Unfortunately, less attention has been paid to models used to measure security overheads. Recently Irvine and Levin proposed a security overhead framework, which can be used for a variety of purposes [19]. However, security overhead model for each security services in the context of real-time computing remains an open issue. To enforce security in real-time applications while making security-aware schedulingalgorithms predictable and practical, in this section we proposed an effective model that is capable of approximately, yet reasonably, measuring security overheads experienced by tasks with an array of security requirements. With the security overhead model in place, schedulers are enabled to be aware of security overheads, thereby incorporating the overheads into the process of scheduling tasks. Particularly, the model can be employed to compute the earliest start times and the minimal security overhead. Without loss of generality, we consider three security services widely deployed in real-time systems, namely, encryption, integrity, and authentication.

3.4 Encryption Overhead

Encryption is used to encrypt real-time applications (executable file) and the data they produced such that a third party is unable to discover users' private algorithmsembedded in the executable applications or understand the data created by the applications Suppose each site has ten optional encryption algorithms, which are listed in

Table 1. Cryptographic algorithms used for encryption service

Cryptographic	s_i^e : SL	$\mu^e(s_i^e)$:MB/s
SEAL	0.1	168.75
RC4	0.2	96.43
Blowfish	0.3	37.5
Knufu/Khafre	0.4	33.75
RC5	0.5	29.35
Rijndael	0.6	21.09
DES	0.7	15
IDEA	0.8	13.5
3DES	0.9	6.25

Table 1. Based on their performance, each cryptographic algorithm is assigned a corresponding security level in the range from 0.1 to 0.9. For example, level 0.9 implies that we use 3DES, which is the strongest yet slowest encryption function among the alternatives. Since computation overhead caused by encryption mainly depends on the cryptographic algorithms used and the size of the data to be protected, Figure 2 (a) shows encryption time in seconds as a function of encryption algorithms and size of data to be secured measured on a 175 MHz Dec Alpha600 machine [25].

Let s_i^e be the encryption security level of t_i, and the computation overhead of the encryption service can be calculated using Equation (7), where l_i is the amount of data whose confidentiality must be guaranteed, and $\mu^e(s_i^e)$ is a function used to map a security level to its corresponding encryption method's performance.

$$c_i^e(s_i^e) = l_i / \mu^e(s_i^e) \ . \tag{7}$$

3.5 Integrity Overhead

Integrity services make it possible to ensure that no one can modify or tamper applications while they are executing on clusters. This can be accomplished by using a variety of hash functions [4]. Ten commonly used hash functions and their

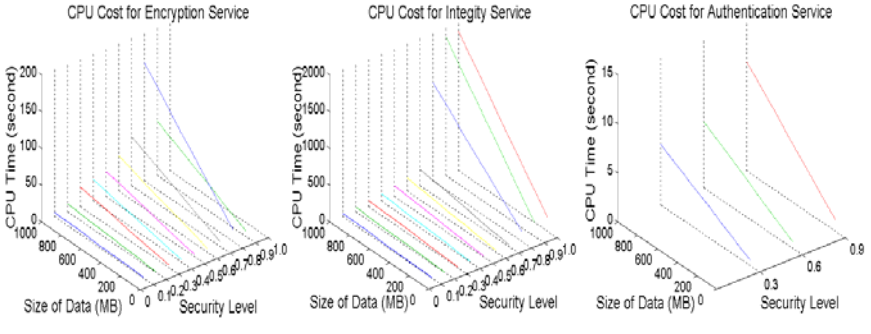


Fig. 2. CPU overhead of security services

Table 2. Ten hash functions used for integrity service

Hash Functions	s_i^g : SL	$\mu^g(s_i^g)$:KB/ms
MD4	0.1	23.90
MD5	0.2	17.09
RIPEMD	0.3	12.00
RIPEMD-128	0.4	9.73
SHA-1	0.5	6.88
RIPEMD-160	0.6	5.69
Tiger	0.7	4.36
Snefru-128	0.8	0.75
MD2	0.9	0.53
Snefru-256	1.0	0.50

performance (evaluated on a 90 MHz Pentium machine) are shown in Table 2. Based on their performance, each hash function is assigned a corresponding security level in the range from 0.1 to 1.0. For example, level 0.1 implies that we use MD4, which is the weakest yet fastest hash function among the alternatives. Level 1.0 means that Snefru-256 is employed for integrity, and Snefru-256 the slowest yet strongest function among the competitors.

Let s_i^g be the integrity security level of t_i, and the computation overhead of the integrity service can be calculated using Equation (8), where l_i is the amount of data whose integrity must be guaranteed, and $\mu^g(s_i^g)$ is a function used to map a security level to its corresponding hash function's performance. The security overhead model for integrity is depicted in Figure 2 (b).

$$c_i^g(s_i^g) = l_i / \mu^g(s_i^g). \tag{8}$$

3.6 Authentication Overhead

Tasks must be submitted from authenticated users and, thus, authentication services are deployed to authenticate users who wish to access the Grid [8]. Table 3 enlists three authentication techniques: weak authentication using HMAC-MD5; acceptable authentication using HMAC-SHA-1, and fair authentication using CBC-MAC-AES. Each authentication technique is assigned a security level s_i^a in accordance with the

Table 3. Three authentication methods

Authentication Methods	s_i^a : Security Level	$c_i^a(s_i^a)$: Time (ms)
HMAC-MD5	0.3	90
HMAC-SHA-1	0.6	148
CBC-MAC-AES	0.9	163

performance. Thus, authentication overhead $c_i^a(s_i^a)$ is a function of security level s_i^a. The security overhead model for authentication is shown in Figure 2(c).

3.7 Modeling Security Overheads

Now we can derive security overhead, which is the sum of the three items above. Suppose task t_i requires q security services, which are provided in sequential order. Let s_i^j and $c_i^j(s_i^j)$ be the security level and overhead of the jth security service, the security overhead c_i experienced by t_i, can be computed using Equation (9). In particular, the security overhead of t_i with security requirements for the three services above is modeled by Equation (10).

$$c_i = \sum_{j=1}^{q} c_i^j(s_i^j), \text{ where } s_i^j \in S_i^j . \tag{9}$$

$$c_i = \sum_{j \in \{a, e, g\}} c_i^j(s_i^j), \text{ where } s_i^j \in S_i^j . \tag{10}$$

It is to be noted that $c_i^e(s_i^e)$, $c_i^g(s_i^g)$, and $c_i^a(s_i^a)$ in Equation (10) are derived from Equations (7)-(8) and Table 2. In section 5, Equation (10) will be used to calculated the earliest start times and minimal security overhead. (See Equations 11 and Equation12)

4 The SAREG Scheduling Algorithm

The *SAREG* algorithm is outlined in Figure 3. The goal of the algorithm is to deliver high quality of security under two conditions: (1) the security level promotion will not miss its deadline; and (2) the security level promotion will not result in any accepted subsequent task to be failed. To achieve the goal, *SAREG* strives to maximize security level (measured by Equation 5) of each accepted job (see Step 21) while maintaining reasonably high guarantee ratios (see Step 12).

Before optimizing the security level of each task of job J_i on M_j, *SAREG* attempts to meet the real-time requirement of J_i. This can be accomplished by calculating the earliest start time (use Equation 11) and the minimal security overhead of J_i (use Equation 12) in Steps 2 and 3, followed by checking if all the tasks of J_i can be completed before the deadline d_i (see Step 4). If the deadline cannot be met by M_j, J_i is rejected by Step 23.

The security level of each task in J_i on M_j is optimized in the following way. Recall that the security service weights used in Equations (2), (4), and (6) reflect the importance of the three security services, directly indicating that it is desirable to give higher priorities to security services with higher weights (see Step 5). In other words, enhancing security levels of more important services tends to yield a maximized security level of the task on M_j.

In the case of a particular security service $v_l \in \{a, e, g\}$, Step 10 escalates the security level $s_{i,k}^{v_l}$ while satisfying the following timing constraints (see Step 12). Step 21

1. **for** each task t_k of job J_i on site M_j **do**
2. Compute $\sigma_{i,k}^{j}$, the earliest start time of J_i on site M_j;
3. Compute the minimal security overhead $c_{i,k}^{min}$ of task t_k;
4. **if** $\sigma_{i,k}^{j} + e_{i,k} + c_{i,k}^{min} \le d_i$ **then** (See **Property 1**)
5. Sort the security service weights in a decreasing order, e.g., $w_i^{v_1} < w_i^{v_2} < w_i^{v_3}$, where $v_l \in \{a,e,g\}, 1 \le l \le 3$;
6. **for** each security service $v_l \in \{a,e,g\}, 1 \le l \le 3,$ **do**
7. $s_{i,k}^{v_l} = \min\{S_i^{v_l}\}$;
8. **for** each security service $v_l \in \{a,e,g\}, 1 \le l \le 3,$ **do**
9. **while** $s_{i,k}^{v_l} < \max\{S_i^{v_l}\}$ **do**
10. increase security level $s_{i,k}^{v_l}$;
11. Use **Equation (10)** to calculate the security overhead of t_k on M_j;
12. **if** $\sigma_{i,k}^{j} + e_{i,k} + \sum_{b\in\{a,e,g\}} c_{i,k}^{b}(s_{i,k}^{b}) > d_i$ (based on **Property 1**) **then**
13. decrease security level $s_{i,k}^{v_l}$; break;
14. **end while**
15. **end for**
16. $SL(s_{i,k}) = \sum_{b\in\{a,e,g\}} w_i^b s_{i,k}^b$
17. **else** Migrate t_k to another site M_r, subjecting to

$$\min_{1\le r\le n, r\ne j}\left\{\sigma_{i,k}^{j,r} + e_{i,k} + \sum_{b\in\{a,e,g\}} c_{i,k}^{b}(s_{i,k}^{b}) + \frac{\delta_i^r}{B_{r,j}}\right\} \le d_i$$

18. **end for**
19. **if Property 2** is satisfied **then /*** All the tasks in J_i can be finished before d_i
20. $y_{i,h} \leftarrow 1$, where $h = j$ or r; /* Accept job J_i */
21. /* Optimize quality of security, see **Equation (4)** */

Find site M_k for J_i, maximize $SL(s_i) = \sum_{k=1}^{p_i} \sum_{b\in\{a,e,g\}} w_i^b s_{i,k}^b$;

22. dispatch job J_i to M_k according to the schedule generated above;
23. **else** $y_{i,h} \leftarrow 0$; /* Reject J_i, since no feasible schedule is available */

Fig. 3. The SAREG scheduling algorithm

is able to maximize the security level of all the tasks in J_i by identifying a site M_h that provides the maximal security level and dispatching J_i to M_h (see Step 22).

The time complexity of the *SAREG* scheduling algorithm is given as follows.

Theorem 1. The time complexity of *SAREG* is *O(knm)*, where n is the number of sites in a Grid, m is the number of tasks running on a site, and k is the number of possible security level ranks for a particular security service v_l ($v_l \in \{a,e,g\}, 1 \le l \le 3$).

Proof. The time complexity of finding the earliest start time for the task on a site is *O(m)* (Step 2). To obtain the minimal security overhead c_i^{min} of the task; the time complexity is a constant *O(1)* (Step 3). Sorting the security service weights in a decreasing order (Step 5) takes a constant time *O(1)* since we only have 3 security services. To increase the task's three security levels to their possible maximal ranks under the constraints (Step 12), the worst case time complexity is *O(3km)* (Step 8 ~ Step 15). To find site M_h on which the security level of task is optimized (Step 20 ~ Step 22), the time complexity is *O(n)*. Thus, the time complexity of the *SAREG* algorithm is as follows: O(*n*)(*O(m) + O(1) + O(1) + O(3km)) + O(n) = O(knm)*. □

Since *n, m* and *k* cannot be very big numbers in practice, the time complexity of *SAREG* should be low based on the expression above. This time complexity indicates that the execution time of *SAREG* is a small value compared with task execution times (e.g., the real world trace used in our simulations shows that the average job execution time is 8031 Sec.). Thus, the CPU overhead of executing *SAREG-EDF* is ignored in our experiments.

5 Performance Evaluation

In the previous Section we proposed the SAREG scheduling algorithm, which integrates security requirements into scheduling for real-time applications on Grids. Now we are in a position to evaluate the effectiveness of SAREG by conducting extensive simulations based on a real world trace from San Diego Supercomputer Center (SDSC SP2 log). The real trace was sampled on a 128-node (66MHz) IBM SP2 from May 1998 through April 2000. To simplify our experiments, we utilized the first three months data with 6400 parallel jobs in simulation.

In purpose of revealing the strength of SAREG, we compared it against two well-know scheduling algorithms, namely, *Min-Min* and *Sufferage* [25] in addition to a traditional real-time scheduling algorithm - the Earliest Deadline First algorithm (*EDF)*. Min-Min and Sufferage are non-preemptive task scheduling algorithms, which were designed to schedule a stream of independent tasks onto a heterogeneous distributed computing system such as a Grid. Note that Min-Min and Sufferage are representative dynamic scheduling algorithms for Grid environments, and they were successfully applied in real world distributed resources management systems such as SmartNet [11]. For the sake of simplicity, throughout this section *Sufferage* is referred to as *SUFFER*.

To emphasize the non-security-aware characteristic of *EDF* algorithm, we refer to the EDF algorithm as *NS-EDF* (non-security-aware *EDF*) in this paper. Although the *NS-EDF* algorithm, a variation of *EDF*, is able to schedule real-time jobs with security requirements, it makes no effort to optimize quality of security. Rather, it randomly selects a security level for each task in a real-time job. The three baseline scheduling algorithms are briefly described below.

(1) *MINMIN*: For each submitted real-time job, a Grid site that offers the earliest completion time is tagged. Among all the mapped tasks, the one that has the minimum earliest completion time is chosen and then allocate to the tagged site. The

MINMIN scheduler randomly selects security levels of security services required by tasks of a real-time job.

(2) *SUFFER*: Allocating a site to a submitted job that would "suffer" most in terms of completion time if that site is not allocated to it. Again, the SUFFER scheduler randomly chooses security levels for security requirements posed by an arriving job.

(3) *NS-EDF*: Tasks with the earliest deadlines are always executed first. We modified the traditional *EDF* algorithm in a way that it can randomly picks values within the security level ranges of services required by tasks. The following expression is held in the NS-EDF algorithm: $\forall 1 \le k \le p_i, b \in \{a, e, g\} : s_{i,k}^b = \text{random}\{S_i^b\}$.

The ultimate goal of comparing SAREG against MINMIN and SUFFER is to demonstrate schedulability performance improvements over existing scheduling algorithms in a real-time computing environment, whereas the purpose of comparing *SAREG* with *NS-EDF* is to show security performance benefits gained by employing SAREG in a Grid environment. This section is organized as follows. Section 5.1 describes our simulator and important system parameters. Section 5.2 is to examine the performance improvements of SAREG over the three baseline algorithms. In Section 5.3 we investigate the performance impacts of the number of computing nodes in a simulated four-site Grid. Section 5.4 addresses the performance sensitivity of the SAREG algorithm to CPU capacities of the nodes in a Grid.

5.1 Simulator and Simulation Parameters

Before presenting the empirical results in detail, we present the simulation model as follows. A competitive advantage of conducting simulation experiments is that performance evaluation on a Grid can be accomplished without additional hardware cost. The Grid simulator was designed and implemented based on the model and the algorithm described in the previous sections. Table 4 summarizes the key configuration parameters of the simulated Grid used in our experiments. The parameters of nodes in Grid are chosen to resemble real-world workstations like IBM SP2 nodes.

We modified a real world trace[1] by adding randomly generated deadlines for all tasks in the trace. The assignment of deadlines is controlled by a deadline base, or laxity, denoted as β, which sets an upper bound on tasks' slack times. We use Equation (11) to generate job J_i's deadline d_i.

$$d_i = a_i + e_i + c_i^{max} + \beta, \cdot \tag{11}$$

where a_i and e_i are the arrival and execution times obtained from the real-world trace. c_i^{max} is the maximal security overhead (measured in ms), which is computed by Equation (12).

$$c_i^{\max} = \sum_{j \in \{a,e,g\}} c_i^j \left(\max\left\{S_i^j\right\}\right) \cdot \tag{12}$$

where $c_i^j\left(\max\left\{S_i^j\right\}\right)$ represents the overhead of the jth security service for J_i when the corresponding maximal requirement is fulfilled.

