Dror Feitelson
Larry Rudolph
Uwe Schwiegelshohn (Eds.)

LNCS 3277

Job Scheduling Strategies for Parallel Processing

10th International Workshop, JSSPP 2004
New York, NY, USA, June 2004
Revised Selected Papers

Springer

Lecture Notes in Computer Science 3277

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

Dror Feitelson   Larry Rudolph
Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

10th International Workshop, JSSPP 2004
New York, NY, USA, June 13, 2004
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

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

Uwe Schwiegelshohn
University of Dortmund
Computer Engineering Institute
44221 Dortmund, Germany
E-mail: uwe.schwiegelshohn@udo.edu

Library of Congress Control Number: 2005925176

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	3-540-25330-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-25330-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11407522 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 10th Anniversary Workshop on Job Scheduling Strategies for Parallel Processing. The workshop was held in New York City, on June 13, 2004, at Columbia University, in conjunction with the SIGMETRICS 2004 conference.

Although it is a workshop, the papers were conference-reviewed, with the full versions being read and evaluated by at least five and usually seven members of the Program Committee. We refer to it as a workshop because of the very fast turnaround time, the intimate nature of the actual presentations, and the ability of the authors to revise their papers after getting feedback from workshop attendees. On the other hand, it was actually a conference in that the papers were accepted solely on their merits as decided upon by the Program Committee.

We would like to thank the Program Committee members, Su-Hui Chiang, Walfredo Cirne, Allen Downey, Eitan Frachtenberg, Wolfgang Gentzsch, Allan Gottlieb, Moe Jette, Richard Lagerstrom, Virginia Lo, Reagan Moore, Bill Nitzberg, Mark Squillante, and John Towns, for an excellent job.

Thanks are also due to the authors for their submissions, presentations, and final revisions for this volume. Finally, we would like to thank the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), The Hebrew University, and Columbia University for the use of their facilities in the preparation of the workshop and these proceedings.

This year saw a continued interest in scheduling in grid and cluster environments, with a growing representation of real-system issues such as workload studies, network topology issues, and the effect of failures. At the same time, there was also a strong representation of research relating to classical multiprocessor systems, and lively discussions contrasting the academic point of view with that of administrators of ‘real’ systems. We hope that the papers in this volume capture this range of interests and approaches, and that you, the reader, find them interesting and useful.

This was the tenth annual workshop in this series, which reflects a consistent and ongoing interest; the organizers believe that the workshop satisfies a real need. The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, and 2862 (and since 1998 they have also been available online). We look forward to the next workshop in 2005, and perhaps even to the next decade of workshops!

September 2004

Dror Feitelson
Larry Rudolph
Uwe Schwiegelshohn

Table of Contents

Parallel Job Scheduling — A Status Report

Dror G. Feitelson[1], Larry Rudolph[2], and Uwe Schwiegelshohn[3]

[1] School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
[2] Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
[3] Computer Engineering Institute
Universität Dortmund
44221 Dortmund, Germany

1 Introduction

The popularity of research on the scheduling of parallel jobs demands a periodic review of the status of the field. Indeed, several surveys have been written on this topic in the context of parallel supercomputers [17, 20]. The purpose of the present paper is to update that material, and to extend it to include work concerning clusters and the grid.

The paper is divided into three major parts. The first part addresses algorithmic and research issues covering the two main approaches: backfilling and gang scheduling. For each, recent advances are reviewed, both in terms of how to perform the scheduling and in terms of understanding the performance results. An underlying theme of the surveyed results is the shift from dogmatic use of rigid formulations to a more flexible approach. This reflects a maturation of the field and improved concern for real-world issues.

The second part of the paper addresses current usage. It presents a short overview of vendor offerings, and then reviews the scheduling frameworks used by top-ranking parallel systems. For vendor offerings, we highlight the distinction between what is done in a research setting and what is actually developed for production use. Regarding actual usage, we consider the alternative options of procurement of an existing system vs. the development of an in-house solution that more directly reflects desired attributes.

The third part of the paper looks both back and forward in time. As with any field, the success, popularity, and influence of a particular approach depends on a range of factors. We review some less successful ones. It is possible that some of these techniques may only be relevant to future machines. The paper, therefor concludes with some observations about the near-term future.

This paper contains a large number of references. In order to highlight the more recent results, i.e. those with publication dates in this millennium, their citation will be superscripted with the last two digits of the publication date.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 1–16, 2005.

2 Advances in Parallel Job Scheduling Research

There are many different ways to schedule parallel jobs and their constituent threads [17], but only a few mechanisms are used in practice and studied in detail. This section reviews backfilling and gang scheduling strategies, their variants, and their connections. The special requirements and strategies for scheduling parallel jobs on a grid are addressed as well.

2.1 Backfilling

The most basic batch scheduling algorithm is First-Come-First-Serve (FCFS) [43] where jobs are considered in order of arrival. Each job specifies the number of processors it requires and is placed in a FIFO queue upon arrival. If there are sufficient available processors to run the job at the head of the queue, the processors are allocated and the job is started. If there are not enough, the scheduler waits for some currently running job to terminate and free additional processors.

Backfilling is an optimization that tries to balance the goals of utilization and maintaining FCFS order. It requires that each job also specifies its maximum execution time. While the job at the head of the queue is waiting, it is possible for other, smaller jobs, to be scheduled, especially if they would not delay the start of the job on the head of the queue. Processors get to be used that would otherwise remain idle.

By letting some jobs execute out of order, other jobs may get delayed. Backfilling will never completely violate the FCFS order where some jobs are never run (a phenomenon known as “starvation”). In particular, jobs that need to wait are typically given a reservation for some future time.

The use of reservations was included in several early batch schedulers [29, 8]. Backfilling, in which small jobs move forward to utilize the idle resources, was introduced by Lifka [33]. This was done in the context of EASY, the Extensible Argonne Scheduling sYstem, which was developed for the first large IBM SP1 installation at Argonne National Lab.

Variations on Backfilling While the concept of backfilling is quite simple, it nevertheless has several variants with subtle differences. We generalize the behavior of backfilling by parameterizing several constants. Judicial choice of parameter values lead to improved performance.

One parameter is the *number of reservations*. In the original EASY backfilling algorithm, only the first queued job received a reservation. Jobs may be scheduled out of order only if they do not delay the job at the head of the queue. The scheduler estimates when a sufficient number of processors will be available for that job and reserves them for this job. Other backfilled jobs may not violate this reservation, they must either terminate before the time of the reservation (known as the “shadow time”), or use only processors that are not required by the first job [33].

Backfilling may cause delays in the execution of other waiting jobs (which are not the first, and therefore do not get a reservation). The obvious alternative is to make reservations for all jobs. This approach has been named "conservative backfilling" [37]^{01}. Simulation results indicate, however, that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison with the more aggressive EASY backfilling. The MAUI scheduler includes a parameter that allows system administrators to set the number of reservations [30]^{01}. Chiang et al. suggest that making up to four reservations is a good compromise [6]^{02}.

An intriguing recent suggestion is adaptive reservations depending on the extent different jobs have been delayed by previous backfilling decisions. If a job is delayed by too much, a reservation is made for this job [50]^{02}. This is essentially equivalent to the earlier "flexible backfilling", in which all jobs have reservations, but backfilling is allowed to violate these reservations up to a certain slack [51]. Setting the slack to the threshold used by adaptive reservations is equivalent to only making a reservation if the delay exceeds this threshold.

Another parameter is the *order of queued jobs*. The original EASY scheduler, and many other systems and designs, use a first come, first served (FCFS) order [33]. A general alternative is to prioritize jobs in some way, and select jobs for scheduling (including as candidates for backfilling) according to this priority order. Flexible backfilling combines three types of priorities: an administrative priority set to favor certain users or projects, a user priority used to differentiate among the jobs of the same user, and a scheduler priority used to guarantee that no job is starved [51]. The Maui scheduler has a priority function that includes even more components [30]^{01}.

A special type of prioritization depends on job characteristics. In particular, Chiang et al. have proposed a whole set of criteria based on resource consumption, that are generalizations of the well-known Shortest Job First (SJF) scheduling algorithm [6]^{02}. These have been shown to improve performance metrics, especially those that are particularly sensitive to the performance of short jobs, such as slowdown.

A final parameter is the amount of *lookahead into the queue*. All previous backfilling algorithms consider the queued jobs one at a time, and try to schedule them. But the order in which jobs are scheduled may lead to loss of resources to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize desired performance metrics. This can be done using dynamic programming, leading to optimal packing and improved performance [47]^{03}.

Effect of User Runtime Estimates Backfilling depends on estimates of how long each job will run to figure out when additional processors will become available, and to verify that backfilled jobs will terminate in time so as not to violate reservations. The source of the estimates is typically the user who submits the job. Jobs that execute beyond their estimated runtime are usually terminated by the system. Many users therefore regard these estimates as upper bounds, rather than as tight estimates.

Initial expectations were that user runtime estimates will nevertheless be tight, as low estimates improve the chance for backfilling. However, comparisons of user estimates with real runtimes show that they tend to be inaccurate, even when users are requested to provide their best possible estimate with no danger of having their job killed if the estimate is too low [18],[37]^{01},[32]^{04}. Attempts to derive better estimates automatically based on historical information from previous runs have not been successful, as they suffered from too many underestimations (which in backfilling would lead to killed jobs).

Probably, the most surprising result demonstrated by several studies has shown that inaccurate runtime estimates actually lead to *improved* average performance [18, 61],[37]^{01}. This is not simply the result of more backfilling due to more holes in the schedule, because inflated runtime estimates not only create holes in the schedule, but also enlarge potential backfill jobs, making it harder for them to fit into the holes. Rather, it is the result of a sequence of events where small backfill jobs prevent the holes from closing up, leading to a strong preference for short jobs and the automatic production of an SJF-like schedule [53]^{04}. This also motivates the construction of algorithms that explicitly favor short jobs such as those proposed by Chiang et al. [6]^{02}.

This does not necessarily indicate that more accurate runtime estimates are impossible and useless. Not all estimates are bad; in most cases, some users provide reasonably accurate estimates while others do not. Some studies indicate that those users who do provide reliable estimates do indeed benefit, as their jobs receive better service from the scheduler [6]^{02}. Also, while it seems that deriving good estimates automatically is not possible for all jobs, it might be possible to do so for short jobs and for jobs that have exhibited especially small variability in the past.

Incidently, inaccurate user runtime estimates have been shown to have surprising effects on performance evaluations [16]^{03},[15]^{03}. In a nutshell, it was seen that for workloads with numerous long single-process jobs, the inaccurate estimates allow for significant backfilling of these jobs under the aggressive EASY backfilling, but not under conservative backfilling. This in turn was detrimental for the performance of short jobs that were delayed by the long backfilled jobs. But if accurate estimates were used the effect was reversed, leading to a situation where short jobs were favored over long ones. This has more to do with evaluation methodology than will scheduling technology.

2.2 Gang Scheduling

The main alternative to batch scheduling is gang scheduling, where jobs are preempted and re-scheduled as a unit, across all involved processors. The notion was introduced by Ousterhout, using the analogy of a working set of memory pages to argue that a “working set” of processes should be co-scheduled for the application to make efficient progress [38]. Subsequent work emphasized gang scheduling, which is an all-or-nothing affair, i.e. either all of the job's processes run or none do.

The point of gang scheduling is that it provides an environment similar to a dedicated machine, in which all a job's threads progress together, and at the same time allows resources to be shared. In particular, preemption is used to improve performance in face of unknown runtimes. This prevents short jobs from being stuck in the queue waiting for long ones, and improves fairness [44]^{00}.

Flexible Algorithms One problem with gang scheduling is that the requirement that all a job's processes always run together causes too much fragmentation. This has led to several proposals for more flexible variants.

One such variant, called "paired gang scheduling" is designed to alleviate inefficiencies caused by I/O activity [56]^{03}. In conventional gang scheduling, processors running processes that perform I/O remain idle for the duration of the I/O operation. In paired gang scheduling jobs with complementary characteristics are paired together, so that when the processes of one perform I/O, those of the other can compute. Given a good job mix, this can lead to improved resource utilization at little penalty to individual jobs.

A more general approach is to monitor the communication behavior of all applications, and try to determine whether they really benefit for gang scheduling [24]^{03}. Gang scheduling is then used for those that need it. Processes belonging to other jobs are used as filler to reduce the fragmentation cause by the gang scheduled jobs.

Dealing with Memory Pressure Early evaluations of gang scheduling assumed that all arriving jobs can be started immediately. Under high loads this could lead to situations where dozens of jobs share each processor. This is unrealistic as all these jobs would need to be memory resident or else suffer from paging, which would interfere with the synchronization among the job's threads.

A simple approach for avoiding this problem is to use admission controls, and only allow additional jobs to start if enough memory is available [3]^{00}. An alternative is placing an oblivious cap on the multiprogramming level (MPL), usually in the range of 3–5 jobs [35]. While this avoids the need to estimate how much memory a new job will need, it is more vulnerable to situations in which memory becomes overcommitted causing excessive paging.

When admission controls are used and jobs wait in the queue the question of queue order presents itself. The simplest option is to use a FCFS order. Improved performance is obtained by using backfilling, and allowing small jobs to move ahead in the queue [59]^{00},[58]^{03}. In fact, using backfilling fully compensates for the loss of performance due to the limited number of jobs that are actually run concurrently [23]^{03}.

All the above schemes may suffer from situations in which long jobs are allocated resources while short jobs remain in the queue and await their turn. The solution is to use a preemptive long-range scheduling scheme. With this construction, the long term scheduler allocates memory to waiting jobs, and then the short term scheduler decides which jobs will actually run out of those that are memory resident. The long term scheduler may decide to swap out a job that has been in memory for a long time, to make room for a queued job that has been waiting for a long time. Such a scheme was designed for Tera (Cray) MTA machine [1].

System Integration The only commercially successful implementation of gang scheduling that we know of so far was the one on the Connection Machine CM-5. Other implementations, e.g. on the Intel Paragon, never moved beyond experimentation because of significant performance overheads, probably due to the cost of gang synchronization and coordination. Recent advances in the implementation of gang scheduling in experimental systems promise to reduce these overheads.

Gang scheduling requires the context switching to be synchronized across the nodes of the machine, and software-implemented synchronization on large machines is expensive. But some modern interconnection networks provide hardware support for global operations, and this can be exploited also in the runtime system. For example, in the STORM, where all parallel system activities are expressed in terms of three basic primitives, which in turn are supported by the hardware of the Quadrics network. In particular, this design has resulted in a very scalable implementation of gang scheduling [25]^{02}.

While high performance networks enable efficient implementation of system primitives, they may cause problems with multiprogramming. The difficulty arises due to the use of user-level communication, in which user processes access the network interface cards (NICs) directly so as to avoid the overheads involved in trapping into the operating system. As a result no protection is available, and only one job can use the NICs. This can be solved by switching communication buffers as part of the gang scheduling's context switch operation [14]^{01}. It is also possible that this problem will be reduced in the future, as the memory available on NICs continues to grow.

Even tighter integration between communication and scheduling is used in the "buffered coscheduling" scheme proposed by Petrini and Feng [39]^{00},[40]^{00}. In this scheme the execution of all jobs is partitioned by the system into phases. In each phase, communication operations are buffered and at the end of the phase all the required communications is scheduled and and performed during the next phase. This leads to complete overlap of computation and communication.

Gang scheduling was originally developed in order to support fine-grain synchronization of parallel applications [19]. But an even greater benefit may be its contribution to reducing interference. The problem is that the nodes of parallel machines and clusters typically run a full operating system, with various user-level daemons that are required for various system services. These daemons may wake up at unpredictable times in order to perform their function. Obviously this interferes with the application process running on the node [36]. If such interferences are not synchronized across nodes, the application will be slowed considerably as different processes are delayed. But with gang scheduling it is possible to run all the daemons on the different nodes at the same time, and eliminate their interference when user jobs are running [41]^{03}. When this is done, the full capabilities of the hardware are achieved.

2.3 Parallel Job Scheduling and the Grid

More recently, parallel computers are becoming part of a so called computational grid. The name grid has been chosen in analogy to the electrical power grid where several power plants provide numerous consumers with electrical power without the consumer being aware of the origin of the power. Similarly, it is the goal of a computational grid or simply Grid to allow users to run their jobs on any suitable computer belonging to the Grid. This way the computational load is balanced across many machines. Clearly, the Grid is mainly of interest for large computational jobs or jobs using a large data set as smaller jobs will usually run locally. However, the Grid is not restricted to this kind of jobs but will cover a wide range of general services. Nevertheless at the moment large computational jobs form the dominant grid application.

Before addressing the scheduling problem in a grid it is necessary to point out some differences between a parallel computer and the grid. A parallel computer has a central resource management system that can control all individual processors. However in a grid, the compute resources typically have different owners and as in most distributed systems there is no central control. Therefore, a compute resource typically has its own local resource management system that implements the policy of its owner. Hence, a grid scheduling architecture must be built on top of those existing local resource management systems. This requires communication between those different layers of the scheduling system in a grid [45]^{03},[55]. As in a distributed system the use of a central grid scheduler may result in a performance bottleneck and lead to a failure of the whole system if the scheduler fails. It is therefore appropriate to use a decentralized grid scheduler architecture and distributed algorithms.

Further, grid resources are heterogeneous in hardware and software which imposes constraints on the suitability of a resource for a given job. In addition, not every user may be accepted on every machine due to the implemented owner policy. A grid scheduler must determine which resources can be used for a specific submitted job while such a problem is usually not encountered in a parallel processor or even in a cluster of computers [10]^{02},[12]^{02}. Moreover, the grid is subject to frequent changes as some compute resources may be temporarily withdrawn from the grid due to maintenance or privileged non-grid use on request of the owner. To obtain these data, the grid scheduler needs a specific grid information service while the necessary up-to-date information is always assumed to be available in a parallel computer.

Today, the main purpose of grid computing is considered to be in the area of cross-domain load balancing. To support this idea the Globus Toolbox provides basic services that allow the construction of a grid scheduler [22]. With the help of those basic services grid schedulers are constructed that run on top of commercial resource management systems, like LSF, PBS or Loadleveler. Further, existing

Systems, like Condor [34, 42], are adapted to include grid scheduling abilities or allow integration with a grid scheduler.

If a parallel computer is embedded in a grid, a large variety of jobs from different users will be run on this machine. Then it will become increasingly difficult to implement the usage policy of an owner with the help of those simple scheduling criteria that are used today, like utilization and response time. Therefore, it can be assumed that the grid will also change job scheduling strategies for parallel computers. However in practice such an effect has not been observed yet.

Large grid application projects, like LCG, Datagrid, GriPhyn, frequently include the construction of some grid scheduler. Unfortunately, the scope of such a scheduler is usually restricted to the corresponding application project. On the other hand, there are academic projects that specifically address scheduling issues like the generation, distribution and selection of resource offerings. To this end various means are used, for instance economic methods.

In another approach, the job itself is responsible for its scheduling. Then we speak of an application scheduler. This is important for jobs which have a complex workflow and are subject to complex parallelization constraints. For example, this is the approach taken in the AppLes project $[7]^{00}$.

As a continuation of some metacomputing ideas it is sometimes considered to use a computational grid as a single parallel processor, where many computational resources, that is parallel computers in the grid, are combined to solve a single very large problem . In this situation, the network performance varies greatly from communication within a parallel computer to communication between two parallel computers. Some models have been derived to evaluate the performance of so called multi site computing $[26, 4, 9, 11, 13]^{03}$. However in practice, such an approach has not been implemented with the possible exception of the preplanned combination of a few specific parallel computers for a specific purpose.

An important component of using the grid as a single parallel resource is co-allocation $[5, 4, 2, 48]^{04}$. This means that resources on several different machines need to be allocated to the same job at the same time. This is hard to accomplish due to the fact that the different resources belong to different owners, and do not have a common resource management infrastructure. The way to circumvent this problem is to try and reserve resources on the different machines, and then to use them only if all required reservations are successful $[49]^{00}$.

3 Parallel Job Scheduling Practice

3.1 Vendor Offerings

Commercial scheduling software for parallel jobs comes in two types: portable, standalone systems, and components in a specific system.

There are two main competitors in the market for scheduling software. One is the Platform Computing Load Sharing Facility (LSF), which is based on the

Utopia project [60]. The other is the Veridian Portable Batch System (PBS) [27]. Both provide similar functionality. In particular, they provide support for various administrative tasks, which is often lacking from research prototypes.

In addition, vendors of parallel supercomputers typically provide some sort of scheduling support with their systems. This includes schedulers on the IBM SP, the Cray Origin, and HP and Sun systems.

3.2 Actual Usage

In order to determine which job scheduling strategies for parallel processors are actually applied in practice we considered the 50 most powerful parallel computers based on actual Top500 list. Information about the strategies used in each case where mainly retrieved from publicly available information sources like the web. In addition many sites were contacted directly and asked to provide further information.

Those parallel computers can be classified into 3 groups:

Parallel Vector Processors There are only 4 entries in this class: the NEC's Earth-Simulator, which is the leader of the Top500 list, and 3 installations of a Cray X1.

Parallel Processors Almost 40% of the considered computers are true parallel processors. All but 4 of which are not IBM SP Power3 or IBM pSeries 690.

Clusters There is a larger variety of types for clusters although Xeon clusters clearly dominate with more than 50% of all cluster installation among the considered computer systems.

Parallel Vector Processors The Cray X1 installations all use the same scheduling system consisting of *PBS Pro* in combination with Cray's *psched* placement scheduler. PBS is used for workload management. This means that it controls the allocation of resources to different users and groups, the performs accounting functions [27]. Psched, originally developed for the Cray T3E, includes a load balancer and a gang scheduler [31]. It monitors the actual usage of the system's nodes, and passes the information to PBS to allow PBS to decide which job should run. Psched is then responsible for the actual placement of this job in the system, i.e. the allocations of specific nodes.

Scheduling is different for the Earth Simulator which is currently the most powerful parallel processor according to the Top500 list. The system uses a queue for small batch requests (S-queue) and a queue for large batch requests (L-queue) [54]^{03}. For the S-queue, ERS-II is used as a scheduling system. Although ERS-II supports gang scheduling this feature is not used for the S-queue. The L-queue has a customized scheduler which does not support gang scheduling. Further, the Earth Simulator scheduling systems support backfilling and checkpointing.

Parallel Processors Most IBM systems use the LoadLeveler scheduler, which supports backfilling. Although LoadLeveler also allows job prioritization, this is not mentioned as a feature in the description of most installations. As most direct replies confirmed job prioritization, we may assume that it is actually used in most systems but nor explicitly mentioned. At least the newer versions of LoadLeveler also support gang scheduling which is also not found in most descriptions. However, at least the Max-Plank-Society in Germany explicitly states that gang scheduling is possible but not used. This shows that at least some installations have decided against gang scheduling.

The Lawrence Livermore National Labs have developed a home grown resource management system called LCRM (Livermore Computing Resource Management System) that supports backfilling, reservation, preemption, and gang scheduling. This system is used for the ASCI White installation and for cluster installations at Lawrence Livermore National Labs. The ASCI White system has batch partition and an interactive partition but uses only a single queue with 3 classes of jobs (*expedited*, *normal* and *stand-by*). However, it does not currently use the preemption feature. The utilization is between 80% and 90%.

Reservation is also used in the installation at ECMWF (European Centre for Medium-Range Weather Forecast) [28]^{04}. Here, LoadLeveler is enhanced by a special job filter. The system separates serial and parallel jobs by assigning them to different classes (2 classes for serial jobs and 3 classes for parallel jobs). The utilization of this system is between 94% and 97.5%. A similar utilization is achieved on the above mentioned parallel processor of the Max-Plank-Society with a more elaborate scheme of job queues.

We were not able to obtain much information on non-IBM parallel processors except that gang scheduling is supported by the ASCI Red system consisting of Intel Xeon processors and using the Paragon operating system.

Clusters Various commercial resource management systems can be found in cluster installations, including various form of PBS [27] and LSF [60]. They are frequently combined with the Maui scheduler [30]^{01}. As already mentioned Lawrence Livermore National Labs use LCRM also for their clusters. In many Linux clusters SLURM (Simple Linux Utility for Resource Management) is especially used for low priority jobs [57]^{03}. The Pittsburgh Supercomputing Center has developed a custom scheduler called Simon on top of OpenPBS in order to support a variety of advanced scheduling features like advance reservation, backfilling, and checkpointing.

In general, it can be stated that the scheduler of most cluster installations support backfilling and job prioritization. Gang scheduling, preemption, advance reservations and checkpointing are more frequently found than in parallel processor installations. In most installations, almost all computing nodes are in a single partition. There are few exceptions. For instance the Pacific Northwest National Lab has additional partitions for management and user log-in nodes (4 nodes) as well as for the Lustre file system nodes (34). However, these partitions are relatively small in comparison to the total number of nodes in the compute partition (940). The cluster at Los Alamos National Labs also has file serving nodes that allow interactive access via LSF.

Los Alamos National Lab also uses more queues (8-9 active queues and 4-5 special purpose queues) than other installations.In addition queues can be specifically set up for a project. In other clusters users can submit their hobs to at most 3 different queues.

The utilization of the systems depends on the applications and ranges from approximately 55% in 2003 (Los Alamos National Lab) to 95% for the last 30 days (Pittsburgh Supercomputing Center).