[1] http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_sp2.html

Table 4. Characteristics of system parameters

Parameter	Value (Fixed) - (Varied)
CPU Speed	(2) – (4, 8, 16)
β (Deadline Base, or Laxity)	(50 second) – (200, 400, 800 second)
Network bandwidth	5 MB/Second
Number of sites	(4) – (8, 16, 32)
Number of nodes	(184)- (256, 320,384)
Mean size of data to be secured	50KB for short jobs, 500KB for medium jobs, 1MB for long jobs
Mean size of input data	100MB for short jobs, 500MB for medium jobs, 1TB for large jobs
Mean size of application code	500KB for short jobs, 5MB for medium jobs, 50MB for large jobs
Required security services	Encryption, Integrity and Authentication
Weights of security services	Authentication=0.2; Encryption=0.5; Integrity=0.3

"Job number", "submit time", "execution time" and "number of requested processors" of jobs submitted to the Grid are taken directly from the trace. "size of input file", "size of application code", "size of output file" and "deadlines" are synthetically generated in accordance with the above model, since these parameters are not available in the trace. Security requirements are randomly generated from 0.1 to 1.0 for each security service. When a job has to be remotely executed to meet its deadline, we must consider its migration cost, which is factored in Equation 11. In order to measure the migration cost of a job, we need to estimate the network bandwidth and the amount of data to be transferred. Vazhkudai *et. al.* [40] measured the end-to-end bandwidth between two remote super-computing centres using GridFTP. It was discovered that the network bandwidth varies from 1.5 to 10.2 MB/sec. In our simulation experiments, the network bandwidth was randomly drawn from a uniform distribution with range 1.5 to 10.2 MB/sec., which can resemble practical network bandwidth in existing distributed systems. The synthesized deadline weakens correlations between real-time requirement and other workload characteristics. However, in the experiments we can examine performance impacts of deadlines on system performance by controlling the deadlines as fundamental simulation parameters (see Section 5.2).

The performance metrics by which we evaluate system performance include:

security value: (see Equation 6).

guarantee ratio: measured as a fraction of total submitted jobs that are found to be schedulable).

overall system performance: defined as a product of security value and guarantee ratio.

5.2 Overall Performance Comparisons

The goal of this experiment is two fold: (1) to compare the proposed SAREG algorithm against the three baseline schemes, and (2) to understand the sensitivity of SAREG to parameter β, or deadline base (Laxity). To stress the evaluation, we as-

sume that each job arrived in the Grid requires the three security services. Without loss of generality, it is assumed that time spent handling page faults is factored in jobs' execution time.

Figure 4 shows the simulation results for these four algorithms on a Grid with 4 sites (184 nodes) where the CPU power is fixed at 100MIPS. We observe from Figure 6 (a) that SAREG and NS-EDF exhibit similar performance in terms of guarantee ratio (the performance difference is less than 2%), whereas the guarantee ratios of SAREG are a lot higher than those of MINMIN and SUFFER algorithms. The reason for the performance improvements of SAREG over MINMIN and SUFFER is two fold. First, SAREG is a real-time scheduler, while MINMIN and SUFFER are non-real-time scheduling algorithms. Second, SAREG judiciously enhances the security levels of accepted jobs under the condition that the deadlines of the accepted jobs are guaranteed.

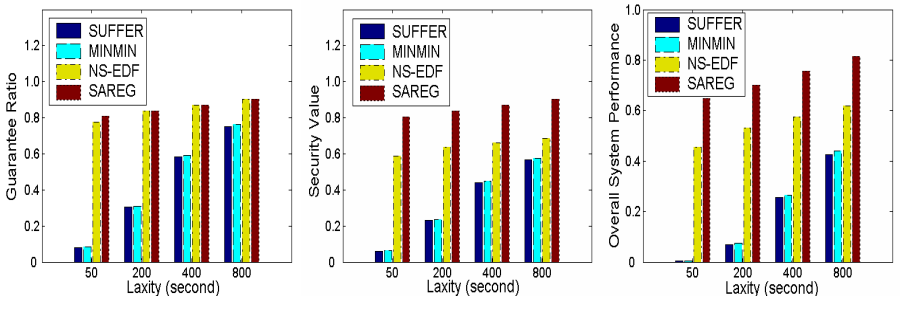


Fig. 4. Performance impact of deadline

Figure 4 (a) illustrates that the guarantee ratios of four algorithms increase with the increasing value of the laxity. This is because the large deadline leads to long slack times, which in turn tend to make the deadlines more likely to be guaranteed.

Figure 4 (b) plots security values of the four alternatives when the deadline base is increased from 50 to 800 Sec. Comparing with the average execution time of all jobs in the trace, which is 8030.8 Sec., the laxity range [50, 800] is reasonable. Figure 4 (b) reveals that SAREG consistently performs better, with respect to quality of security, than all the other three approaches. When the deadlines are tight, the security values of SAREG are much higher than those of MINMIN and SUFFER. In addition, SAREG consistently outperforms NS-EDF when the laxity varies from 50 seconds to 800 seconds. This is because that SAREG can improve accepted jobs' security levels under constraints of their deadlines and resources availability, while NS-EDF makes no effort to optimize the security levels. More specifically, NS-EDF merely randomly chooses a security level within the corresponding security requirement range. Interestingly, when the deadlines become tight, the performance improvements of SAREG over the three competitor algorithms are more pronounced. The results clearly indicate that Grids can gain more performance benefits from our SAREG approach under the circumstance that real-time tasks have urgent deadlines.

The overall system performance improvements achieved by SAREG are plotted in Figure 4(c). The first observation deduced from Figure 4(c) is that the value of overall system performance increases with the laxity. This is mainly because the overall system performance is a product of security value and guarantee ratio, which become higher when the deadlines are loose due to the high laxity value.

A second observation made from figure 4(c) is that the SAREG algorithm significantly outperforms all the other three alternatives. This can be explained by the fact that although the guarantee ratios of SAREG and NS-EDF are similar, SAREG considerably improves security values over the other algorithms, while achieving much higher guarantee ratio than MINMIN and SUFFER. This result suggests that if quality of security is the sole objective in scheduling, SAREG is more suitable for Grids than the other algorithms. By contrast, if schedulability is the only performance objective, SAREG can maintain similar guarantee ratios as those of NS-EDF, whose security performance is the second best among the four algorithms.

Last but not least, Figure 4(c) indicates that the overall performance improvement of SAREG over the other three algorithms becomes more pronounced when the deadlines are tighter, implying that more performance benefits can be obtained by SAREG for real-time applications with small slack times. This is because the SAREG approach is less sensitive to the change in deadlines than the other approaches.

5.3 Scalability

This experiment is intended to investigate the scalability of the SAREG algorithm. We scale the number of sites in the Grid from 4 to 32. Figure 5 plots the performances as functions of the number of sites in the Grid. The results show that the SAREG approach exhibits good scalability.

Figure 5 shows the improvement of SAREG in overall system performance over the other three heuristics. It is observed from Figure 5 that the amount of improvement over MINMIN and SUFFER maintains almost the same level with theincreasing value of the site number. This result can be explained by the non-real-time nature of MINMIN and SUFFER, which schedules tasks that change the expected site ready time status by the least amount that any assignment could.

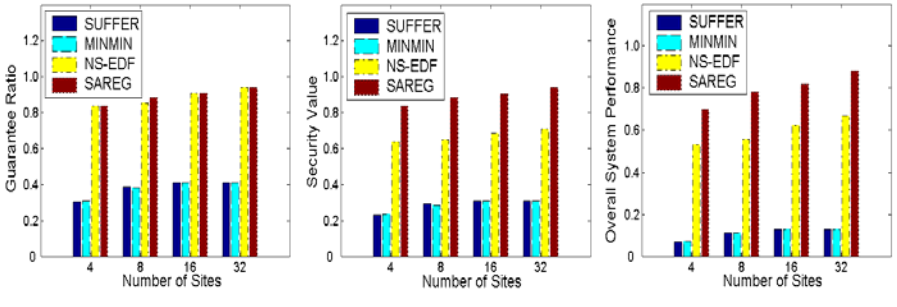


Fig. 5. Performance impact of number of sites

5.4 Sensitivities to CPU Capacity

To examine performance sensitivities of the four algorithms to CPU capacity, in this set of experiments we varied the CPU capacity (measured as speedup over the baseline computational node) from 2 to 16. Specifically, the CPU speed of the IBM SP2 66MHz nodes is normalized to 1. We escalate the CPU capacity of the nodes to a normalized value of 2, 4, 8, and 16, respectively. Therefore, the execution times (including security overhead) could be 1/2, 1/4, 1/8 and 1/16 of that of original values, respectively. Also, we select a 200 seconds laxity and a four-site simulated Grid with total 184 nodes. This experiment is focused on evaluating the performance impact of CPU speedup on the four algorithms under a situation where deadlines are relatively tight and the number of nodes is less than sufficient.

The results reported in Figure 6 reveal that the SAREG algorithm outperforms the other three alternatives in terms of security value and overall system performance. However, the discrepancy of guarantee ratio performance between SAREG and NS-EDF is almost zero. This is because SAREG can accept the same number of submitted tasks as NS-EDF when the node's CPU speed is so fast that the security overhead is trivial and thus has little effect on the guarantee ratio performance. Figure 6 shows that MINMIN and SUFFER only slightly improve their performance when the computing capacity of the Grid is increased. The results can be explained by the following two reasons. First, the trace used in the simulation has approximate fixed job arrival rate, meaning that the decrease jobs' execution time unnecessarily improves guarantee ratio. Second, the deadlines of all submitted tasks are relatively tight. As we can see from Fig 6., the laxity has great influence on MINMIN and SUFFER in terms of the guarantee ratio.

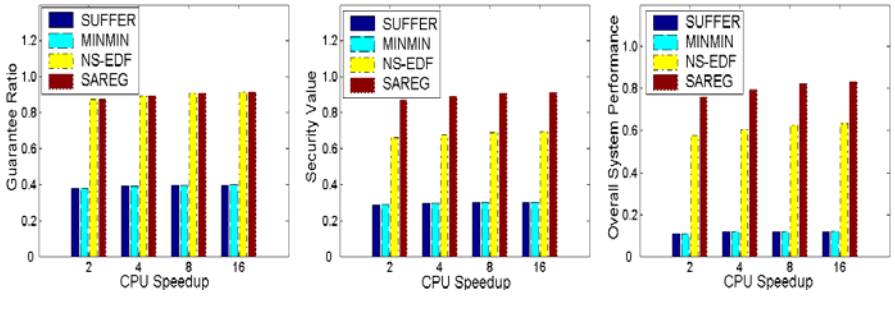


Fig. 6. Performance impact of CPU speedup

6 Summary and Future Work

In this paper, we proposed a novel scheduling algorithm, or SAREG, for real-time applications on computational Grids. The SAREG approach paves the way to the design of security-aware real-time scheduling algorithms for Grid computing environments. To make the SAREG scheduling algorithm practical, we presented a

mathematical model in which a scheduling framework, security-sensitive real-time jobs, and security overheads are formally described. With the mathematical model in place, we can incorporate security overheads into the process of real-time scheduling. We introduced a new performance metric-security value, which was used to measure the quality of security experienced by all real-time jobs whose deadlines can be met.

To quantitatively evaluate the effectiveness of the SAREG algorithm, we conducted extensive simulations based on a real world trace from a supercomputing centre. Experimental results under a wide spectrum of workload conditions show that SAREG significantly enhances quality of security for real-time applications while maintaining high guarantee ratios. More importantly, SAREG-EDF achieves overall system performance over three existing scheduling algorithms (MIN-MIN, Sufferage, and EDF) by averages of 286.34%, 272.14%, and 33.86%, respectively.

Future study in this research is to incorporate more security services into our security overhead model. Additional security services include authorization and auditing services.

Acknowledgements

This work was partially supported by a start-up research fund (103295) from the research and economic development office of the New Mexico Institute of Mining and Technology. We are grateful to five anonymous referees of this paper for their very comprehensive suggestions and comments, which greatly improved the original manuscript.

References

1. Abdelzaher, T.F., Atkins, E. M., Shin, K.G.: QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. IEEE Trans. Computers, Vol. 49, No. 11, Nov. (2000)1170-1183
2. Ahmed, Q., Vrbsky, S.: Maintaining security in firm real-time database systems. Proc. 14th Ann. Computer Security Application Conf., (1998)
3. Azzedin, F., Maheswaran, M.: Towards trust-aware resource management in grid computing systems. Proc. 2nd IEEE/ACM Int'l Symp. Cluster Computing and the Grid, May (2002)
4. Bosselaers, A., Govaerts, R., Vandewalle, J.: Fast hashing on the Pentium. Proc. Advances in Cryptology, LNCS 1109, Springer-Verlag, (1996)298-312
5. Braun, T.D.: A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. Proc. Workshop on Heterogeneous Computing (1999)
6. Caron, E., Chouhan, P.K., Desprez, F.: Deadline Scheduling with Priority for Client-Server Systems on the Grid. Proc. Int'l Workshop Grid Computing, Nov. (2004) 410-414
7. Connelly, K., Chien, A.A.: Breaking the barriers: high performance security for high performance computing. Proc. Workshop on New security paradigms, Virginia, Sept. (2002)
8. Deepakumara, J., Heys, H.M., Venkatesan, R.: Performance comparison of message authentication code (MAC) algorithms for Internet protocol security (IPSEC). Proc. Newfoundland Electrical and Computer Engineering Conf., St. John's, Newfoundland, Nov. (2003)

9. Elkeelany, et al.: Performance analysis of IPSec protocol: encryption and authentication. Proc. IEEE Int'l Conf. Communications, New York, NY, April-May (2002)1164-1168
10. Eltayeb, M., Dogan, A., Ozunger, F.: A data scheduling algorithm for autonomous distributed real-time applications in grid computing. Proc. Int'l Conf. Parallel Processing, Aug. (2004) 388-395
11. Freund, et al.: Scheduling resources in multi-user, heterogeneous computing environments with SmartNet. Proc. Heterogeneous Computing Workshop, March (1998) 184-199
12. George, B., Haritsa, J.: Secure transaction processing in firm real-time database systems. Proc. ACM SIGMOD Conf., May, (1997)
13. Halang, W. A. et al.: Measuring the performance of real-time systems. Int'l Journal of Time-Critical Computing Systems, (2000) 59-68
14. Harbitter, A., Menasce, D. A.: The performance of public key enabled Kerberos authentication in mobile computing applications. Proc. ACM Conf. Computer and Comm. Security, (2001)78-85
15. Harchol-Balter, M., Downey, A.: Exploiting Process Lifetime Distributions for Load Balacing. ACM transaction on Computer Systems, vol. 3, no. 31, (1997)
16. He, L., Jarvis, S. A., Spooner, D. P., Chen, X., Nudd, G.R.,: Dynamic Scheduling of Parallel Jobs with QoS Demands in Muticluster and Grids. Proc. Int'l Workshop Grid Computing, Nov. (2004) 402-409
17. Humphrey, M., Thompson, M.R., Jackson, K.R.: Security for Grids. Proc. of the IEEE, March, (2005) 644-652
18. In, J.U., Avery, P., Cavanaugh, R., Ranka, S.: Policy based scheduling for simple quality of service in grid computing. Proc. Int'l Symp. Parallel and Distributed Processing, April (2004) 23-32
19. Irvine, C., Levin, T.: Towards a taxonomy and costing method for security services. Proc. 15th Annual Computer Security Applications Conference, (1999)
20. Lee, J., Tierney, B., Johnston, W.: Data intensive distributed computing: a medical application example. Proc. High Performance Computing and Networking Conf., April (1999)
21. Li, K.: Experimental performance evaluation of job scheduling and processor allocation algorithms for grid computing on metacomputers. Proc. Int'l Symp. Parallel and Distributed Processing, April (2004)
22. Liden, S.: The Evolution of Flight Management Systems. Proc. IEEE/AIAA 13th Digital Avionics Systems Conf., (1995) 157-169
23. Liu, C. L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM, Vol.20, No.1, (1973) 46-61
24. Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., Freund, R.F.: Dynamicmatching and scheduling of a class of independent tasks onto heterogeneous computing systems. Proc. IEEE Heterogeneous Computing Workshop, Apr. (1999)30–44
25. Nahum, E., O'Malley, S., Orman, H., Schroeppel, R.: Towards High Performance Cryptographic Software. Proc. IEEE Workshop Architecture and Implementation of High Performance Communication Subsystems, August (1995)
26. 26. Nilsson, J., Dahlgren, F.: Improving performance of load-store sequences for transaction processing workloads on multiprocessors. Proc. Int'l Conference on Parallel Processing, 21-24 Sept. (1999) 246-255
27. Qin, X., Jiang, H., Zhu, Y., Swanson, D.R.: Towards Load Balancing Support for I/O-Intensive Parallel Jobs in a Cluster of Workstations. Proc. 5th IEEE Int'l Conf. on Cluster Computing, Dec. (2003) 100-107