4 Conclusions

Parallel job scheduling has been useful for parallel processors. Recently, the strategies and algorithms have been adapted to the grid and to clusters. In the future, microprocessors are likely to contain several processors on a chip with the numbers of processors per chip rapidly increasing over the years. Servers and even personal computers will be parallel processors. These may then be organized into clusters, grids, or tightly coupled microprocessors. To a degree, this is already happening with the introduction of hyperthreading in Intel microprocessors [52].

Job schedulers will have to deal with at least two layers of scheduling and even more if several processor chips are aggregated together into a server. Current research projects are actively building processors with tens or even hundreds of processors.

When used as a fast personal computer, some gang scheduling variant makes sense. When used as a supercomputer, some variant on batch scheduling will be used. This leads to the obvious convergence of both types of scheduling strategies.

To summarize, actual usage patterns in parallel job scheduling have advanced in the past decade, but largely remain within the realm of batch scheduling. Backfilling and prioritization are standard in many systems. More advanced facilities such as reservations and checkpointing are also making inroads. The alternative approach of gang scheduling is common, but not always deployed.

One outcome of this progress is the utilization is improving greatly. While in the past utilization of 50-70% were accepted as the norm [46]^{00}, now many systems report utilization in the 90% range [28]^{04}.

The competition between many different systems and designs testifies to the fact that the field of parallel job scheduling is important and vibrant. The flip side of the coin is that this may reach proportions that actually hamper progress rather than promoting it. As each large installation starts from scratch and develops its own home-grown solution, there is much duplication of effort. At the same time novel ideas are left at the wayside, as developers struggle to get new systems to perform. One may wonder at this point whether the time is not ripe for some standardization effort, that will define a basic architecture for parallel job schedulers, complete with interfaces between the major components. This would allow researchers and developers to focus on that component that they feel compelled to work on, with the assurance that the results of their labor would be usable in combination with other components from different origins.

Another issue that raises concern is that of scalability. A review of the Top500 list of supercomputer installations reveals that since 1997 the largest machines in the world have always been just smaller than 10,000 nodes. While this may reflect power and packaging limitations, it may also be interpreted as indicating that we currently do not really know how to utilize more nodes effectively [21]^{04}. This has two aspects. One is programming models that allow such large numbers of nodes to be harnessed to work in concert, while dealing effectively with the occurrence of failures. The other is management of such large numbers of nodes.

Management is problematic because workload studies on current systems indicate that many jobs run on parallel systems actually have a very limited degree of parallelism. This implies that the load on the management system may grow considerably as more nodes are added. It also conflicts with the common approach of handling larger systems simply by increasing the unit of allocation, e.g. by allocating blocks of 32 nodes rather than single nodes. Scalable and effective management solutions for large systems are therefore an important area for further research and development.

Acknowledgments The authors are thankful to Gary Skouson from Pacific Northwest National Lab, Manuel Vigil from Los Alamos National Lab, Graham Holt from the European Centre for Medium-Range Weather Forecast, and Ingeborg Weidl from the Max-Plank-Institut für Plasmaphysik, for providing them with detailed information about the scheduling methods and tools used in their respective installation.

References

[1] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[2] S. Banen, A. I. D. Bucur, and D. H. J. Epema, "*A measurement-based simulation study of processor co-allocation in multicluster systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 105–128, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[3] A. Batat and D. G. Feitelson, "*Gang scheduling with memory considerations*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 109–114, May 2000.

[4] A. I. D. Bucur and D. H. J. Epema, "*The influence of communication on the performance of co-allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 66–86, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[5] A. I. D. Bucur and D. H. J. Epema, "*The influence of the structure and sizes of jobs on the performance of co-allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 154–173, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.

[6] 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.

[7] W. Cirne and F. Berman, "*Adaptive selection of partition size for supercomputer requests*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 187–207, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.

[8] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[9] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "*Enhanced Algorithms for Multi-Site Scheduling*". In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore*, Springer–Verlag, Lecture Notes in Computer Science LNCS, 2002.

[10] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "*On Advantages of Grid Computing for Parallel Job Scheduling*". In *Proc. 2nd IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGRID2002)*, IEEE Press, Berlin, May 2002.

[11] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour, "*On Effects of Machine Configurations on Parallel Job Scheduling in Computational Grids*". In *International Conference on Architecture of Computing Systems, ARCS*, pp. 169–179, VDE, Karlsruhe, April 2002.

[12] C. Ernemann, V. Hamscher, and R. Yahyapour, "*Economic Scheduling in Grid Computing*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 128–152, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[13] C. Ernemann and R. Yahyapour, *"Grid Resource Management - State of the Art and Future Trends"*, chap. "Applying Economic Scheduling Methods to Grid Environments", pp. 491–506. Kluwer Academic Publishers, 2003.

[14] Y. Etsion and D. G. Feitelson, "*User-level communication in a system with gang scheduling*". In 15th *Intl. Parallel & Distributed Processing Symp.*, Apr 2001.

[15] D. G. Feitelson, *Experimental Analysis of the Root Causes of Performance Evaluation Results: A Backfilling Case Study.* Technical Report 2002–4, School of Computer Science and Engineering, Hebrew University, Mar 2002.

[16] D. G. Feitelson, "*Metric and workload effects on computer systems evaluation*". *Computer* **36(9)**, pp. 18–25, Sep 2003.

[17] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[18] 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.

[19] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[20] D. G. Feitelson and L. Rudolph, "*Parallel job scheduling: issues and approaches*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[21] D. G. Feitelson "*The Supercomputer Industry in Light of the Top500 Data*". In *Comput. in Science & Engineering* 7(1), pp. 42-47, 2004.

[22] I. Foster and C. Kesselman, "*The Globus toolkit*". In *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman (eds.), pp. 259–278, Morgan Kaufmann, 1999.

[23] 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.

[24] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "*Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources*". In 17th *Intl. Parallel & Distributed Processing Symp.*, Apr 2003.

[25] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll, "*STORM: lightning-fast resource management*". In *Supercomputing*, Nov 2002.

[26] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "*Evaluation of Job-Scheduling Strategies for Grid Computing*". In *Proc. 7th Int'l Conf. on High Performance Computing, HiPC-2000*, pp. 191–202, Springer, Berlin, Lecture Notes in Computer Science LNCS 1971, Bangalore, Indien, 2000.

[27] R. L. Henderson, "*Job scheduling under the portable batch system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[28] G. Holt, "*Time-Critical Scheduling on a Well Utilised HPC System Using Resource Reservations*," In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), Springer-Verlag, 2004. Lect. Notes Comput. Sci. (this volume).

[29] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.

[30] D. Jackson, Q. Snell, and M. Clement, "*Core algorithms of the Maui scheduler*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 87–102, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[31] R. Lagerstrom, and S. Gipp, "*PScheD: Political Scheduling on the CRAY T3E*," In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 117–138, Springer-Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[32] 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, 2004. Lect. Notes Comput. Sci. (this volume).

[33] 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.

[34] M. J. Litzkow, M. Livny, and M. W. Mutka, "*Condor - a hunter of idle workstations*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 104–111, Jun 1988.

[35] 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 *Supercomputing'98*, Nov 1998.

[36] R. Mraz, "*Reducing the variance of point-to-point transfers for parallel real-time programs*". *IEEE Parallel & Distributed Technology* **2(4)**, pp. 20–31, Winter 1994.

[37] 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.

[38] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[39] F. Petrini and W-c. Feng, "*Buffered coscheduling: a new methodology for multitasking parallel jobs on distributed systems*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 439–444, May 2000.

[40] F. Petrini and W-c. Feng, "*Time-sharing parallel jobs in the presence of multiple resource requirements*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 113–136, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.

[41] F. Petrini, D. J. Kerbyson, and S. Pakin, "*The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q*". In *Supercomputing*, Nov 2003.

[42] J. Pruyne and M. Livny, "*Parallel processing on dynamic resources with CARMI*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 259–278, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[43] U. Schwiegelshohn and R. Yahyapour, "*Analysis of First-Come-First-Serve Parallel Job Scheduling*". In *Proceedings of the 9^{th} SIAM Symposium on Discrete Algorithms*, pp. 629–638, January 1998.

[44] U. Schwiegelshohn and R. Yahyapour, "*Fairness in Parallel Job Scheduling*". *Journal of Scheduling, 3(5):297-320. John Wiley*, 2000.

[45] U. Schwiegelshohn and R. Yahyapour, *"Grid Resource Management - State of the Art and Future Trends"*, chap. "Attributes for Communication Between Grid Scheduling Instances", pp. 41–52. Kluwer Academic Publishers, 2003.

[46] L. Rudolph, and P. Smith, "*Valuation of Ultra-scale Computing Systems*," In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 39–55, Springer-Verlag, 2003. Lect. Notes Comput. Sci. vol. 1911.

[47] E. Shmueli and D. G. Feitelson, "*Backfilling with lookahead to optimize the performance of parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 228–251, Springer-Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[48] J. M. P. Sinaga, H. H. Mohammed, and D. H. J. Epema, "*A dynamic co-allocation service in multicluster systems*". In 10th *Job Scheduling Strategies for Parallel Processing*, Jun 2004.

[49] Q. Snell, M. Clement, D. Jackson, and C. Gregory, "*The performance impact of advance reservation meta-scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 137–153, Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.

[50] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "*Selective reservation strategies for backfill job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 55–71, Springer-Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[51] D. Talby and D. G. Feitelson, "*Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling*". In 13th *Intl. Parallel Processing Symp.*, pp. 513–517, Apr 1999.

[52] D. M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "*Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*," In 23rd *Annual International Symposium on Computer Architecture*, May, 1996.

[53] D. Tsafrir. in preparation.

[54] A. Uno, T. Aoyagi, and K. Tani, "*Job scheduling on the earth simulator*". *NEC Res. & Develop.* **44(1)**, pp. 47–52, Jan 2003.

[55] U. Schwiegelshohn and R. Yahyapour, "*GGF-GFD.6: Attributes for Communication between Scheduling Instances*". http://www.ggf.org/documents/GFD/GFD-I-6.pdf, Dec 2001.

[56] Y. Wiseman and D. G. Feitelson, "*Paired gang scheduling*". *IEEE Trans. Parallel & Distributed Syst.* **14(6)**, pp. 581–592, Jun 2003.

[57] A. B. Yoo, M. A. Jette, and M. Grondona, "*SLURM: simple Linux utility for resource management*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 44–60, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[58] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "*An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*". *IEEE Trans. Parallel & Distributed Syst.* **14(3)**, pp. 236–247, Mar 2003.

[59] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "*Improving parallel job scheduling by combining gang scheduling and backfilling techniques*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 133–142, May 2000.

[60] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "*Utopia: a load sharing facility for large, heterogeneous distributed computer systems*". *Software — Pract. & Exp.* **23(12)**, pp. 1305–1336, Dec 1993.

[61] D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *Intl. Symp. High Performance Distributed Comput.*, Aug 1999.

Scheduling on the Top 50 Machines

Carsten Ernemann, Martin Krogmann, Joachim Lepping, and
Ramin Yahyapour

Computer Engineering Institute, University Dortmund, 44221 Dortmund, Germany
{carsten.ernemann,martin.krogmann,joachim.lepping,ramin.yahyapour}@udo.edu

Abstract. The well-known TOP500 list ranks the 500 most powerful high-performance computers. However, the list lacks details about the job management and scheduling on these machines. As this statistic is interesting for researchers and system designers, this paper gives an overview and survey on scheduling relevant information for the first 50 entries in the TOP500 list.

1 Introduction

The task of scheduling computational jobs on parallel computers is subject to research for quite a long time. Despite many different approaches from theory, only a few scheduling strategies are practically in use. The actual statistics of the actual implementations are of interest to researchers, system administrators and manufacturers. The most known statistic about high-performance computers is the TOP500 list which is published every half year [2]. The list contains the 500 most powerful computers according to the LINPACK benchmark [5].

Unfortunately, the TOP500 list focuses on the benchmark result, peak performance, machine size, manufacturer and installation site. That is, there are no information about the scheduling systems that are deployed on these machines. To this end, this paper gives an survey about additional information of the top 50 machines on the TOP500 list from November 2003. The information has been collected from available web sites, publications and by querying the corresponding system administrators. The following section gives a description about the data in the list.

2 List Description

TOP500: Position in the TOP500 ranking for the November 2003 edition of the TOP500 list.

Name: Installation name from the TOP500 list.

Country and City: Location of the installation.

Year: Year of installation or last significant update.

Computer Family Model/Manufacturer: Information about the system model and the manufacturer.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 17–46, 2005.

Type: Type of the computer, e.g. parallel computer (MPP), vector computer, cluster.

Inst. Type: Classification of the application field of the installation (research, academic, industry).

Processors: Number of processors.

Op. System: Operating System of the machine.

Max. Mem./Total Mem.: Maximum available main memory on a single processing node/cummulative total memory.

R_{max}/R_{peak}**:** Maximal LINPACK performance achieved and the theoretical peak performance respectively (both in GFlops).

N_{max}/N_{half}**:** LINPACK problem size for achieving R_{max} and for achieving half of R_{max}.

Queues: Information about the existing queues in the job management system.

Scheduling: Information about the used job scheduling system and strategies.

Prioritization: shows whether priorities are assigned to users and/or jobs.

Backfilling: whether backfilling is used as a job scheduling strategy [4,3]

Reservations: whether processor allocations are reservable in advance.

Checkpointing: The local management supports the checkpointing of a job. A file of a checkpointed job is generated that allows a later continuation from that point. The checkpoint file may also be migratable to other resources, but this feature is not required.

Preemption: A job is preempted on a given processor allocation and later continued [1]. In this case the corresponding application is stopped but remains resident on the allocated processors and can be resumed later. This preemption is not synonymous with the preemption in a multitasking system that typically happens in the time range of milliseconds.

Gang Scheduling: A parallel job can be preempted and continued on a given processor allocation. The scheduling system assures that all tasks of a parallel jobs are active at the same time, so that no process of a job has to wait for communication with another process of the job which is not currently active. That is preemption is synchronized for all processes of a job; within a "gang" all processes are active at the same time. This strategy can be used to allow time-shared execution of several parallel applications within different gangs.

Partitions: Many systems use partitioning to split the existing number of processors into groups for special applications. For instance, dedicated partitions for interactive jobs or data-intensive applications.

Average Utilization: Information about the average utilization of the complete machine.

3 List

TOP500[1]**:**	1	**Name:**	Earth Simulator Center	
Country:	Japan	**City:**	Yokohama	**Year:** 2002
Computer Family Model:	Earth-Simulator	**Manufacturer:**	NEC	
Type:	Parallel vector	**Inst. Type:**	Research	
Processors:	5120	**Op. System:**	ESOS (SUPER-UX)	
Max. Mem.:	16 GB	**Total Mem.:**	10 TB	
$\mathbf{R_{max}}$[2] :	35860	$\mathbf{R_{peak}}$[3] :	40960	
$\mathbf{N_{max}}$[4] :	$1{,}0752\times10^6$	$\mathbf{N_{half}}$[5] :	266240	
Queues: • S-queue : small scale batch requests (Max 8 AP and 16 GB within 1 node) • L-queue : large scale batch requests (Max 512 nodes)				
Scheduling: • NQS-II (ERS-II : S-queue, customized scheduler : L-queue), NEC				
Prioritization:	No	**Backfill:**	Yes	
Reservations:	No	**Checkpointing:**	Yes	
Preemption:	No	**Gang Scheduling:**	No	
Partitions: • 2048 Banks				
Average Utilization: not given				

TOP500:	2	**Name:**	Los Alamos National Lab		
Country:	USA	**City:**	Los Alamos, NM	**Year:**	2002
Computer Family Model:	ASCI Q-AlphaServer SC 45, 1.25 GHz	**Manufacturer:**	HP		
Type:	Cluster	**Inst. Type:**	Research		
Processors:	8192	**Op. System:**	Tru64 Unix		
Max. Mem.:	not given	**Total Mem.:**	22 TB		
$\mathbf{R_{max}}$:	13880	$\mathbf{R_{peak}}$:	20480		
$\mathbf{N_{max}}$:	633000	$\mathbf{N_{half}}$:	225000		
Queues: • 8-9 active queues per cluster • 4-5 queues per cluster that are activated for special purposes • Queue configuration is changed according to customer input on current needs averaging once per month. • Queues maybe set up for a project with a deadline to give it on-demand access (without preemption), special debugging queues, queues that allow very long running jobs, etc.					
Scheduling: • LSF (Fair Share Scheduling)					
Prioritization:	Yes	**Backfill:**	Yes		
Reservations:	Yes	**Checkpointing:**	Yes		
Preemption:	Yes	**Gang Scheduling:**	Yes		
Partitions: • No login nodes in the Unix/RMS sense. • All access is through LSF scheduled/controlled jobs. • 128 nodes on each cluster are file serving nodes and permit the interactive login to one or two whole nodes via a LSF interactive job. • This provides immediate access for ”login jobs” since there are adequate resources for our typical interactive development workload. These nodes are not normally used for large parallel jobs. • All queues support LSF interactive access up to the maximum size allowed by the queue. • User can schedule up to 384 whole nodes (1356 processors) interactively via an LSF job using the large queue.					
Average Utilization: For 2003 the utilization was approximately 55% on 8192 processors or 2048 nodes.					
Information from: Manuel Vigil, Los Alamos, NM email: mbv@lanl.gov					

TOP500:	3	**Name:**	Virginia Tech		
Country:	USA	**City:**	Falls Church, VA	**Year:**	2003
Computer Family Model:	1100 Dual 2.0 GHz Apple G5, Mellanox Infiniband 4X	**Manufacturer:**	Self-made		
Type:	Cluster	**Inst. Type:**	Academic		
Processors:	2200	**Op. System:**	Mac OS X		
Max. Mem.:	4 GB	**Total Mem.:**	4,4 TB		
$\mathbf{R_{max}}$:	10280	$\mathbf{R_{peak}}$:	17600		
$\mathbf{N_{max}}$:	520000	$\mathbf{N_{half}}$:	152000		
Queues: not given					
Scheduling: • Deja vu					
Prioritization:	No	**Backfill:**	No		
Reservations:	No	**Checkpointing:**	Yes		
Preemption:	No	**Gang Scheduling:**	No		
Partitions: not given					
Average Utilization: not given					

TOP500:	4	**Name:**	NCSA		
Country:	USA	**City:**	Champaign, IL	**Year:**	2003
Computer Family Model:	PowerEdge 1750, P4 Xeon 3.06 GHz, Myrinet	**Manufacturer:**	Dell		
Type:	Cluster	**Inst. Type:**	Academic		
Processors:	2500	**Op. System:**	Linux (Red Hat 9.0)		
Max. Mem.:	3 GB	**Total Mem.:**	3,75 TB		
$\mathbf{R_{max}}$:	9819	$\mathbf{R_{peak}}$:	15300		
$\mathbf{N_{max}}$:	630000	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: • Maui Scheduler					
Prioritization:	Yes	**Backfill:**	Yes		
Reservations:	No	**Checkpointing:**	No		
Preemption:	Yes	**Gang Scheduling:**	No		
Partitions: not given					
Average Utilization: not given					

TOP500:	5	**Name:**	Pacific Northwest National Lab		
Country:	USA	**City:**	Richland, WA	**Year:**	2003
Computer Family Model:	Integrity rx2600 Itanium2 1.5 GHz, Quadics	**Manufacturer:**	HP		
Type:	Cluster	**Inst. Type:**	Research		
Processors:	1956	**Op. System:**	Linux (Red Hat 7.2)		
Max. Mem.:	not given	**Total Mem.:**	6,8 TB		
$\mathbf{R_{max}}$:	8633	$\mathbf{R_{peak}}$:	11616		
$\mathbf{N_{max}}$:	835000	$\mathbf{N_{half}}$:	140000		
Queues: • three main queues for normal user jobs • A large job queue that has a slightly higher priority and only runs jobs requiring 256 CPU's. • A short queue for jobs of 8 CPU's or less and less than 30 minutes of run time and a normal queue of other user jobs. • All of these jobs will backfill if possible. • In addition to these we have some other queues for testing system issues and for running special jobs that we need to tend. • Also we have the SLURM queue for other extremely low priority jobs that we can kill when we need the node for a "real" job.					
Scheduling: • LSF as a scheduler on top of the Quadrics RMS resource management system. • SLURM resource manager for some of the lowest priority, preemptable backfill, jobs. • SLURM jobs to backfill also but preempt them when LSF jobs are scheduled to run.					
Prioritization:	Yes	**Backfill:**	Yes		
Reservations:	Yes	**Checkpointing:**	No		
Preemption:	Yes	**Gang Scheduling:**	Yes		
Partitions: • Partition for the user login nodes and the management nodes (4 nodes). • Partition for the Lustre filesystem nodes (34 nodes). • The remaining nodes are in a single partition (940 nodes). • These nodes consist of "Fat" nodes (8 GB memory and 400 GB local scratch disk at 200MB/s). • "Thin" nodes (6 GB memory, 12 GB local scratch disk)					
Average Utilization: We average over 95% node utilization for the last 30 days.					
Information from: Gary B. Skouson email: Gary.Skouson@pnl.gov					

TOP500:	6	**Name:**	Los Alamos National Lab		
Country:	USA	**City:**	Los Alamos, NM	**Year:**	2003
Computer Family Model:	Opteron 2 GHz, Myrinet	**Manufacturer:**	Linux Networx		
Type:	Cluster	**Inst. Type:**	Research		
Processors:	2816	**Op. System:**	Linux (Red Hat)		
Max. Mem.:	not given	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	8051	$\mathbf{R_{peak}}$:	11264		
$\mathbf{N_{max}}$:	761160	$\mathbf{N_{half}}$:	109208		
Queues: not given					
Scheduling: • LCRM • SLURM • Fair Share with Half-Life					
Prioritization:	Yes	**Backfill:**	No		
Reservations:	No	**Checkpointing:**	No		
Preemption:	Yes	**Gang Scheduling:**	Yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	7	**Name:**	Lawrence Livermore National Lab		
Country:	USA	**City:**	Livermore, CA	**Year:**	2002
Computer Family Model:	MCR Linux Cluster Xeon 2.4 GHz, Quadrics	**Manufacturer:**	Linux Networx		
Type:	Cluster	**Inst. Type:**	Research		
Processors:	2304	**Op. System:**	Chaos 1.2 (modified Red Hat 7.3)		
Max. Mem.:	4 GB	**Total Mem.:**	4,5 TB		
$\mathbf{R_{max}}$:	7634	$\mathbf{R_{peak}}$:	11060		
$\mathbf{N_{max}}$:	350000	$\mathbf{N_{half}}$:	75000		
Queues: not given					
Scheduling: • LCRM • SLURM • Fair Share with Half-Life					
Prioritization:	Yes	**Backfill:**	No		
Reservations:	No	**Checkpointing:**	No		
Preemption:	Yes	**Gang Scheduling:**	Yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	8	**Name:**	Lawrence Livermore National Lab		
Country:	USA	**City:**	Livermore, CA	**Year:**	2000
Computer Family Model:	ASCI White, SP Power3 375 Mhz	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	8192	**Op. System:**	AIX		
Max. Mem.:	16 GB	**Total Mem.:**	8 TB		
$\mathbf{R_{max}}$:	7304	$\mathbf{R_{peak}}$:	12288		
$\mathbf{N_{max}}$:	640000	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: • DPCS • LoadLeveler • GangLL					
Prioritization:	Yes	**Backfill:**	No		
Reservations:	No	**Checkpointing:**	Yes		
Preemption:	Yes	**Gang Scheduling:**	Yes		
Partitions: • Debug Partition • Batch Partition					
Average Utilization: not given					

TOP500:	9	**Name:**	NERSC/LBNL		
Country:	USA	**City:**	Berkeley, CA	**Year:**	2002
Computer Family Model:	SP Power3 375 Mhz 16way	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	6656	**Op. System:**	AIX		
Max. Mem.:	16 GB - 64 GB	**Total Mem.:**	7 TB		
$\mathbf{R_{max}}$:	7304	$\mathbf{R_{peak}}$:	9984		
$\mathbf{N_{max}}$:	640000	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: • LoadLeveler					
Prioritization:	No	**Backfill:**	Yes		
Reservations:	No	**Checkpointing:**	No		
Preemption:	No	**Gang Scheduling:**	Yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	10	**Name:**	Lawrence Livermore National Lab	
Country:	USA	**City:**	Livermore, CA	**Year:** 2003
Computer Family Model:	xSeries Cluster Xeon 2.4 GHz, Quadrics	**Manufacturer:**	IBM/ Quadrics	
Type:	Cluster	**Inst. Type:**	Research	
Processors:	1920	**Op. System:**	not given	
Max. Mem.:	4 GB	**Total Mem.:**	3,75 TB	
$\mathbf{R_{max}}$:	6586	$\mathbf{R_{peak}}$:	9216	
$\mathbf{N_{max}}$:	425000	$\mathbf{N_{half}}$:	90000	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	11	**Name:**	National Aerospace Lab of Japan	
Country:	Japan	**City:**	Tokyo	**Year:** 2002
Computer Family Model:	PRIMEPOWER HPC2500 1.3 GHz	**Manufacturer:**	Fujitsu	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	2304	**Op. System:**	not given	
Max. Mem.:	not given	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	5406	$\mathbf{R_{peak}}$:	11980	
$\mathbf{N_{max}}$:	658800	$\mathbf{N_{half}}$:	100080	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	12	**Name:**	Pittsburgh Supercomputing Center
Country:	USA	**City:**	Pittsburgh, PA **Year:** 2001
Computer Family Model:	AlphaServer SC45, 1GHz	**Manufacturer:**	HP
Type:	Cluster	**Inst. Type:**	Academic
Processors:	3016	**Op. System:**	Tru64 UNIX
Max. Mem.:	32 GB	**Total Mem.:**	3 TB
$\mathbf{R_{max}}$:	4463	$\mathbf{R_{peak}}$:	6032
$\mathbf{N_{max}}$:	280000	$\mathbf{N_{half}}$:	85000
Queues: • one large job queue (>= 256 nodes (>= 1024 cpus)) • one smaller job queue (< 256 nodes (< 1024 cpus))			
Scheduling: • OpenPBS with the custom scheduler Simon (written in TCL). • Simon features advance reservations, backfilling, and co-scheduling special purpose visualization nodes. • Supports various job prioritizations based on job size and queue priority to accommodate the user base and desired workload mix.			
Prioritization:	Yes	**Backfill:**	Yes
Reservations:	Yes	**Checkpointing:**	Yes
Preemption:	No	**Gang Scheduling:**	No
Partitions: • One partition to which jobs are scheduled. • 1 node (an SMP) is comprised of 4 cpus and 4 GB of memory. • Scheduling at the node level so that no nodes are shared.			
Average Utilization: • Typical utilization runs about 90%. • Allocating nodes is done by using a reserved resource model. That is, once a node has been allocated to a job, it's up to the user to decide how to use the resources of the node or nodes assigned as they are assigned exclusively to the user. • Billing and measuring utilization is based on the number of nodes allocated to jobs.			
Information from: Chad Vizino email: vizino@psc.edu			