28. Qin, X., Jiang, H.: Improving Effective Bandwidth of Networks on Clusters using Load Balancing for Communication-Intensive Applications. Proc. 24th IEEE Int'l Performance, Computing, and Communications Conf., Phoenix, Arizona, April (2005)
29. Qin, X.: Improving Network Performance through Task Duplication for Parallel Applications on Clusters. *Proc. 24th IEEE Int'l Performance, Computing, and Communications Conference,* Phoenix, Arizona, April (2005)
30. Qin, X., Jiang, H., Swanson, D. R.: An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems. Proc. 31st Int'l Conf. Parallel Processing, Computing, and Communications Conf., Phoenix, Arizona, April (2005) 360-368
31. Qin, X., Jiang, H.: Dynamic, Reliability-driven Scheduling of Parallel Real-time Jobs in Heterogeneous Systems. Proc. 30th Int'l Conf. Parallel Processing, Sept. (2001) 113-122
32. Qin, X., Jiang, H.: Data Grids: Supporting Data-Intensive Applications in Wide Area Networks. High Performance Computing: Paradigm and Infrastructure (2004)
33. Ramamritham, K., Stankovic, J. A.: Dynamic task scheduling in distributed hard real-time system. IEEE Software, Vol. 1, No. 3, July (1984)
34. Schreur, J.: B737 Flight Management Computer Flight Plan Trajectory Computation and Analysis. Proc. Am. Control Conf., (1995) 3419-3429
35. Son, S.H., Zimmerman, R., Hansson, J.: An adaptable security manager for real-time transactions. Proc. 12th Euromicro Conf. Real-Time Systems, June (2000) 63–70
36. Son, S. H., Mukkamala, R., David, R.: Integrating security and real-time requirements using covert channel capacity. IEEE Trans. Knowledge and Data Engineering, Vol. 12 , No. 6, (2000) 865–879
37. Stankovic, J. A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms. Kluwer Academic Publishers (1998)
38. Theys, M. D., Tan, M., Beck, N., Siegel, H. J., Jurczyk, M.: A mathematical model and scheduling heuristic for satisfying prioritized data requests in an oversubscribed communication networks. IEEE Trans. Parallel and Distributed Systems, Vol. 11, No. 9, Oct. (2000) 969-988
39. Thomadakis, M.E., Liu, J.C.: On the efficient scheduling of non-periodic tasks in hard real-time systems. Proc. 20th IEEE Real-Time Systems Symp., (1999) 148-151
40. Vazhkudai, S., Schopf, J. M., Foster, I.: Predicting the Performance of Wide Area Data Transfers. Proceedings of the 16th International Parallel and Distributed Processing Symposium, April (2002)
41. Wright, R., Shifflett, D.J., Irvine, C.E.: Security Architecture for a Virtual Heterogeneous Machine. Proc. 14th Ann. Computer Security Applications Conference (1998)
42. Wu, M., Sun, X.H.: Memory conscious task partition and scheduling in Grid Environments. Proc. Int'l Workshop Grid Computing, Nov. (2004) 138-145
43. Xie, T., Sung, A., Qin, X.: Dynamic Task Scheduling with Security Awareness in Real-Time Systems. Proc. Int'l Symp. Parallel and Distributed Processing, Denver, April (2005)
44. Xie, T., Qin, X., Sung, A.: SAREC: A Security-aware Scheduling Strategy for Real-Time Applications on Clusters. Proc. 34th Int'l Conf. Parallel Processing, Norway, (2005)5-12
45. Xie, T., Qin, X.: A New Allocation Scheme for Parallel Applications with Deadline and Security Constraints on Clusters. The 2005 IEEE International Conference on Cluster Computing , Boston, USA, September 27-30 (2005)

Unfairness Metrics for Space-Sharing Parallel Job Schedulers

Gerald Sabin and P. Sadayappan

The Ohio State University, Columbus OH 43201, USA
{sabin, saday}@cse.ohio-state.edu

Abstract. Sociology, computer networking and operations research provide evidence of the importance of fairness in queuing disciplines. Currently, there is no accepted model for characterizing fairness in parallel job scheduling. We introduce two fairness metrics intended for parallel job schedulers, both of which are based on models from sociology, networking, and operations research. The first metric is motivated by social justice and attempts to measure deviation from arrival order, which is perceived as fair by the end user. The second metric is based on resource equality and compares the resources consumed by a job with the resources deserved by the job. Both of these metrics are orthogonal to traditional metrics, such as turnaround time and utilization.

The proposed fairness metrics are used to measure the unfairness for some typical scheduling policies via simulation studies. We analyze the fairness of these scheduling policies using both metrics, identifying similarities and differences.

1 Introduction

There has been considerable research [1, 2, 3, 4] focused on various aspects of job schedulers for parallel machines, much of it being reported at the annual Workshop on Job Scheduling Strategies for Parallel Processing [5]. These studies have generally focused on user performance metrics (such as turnaround time and slowdown) and system metrics (such as utilization and loss of capacity).

Scheduling algorithms have been developed with the intent of providing users better functionality and enhanced performance, while improving system utilization. But the issue of fairness has not been much addressed in the context of job scheduling. We believe that this is in large part due to the lack of accepted metrics for characterizing fairness. Whereas performance metrics such as wait time, response time, slowdown and utilization are widely accepted and used by the job scheduling research community, fairness metrics are not well established. Researchers have analyzed worst case and 95th percentile performance data for the more standard user metrics [6, 7] in an attempt to indirectly characterize scheduling schemes with respect to the issues of starvation and fairness. However, there has been very little work which proposes direct fairness metrics for job schedulers.

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 238–256, 2005.

In contrast to the work in the job scheduling community, there is a body of existing work on the topic of fairness in fields such as sociology, computer network scheduling and queuing theory. The Resource Allocation Queuing Fairness Metric (RAQFM) [8, 9, 10] is a model for assessing fairness in queuing systems. RAQFM evenly divides available resources amongst all jobs to determine the "fair" amount of resources which should have been assigned to each job.

Studies [11, 12] show that fairness in queues is very important to humans. For instance, a long wait at a supermarket is made more tolerable if all customers are served in order. Further, it has been observed that customers get upset if they are treated unfairly, even if the absolute wait time is short. Research by Larson [12] has shown that fast food restaurant customers prefer a longer, single queue rather than a multiple queue system with a shorter wait time. This can be attributed to a perceived increase in fairness due to the knowledge that the users are served in arrival order. Mann [11] studied queues for long over night waits at multiple events. He and his associates note the importance of preserving the FCFS queuing order. They analyze the impact of queue position on "queue jumping" and report that the response to "queue jumping" can range from "jeering and verbal threats" to physical altercations. While these queuing situations have significant differences from a non-preemptive space sharing parallel job scheduler, there also exist important similarities. For instance, it is postulated that the human responses to unfairness result from contention for "valued" resources (such as the event tickets in the previous example and the compute cycles in job scheduling) and the perception that a user who has waited longer somehow deserves the resources more.

Thus it is important that fairness be considered in addition to common user metrics (such as turn around time and wait time). By characterizing a center's scheduling policy through suitable fairness metrics, system administrators could assure their users that they are being treated fairly. In this paper, we present two rather distinctly different approaches to quantifying fairness in job scheduling, and assess several standard job scheduling strategies by use of these fairness metrics.

The remainder of this paper is organized as follows: Sect. 2 provides a motivation for fairness by introducing past fairness work in other disciplines. Section 3 introduces the proposed fairness metrics for parallel job schedulers. Sections 4 and 5 provide a simulation study showing the relative fairness of prevalent scheduling schemes. Section 6 provides possible directions for future work, and Sect. 7 concludes the paper.

2 Motivation and Related Work

Networking, sociology and operational research provide two broad categories of fairness. The first is based on social justice and is centered on the user's perception of fairness in terms of expected order of service (arrival order). This type of fairness model is common in sociology and operations research studies. The second fairness category is based on equally dividing the resources (servers)

amongst all active users. This concept is based on the fundamental belief that all users equally deserve the resources whenever they are present. Resource equality based metrics are prevalent in both computer networking and queuing systems.

Social justice is the foundation for the "slips and skips" view of fairness [12, 13, 14] in queuing systems. They introduce "slips and skips" as a means to measure "social justice" in queuing systems. A slip occurs when the queuing position is worsened due to being overtaken by a later arriving entity. A skip is the result of an improved queue position due to the act of overtaking an earlier arriving entity. Rafaeli et. al. [15] performed studies which concluded that perceived fairness in queues is essential and can actually be more important than wait time. Therefore, it seems that it would be helpful to show users how fair a system is and more specifically how fairly an individual's jobs were treated.

The RAQFM [8, 9, 10] is an example of the second class of fairness categories. It calculates fairness in queuing systems by taking the difference between the "deserved" service attributed to a job and subtracting the actual service provided to the job. The deserved service is computed by equally dividing all resources amongst all active users in the system at each time quantum. The deserved resources are defined as $\delta_i = \int_{a_i}^{d_i} 1/N(t)dt$, where $N(t)$ is the number of jobs in the system, a_i is the arrival time and d_i is the departure or completion time. The discrimination is defined as $D_i = \delta_i - s_i$, where s_i is the service time of job i. Three important observations made by Raz et. al. [10] are: 1) if service times are equal then FCFS is the most fair non-preemptive policy, 2) if arrivals times are the same then SJF is the most fair queuing priority, and, 3) a processor sharing scheme is the only ideally fair queuing policy.

There is also queuing fairness work in the context of streaming networks [16, 17, 18]; much of this work is based on max-min fairness, which is similar in spirit to RAQFM. This approach attempts to "evenly" divide bandwidth amongst active flows or packets such that a job can not receive more resources if it would result in reducing the resources of the least serviced job.

RAQFM is a metric which incorporates job seniority and a job's service time. In contrast, other queuing fairness [19, 20, 21] metrics based on a job's slowdown do not explicitly take both job seniority and the job's service time into account. Weirman and Balter [19] use the max mean fairness of all jobs, and consider a policy fair if all job categories have a mean slowdown of no more than $1/(1-\rho)$, where ρ is the load. They show that using this method no non-preemptive schedule is fair, processor sharing is fair, and SJF is more fair the FCFS. Schwiegelshohm and Yahyapour[22] introduce a parallel job scheduling algorithm that can bound a job's slowdown.

Sabin et. al.[23] measure the fairness of individual sites in a multi-site job scheduling environment. The focus of the multi-site fairness work is to ensure that lightly loaded local sites are not too adversely affected by joining a job sharing consortium. That work does not address the relative ordering (and fairness) of individual jobs, as we do in this paper. The metric suggested in [23] compares the performance of each job with job sharing to the performance of each job without job sharing. The authors conclude that the lack of focus on fair-

ness may have lead to unfair decisions in current meta-schedulers, and suggest techniques to improve fairness.

3 Unfairness Metrics

We introduce two fairness metrics. The first metric is related to "slips" and "skips" discussed earlier, and the second metric is based on an equal distribution of system resources amongst all active jobs in the system, similar to RAQFM [8, 9, 10] and network flow scheduling. The metrics are very different in nature and quantify significantly different notions of fairness in the schedule. However, both metrics appear very appropriate to consider when performing a fairness analysis for a schedule. We compare different scheduling schemes against both fairness metrics, to identify commonalities and differences.

These metrics are intended to be orthogonal to the traditional performance metrics. There may be cases where a more fair scheme also exhibits better user metrics such as turnaround time. However, there may be scenarios where a more fair scheme has worse performance metrics. For instance, a non-backfilling FCFS policy will be entirely socially just, but has a very poor average turnaround time. To fully evaluate a scheduling strategy, the fairness should be evaluated in addition to more traditional user and system metrics. There is a large body of research which evaluates the other user and system metrics for the scheduling policies studied in this paper; therefore this paper focuses on fairness and not on the variations in the other metrics. This balance is not unlike balancing wait time and fairness in physical queues as done in [12] and more formally in [15]. Rafaeli's studies [15] showed that anger due to queuing is more closely related to the perception of unfairness rather than the absolute wait time.

3.1 Fair Start Time Analysis

Fair start time (FST) [24] based metrics are proposed as a means to measure social justice for parallel job schedulers. In order to motivate our approach, we use an analogy. Consider a deli line, having many workers, each performing only one task. Would it be fair for a person who arrived later to have access to the cashier earlier than the first person in the line? In general, this would be considered unfair and socially unjust. But if the later person is only ordering a soda, and arrives at the cashier in time to have the sale completed before the earlier person's sandwich order is ready, there isn't really any social injustice. The later person obtained a resource "out of order", but did not delay the earlier arriving person. This is analogous to a later arriving job that backfills but has no negative effect on the start time of earlier arriving jobs. We will refer to this situation as a "benign backfill". Therefore, this metric is similar to measuring the "slips", as we are trying to measure the effect of jobs being run out of the preferred order (FCFS). However, as demonstrated in the analogy, there are significant differences which affect how we define this metric.

Parallel job schedulers must inherently deal with multi-resource jobs and multiple "servers" (processors). In this context, jobs use multiple resources

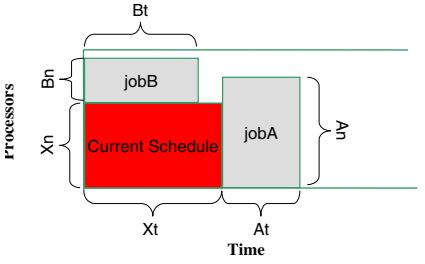
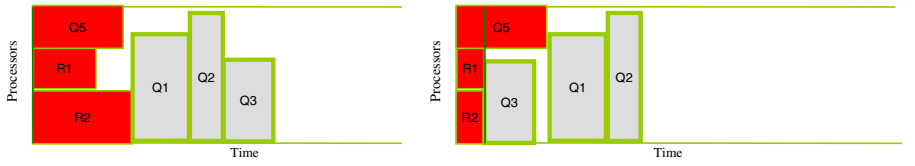


Fig. 1. An example of a benign backfill due to multiple resources where the apparent skip by jobB does not affect jobA

(processors); this means that all "skips" are not necessarily harmful. It is possible for a job to "skip" ahead and not affect any other job in the schedule. In a single resource queuing system, any job receiving service "out of order" inherently delays all other earlier arriving jobs. In a multi-resource job scenario, this is not always the case. It is possible that a "skip" is a benign backfill. For instance, assume that we have Sn total resources (nodes) and $Sn - Xn$ free nodes for Xt seconds, $jobA$ needs An nodes for At time, and $jobB$ needs Bn nodes for Bt time. Assume $Bt < Xt$ and $Bn < (Sn - Xn) < An$ and $jobA$ arrived before $jobB$. The start time of $jobA$ is not affected by $jobB$ running before $jobA$, even though $jobB$ arrived later (see Fig. 1). Therefore, the early start of $jobB$ represents a benign backfill and is not unfair.

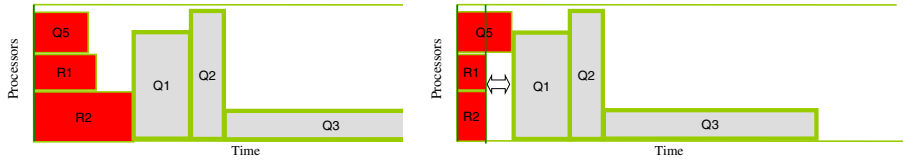
Informally, how fairly a job is treated depends on whether the job could have run at an earlier time had no later arriving job been serviced. If it is delayed because of a later arriving job, it is considered to be treated unfairly. This is analogous to the deli line, and answers the question "did any later arriving job affect my start time". Based on the previous discussion, benign backfilling is obviously possible, and is in fact the desired result of backfilling. Therefore, simply counting how many jobs leapfrogged ahead of a particular job is not an adequate metric to characterize unfairness. Further, it is difficult to identify the benign backfills, as a backfill may only adversely affect some of the active jobs, due to inexact users estimates and the dynamic nature of online job scheduling.

Strict Fair Start Time. Since we can not directly measure the number of "skips" and/or "slips", we decide to measure the effect of later arriving jobs on the start time of each job. To measure this effect we run a "what if" simulation after all jobs have completed. Therefore the "what if" simulation has all the necessary data (actual runtimes, resource constraints, etc.) to determine exactly what would have happened had no later arriving jobs been in the system. This allows us to determine if, and by how much, the job was affected by later arriving jobs. This simulation takes into account the state of the schedule (running jobs and idle resources) when the job arrived. We are able to start the simulation from this state and determine when the job would have started if no later arriving jobs were ever in the system. We call the start time in this "what if" schedule the Fair Start Time (FST) for this job. If the actual start time of the job is not



(a) (b)

Fig. 2. An example of a job starting before it's fair start time. Job Q5 backfills in the top figure. In the bottom figure, job R1 and R2 complete early, allowing Q3 to start earlier than it would have if Q5 were not allowed to backfill.



(a) (b)

Fig. 3. A latter arriving job (Q5) delaying earlier arriving jobs (Q1). The delay is caused by the inaccurate estimates of R1 and R2. The top figure shows job Q5 backfilling into the scheduling window and the bottom figure shows the schedule after R1 and R2 complete early.

greater than this FST, the job was not delayed by any later arriving jobs, and any backfills (out of order executions) must have been benign with respect to this job.

It is possible that due to the dynamics of later arriving jobs, that the job actually had an earlier start time in the actual trace. This is because later arriving jobs could have created "holes" in the schedule for this job to backfill into (see Fig. 2). Therefore, if the job's FST is less than is actual start time is has been treated unfairly (see Fig. 3), however if the jobs FST is greater than the actual start time, the job has been given preferential treatment.

Relaxed Fair Start Time. We next define a less strict notion of the FST, motivated by a couple of issues pertaining to the strict FST metric defined above:

- In the process of generating a job's strict FST, it may have skipped ahead via backfilling. If in actual execution, the job was not able to leapfrog ahead of earlier jobs via backfilling, but was not otherwise delayed, should it really be considered to be unfairly treated?
- If the job backfills when computing its strict FST, and again during actual execution, this backfilling might force an earlier arrived job to violate its strict FST. Thus it may be the case that the collection of strict FST's for the jobs of a trace implies an inherently infeasible schedule.