TOP500:	13	**Name:**	NCAR		
Country:	USA	**City:**	Boulder, CO	**Year:**	2003
Computer Family Model:	pSeries 690 Turbo 1.3 GHz		**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Research		
Processors:	1600	**Op. System:**	AIX		
Max. Mem.:	2 GB		**Total Mem.:**	3 TB	
$\mathbf{R_{max}}$:	4184		$\mathbf{R_{peak}}$:	8320	
$\mathbf{N_{max}}$:	550000		$\mathbf{N_{half}}$:	93000	
Queues: 27					
Scheduling: • LoadLeveler					
Prioritization:	No		**Backfill:**	Yes	
Reservations:	No		**Checkpointing:**	No	
Preemption:	No		**Gang Scheduling:**	No	
Partitions: not given					
Average Utilization: not given					

TOP500:	14	**Name:**	Cinese Academy of Science		
Country:	China	**City:**	Beijing	**Year:**	2003
Computer Family Model:	DeepComp 6800, Itanium2 1.3 GHz, QsNet		**Manufacturer:**	Legend	
Type:	Cluster	**Inst. Type:**	Academic		
Processors:	1024	**Op. System:**	not given		
Max. Mem.:	not given		**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	4183		$\mathbf{R_{peak}}$:	5324,8	
$\mathbf{N_{max}}$:	491488		$\mathbf{N_{half}}$:	not given	
Queues: not given					
Scheduling: not given					
Prioritization:	not given		**Backfill:**	not given	
Reservations:	not given		**Checkpointing:**	not given	
Preemption:	not given		**Gang Scheduling:**	not given	
Partitions: 2 • Climate Simulation Laboratory jobs • Community Computing Jobs					
Average Utilization: not given					

TOP500:	15	**Name:**	Comm. a l'Energie Atomique		
Country:	France	**City:**	St.-Paul-lez-Durance	**Year:**	2001
Computer Family Model:	AlphaServer SC45, 1GHz		**Manufacturer:**	HP	
Type:	Cluster	**Inst. Type:**	Research		
Processors:	2560	**Op. System:**	Tru64 UNIX 5.1a		
Max. Mem.:	not given		**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	3980		$\mathbf{R_{peak}}$:	5120	
$\mathbf{N_{max}}$:	360000		$\mathbf{N_{half}}$:	85000	
Queues: • LSF batch management system					
Scheduling: not given					
Prioritization:	not given		**Backfill:**	not given	
Reservations:	not given		**Checkpointing:**	not given	
Preemption:	not given		**Gang Scheduling:**	not given	
Partitions: not given					
Average Utilization: not given					

TOP500:	16	**Name:**	HPCx		
Country:	UK	**City:**	Edinburgh	**Year:**	2002
Computer Family Model:	pSeries 690 Turbo 1.3 GHz		**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Academic		
Processors:	1280	**Op. System:**	AIX		
Max. Mem.:	1 GB		**Total Mem.:**	1,2 TB	
$\mathbf{R_{max}}$:	3406		$\mathbf{R_{peak}}$:	6656	
$\mathbf{N_{max}}$:	317000		$\mathbf{N_{half}}$:	not given	
Queues: not given					
Scheduling: • LoadLeveler					
Prioritization:	no		**Backfill:**	yes	
Reservations:	no		**Checkpointing:**	no	
Preemption:	no		**Gang Scheduling:**	no	
Partitions: not given					
Average Utilization: not given					

TOP500:	17	**Name:**	Forecast Systems Laboratory		
Country:	USA	**City:**	Washington, DC	**Year:** 2002	
Computer Family Model:	Aspen Systems, Dual Xeon 2.2 GHz,Myrinet2000		**Manufacturer:**	HPTi	
Type:	Cluster	**Inst. Type:**	Research		
Processors:	1536	**Op. System:**	Linux (Red Hat 6)		
Max. Mem.:	1 GB		**Total Mem.:**	0,75 TB	
$\mathbf{R_{max}}$:	3337		$\mathbf{R_{peak}}$:	6758	
$\mathbf{N_{max}}$:	285000		$\mathbf{N_{half}}$:	75000	
Queues: not given					
Scheduling: • PBS Pro					
Prioritization:	no		**Backfill:**	yes	
Reservations:	no		**Checkpointing:**	no	
Preemption:	no		**Gang Scheduling:**	no	
Partitions: not given					
Average Utilization: not given					

TOP500:	18	**Name:**	Naval Oceanographic Office		
Country:	USA	**City:**	Stennis SC, MS	**Year:** 2002	
Computer Family Model:	pSeries 690 Turbo 1.3 GHz		**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research		
Processors:	1184	**Op. System:**	AIX 5.1		
Max. Mem.:	8 GB-64 GB		**Total Mem.:**	1,4 TB	
$\mathbf{R_{max}}$:	3160		$\mathbf{R_{peak}}$:	6156,8	
$\mathbf{N_{max}}$:	not given		$\mathbf{N_{half}}$:	not given	
Queues: 7 • batch • priority • bigmem • share • transfer • debug • background					
Scheduling: • LoadLeveler					
Prioritization:	no		**Backfill:**	yes	
Reservations:	no		**Checkpointing:**	no	
Preemption:	no		**Gang Scheduling:**	no	
Partitions: not given					
Average Utilization: not given					

TOP500:	19	**Name:**	Government		
Country:	USA	**City:**	not given	**Year:**	2003
Computer Family Model:	Cray X1	**Manufacturer:**	Cray Inc.		
Type:	Parallel vector	**Inst. Type:**	not given		
Processors:	252	**Op. System:**	UNICOS/mp		
Max. Mem.:	not given	**Total Mem.:**	5 TB		
$\mathbf{R_{max}}$:	2932,9	$\mathbf{R_{peak}}$:	3225,6		
$\mathbf{N_{max}}$:	338688	$\mathbf{N_{half}}$:	44288		
Queues: not given					
Scheduling: • PBS Pro • Load Balancer • Gang Scheduler					
Prioritization:	no	**Backfill:**	no		
Reservations:	no	**Checkpointing:**	no		
Preemption:	no	**Gang Scheduling:**	yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	20	**Name:**	Oak Ridge National Laboratory		
Country:	USA	**City:**	Oak Ridge, TN	**Year:**	2003
Computer Family Model:	Cray X1	**Manufacturer:**	Cray Inc.		
Type:	Parallel vector	**Inst. Type:**	Research		
Processors:	252	**Op. System:**	UNICOS/mp		
Max. Mem.:	not given	**Total Mem.:**	5 TB		
$\mathbf{R_{max}}$:	2932,9	$\mathbf{R_{peak}}$:	3225,6		
$\mathbf{N_{max}}$:	338688	$\mathbf{N_{half}}$:	44288		
Queues: not given					
Scheduling: • PBS Pro • Load Balancer • Gang Scheduler					
Prioritization:	no	**Backfill:**	no		
Reservations:	no	**Checkpointing:**	no		
Preemption:	no	**Gang Scheduling:**	yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	21	**Name:**	Cray Inc.	
Country:	USA	**City:**	Seattle, WA	**Year:** 2003
Computer Family Model:	Cray X1	**Manufacturer:**	Cray Inc.	
Type:	Parallel vector	**Inst. Type:**	Vendor	
Processors:	252	**Op. System:**	UNICOS/mp	
Max. Mem.:	not given	**Total Mem.:**	5 TB	
$\mathbf{R}_{\mathbf{max}}$:	2932,9	$\mathbf{R}_{\mathbf{peak}}$:	3225,6	
$\mathbf{N}_{\mathbf{max}}$:	338688	$\mathbf{N}_{\mathbf{half}}$:	44288	
Queues: not given				
Scheduling: • PBS Pro • Load Balancer • Gang Scheduler				
Prioritization:	no	**Backfill:**	no	
Reservations:	no	**Checkpointing:**	no	
Preemption:	no	**Gang Scheduling:**	yes	
Partitions: not given				
Average Utilization: not given				

TOP500:	22	**Name:**	Korea Institute of Science	
Country:	Korea	**City:**	Seoul	**Year:** 2003
Computer Family Model:	eServer Cluster 1350 xSeries Xeon 2.4 GHz, Myrinet	**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Research	
Processors:	1024	**Op. System:**	Linux (Red Hat 7.3)	
Max. Mem.:	not given	**Total Mem.:**	1024 GB	
$\mathbf{R}_{\mathbf{max}}$:	3067	$\mathbf{R}_{\mathbf{peak}}$:	4915,2	
$\mathbf{N}_{\mathbf{max}}$:	300000	$\mathbf{N}_{\mathbf{half}}$:	not given	
Queues: not given				
Scheduling: • PBS Pro • Maui Scheduler				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: no partitions				
Average Utilization: not given				

TOP500:	23	**Name:**	ECMWF		
Country:	UK	**City:**	Reading	**Year:**	2002
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	960	**Op. System:**	AIX		
Max. Mem.:	8 GB	**Total Mem.:**	2,7 TB		
$\mathbf{R_{max}}$:	2560	$\mathbf{R_{peak}}$:	4992		
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given		
Queues: 5 classes • classes os and ns in the 3 LPAR for serial jobs • classes op, debug and np in the 116 LPAR for parallel jobs.					
Scheduling: • The standard IBM LL backfill scheduling scheme aided by own combined job-filter • runtime history files that ensures most job are given an accurate wall_clock_limit plus a base-time of 24 hours.					
Prioritization:	yes	**Backfill:**	yes		
Reservations:	yes	**Checkpointing:**	no		
Preemption:	no	**Gang Scheduling:**	no		
Partitions: • Each system has 30 × p690 compute frames and 2 × Nighthawk I/O frames. • The 30 × p690 frames are subdivided. • 4 LPAR/frame, so 120 compute LPAR in total, each with 8 CPU so in total 960 CPUs. • 2 memory types in the 30 × p690 frames. • 27 frames have 32 GB memory and 3 frames 128 GB memory.					
Average Utilization: between 94% and 97.5%					
Information from: Graham Holt Technical Group Leader HPCF Scheduling Specialist ECMWF, Shinfield Park, Reading, Berkshire RG2 9AX, UK email: graham.holt@ecmwf.int					

TOP500:	26	**Name:**	Texas Advanced Computing Center		
Country:	USA	**City:**	Austin, Texas	**Year:**	2003
Computer Family Model:	PowerEdge 1750, Pentium4 Xeon 3.06 GHz, Myrinet	**Manufacturer:**	Dell-Cray		
Type:	Cluster	**Inst. Type:**	Academic		
Processors:	600	**Op. System:**	Linux		
Max. Mem.:	not given	**Total Mem.:**	0,6 TB		
R_{max} :	2455	**R_{peak} :**	3672		
N_{max} :	252000	**N_{half} :**	not given		
Queues: not given					
Scheduling: • Job Mix Scheduler					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	27	**Name:**	Sandia National Laboratory		
Country:	USA	**City:**	Livermore, CA	**Year:**	1999
Computer Family Model:	ASCI Red, Pentium II Xeon	**Manufacturer:**	Intel		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	9632	**Op. System:**	Paragon OS		
Max. Mem.:	256 MB/ 512 MB	**Total Mem.:**	1,2 TB		
R_{max} :	2379	**R_{peak} :**	3207		
N_{max} :	362880	**N_{half} :**	75400		
Queues: not given					
Scheduling: • Gang Scheduler					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	yes		
Partitions: not given					
Average Utilization: not given					

TOP500:	28	**Name:**	Oak Ridge National Laboratory		
Country:	USA	**City:**	Oak Ridge, TN	**Year:**	2002
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	864	**Op. System:**	AIX		
Max. Mem.:	8 GB	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	2310	$\mathbf{R_{peak}}$:	4492,8		
$\mathbf{N_{max}}$:	275000	$\mathbf{N_{half}}$:	62000		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	29	**Name:**	IBM		
Country:	Canada	**City:**	Markham, Ontario	**Year:**	2003
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Vendor		
Processors:	864	**Op. System:**	AIX		
Max. Mem.:	8 GB	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	2310	$\mathbf{R_{peak}}$:	4492,8		
$\mathbf{N_{max}}$:	275000	$\mathbf{N_{half}}$:	62000		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	30	**Name:**	Louisiana State University	
Country:	USA	**City:**	Baton Rouge, LA	**Year:** 2002
Computer Family Model:	P4 Xeon 1.8 GHz Myrinet	**Manufacturer:**	Atipa	
Type:	Cluster	**Inst. Type:**	Academic	
Processors:	1024	**Op. System:**	Linux (Red Hat 7.2)	
Max. Mem.:	2 GB	**Total Mem.:**	1 TB	
$\mathbf{R}_{\mathbf{max}}$:	2207	$\mathbf{R}_{\mathbf{peak}}$:	3686,4	
$\mathbf{N}_{\mathbf{max}}$:	280000	$\mathbf{N}_{\mathbf{half}}$:	56000	
Queues: not given				
Scheduling: • PBS Pro				
Prioritization:	no	**Backfill:**	yes	
Reservations:	no	**Checkpointing:**	no	
Preemption:	no	**Gang Scheduling:**	no	
Partitions: not given				
Average Utilization: not given				

TOP500:	31	**Name:**	Max-Planck-Gesellschaft MPI/IPP	
Country:	Germany	**City:**	Garching	**Year:** 2003
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	832	**Op. System:**	AIX	
Max. Mem.:	21 × 64 GB + 2 × 96 GB + 2 × 256 GB	**Total Mem.:**	2 TB	
$\mathbf{R}_{\mathbf{max}}$:	2198,4	$\mathbf{R}_{\mathbf{peak}}$:	4326,4	
$\mathbf{N}_{\mathbf{max}}$:	not given	$\mathbf{N}_{\mathbf{half}}$:	not given	
Queues: • 12 queues with different number of nodes (processors) and different runtimes. • One special queue for the two ”fat” nodes with 256 GB main memory each.				
Scheduling: • IBM Loadleveler				
Prioritization:	yes	**Backfill:**	yes	
Reservations:	no	**Checkpointing:**	no	
Preemption:	no	**Gang Scheduling:**	not in use	
Partitions: • 25 compute (batch) nodes and 2 I/O nodes				
Average Utilization: 93% on 25 compute nodes				
Information from: Dr. Ingeborg Weidl, Max-Planck-Gesellschaft, D-85748 Garching email: weidl@rzg.mpg.de				

TOP500:	32	**Name:**	NASA	
Country:	USA	**City:**	Greenbelt, MD	**Year:** 2002
Computer Family Model:	AlphaServer SC45, 1GHz	**Manufacturer:**	HP	
Type:	Cluster	**Inst. Type:**	Research	
Processors:	1392	**Op. System:**	Tru64 UNIX 5.1a	
Max. Mem.:	not given	**Total Mem.:**	0,6 TB	
$\mathbf{R_{max}}$:	2164	$\mathbf{R_{peak}}$:	2784	
$\mathbf{N_{max}}$:	320000	$\mathbf{N_{half}}$:	40000	
Queues: • LSF batch management system				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	33	**Name:**	Lawrence Livermore National Lab	
Country:	USA	**City:**	Livermore, CA	**Year:** 1999
Computer Family Model:	ASCI Blue-Pacific SST, IBM SP 604e	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	5808	**Op. System:**	AIX 5	
Max. Mem.:	1,5-2,5 GB (432 nodes with 2,5 GB)	**Total Mem.:**	1,9 TB	
$\mathbf{R_{max}}$:	2144	$\mathbf{R_{peak}}$:	3856,5	
$\mathbf{N_{max}}$:	431344	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: • Parallel Op. System (POE)				
Prioritization:	no	**Backfill:**	no	
Reservations:	no	**Checkpointing:**	no	
Preemption:	no	**Gang Scheduling:**	yes	
Partitions: • 976 4-CPU SMP nodes consisting of 2 × 488-node sectors, S and K • 4 Login Nodes				
Average Utilization: not given				

TOP500:	34	**Name:**	US Army Research Laboratory	
Country:	USA	**City:**	Adelphi, MD	**Year:** 2002
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	800	**Op. System:**	AIX 5	
Max. Mem.:	8 GB	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	2140	$\mathbf{R_{peak}}$:	4160	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	35	**Name:**	NCSA	
Country:	USA	**City:**	Champaign, IL	**Year:** 2003
Computer Family Model:	TeraGrid, Itanium2 1.3 GHz, Myrinet	**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Academic	
Processors:	512	**Op. System:**	Suse SLES 8	
Max. Mem.:	4 GB/ 12 GB	**Total Mem.:**	2 TB	
$\mathbf{R_{max}}$:	2110	$\mathbf{R_{peak}}$:	2662,4	
$\mathbf{N_{max}}$:	308350	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: • PBS Pro • Maui Scheduler				
Prioritization:	yes	**Backfill:**	yes	
Reservations:	no	**Checkpointing:**	no	
Preemption:	yes	**Gang Scheduling:**	no	
Partitions: not given				
Average Utilization: not given				

TOP500:	36	**Name:**	Atomic Weapons Establishment	
Country:	UK	**City:**	Reading	**Year:** 2002
Computer Family Model:	SP Power3 375 Mhz 16way	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	1920	**Op. System:**	AIX	
Max. Mem.:	16 GB (2 Nodes of 64 GB)	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	2106	$\mathbf{R_{peak}}$:	2880	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: 120 nodes with 16 proccessors				
Average Utilization: not given				

TOP500:	37	**Name:**	Deutscher Wetterdienst	
Country:	Germany	**City:**	Offenbach	**Year:** 2003
Computer Family Model:	SP Power3 375 Mhz 16way	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	1920	**Op. System:**	AIX 5.1	
Max. Mem.:	not given	**Total Mem.:**	1,24 TB	
$\mathbf{R_{max}}$:	2106	$\mathbf{R_{peak}}$:	2880	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	38	**Name:**	University at Buffalo	
Country:	USA	**City:**	Buffalo, NY	**Year:** 2002
Computer Family Model:	PowerEdge 2650 Cluster	**Manufacturer:**	Dell	
	P4 Xeon 2.4 GHz - Myrinet			
Type:	Cluster	**Inst. Type:**	Academic	
Processors:	600	**Op. System:**	Linux (RedHat 7.3, 2.4 Kernel)	
Max. Mem.:	2 GB	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	2004	$\mathbf{R_{peak}}$:	2880	
$\mathbf{N_{max}}$:	253400	$\mathbf{N_{half}}$:	42200	
Queues: not given				
Scheduling: • PBS Pro • Maui Scheduler				
Prioritization:	yes	**Backfill:**	yes	
Reservations:	no	**Checkpointing:**	no	
Preemption:	yes	**Gang Scheduling:**	no	
Partitions: 258 Nodes				
Average Utilization: not given				

TOP500:	39	**Name:**	NC for Environmental Prediction		
Country:	USA	**City:**	Camp Springs, MD	**Year:**	2002
Computer Family Model:	pSeries 690 Turbo 1.3 GHz	**Manufacturer:**	IBM		
Type:	Parallel	**Inst. Type:**	Research		
Processors:	704	**Op. System:**	AIX		
Max. Mem.:	8 GB	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	1849	$\mathbf{R_{peak}}$:	3660,8		
$\mathbf{N_{max}}$:	240000	$\mathbf{N_{half}}$:	32500		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	40	**Name:**	SARA		
Country:	Netherlands	**City:**	Almere	**Year:**	2003
Computer Family Model:	SGI Altix 1.3 GHz	**Manufacturer:**	SGI		
Type:	Parallel	**Inst. Type:**	Academic		
Processors:	416	**Op. System:**	Linux (Red Hat)		
Max. Mem.:	not given	**Total Mem.:**	0,83 TB		
$\mathbf{R_{max}}$:	1793	$\mathbf{R_{peak}}$:	2163		
$\mathbf{N_{max}}$:	298799	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: 6 batch nodes / 1 interactive node					
Average Utilization: not given					

TOP500:	41	**Name:**	KISTI Supercomputing Center	
Country:	South Korea	**City:**	Daejeon City	**Year:** 2003
Computer Family Model:	pSeries 690 Turbo 1.7 GHz	**Manufacturer:**	IBM	
Type:	Parallel	**Inst. Type:**	Research	
Processors:	544	**Op. System:**	AIX	
Max. Mem.:	8 GB	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	1760	$\mathbf{R_{peak}}$:	3699,2	
$\mathbf{N_{max}}$:	400000	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	42	**Name:**	Semiconductor Company	
Country:	USA	**City:**	not given	**Year:** 2003
Computer Family Model:	xSeries Cluster Xeon 2.4 GHz, Gig-E	**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Industry	
Processors:	1834	**Op. System:**	Linux	
Max. Mem.:	not given	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	8803,2	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	43	**Name:**	WETA Digital		
Country:	New Zealand	**City:**	Wellington	**Year:**	2003
Computer Family Model:	BladeCenter Cluster Xeon 2.8 GHz, Gig-E	**Manufacturer:**	IBM		
Type:	Cluster	**Inst. Type:**	Industry		
Processors:	1176	**Op. System:**	Linux (Red Hat)		
Max. Mem.:	6 GB	**Total Mem.:**	3,4 TB		
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	6585,6		
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	44	**Name:**	Semiconductor Company		
Country:	USA	**City:**	not given	**Year:**	2003
Computer Family Model:	xSeries Cluster Xeon 2.8 GHz, Gig-E	**Manufacturer:**	IBM		
Type:	Cluster	**Inst. Type:**	Industry		
Processors:	1140	**Op. System:**	Linux		
Max. Mem.:	not given	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	6384		
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	47	**Name:**	PGS		
Country:	USA	**City:**	Houston, TX	**Year:**	2003
Computer Family Model:	xSeries Cluster Xeon 3.06 GHz, Gig-E		**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Industry		
Processors:	1024	**Op. System:**	Linux		
Max. Mem.:	not given		**Total Mem.:**	not given	
R_{max} :	1755		**R_{peak} :**	6266,88	
N_{max} :	not given		**N_{half} :**	not given	
Queues: not given					
Scheduling: not given					
Prioritization:	not given		**Backfill:**	not given	
Reservations:	not given		**Checkpointing:**	not given	
Preemption:	not given		**Gang Scheduling:**	not given	
Partitions: not given					
Average Utilization: not given					

TOP500:	48	**Name:**	WETA Digital		
Country:	New Zealand	**City:**	Wellington	**Year:**	2003
Computer Family Model:	BladeCenter Cluster Xeon 2.8 GHz, Gig-E		**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Industry		
Processors:	1080	**Op. System:**	not given		
Max. Mem.:	not given		**Total Mem.:**	not given	
R_{max} :	1755		**R_{peak} :**	6048	
N_{max} :	not given		**N_{half} :**	not given	
Queues: not given					
Scheduling: not given					
Prioritization:	not given		**Backfill:**	not given	
Reservations:	not given		**Checkpointing:**	not given	
Preemption:	not given		**Gang Scheduling:**	not given	
Partitions: not given					
Average Utilization: not given					

TOP500:	52	**Name:**	CGG	
Country:	USA	**City:**	Houston, TX	**Year:** 2003
Computer **Family Model:**	xSeries Cluster Xeon 2.4 GHz, Gig-E	**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Industry	
Processors:	1100	**Op. System:**	Linux	
Max. Mem.:	not given	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	5280	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	53	**Name:**	Arizona State University/TGEN	
Country:	USA	**City:**	Phoenix, AZ	**Year:** 2003
Computer **Family Model:**	xSeries Cluster Xeon 2.4 GHz, Gig-E	**Manufacturer:**	IBM	
Type:	Cluster	**Inst. Type:**	Academic	
Processors:	1100	**Op. System:**	Linux	
Max. Mem.:	not given	**Total Mem.:**	not given	
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	5030,4	
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given	
Queues: not given				
Scheduling: not given				
Prioritization:	not given	**Backfill:**	not given	
Reservations:	not given	**Checkpointing:**	not given	
Preemption:	not given	**Gang Scheduling:**	not given	
Partitions: not given				
Average Utilization: not given				

TOP500:	54	**Name:**	Paradigm Geophysical		
Country:	USA	**City:**	Houston, TX	**Year:**	2003
Computer Family Model:	BladeCenter Cluster Xeon 2.4 GHz, Gig-E	**Manufacturer:**	IBM		
Type:	Cluster	**Inst. Type:**	Research		
Processors:	1024	**Op. System:**	not given		
Max. Mem.:	not given	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	4915,2		
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

TOP500:	55	**Name:**	TotalFinaElf		
Country:	France	**City:**	not given	**Year:**	2003
Computer Family Model:	xSeries Cluster Xeon 2.4 GHz, Gig-E	**Manufacturer:**	IBM		
Type:	Cluster	**Inst. Type:**	Industry		
Processors:	1024	**Op. System:**	not given		
Max. Mem.:	not given	**Total Mem.:**	not given		
$\mathbf{R_{max}}$:	1755	$\mathbf{R_{peak}}$:	4915,2		
$\mathbf{N_{max}}$:	not given	$\mathbf{N_{half}}$:	not given		
Queues: not given					
Scheduling: not given					
Prioritization:	not given	**Backfill:**	not given		
Reservations:	not given	**Checkpointing:**	not given		
Preemption:	not given	**Gang Scheduling:**	not given		
Partitions: not given					
Average Utilization: not given					

4 Acknowledgements

While some of the data have been gathered from the available web pages, the authors are grateful to the different contributions from system administrations. The names are given in the tables for the corresponding entries.