Therefore, we define a "relaxed" FST metric, very similar to the strict FST in most ways. The only difference is that the job we are determining the FST for is not allowed to backfill (it can not start until all other jobs submitted before it have started). This gives an FST that is strictly greater than or equal to the Strict FST, thus we call this the Relaxed FST. It should be noted that while the job in question will not directly force another job to be treated unfairly, the set of relaxed FST's is not necessarily a feasible schedule.

The pseudo-code below shows a possible implementation to compute the strict and relaxed fair start time. This would need to be added to the scheduler. The code assumes that a snapshot of the schedule (with information sufficient to start a simulation) is taken upon job arrival. This snapshot is retrieved with the getScheduleSnapshotAtArrival() method. As can be seen, the only difference between the relaxed and strict metric is when the newly arriving job is added to the "what if" simulation. It should be noted that this must be performed after all jobs which arrived before j have completed, so that the scheduler can perform an accurate simulation using the actual runtime of the jobs.

A consequence of both of the proposed approaches (Strict and Relaxed) to characterizing fairness is that the FST of a particular job in a trace will generally be different under different scheduling strategies. Consider a job X that arrives at t=60 minutes. It is possible for it to have a fair-start time of 100 minutes and an actual start time of 120 minutes with scheduling strategy A, and a fair-start time of 200 minutes and an actual start time of 180 minutes with scheduling scheme B. Under our model of fairness, the job is unfairly treated by scheduling scheme A, but is fairly treated by scheduling scheme B, although the actual start time for scheme B is much worse. Is this reasonable? We believe that the conclusion regarding fairness is not inappropriate, since fairness and performance are orthogonal considerations. Although the user might prefer scheduling scheme A because it provides a better response time, it does not mean that it is a fairer scheme. Moreover, there has been sociological research that suggests users might actually prefer the fair scheme, similar to the longer wait times in the fast food restaurant [12].

Is it possible to use a common reference fair-start time for each job when comparing the fairness of different scheduling schemes? We believe not, since the relative performance of the base scheme would confound the evaluation. A strict FCFS policy (i.e. with no backfilling) is a completely fair scheduling scheme, in terms of social justice. So this might seem to be a logical choice for a reference scheme. However, FCFS no-backfill generates an inherently poor schedule with very high average response time due to low utilization. Therefore, backfilling scheduling schemes would likely result in most jobs having better response times than that with FCFS no-backfill. This would be true even with schedules that are blatantly unfair. For example, consider an FCFS-aggressive schedule on a trace with many pairs of identically shaped jobs that arrive in close proximity. With most implementations of aggressive backfilling, these pairs of identical, close proximity jobs would start in arrival order, i.e. be treated fairly. Now consider a modified schedule where we swap the positions of the job pairs

```
  # Paramaters:
  # j - the job we are computing the FST for
  # isRelaxed - should the relaxed or strict metric be computed
  #
  # Return:
  # The FST for job j

public long calcFST(Job j, boolean isRelaxed){
  #Get the state of the scheduler when job j arrived
  Schedule schedule = getScheduleSnapshotAtArrival(j)

  # If we are calucating the strict FST, insert job j into
  # the scheduler so that it is allowed to backfill.  If we are
  # calulating the relaxed FST, we need to allow all jobs in
  # the schedule to backfill and start before job j is added
  if !isRelaxed
    schedule.addJob(j)

  # Run the simulation to completetion
  schedule.simulateUntilAllJobsHaveStarted()

  if(not isRelaxed)
    # Simply return when the job started
    return j.getStartTime()
  else #isRelaxed
      # All jobs which arrived before j have already been forced
      # to start, add j to the schedule to see when it will start
    schedule.add(j);
    # Run the simulation to completetion
    schedule.simulateUntilAllJobsHaveStarted()
    return j.getStartTime()
}
```

Fig. 4. This method calculates the fair start time (relaxed or strict) for a given job

in the aggressive backfill schedule - i.e. have the later job run in the earlier job's slot and vice versa. This modified schedule is unquestionably unfair with respect to these job pairs. But in the corresponding FCFS-no-backfill schedule, it is very likely that all jobs would have later start times than the FCFS-aggressive schedule. So if we just compare a job's start-time with its reference start-time under the FCFS-no-backfill schedule, all jobs would be considered to be fairly treated for this contrived schedule that is obviously unfair.

Thus it is problematic to use a single reference schedule in comparing fairness of different scheduling schemes. The metric for fairness should be independent of the performance (from the classical user/system metrics) of the scheduling policies. The goal of characterizing fairness is not to determine whether one policy is more effective than another with respect to user/system performance metrics.

Fairness and performance are both important in determining the attractiveness of a scheduling strategy, but they are independent factors.

Cumulative Metric. Above we have defined ways to measure the FST for each job. We can then determine the unfairness for each job by subtracting the FST from the actual start time. If the $ActualStartTime - FST$ is less than zero, the job has been given preferential treatment; if it equals zero the job has been treated fairly; and if it is greater than zero it has been treated unfairly.

We define the overall average unfairness as the sum of the unfairness divided by the number of jobs.

$$OverallUnfairness = \frac{\sum_{i \in jobs} max(ActualStartTime_i - FST_i, 0)}{\sum_{j \in jobs} 1} \quad . \qquad (1)$$

Therefore, jobs which are given preferential treatment can not bring down the metric, as we only sum over unfairly treated jobs. Also, we divide the sum by the total number of jobs. So if only one job is treated unfairly by X, the overall unfairness is X/N, while a scheme where all jobs are treated unfairly by X will have an overall unfairness of X.

3.2 Resource Equality

The FST metrics explained above measure effects similar to "skips" and "slips" in queues. A very different model for fairness is that over any period of time, ideally an equal fractional time slice should be provided for each job in the system. The idea behind the Resource Equality (RE) based metric is to evenly divide the resources amongst all jobs which are active in the system. Active jobs are defined as jobs which have arrived but have not exited the system (completed), i.e. running and queued jobs. In a single resource context (i.e. all sequential jobs), each job simply deserves $1/N$ (where N is the number of active jobs in the system) of the resources for each time quantum (the length of which is defined by job arrival and departures). However, in a multi-resource context (i.e. with parallel jobs), some jobs are able to use more resources in a time quantum than other jobs. For instance, if there are a 5-processor and a 15-processor job active in a 20 processor system, it does not make sense to insist that each job deserves 1/2 of the system for the time quantum. The 5 processor job could not use more than 1/4th of the system, and is not being treated unfairly if it only receives 1/4th of the system.

Conceptually, the resource equality based metric can be viewed as an ideal virtual time slicing scheme in a round robin fashion (processor sharing). Here each job receives an infinitesimally narrow time slice, and a width that equals the number of actual requested nodes. It is an idealized time slicing scheme, because we ignore packing issues by allowing jobs to fill fragmented slices and complete in the next time slice. We add the condition that no job can deserve more cycles than it is able to consume (a 8 process job, in a t length time quantum, deserves a maximum of $8t$ cycles). Further, we follow the model that only the total used

resources should be divided amongst active jobs, rather than the total system resources, as proposed in RAQFM in a multi-server context. Therefore, it is feasible for a scheme to be fair even if resources are wasted and the resource equality unfairness metric is independent of utilization and loss of capacity.

The deserved time for job i is more formally defined as below:

$$d_i = \int_{a_i}^{c_i} min(\frac{nodes_i}{\sum_{j \in ActiveJobs} nodes_j} \times used\_Nodes, nodes_i)dt \ . \quad (2)$$

The unfairness for each individual job is defined as the deserved resources (d_i) minus the consumed resources ($s_i = runtime_i \times processors_i$):

$$D_i = d_i - s_i \ . \quad (3)$$

We then define the overall unfairness in a manner similar to the FST based metrics:

$$OverallUnfairness = \frac{\sum_{i \in jobs}(max(D_i, 0))}{\sum_{j \in jobs} 1} \ . \quad (4)$$

This metric is partially independent of the actual scheduling policy used. The metric is based solely on when a job arrives and departs and how many other jobs are active in the system. Therefore, this metric can easily be computed using post processing, by using the scheduler output trace in the Standard Workload Format [25]. This is different from the FST based algorithm that explicitly uses the current scheduling policy and needs the state of the schedule in order to generate the FST when each jobs arrives.

4 Simulation Environment

We used the proposed fairness metrics to analyze several standard space sharing, parallel job scheduling schemes. We analyzed three depths of reservations: no guarantees (no reservations), EASY/aggressive backfilling (1 reservation), and conservative backfilling (all jobs have a reservation). For each of the reservation depths, we simulated three queuing orders, First Come First Served (FCFS), Shortest Job First (SJF), and Largest eXpansion Factor (LXF)[26]. For each queuing order, the queue was dynamically sorted by the specified criterion.

We performed simulations on 5000 job subsets from each of the following traces: 128 node SDSC SP2, the 512 node CTC SP2 (with 430 batch nodes), the 1152 SDSC Blue, and the 178 node OSC machines available from the Feitelson workload archives [25]. We modified the traces to simulate offered loads of 70%, 80%, 90%, 95%, and 98%. We show representative data using the 80% (medium load) and 95% (high load) simulations using the CTC SP2 and SDSC Blue input traces.

To modify the workload of an input trace, we multiplied both the user supplied runtime estimate and the actual runtime by a suitable factor to achieve a desired

offered load. For instance, let us assume that the original trace had a utilization of 65%. To achieve an offered utilization of 90%, the actual runtime and estimated wallclock limit was multiplied by 0.9/0.65. Thus we used runtime expansion to increase the mean load. Using runtime expansion in lieu of shrinking the inter-arrival keeps the duration of a trace consistent. When shrinking the inter-arrival times, the duration of the input trace is proportional to the change in the offered load. When using runtime expansion, the duration of a long simulation remains roughly same. We note that the schedules generated with either approach is equivalent. Let i be the basic time unit when shrinking inter-arrival time, r be the basic time unit when using runtime expansion, and x be the factor by which the run times were expanded and the inter-arrival times were shrunk. Simply letting $x * i == r$ gives identical simulation states.

5 Evaluation

Figures 5 and 6 show the overall average unfairness using the strict FST based metric. These figures show that no-guarantee backfilling is worse than aggressive or conservative backfilling, especially for the SDSC Blue trace. There are cases where SJF no-guarantee is better than SJF conservative and SJF aggressive backfilling; however SJF is generally more unfair than FCFS and LXF. There are no consistent trends showing LXF or FCFS to be more unfair than the other. Figures 7 and 8 show the strict FST based unfairness metric categorized by job width (number of nodes). Only figures for FCFS data are shown, as the relative results for comparing no guarantee, aggressive and conservative backfilling remain similar for LXF or SJF queuing priority. Table 1 shows the number of jobs in each category. These figures show that wide jobs (i.e. jobs which need many nodes) are treated extremely unfairly using a no-guarantee backfilling scheme. Therefore, even though the overall unfairness for no-guarantee backfilling may look acceptable using the strict FST metric for the CTC trace, wide jobs are treated very unfairly. Therefore, we can conclude that the strict FST metric shows that SJF is an unfair queuing order and no-guarantee backfilling is unfair. However, no consistent conclusions can be drawn regarding the relative unfairness of an LXF and an FCFS queuing priority, or between aggressive and conservative backfilling.

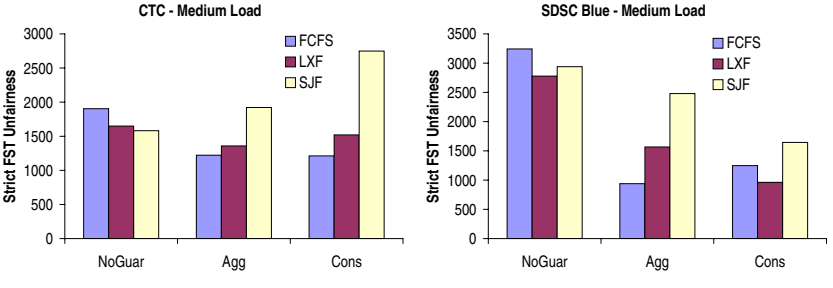


Fig. 5. Average strict fair start miss time under medium load

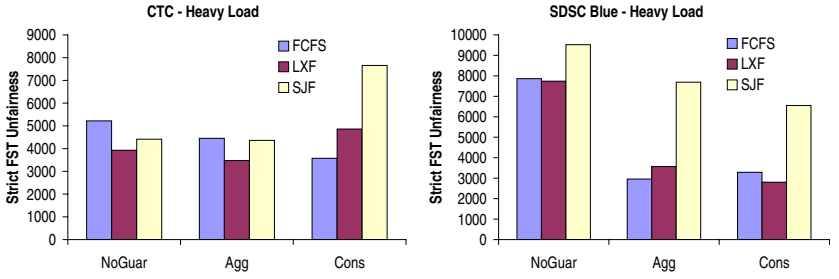


Fig. 6. Average strict fair start miss time under high load

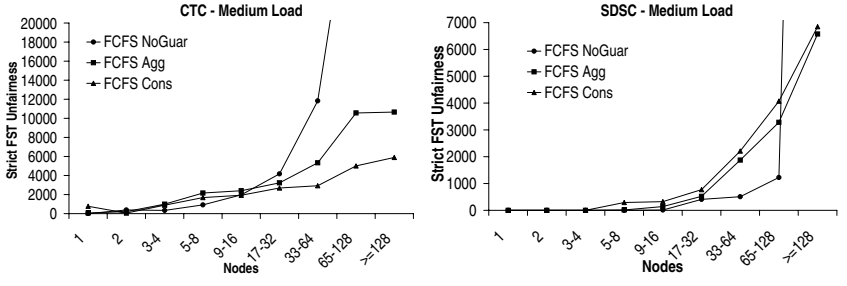


Fig. 7. Average strict fair start miss time for different width jobs under medium load

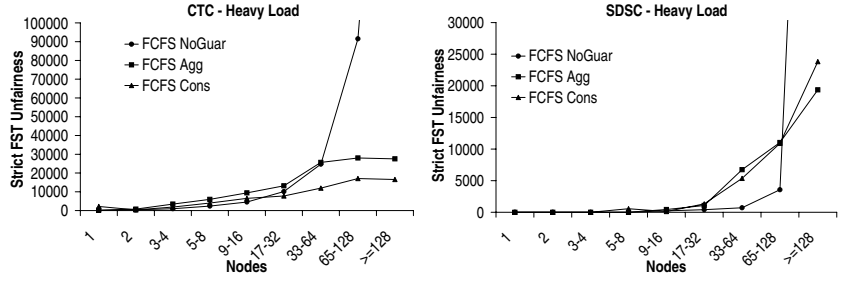


Fig. 8. Average strict fair start miss time for different width jobs under high load

Table 1. The number of jobs for width (processor) categories presented

	1	2	3-4	5-8	9-16	17-32	33-64	65-128	> 128
CTC SP2	2209	435	722	487	609	309	157	45	30
SDSC Blue	3	0	0	2489	653	569	616	396	277

Figures 9 and 10 show the overall average unfairness using the relaxed FST based metric. The relaxed metric shows that no-guarantee backfilling or a SJF queuing priority are unfair, similar to the conclusions with the strict metric. However, for all simulations, the relaxed metric more clearly shows no-guarantee backfilling to be unfair than does the strict metric. Also, the relaxed metric

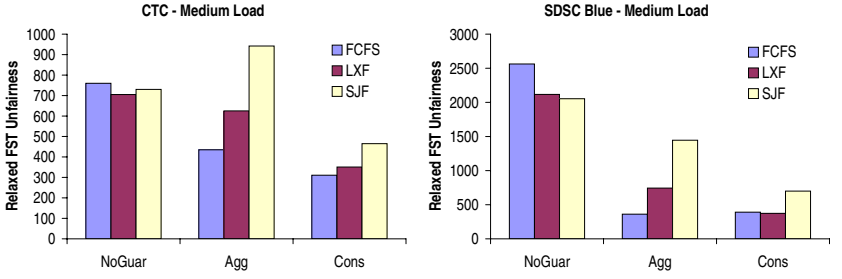


Fig. 9. Average relaxed fair start miss time under medium load

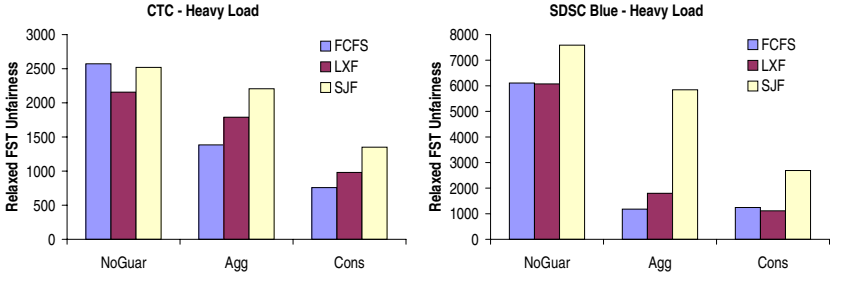


Fig. 10. Average relaxed fair start miss time under high load

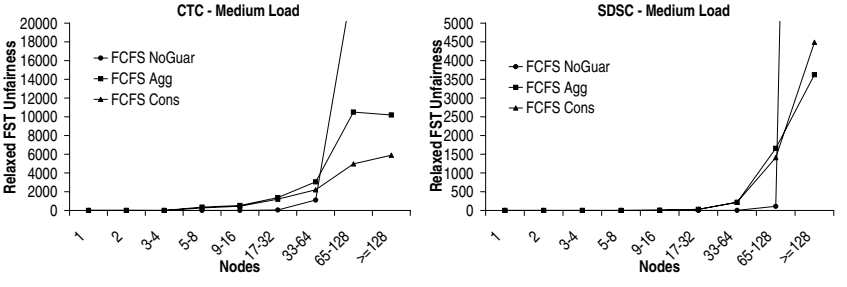


Fig. 11. Average relaxed fair start miss time for different width jobs under medium load

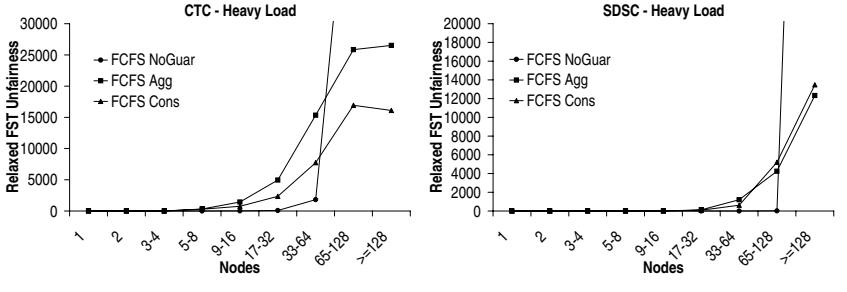


Fig. 12. Average relaxed fair start miss time for different width jobs under high load

shows a clearer separation between an LXF and FCFS queuing priority, and between an aggressive and conservative backfilling policy. Using this metric, FCFS is either less unfair or extremely close to LXF and conservative backfilling is less unfair than aggressive backfilling; however they are both close in some situations. Figures 11 and 12 show the relaxed FST based unfairness metric for width categories. Again only FCFS data is shown, as the trends for different depths of reservation are consistent for LXF and SJF. These figures further show the unfairness of no-guarantee backfilling, as the wide jobs are treated extremely unfairly. The conclusions that can be drawn from this metric are that an SJF queuing policy or no-guarantee backfilling are unfair scheduling policies. Further, LXF appears to be slightly more unfair than FCFS and aggressive backfilling appears to be slightly more unfair than conservative backfilling.