References

1. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science (LNCS)*, pages 1–34. Springer, Berlin, April 1997.
2. 22nd TOP500 List introduced during the Supercomputer Conference (SC2003) in Phoenix, AZ. http://www.top500.org November 2003.
3. D.G. Feitelson and A.M. Weil. Utilization and Predictabillity in Scheduling the IBM SP2 with Backfilling. In *Proceedings of IPPS/SPDP 1998*, IEEE Computer Society, pages 542–546, 1998.
4. D. A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, Berlin, Lecture Notes in Computer Science LNCS 949, 1995.
5. A. Petitet and R.C. Whaley and J. Dongarra and A. Cleary HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers http://www.netlib.org/benchmark

Parallel Computer Workload Modeling with Markov Chains

Baiyi Song, Carsten Ernemann*, and Ramin Yahyapour

Computer Engineering Institute, University Dortmund, 44221 Dortmund, Germany
{song.baiyi, carsten.ernemann, ramin.yahyapour}@udo.edu

Abstract. In order to evaluate different scheduling strategies for parallel computers, simulations are often executed. As the scheduling quality highly depends on the workload that is served on the parallel machine, a representative workload model is required. Common approaches such as using a probability distribution model can capture the static feature of real workloads, but they do not consider the temporal relation in the traces. In this paper, a workload model is presented which uses Markov chains for modeling job parameters. In order to consider the interdependence of individual parameters without requiring large scale Markov chains, a novel method for transforming the states in different Markov chains is presented. The results show that the model yields closer results to the real workloads than other common approaches.

1 Introduction

The use of parallel computers and workstation clusters has become a common approach for solving many problems. The efficient allocation of processing nodes to jobs is the task of the scheduling system. Here, the quality of the scheduling system has a high impact on the overall performance of the parallel computer. To this end, many researchers have developed various job scheduling subsystems for such parallel computers [28, 15, 17]. As already pointed out in [16, 7], the performance of a scheduling algorithm highly depends on the workload it is applied to. There is no single scheduling algorithm that is best for all scenarios. To this end, the evaluation of scheduling algorithms for different workloads is an important step in designing a scheduling system. Therefore, much effort has been put in the characterization and modeling of the workload of parallel computers [4, 1, 6, 24].

A typical approach for the performance evaluation of a scheduling system is the application of an existing workload trace which has been recorded on an existing machine [29, 25, 13]. However, while this represents a realistic user behavior on a real machine, there are several drawbacks. For instance, such a workload trace cannot directly be applied to configurations different from the

* Carsten Ernemann is a member of the Collaborative Research Center 531, "Computational Intelligence", at the University of Dortmund with financial support of the Deutsche Forschungsgemeinschaft (DFG).

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 47–62, 2005.

original machine. In addition, the size of the workload, that is the number of jobs in the trace, cannot be scaled easily.

Therefore, often a statistical workload model is adopted as an alternative. The most common approach is the use of a probability distribution function model (PDF) [16]. However the PDF model often omits the dynamic characteristics of workloads. That is, the sequential correlation of different jobs is not taken into account. In this paper, we propose an extended job model based on Markov chains which uses information from the previous job to consider the sequential dependencies for the next job submission. After a discussion of the necessary background in Section 2, we discuss in Section 3 the relevant model parameters. In Section 4, the model is constructed. The quality of the model is evaluated by comparing its outcome with real workload data in Section 5. The paper ends with a short conclusion.

2 Background

Many parallel computers or supercomputers use a space-sharing strategy for efficient execution of parallel computational jobs. This means that a job runs exclusively on the allocated processor set. Moreover, jobs are executed until completion without any preemption. The scheduling problem is an online scenario in which the jobs are not known in advance and are continuously submitted to the scheduling systems by the users.

A workload model is an abstract description of the parameters of the jobs in the workload. A job consists of several parameters, for instance the number of required processing nodes, the job runtime, or memory requirements. In this paper, we concentrate on the modeling of the required number of nodes and the corresponding runtime. However, our approach is general and can easily be extended to consider other parameters as well. Note, that we do not model the submission time or inter-arrival time of jobs. For this task several other adequate models are available [3, 6].

As mentioned before, often a probability distribution function model is chosen for modeling workload parameters. Thereby, the parameters are typically considered independently and, consequently, individual distributions are created for each parameter. For example, Jann et al. used a hyper-Erlang distribution to match the first 3 moments of an observed distribution [20]. Alternatively, Uri Lublin and Dror Feitelson used a three-stage hyper-gamma distribution to fit the original data [24].

Besides the isolated modeling of each attribute, the correlations between different attributes are also very important. Lo et al. [23] demonstrated how the different degrees of correlation between job size and job runtime might lead to discrepant conclusions about the evaluation of scheduling performance. To consider such correlations, Jann et al. [20] divided the job sizes into subranges and then created a separate model for the inter-arrival time and the service time in each range, which may have a risk of over-fitting and too many unknown parameters. Furthermore, Lublin and Feitelson in [24] considered the runtime attribute according to a two-stage hyper-gamma distribution with a linear relation between the job size and the parameters of the runtime distribution so that the longer runtime can be emphasized by using the distribution with the higher mean.

Although the PDF models can be adapted to fit the observed original distribution, the sequential dependencies in workload is lost. For instance, in [15] Feitelson et al. showed that users tend to submit jobs which are similar to its predecessor. Therefore a more realistic model is sought which incorporates the correlation within the sequence of job submissions.

3 Analysis

The analysis of available workload traces shows several temporal relations of job parameters which are very complex. We examined seven traces which are publicly available in [30]. Each contains several thousands of job which have been submitted during a time frame of several months, as shown in Table 1. For analyzing statistical parameter of data series including temporal relations, the software named R [19] has been chosen to extract the statistical information.

Identifier	NASA	CTC	KTH	LANL	SDSC SP2	SDSC 95	SDSC 96
Machine	iPSC/860	SP2	SP2	CM-5	SP2	SP2	SP2
Period	10/01/93 12/31/93	06/26/96 05/31/97	09/23/96 08/29/97	04/10/94 09/24/96	04/28/98 04/30/00	12/29/94 12/30/95	12/27/95 12/31/96
Processors	128	430	100	1024	128	416	416
Jobs	42264	79302	28490	201378	67667	76872	38719

Table 1. Workloads used in this Research.

By analyzing the workload data, it has been found that within a short examined time frame there is only a limited variance in the number of required node and runtime that a user requests. This has also been found by [16] who considered a time frame of one week. Moreover, jobs are often identical if subsequently submitted by a user, [15]. For considering such continuous submission of identical jobs, Feitelson used a Zipf distribution to model the number of repetitions [15]. However, our examination holds for all jobs in a workload trace without differentiating the submitting user. This is probably caused by the fact that only a limited number of users is active system within a time frame. Moreover, the inter-arrival time between jobs is relatively small.

In addition, we found that not all jobs are submitted with the same continuity. For a job series J in a workload, we extract the requested number of nodes $u_j \in U$ for each job j. We examined the average continued appearance of a node requirement in this sequence U. That is, for each job node requirement in the workload trace the number of direct repetitions is considered. Note, that we consider all job submissions in a workload and not jobs submitted by the same user. As workloads contain predominantly jobs with a power of 2 number

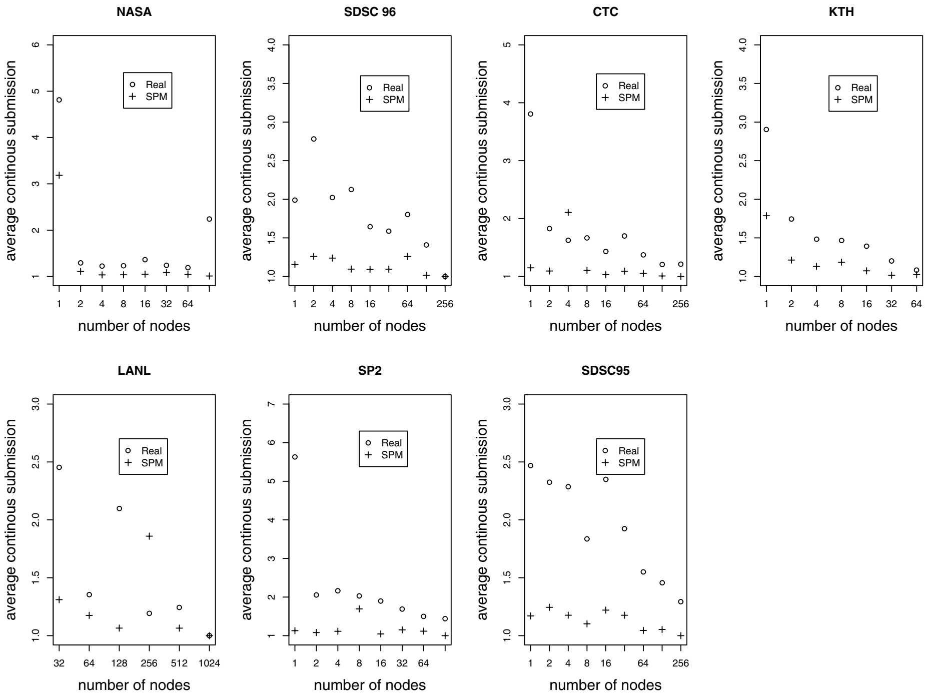


Fig. 1. Continuous Submission of Jobs with Similar Node Requirements.

of nodes, we restrict the examination on such jobs requiring 1 node, 2 nodes, 4 nodes, etc. That is, for now we neglect all jobs which do not have a power of 2 node requirement. In Figure 1 the average subsequent appearances of a job requirement in a real workload is shown. As a reference the average number of occurrences is shown if a simple probability distribution model is used for modeling the node requirements. This strategy (denoted as SPM in the Figure) does model each parameter independently according to the statistical occurrences in the original trace. It can be seen, that sequences of the same node requirement occur significantly more often in a real workload than it would be in the PDF model. This shows that a simple distribution model does not correctly represent this effect. Furthermore, jobs in the real traces with less nodes requirements have a higher probability that the subsequent node is identical than jobs requiring more nodes. That is, jobs with less parallelism have a higher probability to be repeatedly submitted.