The relaxed and strict metric show similar results; however the relaxed metric gives a clearer picture with respect to the differences between LXF and FCFS, and between conservative and aggressive backfilling.

Figures 13 and 14 show data for the resource equality based metric, for the CTC and SDSC traces, for medium and high loads. This metric shows that a no-guarantee backfilling scheduling policy is very unfair with respect to the RE based metric. However, the RE based metric does not show any significant distinction between aggressive and conservative backfilling. Further, this metric does not show significant differences between different queuing priority orders. Thus, similar to the FST based metrics, there is little difference between ag-

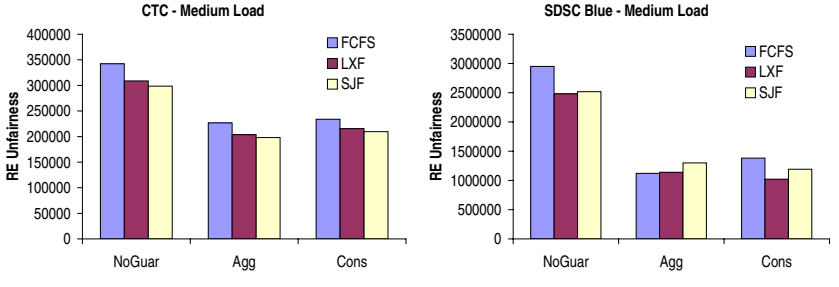


Fig. 13. Resource equality based unfairness under medium load

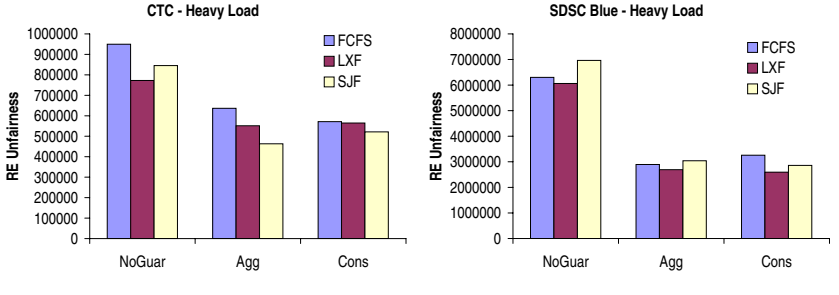


Fig. 14. Resource equality based unfairness under high load

gressive and conservative backfilling and no-guarantee backfilling appears to be unfair. However, SJF does not appear to be an unfair scheme using the resources equality based metric, which is very different from the conclusions drawn from the FST based fairness metrics.

Figures 15 and 16 show the overall resource equality based metric for the CTC and SDSC traces at the medium and high loads for varying width jobs - again only FCFS figures are shown as LXF and SJF shows similar trends. Again, it can be seen that no-guarantee backfilling treats wide job very unfairly.

All three metrics show that no-guarantee backfilling is an unfair scheduling policy, especially for wide jobs. This is not a surprising result, as no-guarantee backfilling does not provide a mechanism for wide jobs to start in a timely manner in the presence of narrower jobs. There is no consistent difference between aggressive and conservative backfilling when using two of the metrics; however the relaxed FST metric shows that conservative backfilling may be slightly less unfair. An SJF queuing order is much more unfair in the FST based metrics, while it is comparable to FCFS and LXF with the RE based metric. There is also little difference between LXF and FCFS: two of the three fairness metrics do not show any consistent trends, while the third metric (relaxed FST) shows that FCFS may be slightly less unfair than LXF. The apparent closeness of LXF and FCFS is a surprising result, as FCFS attempts to run jobs in arrival order; it appears that the inaccurate user estimates leads to a significant amount of unfairness even with an FCFS priority queue. This is due to the scheduler allowing a job to backfill. The apparently

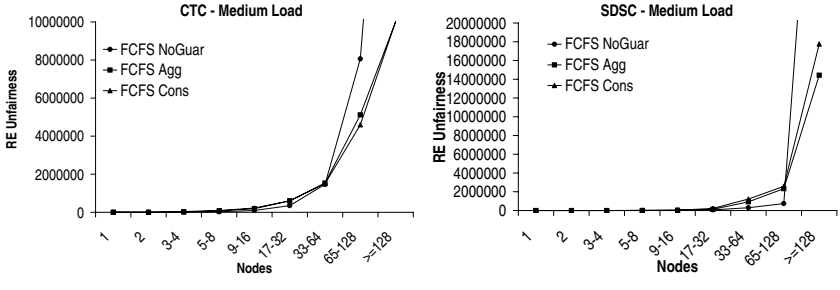


Fig. 15. Resource equality based unfairness for different width jobs under medium load

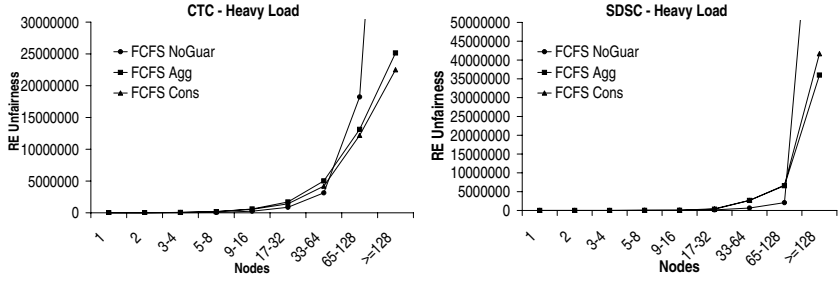


Fig. 16. Resource equality based unfairness for different width jobs under high load

benign backfill can delay earlier jobs when other running jobs have not accurately estimated their runtimes [24]. The unfairness in a SJF schedule is possibly caused by temporarily starving some of the longer jobs while giving preferential treatment to shorter jobs. In conclusion, SJF and no-guarantee backfilling generally exhibit greater unfairness, while there is little difference between FCFS vs. LXF and between conservative vs. aggressive backfilling.

5.1 Rank Correlation

Here we assess how sensitive these metrics are to the choice of jobs in a trace. Each trace was divided into 10 "random" subsets, based solely on the arrival order of the jobs. Table 2 shows the Spearman [27] correlation between each subset and the overall results, using all 45 simulations for each trace (combinations of load, backfilling policy, and queue priority policy). The rank correlation was performed between each of the subsets and the overall average. The data points being ranked are the 45 simulations on a trace by trace basis. The 45 simulations were obtained from 5 offered loads, the three queuing priorities, and the three backfilling policies. The simulations were ranked (individually) between 1 and 45 for each of the 10 subsets and for the overall average. The Spearman rank order correlation was than computed between each $subset_i$ and the overall average, to show the correlation between job subsets and the overall average.

The high correlation indicates that the order of the metrics would be similar even if we chose to only rank schemes by the unfairness of a subset of the jobs. Therefore, unfairness is similar for different subsets and does not vary significantly.

Table 2. Spearman correlation data between subsets of data and the overall unfairness

	CTC			SDSC		
	FST Strict	**FST Relaxed**	**Resource Equality**	**FST Strict**	**FST Relaxed**	**Resource Equality**
Subset 0	0.96	0.96	0.96	0.95	0.91	0.93
Subset 1	0.97	0.95	0.97	0.98	0.96	0.98
Subset 2	0.93	0.87	0.89	0.97	0.96	0.96
Subset 3	0.93	0.76	0.88	0.97	0.98	0.98
Subset 4	0.95	0.85	0.88	0.97	0.97	0.97
Subset 5	0.94	0.83	0.91	0.96	0.95	0.98
Subset 6	0.97	0.96	0.98	0.97	0.96	0.95
Subset 7	0.96	0.97	0.97	0.99	0.98	0.97
Subset 8	0.98	0.91	0.95	0.94	0.88	0.93
Subset 9	0.94	0.95	0.96	0.85	0.74	0.76

6 Future Work

We hope to extend the FST based unfairness work to be based on other fairness priorities (other than FCFS). The current metric assumes that the fair order is

FCFS. The queue priority can be changed, but the FST based schemes run "what if" simulations assuming that no later arriving jobs exist. We would like to be able to use other fairness priorities, so that system administrators can select a fair order (possibly based on time in the queue or administrative priorities) and measure the unfairness based on these altered fairness priority assumptions. One issue that is difficult when addressing this issue is handling dynamic priorities and arriving jobs.

We would also like to expand these metrics and ideas to other related areas beyond "simple" non-preemptive space shared schedulers. For instance, we plan on measuring unfairness and identifying other fairness concerns for parameter sweep applications and moldable job schedulers.

It would also be interesting to perform a comparison of slowdown based metrics to the two metrics defined in this paper. The slowdown based metrics do not entirely fit into either of the two presented categories (social justice or resource equality).

This work has focused on the fairness of individual jobs. The metrics are based on related fields where each job is treated individually and there is no distinction between users and jobs. However, in parallel job scheduling, there often are heavy users who may submit many jobs. While job level fairness is reasonable, and provides a firm foundation in other fields, a user based fairness metric would also seem reasonable. We believe that our work on job fairness can serve as a good foundation for continued work on user based fairness metrics.

7 Conclusion

In this paper, we introduced two fairness metrics for job scheduling, each motivated by past fairness research in sociology, computer networking, and/or queuing systems. The first metric (Fair Start Time based) is based on a deli-line notion of fairness or social justice, where jobs should be serviced in order. We attempt to measure the effects of out-of-order execution caused by backfilling and inaccurate user estimates. This is similar to measuring "skips" and "slips". The second metric is based on equally sharing the resources amongst active jobs. This metric compares the resources a job deserves during its lifetime to the resources it actually consumes. It exhibits similarities to the RAQFM metrics used in queuing systems and fairness amongst network flows.

These two types of metrics are based on different principles and measure different features. Both metrics are orthogonal to traditional user metrics (turnaround time or slowdown) or system metrics (utilization). The extent of similarities in the observations using the two metrics is somewhat surprising. Both metrics indicate that LXF and FCFS have comparable unfairness, while results for an SJF queue depend on the unfairness metric being used. Further, there is not a consistent difference in unfairness between aggressive and conservative backfilling, while no-guarantee backfilling is consistently more unfair (especially in regards to the unfairness towards wide jobs).

Thus we have introduced unfairness metrics for parallel job scheduling, and were able to draw some consistent results across multiple traces and different

loads. The data shows that there may not be significant unfairness concerns for some previously reported schemes (LXF) which are often not implemented due to unfounded unfairness concerns. We hope these unfairness metrics are able to improve the acceptance of other scheduling schemes, where unfairness may be a concern. Further, these metrics can be used to help alleviate user concerns about unfairness of existing scheduling schemes, hopefully increasing user satisfaction.

Acknowledgments. We would like to thank Dror Feitelson for the workload archive, Travis Earheart and Nancy Wilkins-Diehr for the SDSC Blue workload logs, Victor Hazlewood from HP Labs for the SDSC SP2 logs, and Dan Dwyer and Steve Hotovy for the CTC workload logs. We thank the referees for their suggestions for improvement.

References

1. Talby, D., Feitelson, D.: Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In: Proceedings of the 13th International Parallel Processing Symposium. (1999)
2. Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. In: IEEE Transactions on Parallel and Distributed Systems. Volume 12. (2001) 529–543
3. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: Scheduling of parallel jobs in a heterogeneous multi-site environement. In Feitelson, D., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing, 9th International Workshop. Volume 2862., Seattle, WA, USA, Springer-Verlag Heidelberg (2003) 87 – 104
4. Islam, M., Balaji, P., Sadayappan, P., Panda, D.K.: QoPS: A QoS based scheme for parallel job scheduling. In Feitelson, D., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing, 9th International Workshop. Volume 2862., Seattle, Washington (2003)
5. Feitelson, D.: Workshops on job scheduling strategies for parallel processing. (www.cs.huji.ac.il/ feit/parsched/)
6. Shmueli, E., Feitelson, D.: Backfilling with lookahead to optimize the performance of parallel job scheduling. In Feitelson, D., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing. Springer-Verlag (2003) 228–251 Lect. Notes Comput. Sci. vol. 2862.
7. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for job scheduling. In: 2002 Intl. Workshops on Parallel Processing. (2002) held in conjunction with the 2002 Intl. Conf. on Parallel Processing, ICPP 2002.
8. Raz, D., Levy, H., Avi-Itzhak, B.: A resource-allocation queueing fairness measure. In: Proceedings of Sigmetrics 2004/Performance 2004 Joint Conference on Measurement and Modeling of Computer Systems, New York, NY (2004) 130–141 Also appears as *Performance Evaluation Review Special Issue* 32(1):130-141.
9. Avi-Itzhak, B., Levy, H., Raz, D.: Quantifying fairness in queueing systems: Principles and applications. Technical Report RRR-26-2004, RUTCOR, Rutgers University (2004)

10. Raz, D., , Levy, H., Avi-Itzhak, B.: RAQFM: a resource allocation queueing fairness measure. Technical Report RRR-32-2004, RUTCOR, Rutgers University (2004)
11. Mann, L.: Queue culture: The waiting line as a social system. The American Journal of Sociology **75** (1969) 340–354
12. Larson, R.: Perspectives on queues: Social justice and the psychology of queueing. Operations Research **35** (1987) 895–905
13. Gordon, E.S.: Slips and Skips in Queues. PhD thesis, Massachusetts Institute of Technology (1989)
14. Whitt, W.: The amount of overtaking in a network of queues. Networks **14** (1984) 411–426
15. Rafaeli, A., Kedmi, E., Vashdi, D., Barron, G.: Queues and fairness: A multiple study experimental investigation. (http://queues-fairness.rafaeli.net/)
16. Greenberg, A.G., Madras, N.: How fair is fair queueing? Association for Computing Machinery **39** (1992) 568–598
17. Demers, A., Keshav, S., Shenker, S.: Analysis and simulation of a fair queueing algorithm. Internetworking Research and Experience **1** (1990) 3–26
18. Nandagopal, T., Lu, S., Bharghavan, V.: A unified architecture for the design and evaluation of wireless fair queueing algorithms. Wireless Networks **8** (2002) 231–247
19. Wierman, A., Harchol-Balter, M.: Classifying scheduling policies with respect to unfairness in an M/GI/1. In: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. (2003) 238 – 249
20. Bansal, N., Harcol-Balter, M.: Analysis of SRPT scheduling: Investigating unfairness. In: SIGMETRICS. (2001)
21. Harchol-Balter, M., Sigman, K., Wierman, A.: Asymptotic convergence of scheduling policies with respect to slowdown. In: IFIP WG 7.3 International Symposium on Computer Modeling, Measurement and Evaluation. (2002)
22. Schwiegelshohn, U., Yahyapour, R.: Fairness in parallel job scheduling. Journal of Scheduling **5** (2000) 297–320
23. Sabin, G., Sahasrabudhe, V., Sadayappan, P.: On fairness in distributed job scheduling across multiple sites. In: Cluster. (2004)
24. Sabin, G., Kochhar, G., Sadayappan, P.: Job fairness in non-preemptive job scheduling. In: International Conference on Parallel Processesing. (2004)
25. Feitelson, D.G.: Logs of real parallel workloads from production systems. (URL: http://www.cs.huji.ac.il/labs/parallel/workload/)
26. Hansen, B.: An analysis of response ratio. In: IFIP Congress. (1971)
27. Weisstein, E.W.: Spearman rank correlation coefficient. http://mathworld.wolfram.com/SpearmanRankCorrelat ionCoefficient.html) From MathWorld–A Wolfram Web Resource.

Pitfalls in Parallel Job Scheduling Evaluation

Eitan Frachtenberg[1] and Dror G. Feitelson[2]

[1] Modeling, Algorithms, and Informatics Group,
Los Alamos National Laboratory
eitanf@lanl.gov
[2] School of Computer Science and Engineering,
The Hebrew University, Jerusalem, Israel
feit@cs.huji.ac.il

Abstract. There are many choices to make when evaluating the performance of a complex system. In the context of parallel job scheduling, one must decide what workload to use and what measurements to take. These decisions sometimes have subtle implications that are easy to overlook. In this paper we document numerous pitfalls one may fall into, with the hope of providing at least some help in avoiding them. Along the way, we also identify topics that could benefit from additional research.

Keywords: parallel job scheduling, performance evaluation, experimental methodology, dynamic workload, static workload, simulation.

1 Introduction

Parallel job scheduling is a rich and active field of research that has seen much progress in the last decade [31]. Better scheduling algorithms and comparative studies continually appear in the literature as increasingly larger scale parallel systems are developed and used [70]. At the same time, parallel job scheduling continues to be a very challenging field of study, with many effects that are still poorly understood. The objective, scientific difficulties in evaluating parallel job schedulers are exacerbated by a lack of standard methodologies, benchmarks, and metrics for the evaluation [9,27].

Throughout the years, the authors have encountered (and sometimes committed) a variety of methodological leaps of faith and mistakes that are often recurring in many studies. In this paper we attempt to sum up the experience gleaned from ten years of the workshop on job scheduling strategies for parallel processing (JSSPP).[1] Our main goal is to expose these topics in a single document with the hope of helping future studies avoid some of these pitfalls.

To limit this paper to a reasonable size, we chose to exclude from the scope of this paper topics in static and DAG scheduling, which is a separate field that is not generally covered by the JSSPP workshop. Additionally, grid scheduling is not specifically targeted, although many of the topics that are covered bear relevance to grid scheduling.

[1] www.cs.huji.ac.il/~feit/parsched/

D. Feitelson et al. (Eds.): JSSPP 2005, LNCS 3834, pp. 257–282, 2005.

Naturally, not all pitfalls are relevant to all classes of evaluation or scheduling strategies (e.g., time slicing vs. space slicing). Others, such as those related to the choice of workload, affect almost any quantitative evaluation. As a notational convention, we marked each pitfall with one to three lightning bolts, representing our perception of the severity of each item.

Several of the pitfalls we list may not always represent a methodological mistake. Some choices may be correct in the right context, while others may represent necessary compromises. Some of these suggestions may even seem contradictory, depending on their context. Many topics and methodological choices remain open to debate and beg for further research. But it is necessary that we remain cognizant of the significance of different choices. We therefore recommend not to consider our suggestions as instructions but rather as guidelines and advisories, which are context-dependent.

We have made a deliberate choice not to point out what we perceive as mistakes in others' work. Instead, we described each pitfall in general terms and without pointing to the source of examples. When detailing suggestions, however, we attempted to include positive examples from past works.

The list of pitfalls we present in this paper is by no means exhaustive. The scope of this paper is limited to issues specific to our field, and does not cover methodological topics in general. For example, we exclude general problems of statistics, simulation techniques, or presentation. Similarly, issues that are are not very significant to parallel job scheduling evaluation and minor methodological problems were left out. Still, we wish to enumerate at this point the following general principles, that should be considered for any performance evaluation study:

- Provide enough details of your work to allow others to reproduce it.
- Explore the parameter space to establish generality of results and sensitivity to parameters.
- Measure things instead of assuming they are so, even if you are sure.

The designer of a parallel job scheduler is faced with myriad choices when reaching the evaluation stage. First, the researcher must choose a workload (or more than one) on which the scheduler is evaluated. This workload may reflect choices typical for a researcher's site or could try to capture the salient properties of many sites. Either way, the choice of a good workload structure entails many tricky details. Inexorably related to the workload structure are the workload applications that must next be chosen. The evaluation applications, whether modeled, simulated, or actually run, also have a large impact's on the evaluation's results, and must be chosen carefully. Next, experiments must be designed, and in particular, a researcher must choose the factors (input parameters) to be evaluated and the metrics against which the scheduler is measured. Here too, different choices can have radically different results. For example, scheduler A might have lower average response time than scheduler B, but also be more unfair. A different choice of input parameters could reverse this picture. A good choice of input parameters and metrics can typically not be done in isolation of the measurements, but rather, after careful examination of the effect on each of

the evaluated system. Lastly, the measurement process itself entails many traps for the unwary researcher, which could be averted with a systematic evaluation.

We group pitfalls into groups that loosely reflect the order of choices made for a typical job scheduling study. We start in Section 2 with pitfalls relating to workload structure. Section 3 delves into what comprises the workload, namely, the applications. The input parameters for the evaluation are considered in Section 4. Next, Sections 5 and 6 discuss methodological issues relating to measurement and metric choices. Finally, we conclude in Section 7.

2 Workload Structure

Workload issues span all methods of parallel job scheduling evaluation. The choices and assumptions represented in the tested workloads may even determine the outcome of the evaluation [21,30]. With experimental evaluations on real systems, our choice of workload is often limited by practical issues, such as time or machine availability constraints. Analysis may be limited by considerations of mathematical tractability. Simulation studies are typically less limited in their choice of workload [75].

As a general rule of thumb, we recommend using multiple different workloads, each of which is long enough to produce statistically meaningful results. Several workload traces and models can be obtained from the Parallel Workload Archive [54]. Whether using an actual trace or a workload model, care must be given to specific workload characteristics. For example, a workload dominated by power-of-two sized jobs will behave differently from one containing continuously sized jobs [47]. It is important to understand these workload characteristics and their effect on the evaluation's results, and if possible, choose different workload models and compare their effect on the evaluation [1,21].

This section lists the challenges that we believe should be considered when defining the workload structure. Many of these pitfalls can be summarized simply as "employing overly-simplistic workload models". Given that better data is often available, there is no justification to do so. Therefore, whenever possible we would suggest to use realistic workloads, and moreover, to use several different ones.

Pitfall 1 ϟϟϟ

Using invalid statistical models

Problem. Workload models are usually statistical models. Workload items are viewed as being sampled from a population. The population, in turn, is described using distributions of the various workload attributes.

A model can be invalid, that is, not representative of real workloads, in many different ways. The most obvious is using the wrong distributions. For example, many researchers use the exponential distribution for interarrival times, assuming that arrivals are a Poisson process. This ignores data about self similarity (pitfall 4), and also ignores irregularities and feedback effects, as well as job resubmittals [22,38]. Some studies even use uniform

distributions, e.g., for the parallelism (size) of jobs, which is entirely unrepresentative; a more representative distribution favors small jobs (e.g. the log-uniform), and moreover is modal, emphasizing powers of two [14,16,47].

To be fair, finding the "right" distribution is not always easy, and there are various methodological options that do not necessarily produce the same results. For example, even distributions that match several moments of the workload data may not match the shape of the distributions, especially the tail. The story does not end with distributions either: it is also important to model correlations between different attributes, such as job size, interarrival time, and length [14,20,47].

Suggestions. If a workload model is used, one must ensure that it is a good one. The alternative is to use a real trace. This has the advantage of including effects not known to modelers, but also disadvantages like abnormal data (pitfall 5).

Research. Workload modeling is still in its infancy. There is much more to learn and do, both in terms of modeling methodology and in terms of finding what is really important for reliable performance evaluations. Some specific examples are mentioned in the following pitfalls.

Pitfall 2 ϟϟϟ

Using only static workloads

Problem. Static workloads are sets of jobs that are made available together at the beginning of the evaluation, and then executed with no additional jobs arriving later — akin to off-line models often assumed in theoretical analyses. This is significantly different from real workloads, where additional jobs continue to arrive all the time.

Static workloads are used for two reasons. One is that they are easier to create (there is no need to consider arrivals), and are much smaller (fewer jobs take less time to run). The other is that they are easier to analyze, and one can in fact achieve a full understanding of the interaction between the workload and the system (e.g. [33]). While this may be useful, it cannot replace a realistic analysis using a dynamic workload. This is important because when a static workload is used, the problematic jobs tend to lag behind the rest, and in the end they are left alone and enjoy a dedicated system.

Suggestions. It is imperative to also use dynamic workloads, whether from a trace or a workload model.

Research. An interesting question is whether there are any general principles regarding the relationship of static vs. dynamic workloads. For example, are static workloads always better or worse?

Pitfall 3 ϟϟ

Using too few different workloads

Problem. Workloads from different sites or different machines can be quite different from each other [68]. Consequently, results for one workload are not

necessarily valid for other workloads. For example, one study of backfilling based on three different workloads (one model and two traces) showed EASY and conservative to be similar [32], but a subsequent study found that this happens to be a nonrepresentative sample, as other workloads bring out differences between them [51].

Suggestions. Whenever possible, use *many* different workloads, and look for invariants across all of them; if results differ, this is a chance to learn something about either the system, the workload, or both [21].

Research. It is important to try to understand the interaction of workloads and performance results. *Why* are the results for different workloads different? This can be exploited in adaptive systems that learn about their workload.

Pitfall 4 

Ignoring burstiness and self-similarity

Problem. Job arrivals often exhibit a very bursty nature, and realistic workloads tend to have high variance of interarrival times [22,67]. This has a strong effect on the scheduler, as it sometimes has to handle high transient loads. Poisson models make the scheduler's life easier, as fluctuations tend to cancel out over relatively short time spans, but are not realistic.

Suggestions. Burstiness is present in workload traces, but typically not in models. Thus real traces have an advantage in this regard.

Research. One obvious research issue is how to incorporate burstiness and self-similarity in workload models. This has been done in network traffic models (e.g. [76]), but not yet in parallel job models.

Another issue is the effect of such burstiness on the evaluation. If a model creates bursts of activity randomly, some runs will lead to larger bursts than others. This in turn can lead to inconsistency in the results. The question then is how to characterize the performance concisely.

Pitfall 5 

Ignoring workload flurries and other polluted data

Scope. When using a real workload trace.

Problem. We want workload traces to be "realistic". What we mean is that they should be representative of what a scheduler may be expected to encounter. Regrettably, real traces often include subsets of data that cannot be considered representative in general. Examples include:

- Heavy activity by system administrators [24].
- Heavy activity by cleanup scripts at night.
- Heavy and unusual activity by a single user that dominates the workload for a limited span of time (a workload flurry) [73].

Suggestions. Data needs to be sanitized before it is used, in the sense of removing obvious outlier data points. This is standard practice in statistical analysis. Logs in the Parallel Workloads Archive have cleaned versions, which are recommended.

Research. The question of what to clean is not trivial. At present, this is done manually based on human judgment, making it open to debate; additional research regarding considerations and implications can enrich this debate. Another interesting question is the degree to which cleaning can be automated.

Pitfall 6

using oblivious open models with no feedback

Problem. The common way to use a workload model or trace is to "submit" the jobs to the evaluated scheduler as defined in the model or trace, and see how the scheduler handles them. This implicitly assumes that job submittals are independent of each other, which in fact they are not.
Real workloads have self-throttling. When users see the system is not responsive, they reduce the generation of new load. This may help spread the load more evenly.

Suggestions. Use a combination of open and closed model. A possible example is the repeated jobs in the Feitelson model where each repetition is only submitted after the previous one terminates [16].

Research. Introducing feedback explicitly into workload models is an open question. We don't know how to do it well, and we don't know what its effects will be.

Pitfall 7

Limiting machine usage assumptions

Problem. A related issue to pitfall 3 is the embedding of workload assumptions that are too specific to the workload's site typical usage. Some sites run the same applications (or class of applications) for months, opting to use the machine as a capability engine [36,58]. Others use far more heterogeneous workloads representing a machine running in capacity mode [47,54]. The application and workload characteristics of these two modes can be quite different, and not all evaluations can address both.

Suggestions. If a specific usage pattern is assumed, such as a site-specific or heterogeneous workload, it should be stated, preferably with an explanation or demonstration of where this model is valid [39]. Whenever possible, compare traces from different sites [10,15,47].

3 Applications

The application domain of parallel job schedulers is by definition composed of parallel jobs. This premise predicates that applications used for the evaluation of schedulers include some necessary aspects of parallel programs. An obvious minimum is that they use several processors concurrently. But there are others too.

Virtually all parallel applications rely on communication, but the communication pattern and granularity can vary significantly between applications. The degree of parallelism of an application also varies a lot (depending on application type) between sequential applications—with zero parallel speedup—to highly parallel and distributed applications—with near-linear speedup. Other parameters where parallel applications show high variability include services time, malleability, and resource requirements, such as memory size and network bandwidth. In addition, typical applications for a parallel environment differ from those of a grid environment, which in turn differ from distributed and peer-to-peer applications. This high variability and wide range of possible applications can translate to very dissimilar results for evaluations that differ only in the applications they evaluate. It is therefore vital to understand the different factors that applications imply on the evaluation.

Many job scheduling studies regard parallel jobs as rectangles in processors × time space: they use a fixed number of processors for a certain interval of time. This is justifiable when the discussion is limited to the workings of the scheduler proper, and jobs are assumed not to interact with each other or with the hardware platform. In reality, this is not always the case. Running the same jobs on different architectures can lead to very different run times, changing the structure of the workload [78]. Running applications side by side may lead to contention if their partitions share communication channels, as may happen for mesh architectures [45]. Contention effects are especially bad for systems using time slicing, as they may also suffer from cache and memory interference.

On the other hand, performing evaluations using detailed applications causes two serious difficulties. First, it requires much more detailed knowledge regarding what application behaviors are typical and representative [30], and suffers the danger of being relevant to only a small subset of all applications. Second, it requires much more detailed evaluations that require more time and effort. The use of detailed application models should therefore be carefully considered, including all the tradeoffs involved.

Pitfall 8 ϟϟϟ

Using black-box applications

Scope. When contention between jobs and interactions with the hardware platform are of importance.

Problem. An evaluation that models applications as using P processors for T time is oblivious of anything that happens in the system. This assumption is reasonable for jobs running in dedicated partitions that are well-isolated from each other. However, it does not hold in most systems, where communication links and I/O devices are shared. Most contemporary workload models do not include such detailed data [9], so the interaction between different applications with different properties and the job scheduler are virtually impossible to capture from the workload model alone.

In a simulation context that does not employ a detailed architecture simulator that runs real applications (often impractical when testing large parallel

machines), shortcut assumptions are regularly made. For example, assuming that applications are all bag-of-tasks.

If synthetic applications are measured or simulated, the benchmark designer is required to make many application-related choices, such as memory requirements and degree of locality, communication granularity and pattern, etc. [18]. Another example is the role of I/O in parallel applications, that is often ignored but may actually be relevant to a job scheduler's performance [42,81], and present opportunities for improved utilization [77].

Suggestions. Offer descriptions or analysis of relevant application properties, like network usage [3,42], parallelism [79], I/O [42,53], or memory usage [17,57,61]. If approximating the applications with synthetic benchmark programs or in a simulator, application traces can be used [79]. Based on these traces, a workload space can be created where desired parameters are varied to test the sensitivity of the scheduler to those parameters [81]. In the absence of traces, stochastic models of the relevant applications can be used [17,35,53,57].

Research. Current knowledge on application characteristics and correlations between them is rudimentary. More good data and models are required.

Pitfall 9 ///

Measuring biased, unrepresentative, or overly homogeneous applications

Problem. Many evaluations use applications that are site-specific or not very demanding in their resource requirements. Even benchmarks suites such as the NAS parallel benchmarks (NPB) [6] or ESP [78] can be representative mostly of the site that produced them (NASA and NERSC respectively for these examples), or ignore important dynamic effects, such as realistic job interarrival time [18].

Using "toy" and benchmark applications can be useful for understanding specific system properties and for conducting a sensitivity analysis. However, writing and experimenting with toy applications that have no relevant properties such as specific memory, communication, or parallelization requirements (e.g., a parallel Fibonnacci computation) helps little in the evaluation of a parallel job scheduler. Actual evaluations of scientific applications with representative datasets often take prohibitively long time. This problem is exacerbated when evaluating longer workloads or conducting parameter space explorations [8]. In some cases, researchers prefer to use application kernels instead of actual applications, but care must be taken to differentiate those from the real applications [5].

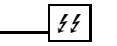
The complexity of evaluating actual applications often leads to compromises, such as shortening workloads, using less representative (but quicker to process) datasets, and selecting against longer-running applications, resulting in a more homogeneous workload that is biased toward shorter applications. On the other hand, a researcher from a real-site installation may prefer to use applications that have more impact on the site [36,39]. This

choice results in an analysis that may be more meaningful to the researcher's site than to the general case.

Suggestions. If possible, evaluate more applications. If choice of applications is limited, qualify the discussion to those limits. Identify (and demonstrate) the important and unimportant aspects of chosen applications. When possible, use longer evaluations or use more than one architecture [39].

Conversely, some situations call for using simple, even synthetic applications in experiments. This is the case for example when we wish to isolate a select number of application characteristics for evaluation, without the clutter of irrelevant and unstudied parameters, or when an agreed benchmark for the studied characteristics is unavailable [7]. To make the workload more heterogeneous, while still maintaining reasonable evaluation delays, a researcher may opt to selectively shorten the run time of only a portion of the applications (for example, by choosing smaller problem sizes).

Research. Research on appropriate benchmarks is never ending.

Pitfall 10 

Ignoring or oversimplifying communication

Problem. Communication is one of the most essential properties that differentiates parallel jobs from sequential (and to some extent, distributed) applications. Using a single simplistic communication model, such as uniform messaging, can hide significant contention and resource utilization problems. Assuming that communication takes a constant proportion of the computation time is often unrealistic, since real applications can be latency-sensitive, bandwidth-sensitive, or topology-sensitive. In addition, contention over network resources with other applications can dramatically change the amount of time an application spends communicating.

In a simulation or analysis context where communication is modeled, one should consider that communication's effect on application performance can vary significantly by the degree of parallelism and interaction with other applications in a dynamic workload. This is only important when communication is a factor, e.g., in coscheduling methods.

Suggestions. Whenever possible, use real and representative applications. If communication is a factor, evaluate the effect of different assumptions and models on the measured metrics [42,64], before possibly neglecting factors that don't matter. If a parameter space exploration is not feasible, model the communication and detail all the assumptions made [26,60], preferably basing the model on real workloads or measurements. Confidence intervals can also be used to qualify the results [69].

Research. How to select representative communication patterns? Can we make do with just a few? A negative answer can lead to an explosion of the parameter space. Some preliminary data about communication patterns in real applications is available [12], but much more is needed to be able to identify common patterns and how often each one occurs.

Pitfall 11 ⚡⚡

Using Coarse-grained applications

Scope. Time slicing systems.

Problem. Fine-grained application are the most sensitive to scheduling [25]. Omitting them from an evaluation of a time slicing scheduler will probably not produce credible results. This pitfall may not be relevant for schedulers where resources are dedicated to the job, such as space slicing methods.

Suggestions. Make sure that fine-grained applications are part of the workload. Alternately, conduct a sensitivity analysis to measure how the granularity of the constituent applications affects the scheduler.

Research. What are representative granularities? What is the distribution? Again, real data is sorely needed.

Pitfall 12 ⚡⚡

Oversimplifying the memory model

Problem. Unless measuring actual applications, an evaluation must account for applications' memory usage. For example, allowing the size of malleable jobs to go down to 1 may be great for the job scheduler in terms of packing and speedup, but real parallel applications often require more physical memory than is available on one node [58] (see pitfall 16). Time slicing evaluations must take particular care to address memory requirements, since the performance effects of thrashing due to the increased memory pressure can easily offset any performance gains from the scheduling technique. A persistent difficulty with simulating and analyzing memory usage in scientific applications is that it does not lend itself to easy modeling, and in particular, does not offer a direct relation between parallelism and memory usage [17].

Suggestions. State memory assumptions e.g., that a certain multiprogramming level (MPL) is always enough to accommodate all applications. Perform a parameter-space exploration, or use data from actual traces [17,43]. Malleable jobs may require that an assumption be made and stated on the minimal partition size for each job so that it still fits in memory.

Research. Collecting data on memory usage, and using it to find good memory-usage models.

Pitfall 13 ⚡

Assuming malleable or moldable jobs and/or linear speedup

Problem. Malleable or moldable jobs can run on an arbitrary number of processors dynamically or at launch time, respectively. While many jobs are indeed moldable, some jobs have strict size requirements and can only take a limited range of sizes, corresponding to network topology (e.g., torus, mesh, hypercube), or application constraints (e.g., NPB's applications [6]). Another unlikely assumption for most MPI jobs is that they can change their size dynamically (*malleable* or *evolving* jobs [29]).

For cases where malleability *is* assumed (e.g. for the evaluation of dynamic scheduling) linear speedup is sometimes assumed as well. This assumption is wrong. Real workloads have more complex speedup behavior. This also affects the offered load, as using a different number of processors leads to a different efficiency [30].

Suggestions. If job malleability is assumed, state it [10], and model speedup and efficiency realistically [13,52,66]. If real applications are simulated based on actual measurements, use an enumeration of application speedup for all different partition sizes.

Pitfall 14

Ignoring interactive jobs

Problem. Interactive jobs are a large subset of many parallel workloads. Interactive jobs have a significant effect on scheduling results [55] that needs to be accounted for if they are mixed with the parallel jobs. Some machines however have separate partitions for interactive jobs, so they don't mix with the parallel workload.

Suggestions. An evaluator needs to be aware of the special role of interactive jobs and make a decision if they are to be incorporated in the evaluation or not. To incorporate them, one can use workload traces or models that include them (e.g., [48]). If choosing not to incorporate them, the decision should be stated and explained. A model or a trace can then be used from a machine with a noninteractive partition [60]. It should be noted that the presence (or lack) of interactive jobs should also affect the choice of metrics used. More specifically, response time (or flow) is of lesser importance for batch jobs that it is for interactive ones. Moreover, interactive jobs account for many of the short jobs in a workload, and removing those will have a marked effect on the performance of the chosen scheduling algorithm and metric [33].

Pitfall 15

Using actual runtime as a user estimate

Scope. Evaluating backfilling schedulers that require user estimates of runtime.

Problem. User estimates are rarely accurate predictions of program run times [41,51]. If an evaluation uses actual run times from the trace as user estimates, the evaluation results may differ significantly from a comparable evaluation with the actual user estimates [21].

Suggestions. Evaluators should strive to use actual user estimates, when available in the trace (these are currently available in 11 of the 16 traces in the workload archive). Lacking user estimates, the evaluator can opt to use a model for generating user estimates [72]. Another alternative is to test a range of estimates (e.g., by a parametrized model) and either describe their effect on the result, or demonstrate that no significant effect exists. Simply multiplying the run times with different factors to obtain overestimation [33,51] can lead to artificial, unrepresentative performance improvements [72].

4 Evaluation Parameters

Even after meticulously choosing workloads and applications, an evaluation is only as representative as the input parameters that are used in it. Different scheduling algorithms can be very sensitive to parameters such as input load, multiprogramming level, and machine size. It is therefore important to understand the effect of these parameters and the reasonable ranges they can assume in representative workloads.

Pitfall 16 ⚡⚡⚡

Unrealistic multiprogramming levels

Problem. Increasing the multiprogramming level in time slicing schedulers also increases the memory pressure. High MPLs have value when studying the effect of the MPL itself on scheduling algorithms, and as a limiting optimal case. But for actual evaluations, high MPLs are unrealistic for many parallel workloads, especially in capability mode, where jobs could potentially use as much memory as they can possibly allocate. For many time slicing studies it is not even required to assume very high MPLs: several results show that increasing the MPL above a certain (relatively low) value offers little additional benefit, and can in fact degrade performance [33,50,65].

Since MPL can be interpreted as the allowed degree of resource oversubscribing, with space slicing scheduling a higher MPL translates to more jobs waiting in the scheduler's queues. In this case, the MPL does not have an effect on memory pressure, but could potentially increase the computation time of the space allocation algorithm.

Suggestions. Ideally, a comprehensive model of application memory requirements can be incorporated into the scheduler, but this remains an open research topic. For time slicing algorithms, bound the MPL to a relatively low value [80], say 2–4. Alternately, use a technique such as admission control to dynamically bound the MPL based on memory resources [7] or load [79]. Another option is to incorporate a memory-conscious mechanism with the scheduler, such as swapping [2] or block paging [75].

Research. More hard data on memory usage in parallel supercomputers is required. Desirable data includes not only total memory usage, but also the possible correlation with runtime, and the questions of locality, working sets, and changes across phases of the computation.

Pitfall 17 ⚡⚡

Scaling traces to different machine sizes

Problem. The parameters that comprise workloads and traces do not scale linearly with machine size. Additionally, scaling down workloads by trimming away the jobs that have more processors than required can distort the workload properties and affect scheduling metrics [39].

Suggestions. If using simulation or analysis, adjust simulated machine size to the one given in trace [10]. If running an experimental evaluation or for experimental reasons the workload's machine size needs to be fixed (e.g., to compare different workloads on the same machine size), use a scaling model to change the workload size [15]. If possible, verify that the scaling preserves the metrics being measured. Alternatively, use a reliable workload model to generate a synthetic workload for the desired machine size. For example, Lublin postulated a piecewise log-uniform distribution of job sizes, and specified the parameters of the distribution as a function of the machine size [48].

Research. No perfect models for workload scaling exist yet. Such models should be checked against various real workloads. Specifically, what happens with very large systems? does it depend on the machine's usage? e.g., do capability machines have more large jobs than general usage machines?

Pitfall 18 

Changing a single parameter to modify load

Problem. It is often desired to repeat an experiment with various offered loads, in order to evaluate the sensitivity and saturation of a job scheduler. This is often done by expanding or condensing the distribution of one of these parameters: job interarrival time, job run time, and degree of parallelism [47,68]. In general however, the following problems arise:

- changing P (job size) causes severe packing problems that dominate the load modification, especially since workloads tend to have many powers of 2, and machines tend to be powers of 2.
- changing T (job runtime) causes a correlation of load and response time.
- changing I (job interarrivals) changes the relative size of jobs and the daily cycle; in extreme cases jobs may span the whole night.

In addition, changing any of these parameters alone can distort the correlations between these parameters and incoming load [68].

Suggestions. One way to avoid this problem is to use model-derived workloads instead of trace data. Ideally, a workload model should be able to produce a representative workload for any desired load. However, if we wish to use an actual trace for added realism or comparison reasons, we cannot suggest a bulletproof method to vary load. To the best of our knowledge, the question of adjusting traces load in a representative manner is still open, so we may have to compromise on changing a single parameter, and advise the reader of the possible caveat.

If an evaluation nevertheless requires a choice of a single-value parameter change to vary load, changing interarrivals is in our opinion the least objectionable of these.

Research. How to correctly modify the load is an open research question.

Pitfall 19 

Using FCFS queuing as the basis of comparison

Problem. First-come-first-serve with no queue management makes no sense for most dynamic workloads, since many proven backfilling techniques exist and offer better performance [31]. It only makes sense when the workload is homogeneous with large jobs, since backfilling is mostly beneficial when the workload has variety.

Suggestions. Employ any reasonable backfilling method, such as EASY or conservative [44,51].

Pitfall 20 

Using wall-clock user estimates for a time-slicing backfilling scheduler

Problem. Backfilling requires an estimate of run times to make reservations for jobs. These run times cannot be guaranteed however in a time-slicing scheduler, even with perfect user estimates, because run times change with the dynamic MPL.

Suggestions. One heuristic to give an upper bound for reservation times is to multiply user estimates by the maximum MPL [33].

Research. Come up with more precise heuristics for estimating reservation times under time slicing.

5 Metrics

Different metrics are appropriate for different system models [30]. Makespan is suitable for off-line scheduling. Throughput is a good metric for closed systems. When considering on-line open systems, which are the closest model to how a real system operates, the metrics of choice are response time and slowdown.

Pitfall 21 

Using irrelevant/wrong/biased metrics

Problem. Some metrics do not describe a real measured value, such as utilization (see pitfall 22). Other metrics may not mean the same thing in different contexts (e.g., slowdown in a time slicing environment vs. non-time-slicing [21,82], or makespan for open vs. closed workloads).

Suggestions. Metrics should be used in an appropriate context of workload and applications [30]. Even within the context of a workload, there is room to measure metrics separately (or use different metrics) for different classes of applications, e.g., based on their type or resource requirements [33,75].

Pitfall 22 

Measuring utilization, throughput, or makespan for an open model

Problem. This is a special case of the previous pitfall, but occurs frequently enough to merit its own pitfall.

"Open models" correspond to systems that operate in an on-line mode [23]. This means that jobs arrive in a continuous but paced manner. In this context, the most significant problem with utilization and throughput as metrics is that they are a measure of the offered load more than of any effect the scheduler may have [30]. In fact, utilization should *equal* the offered load unless the system is saturated (pitfall 25); thus measuring utilization is useful maily as a sanity check. Makespan is also irrelevant for an open model, as it is largely determined by the length of the workload: how many jobs are simulated or measured.

Suggestions. The relevant metric in an open system is not the utilization, but the saturation point: the maximum utilization that can be achieved. In practice, this is reflected by the "knee" of the response-time or slowdown curve [59]. However, identification of the knee is somewhat subjective. A more precise definition is the limiting value of the asymptote, which can also be identified by the departure from the diagonal of the utilization curve, where the measured load becomes lower than the offered load. Somtimes it can be found analytically based on the distribution of job sizes [28].

For open systems, other metrics can and should also be used, and are relevant for all loads up to the saturation point. These include slowdown, response time, and wait time. But note that measuring these metrics beyond (and even close to) the saturation point leads to meaningless results that reflect only the size of the workload. Measuring utilization is may only be relevant in the context of addmission controls (pitfall 25).

In a closed system, throughput is the most important metric. For static workloads, where all the jobs are assumed to arrive at the same time, makespan can be used [30]. In this case, makespan and utilization provide the same information.

Usage note. Yet another difficulty with utilization is that there are some variations in its definition. Basically, utilization is that fraction of the available resources that is actually used. One question is then what are the available resources, and specifically, whether or not to take machine inavailability into account [59]; we would suggest to do so, but availability data is not always available.

Another issue is that "actually used" can be interpreted in different ways. A commonly used option is to consider all nodes that are *allocated* to running jobs. However, some parallel machines allow processor allocation only in fixed quanta, or specific dimensions, thereby forcing the allocation of more processors than the job actually requires. For example, BlueGene/L allocates processors in units of 512 [40], and the Cray T3D allocates power-of-two processors, starting from two [22]. The question then arises, how to measure utilization in light of the fact that many jobs in parallel workloads actually have a very low degree of parallelism [43,48], or just different sizes than those of the machine's allocated sizes, and thus necessarily have unused processors allocated to them.

The above leads to the alternative of only counting nodes that are actually *used*, thus explicitly accounting for effects such as the internal fragmentation

cited above. An even more extreme definition only considers actual CPU utilization, in an attempt to factor out effects such as heavy paging that reduces CPU utilization [29]. However, the required data is typically not available.

When referring to utilization, one should therefore be specific about what exactly is measured.

Pitfall 23 ⚡⚡

Using the mean for asymmetrically distributed (skewed) results

Problem. Some metrics are asymmetrically distributed, sometimes even heavy tailed. One example is slowdown, where short jobs have disproportionately longer slowdowns than the rest of the workload. Averaging these values yields a distorted picture, since the mean is disproportionately affected by the tail [11,33].

Suggestions. Try to describe the distribution instead of using a mean. Alternative metrics can often be devised to bypass this problem, such as bounded slowdown [19] and weighted response time [60]. Other statistical tools can be used to describe the measurements more accurately, such as median, geometric mean, or box plot. Another alternative is to divide the results into bins (e.g., short/long narrow/wide jobs [33,61]) and analyze each bin separately.

Pitfall 24 ⚡⚡

Inferring scalability trends from O*(1) nodes*

Problem. Performance results rarely scale linearly [5]. Measuring a near-constant growth of a metric of a small number of nodes and inferring scalability of the property is risky. This generalization naturally is even more applicable for simulation and analysis based studies, that hide an assumption about the scalability of the underlying hardware or mechanisms.

Suggestions. Barring actual measurement, use detailed models or qualified estimates. In case of simulation, the most reasonable approach is to simulate larger machines.

Research. Indeed, how can we learn about scalability with minimal effort? what can we do if we cannot get a 1000+ node machine to run on?

6 Measurement Methodology

In both simulations and actual measurements, performance is evaluated by having the scheduler schedule a sequence of jobs. Previous sections have listed pitfalls related to the workload itself, i.e. which jobs should appear in this sequence, and to the metrics used to measure performance. This final section is about the context of the measurements.

The considerations involved in these pitfalls are well-known in the simulation literature, and can be summarized as ensuring that we are simulating (and measuring) the system in its steady state. The problem is that with traces it may

be hard to achieve a steady state, or easy to overlook the fact that the state is not steady.

Pitfall 25 ///

Measuring saturated workloads

Problem. This is one of the most common errors committed in the evaluation of parallel job scheduling schemes. In queuing theory terms, a system is stable only if the arrival rate is lower than the service rate ($\lambda < \mu$, or alternatively $\rho = \frac{\lambda}{\mu} < 1$). If this condition is violated the system is unstable — it has no steady state.

The saturation point is the maximal load that the system can handle. Keep in mind that many parallel workloads and machines do not even get close to 100% utilization [39,59], due to loss of resources to fragmentation (pitfall 26). Evaluating the system for loads beyond the saturation point is unrealistic and typically leads to meaningless results.

Measuring the behavior of a parallel system with an offered load that is higher than the saturation point would yield infinitely-growing queues on a real system, and metrics such as average wait time and slowdown will grow to infinity. Thus the results of the measurement would *depend on the length of the measurement* — the longer your workload, the worse it gets. However, it is easy to miss this situation in a real measurement, because all the evaluations we perform are finite, and finite workloads always converge in the end. But results we measure on such workloads are actually invalid.

Suggestions. Identify the saturation point by comparing the offered load to the achieved utilization (see pitfall 22) and discard data points from saturated experiments, effectively limiting the experimental results to the relevant domain [22,33].

Transient saturation for a limited time should be allowed, as it is a real phenomenon. However, if this happens toward the end of a simulation/measurement, it may indicate a saturated experiment.

The only case where measurement with an offered load higher than the saturation point are relevant is when we are considering admission policies. This means that the system is designed to deal with overload, and does so by discarding part of its input (i.e. some jobs are simply not serviced). In this case relevant metrics are the fraction of jobs that are serviced and the achieved utilization.

Pitfall 26 //

Ignoring internal fragmentation

Problem. Even an optimal scheduler that eliminates external fragmentation entirely (i.e., all processors are always allocated) might suffer pitiful response times and throughput if processor efficiency is not taken into account. Most applications scale sublinearly with processors and/or include inherent internal fragmentation (e.g., due to the "memory wall"). When using such

applications, measuring system-centric metrics only (such as machine utilization in terms of processor allocation, without considering efficiency) can produce results that indeed favor the system view, while specific applications suffer from poor response times.

Suggestions. To the extent that a scheduler's designer can alleviate these phenomena, reducing internal fragmentation should be given the same consideration as reducing external fragmentation. For example, adaptive and dynamic partitioning can increase application efficiency by matching the partition size to the degree of parallelism of applications, although this poses certain requirements on the scheduler and applications [52,66]. If possible, include a dynamic coscheduling scheme [4,34,63,65] in the evaluation, as these tend to reduce internal fragmentation.

Whether using any of these methods or not, an experimental evaluator should remain cognizant of internal fragmentation, and address its effect on the evaluation.

Pitfall 27 

Using exceedingly short workloads

Problem. Some phenomena, including finding the saturation point (pitfall 22) or fragmentation of resources (such as contiguous processors or disk space in a file system [62]), only appear with long enough workloads. Thus we need long workloads not only to achieve a steady state, but even more so to see the realistic conditions that a real system faces.

Suggestions. For an analysis or simulation study, use thousands of jobs, e.g. use a rule of thumb of 30 batches of 5000 jobs each (but note that even this may not suffice for some skewed distributions) [49]. This is more difficult for an experimental evaluation. In this case, use as many jobs as practical and/or use a higher load to pack more jobs into the same amount of time (but note pitfall 18).

Research. Immediate research issues are related to aging. How to do aging quickly? How to quantify appropriate aging?

At a broader level, these phenomena are related to new research on software rejuvenation [71] — rebooting parts of the system to restore a clean state. In real systems this may also happen, as systems typically don't stay up for extended periods. As each reboot causes the effects of aging to be erased, it may be that short workloads are actually more representative! This would also imply that transient conditions that exist when the system is starting up or shuting down should be explicitly studied rather than being avoided as suggested in pitfalls 28 and 29.

Pitfall 28 

Not discarding warm-up

Problem. A special case of pitfall 27 that is important enough to be noted separately.

Initial conditions are different in a consistent manner, so even averaging over runs will be biased. Thus it is wrong to include the initial part of a measurement or simulation in the results, as it is not representative of steady state conditions.

Suggestions. The first few data points can be discarded to account for warmup time. There are various statistical methods to decide how much to discard, based on estimates of whether the system has entered a steady state [56]. As an approximation, a running average of the performance metric can be drawn to see when it stabilizes (which will not necessarily be easy to identify because of diverging instantaneous values).

Research. Which statistical methods are specifically suitable to identify the steady state of parallel workloads? And is there really a steady state with real workloads? Alternatively, is rebooting common enough that it is actually important to study the initial tranzient conditions?

Pitfall 29 

Not avoiding cool-down

Problem. Another special case of pitfall 27.

Towards the end of a measurement/simulation run, problematic jobs remain and enjoy a dedicated system with less competition. This also causes the measured load to deviate from the intended one. Therefore a good indication of this problem is that the system appears to be saturated (pitfall 25).

Suggestions. Stop the measurements at the last arrival at the latest. Count terminations to decide when enough jobs have been measured, not arrivals. Then, check jobs left in queue, to see if there are systematic biases.

Pitfall 30 

Neglecting overhead

Problem. Overhead is hard to quantify. Overhead and operation costs come in many forms, not all of them even known in advance. Some overhead effects are indirect, such as cache and paging effects, or contention over shared resources.

Suggestions. Model the distribution of overhead [66], incorporate data from traces or actual machines [37], or use a range of overhead values [50].

Research. What are overheads of real systems? not much data on overhead is available, and it continuously changes with technology.

Pitfall 31 

Measuring all jobs

Problem. One aspect of this pitfall has already been covered as pitfall 5: that some jobs are unrepresentative, and the data should be cleaned. But there is more to this.

As loads fluctuate, many of the jobs actually see an empty or lightly loaded system. In these cases the scheduler has no effect, and including them just

dilutes the actual measurements. Additionally, different jobs may experience very different conditions, and it is questionable whether it is meaningful to average all of them together.

Another aspect of this pitfall is the distinction between jobs that arrive during the day, at night, and over the weekend. Many sites have policies that mandate different levels of service for different classes of jobs at different times [46,74]. Obviously evaluating such policies should take the characteristics of the different job classes into account, and not bundle them all together.

Suggestions. Partition jobs into classes according to conditions, and look at performance of each class separately. For example, only look at jobs that are high-priority, prime-time, interactive, or belong to a certain user. The emphasis here is not on presenting results for all possible categories, but rather, on identifying the important conditions or categories that need to be taken into account, and then presenting the results accordingly.

Research. To the best of our knowledge, nobody does this yet. Many new pitfalls may be expected in doing it right.

Pitfall 32

Comparing analysis to simulations

Problem. Comparisons are important and welcome, but we must be certain that we are validating the correct properties: Both simulation and analysis could embody the same underlying hidden assumptions, especially if developed by the same researchers. Experimental evaluations tend to expose unaccounted-for factors.

Suggestions. Compare analysis and/or simulation to experimental data [1].

7 Conclusion

As the field of parallel job scheduling matures, it still involves many poorly understood and often complex factors. Despite this situation, and perhaps because of it, we need to approach the study of this field in a scientific, reproducible manner. This paper concentrates on 32 of the more common pitfalls in parallel job scheduling evaluation, as well as a few of the more subtle ones. It is unlikely that any study in this field will be able to follow all the complex (and sometimes contradicting) methodological suggestions offered in this paper. Nor is it likely that any such study will be immune to other methodological critique. Like other fields in systems research, parallel job scheduling entails compromises and tradeoffs. Nevertheless, we should remain circumspect of these choices and make them knowingly. It is the authors' hope that by focusing on these topics in a single document, researchers might be more aware of the subtleties of their evaluation. Ideally, disputable assumptions that are taken in the course of a study can be justified or at least addressed by the researcher, rather than remain undocumented. By promoting more critical evaluations and finer attention to methodological issues, we hope that some of the "black magic" in the field will be replaced by reliable and reproducible reasoning.

References

1. K. Aida, H. Kasahara, and S. Narita. Job scheduling scheme for pure space sharing among rigid jobs. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 98–121. Springer-Verlag, 1998.
2. G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith. Scheduling on the Tera MTA. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 19–44. Springer-Verlag, 1995.
3. C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Informing algorithms for efficient scheduling of synchronizing threads on multiprogrammed SMPs. In *30th International Conference on Parallel Processing (ICPP)*, pp. 123–130, Valencia, Spain, September 2001.
4. A. C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, August 2001.
5. D. H. Bailey. Misleading performance in the supercomputing field. In *IEEE/ACM Supercomputing*, pp. 155–158, Minneapolis, MN, November 1992.
6. D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS parallel benchmark results. In *IEEE/ACM Supercomputing*, pp. 386–393, Minneapolis, MN, November 1992.
7. A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *14th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 109–114, May 2000.
8. T. B. Brecht. An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 139–165. Springer-Verlag, 1997.
9. S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Fifth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pp. 67–90. Springer-Verlag, 1999.
10. W. Cirne and F. Berman. Adaptive selection of partition size for supercomputer requests. In *Sixth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pp. 187–207. Springer-Verlag, 2000.
11. M. E. Crovella. Performance evaluation with heavy tailed distributions. In *Seventh Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pp. 1–10. Springer Verlag, 2001.
12. R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *20th International Symposium on Computer Architecture (ISCA)*, pp. 2–13, May 1993.
13. A. B. Downey. A model for speedup of parallel programs. Technical Report UCB/CSD-97-933, University of California, Berkeley, CA, January 1997.
14. A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *Performance Evaluation Review*, 26(4):14–29, March 1999.
15. C. Ernemann, B. Song, and R. Yahyapour. Scaling of workload traces. In *Ninth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pp. 166–182. Springer-Verlag, 2003.

16. D. G. Feitelson. Packing schemes for gang scheduling. In *Second Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pp. 89–110. Springer-Verlag, 1996.
17. D. G. Feitelson. Memory usage in the LANL CM-5 workload. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 78–94. Springer-Verlag, 1997.
18. D. G. Feitelson. A critique of ESP. In *Sixth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pp. 68–73. Springer-Verlag, 2000.
19. D. G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Seventh Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pp. 188–1205. Springer Verlag, 2001.
20. D. G. Feitelson. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459 of *Lecture Notes in Computer Science*, pp. 114–141. Springer-Verlag, September 2002.
21. D. G. Feitelson. Metric and workload effects on computer systems evaluation. *Computer*, 36(9):18–25, September 2003.
22. D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 238–261. Springer-Verlag, 1997.
23. D. G. Feitelson and A. W. Mu'alem. On the definition of "on-line" in job scheduling problems. *ACM SIGACT News*, 36(1):122–131, March 2005.
24. D. G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 337–360. Springer-Verlag, 1995.
25. D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.
26. D. G. Feitelson and L. Rudolph. Coscheduling based on run-time identification of activity working sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.
27. D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 1–18. Springer-Verlag, 1995.
28. D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, May 1996.
29. D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Second Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pp. 1–26. Springer-Verlag, 1996.
30. D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 1–24. Springer-Verlag, 1998.
31. D. G. Feitelson, L. Rudolph, and U. Schwigelshohn. Parallel job scheduling – A status report. In *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pp. 1–16. Springer-Verlag, 2004.

32. D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium (IPPS)*, pp. 542–546, April 1998.
33. E. Frachtenberg, D. G. Feitelson, J. Fernandez-Peinador, and F. Petrini. Parallel job scheduling under dynamic workloads. In *Ninth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pp. 208–227. Springer-Verlag, 2003.
34. E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed Systems*, To appear.
35. A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pp. 120–32, San Diego, CA, May 1991.
36. G. Holt. Time-critical scheduling on a well utilised HPC system at ECMWF using LoadLeveler with resource reservation. In *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pp. 102–124. Springer-Verlag, 2004.
37. A. Hori, H. Tezuka, and Y. Ishikawa. Overhead analysis of preemptive gang scheduling. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 217–230. Springer-Verlag, 1998.
38. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan. Modeling of workload in MPPs. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 95–116. Springer-Verlag, 1997.
39. J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Fifth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pp. 1–16. Springer-Verlag, 1999.
40. E. Krevat, J. G. Castaños, and J. E. Moreira. Job scheduling for the BlueGene/L system. In *Eighth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pp. 38–54. Springer Verlag, 2002.
41. C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pp. 253–263. Springer-Verlag, 2004.
42. W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for gang scheduled workloads. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 215–237. Springer-Verlag, 1997.
43. H. Li, D. Groep, and L. Wolters. Workload characteristics of a multi-cluster supercomputer. In *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pp. 176–193. Springer-Verlag, 2004.
44. D. Lifka. The ANL/IBM SP scheduling system. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 295–303. Springer-Verlag, 1995.

45. W. Liu, V. Lo, K. Windisch, and B. Nitzberg. Non-contiguous processor allocation algorithms for distributed memory multicomputers. In *IEEE/ACM Supercomputing*, pp. 227–236, November 1994.
46. V. Lo and J. Mache. Job scheduling for prime time vs. non-prime time. In *Fourth Proceedings of the IEEE International Conference on Cluster Computing*, pp. 488–493, September 2002.
47. V. Lo, J. Mache, and K. Windisch. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 25–46. Springer-Verlag, 1998.
48. U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, November 2003.
49. M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools*. MIT Press, 1987.
50. J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette. An infrastructure for efficient parallel job execution in terascale computing environments. In *IEEE/ACM Supercomputing*, Orlando, FL, November 1998.
51. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.
52. T. D. Nguyen, R. Vaswani, and J. Zahorjan. Parallel application characterization for multiprocessor scheduling policy design. In *Second Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pp. 175–199. Springer-Verlag, 1996.
53. N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.
54. Parallel workload archive. `www.cs.huji.ac.il/labs/parallel/workload`.
55. E. W. Parsons and K. C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 127–145. Springer-Verlag, 1995.
56. K. Pawlikowski. Steady-state simulation of queueing processes: A survey of problems and solutions. *ACM Computing Surveys*, 22(2):123–170, June 1990.
57. V. G. J. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pp. 5–18, Nashville, TN, May 1994.
58. A. program. ASCI technology prospectus: Simulation and computational science. Technical Report DOE/DP/ASC-ATP-001, National Nuclear Security Agency, July 2001.
59. L. Rudolph and P. Smith. Valuation of ultra-scale computing systems. In *Sixth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pp. 39–55. Springer-Verlag, 2000.
60. U. Schwiegelshohn and R. Yahyapour. Improving first-come-first-serve job scheduling by gang scheduling. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 180–198. Springer-Verlag, 1998.

61. S. K. Setia. The interaction between memory allocation and adaptive partitioning in message-passing multicomputers. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 146–164. Springer-Verlag, 1995.
62. K. A. Smith and M. I. Seltzer. File system aging—Increasing the relevance of file system benchmarks. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pp. 203–213, June 1997.
63. P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 106–126. Springer-Verlag, 1995.
64. A. C. Sodan and L. Lan. LOMARC—Lookahead matchmaking for multi-resource coscheduling. In *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pp. 288–315. Springer-Verlag, 2004.
65. M. Squillante, Y. Zhang, S. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira. Modeling and analysis of dynamic coscheduling in parallel and distributed environments. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pp. 43–54, Marina Del Rey, CA, June 2002.
66. M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 219–238. Springer-Verlag, 1995.
67. M. S. Squillante, D. D. Yao, and L. Zhang. Analysis of job arrival patterns and parallel scheduling performance. *Performance Evaluation*, 36–37:137–163, 1999.
68. D. Talby, D. G. Feitelson, and A. Raveh. Comparing logs and models of parallel workloads using the Co-Plot method. In *Fifth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pp. 43–66. Springer-Verlag, 1999.
69. S. Tongsima, C. Chantrapornchai, and E. H.-M. Sha. Probabilistic loop scheduling considering communication overhead. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 158–179. Springer-Verlag, 1998.
70. Top 500 supercomputers. `www.top500.org`.
71. K. S. Trivedi and K. Vaidyanathan. Software reliability and rejuvenation: Modeling and analysis. In *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459 of *Lecture Notes in Computer Science*, pp. 318–345. Springer-Verlag, 2002.
72. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. In *11th Workshop on Job Scheduling Strategies for Parallel Processing*. 2005.
73. D. Tsafrir and D. G. Feitelson. Workload flurries. Technical Report 2003-85, Hebrew University, November 2003.
74. M. Wan, R. Moore, G. Kremenek, and K. Steube. A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm. In *Second Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pp. 48–64. Springer-Verlag, 1996.
75. F. Wang, M. Papaefthymiou, and M. Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In *Third Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pp. 277–298. Springer-Verlag, 1997.

76. W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. In *ACM SIGCOMM*, pp. 100–113, 1995.
77. Y. Wiseman and D. G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, June 2003.
78. A. T. Wong, L. Oliker, W. T. C. Kramer, T. L. Kaltz, and D. H. Bailey. ESP: A system utilization benchmark. In *IEEE/ACM Supercomputing*, pp. 52–52, Dallas, TX, November 2000.
79. K. K. Yue and D. J. Lilja. Loop-level process control: An effective processor allocation policy for multiprogrammed shared-memory multiprocessors. In *First Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pp. 182–199. Springer-Verlag, 1995.
80. Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. In *Seventh Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pp. 133–158. Springer Verlag, 2001.
81. Y. Zhang, A. Yang, A. Sivasubramaniam, and J. Moreira. Gang scheduling extensions for I/O intensive workloads. In *Ninth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pp. 166–182. Springer-Verlag, 2003.
82. B. B. Zhou, R. P. Brent, D. Walsh, and K. Suzaki. Job scheduling strategies for networks of workstations. In *Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pp. 143–157. Springer-Verlag, 1998.

Author Index