Even if those continuously appearing elements in U are removed, sequential dependencies can be found. To this end, only one element are kept for each sequence of identical node requirements. That is we create U' from U. For example, an excerpt in a series of node requirements of 1, 1, 1, 2, 2, 5, 5, 5, 8, 16, 16, 16, 2, 2 is transformed to 1, 2, 5, 8, 16, 2. Note, here we also consider jobs

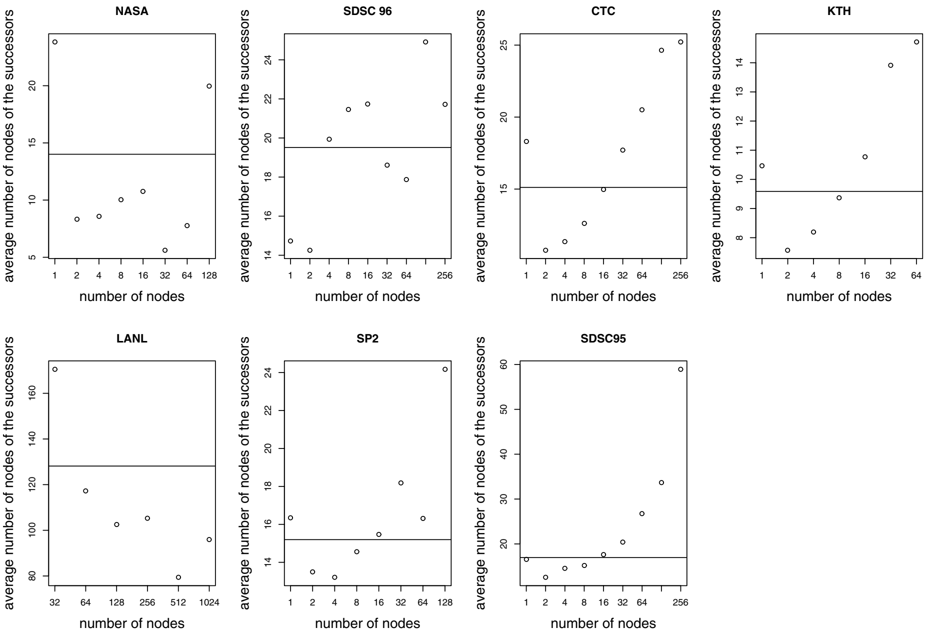


Fig. 2. Temporal Relation of Node Requirements in U''.

with node requirements that are not power of 2. In a next step we transform U' to U'' by $U'' = \{2^{\lfloor log_2(u_i') \rfloor} | u_i' \in U'\}$. That is, each node requirement is rounded to the next lower power of 2. For each distinct node requirement, we calculate the average number of nodes requested by its successor. Figure 2 shows that the successors of those jobs with a large node requirement also tend to request a large number of nodes for most workloads. That is, in most traces jobs with high node requirements are followed by jobs with also a high or even higher node requirement. The lines in the figure show the overall average for the node requirements in each workload. However, the behavior for NASA, SDSC96 and LANL is not clear. For the NASA workload it can be noted that the workload in general shows an unusual behavior as jobs are only submitted if enough free resources are available. That is, jobs start immediately after their submission. Moreover, only a small number of different node requirements occur in the traces.

There are also temporal relations in the runtime of jobs. Here, we grouped jobs by the integer part of the logarithm of their runtime and for each group the average runtime of its successors has been calculated. The result is shown in Figure 3.

Such sequential dependencies may become very important for optimizing many scheduling algorithms, like e.g. backfilling. For instance, algorithms can utilize probability information about future job arrivals. Such data can be included in heuristics about current job allocations. Therefore a method to capture

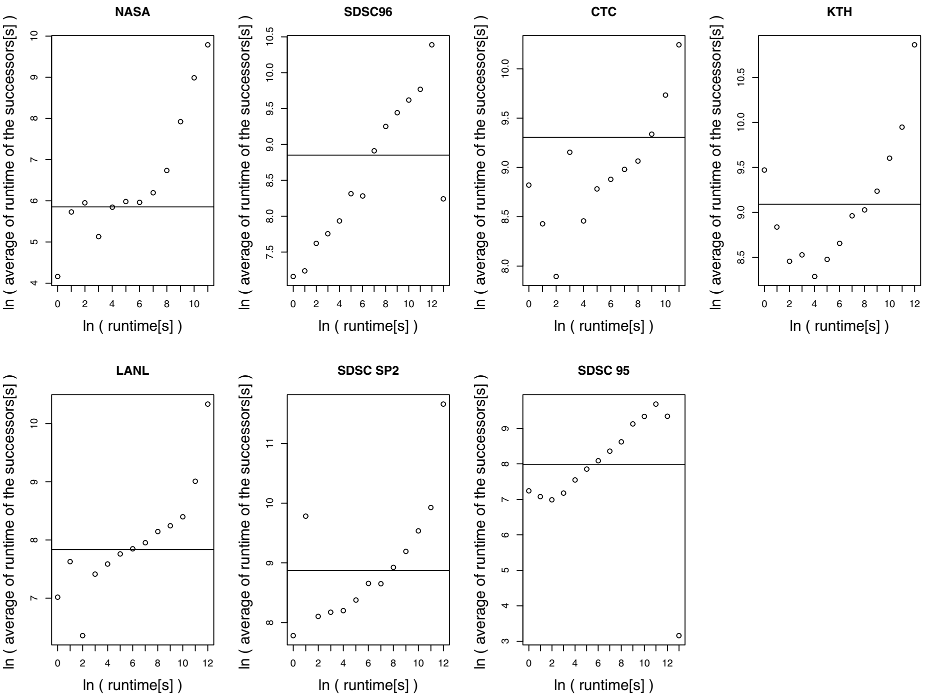


Fig. 3. Temporal Relation in the Sequence of Runtime Requirements.

the sequential dependencies in the workload would be beneficial. As shown in Figures 2 and 3, the characteristic of the relation of subsequent jobs varies for different node requirements and runtimes.

4 Modeling with Markov Chains

There are several classical methods to analyze stochastic processes. For example, the ARIMA model [2] uses lags and shifts in the historical data to uncover patterns (e.g. moving averages, seasonality) and predict the future. However the theory of ARIMA model is based on the assumption that the process is stationary, which does not hold for workloads as shown in [18]. Another common approach is the use of Neural Networks to analyze and model sequential dependencies [31, 5]. However, it is difficult to adapt and extend such a model. Instead, Markov chains [21] have been chosen for modeling the described patterns in Section 3. Those chains have the important characteristic that a transition to the next state just depends on the previous state. Therefore Markov Chains have some kind of memory and the transition probabilities to move from one state to the other within the whole model can be described by a transition matrix. The element (i, j) within the matrix describes the probability to move from state i to state j if the system was in state i.

In our workload modeling we use two Markov chains to represent the number of nodes and the runtime requirements respectively. However, as shown above it is necessary to correlate both chains. Similar requirements for combining Markov chains also occur in other application areas [8, 27, 26]. For instance, advanced speed recognition systems use so-called Hidden Markov Models (HMM) to represent not only phonemes, the smallest sound units of which words are composed, but also their combination to words. The method to correlated those different Markov models is called "embedded" HMM, in which each state in the model (super states) can represent another Markov model (embedded states). However, this method is not suitable to our problem of workload modeling, as it will dramatically increase the number of parameters. Correspondingly, the model becomes very hard to train. Therefore, in the following we propose a new algorithm to correlate two Markov chains without increasing the number of states.

As mentioned above, our model is based on two independent Markov chains. Here, the first chain is used to model the requested number of nodes. The second chain represents the runtime requirements. At the beginning, these two chains are independently constructed and then combined in order to create a model for generating the next incoming job. Note, as mentioned before, the following method does not consider the modelling of inter-arrival times. Moreover, it is intended for combination with existing submission time models.

4.1 Markov Chain Construction

First, the construction of the Markov chain for the requested number of nodes is given. The corresponding second Markov chain for runtime requirements is similarly generated.

Assume a chain of p jobs where the series of requested nodes is described by the sequence $T = \{t_1, t_2, \cdots t_p\}$. One of the key issues during the construction of a Markov chain is to identify a small set of relevant states in this series. Otherwise the Markov chain would require too many different states if all distinct node requirements in the traces would be considered.

To this end, a transformation of T is used which classifies all node requirements into power of 2 groups. Thus, the reduced sequence $S = \{s_1, s_2 \cdots s_p\}$ is constructed from T as follows:

$$s_i = 2^{\lfloor log_2 t_i \rfloor}, i \in [1, p] \tag{1}$$

Now each *distinct element* in S can be considered as a separate state within the corresponding Markov chain. The set of states L of this Markov chain can be constructed as follows:

$$L = \{l_1, l_2, \cdots l_q | q \leq p; \forall (i,j) : i \neq j \wedge i,j \in [1,q]; l_i \neq l_j\} \tag{2}$$

The different values representing the states can be calculated by:

$$(l_j = 2^{j-1} | \exists s_x \in S : s_x = 2^{j-1}, \forall j : j \leq log_2 t_{max}) \tag{3}$$

Using this transformation, the original sequence T can now be represented by using S and L. In order to consider the state changes in the original workload, we use series U to denote the order of occurrences of elements in L within the original stream of job submissions. Thus, we define $U = \{u_1, \cdots, u_p\}$ whereas the different states in U can be described by $u_i = j$ with $l_j = s_i; s_i \in S$ and $l_j \in L$ for $i \in [1, p]$. That is, the sequence U consists of the indices of the elements in L corresponding to the job sequence. As an example, consider Table 2 where the process of reducing the distinct number of node requirements in T is shown.

Index	1	2	3	4
$t_i \in T$	2	4	6	16
$s_i \in S$	2	4	4	16
$l_j \in L$	2	4	16	-
$u_i \in U$	1	2	2	3

Table 2. Example for Deriving the States of the Markov Chains.

The presented method leads to the construction of the transition matrix P for the Markov chain, where the values can be calculated as: $p_{ij} = n_{ij}/n_i$ where n_{ij} and n_i are defined as

$$n_i = |\{s_j = l_i, j \in [1, p]; l_i \in L\}| \text{ and} \tag{4}$$

$$n_{ij} = |\{j | s_o = l_i \wedge s_{o+1} = l_j, o \in [1, p-1]; i, j \in [1, q]\}| \; . \tag{5}$$

The transformations from T to S cause a loss of information about the precise number of requested nodes, as they have been reduced to power of 2 values. In addition, a *quality ratio* c_j is calculated for each state in the Markov chain. This ratio indicates how often the real value in the original group is exactly matched by the representing number of nodes in this state of the chain. More precisely, the set of quality ratios is calculated by:

$$C = \{c_j | c_j = \frac{|\{i | t_i = l_j, i \in [1, p]\}|}{|\{i | s_i = l_j, i \in [1, p]\}|}, j \in [1, q]\}. \tag{6}$$

The definition of the quality ratios c_j, $j \in [1, q]$ is needed to model the exact node number of a job. If the system is in state j, the corresponding value l_j is used as the system output with the probability of c_j. With the probability of $(1 - c_j)$ a uniform distribution between $[l_j, l_{j+1}[$ is used to create the final value for the requested number of nodes.

As mentioned earlier the same method for constructing the Markov chain can be applied to the runtime requirements. This yields a second Markov chain. The dimensions of these matrices for the considered workload representations are presented in Table 3.

	Dimension of runtime chain	Dimension of the number of nodes chains	cor_0	cor_1
NASA	16	8	0.639	0.744
SDSC 96	19	9	0.509	0.475
CTC	17	9	−0.077	−0.041
KTH	18	8	0.037	0.142
LANL	18	6	0.136	0.325
SP2	19	8	0.177	0.292
SDSC95	19	9	0.465	0.510

Table 3. Some parameters in our workload modeling.

4.2 Correlation Between Runtime and Number of Nodes

It has been found that the runtime and number of nodes have a weak positive correlation in all examined workloads, that is, the jobs requiring more nodes have a longer runtime time on average [17]. It has been shown that such a correlation has an impact on the performance of the scheduling algorithms [15]. Therefore as key feature, this correlation should be reflected in our model.

To this end, the two independent Markov chains, for node and runtime requirements, must be combined to incorporate the correlation. A straight forward approach would be the merging of the two Markov chains into a single Markov chain. However, this would yield a very high dimensional chain based on all combinations of the states in the original two chains. Such a Markov chain is very difficult to analyze and such an approach would not scale for incorporating additional other job parameters.

Our algorithm is applied after a transition in both Markov chains. It adjusts the new state in the Markov chain for node requirements according to the latest transition in the chain for the runtime. As an example, consider the observation that jobs requiring a longer runtime also tend to request a large number processing nodes. Therefore, if the Markov chain for the job runtime is in a state representing a longer runtime, the state of the Markov chain for the node requirements would also move to a state of requesting more nodes with a certain probability, an vice versa. If the runtime requirement changes dramatically in a chain, the request for nodes will have a tendency to change as well. That is, the transformation of the states in the different Markov chains incorporate the correlations between the examined parameter.

The detailed correlation algorithm for the Markov chains is given in the following:

First, the correlation value of the requested number of nodes sequence and the required runtime sequence is calculated from the original workload. In the following the transformation path for the number of nodes is denoted by N, while R represented the runtime transformation path.

The procedure of adjusting the state in the node requirement chain depends on the independent state changes in the two chains. Here, we distinguish three cases. First, if the Markov chain for the number of nodes did not change, no adjustment is applied. Second, if the state changed only in the chain for node requirement and not for runtime, then the state in the node requirement chain is adjusted based on the state in the runtime chain and the correlation factor cor_0 between the chains. Third, if states changed in both chains, the correlation of first-order differences cor_1 and the last change in the runtime chain are used to adjust the state in the node requirement chain.

The following mathematical description explains in more detail how the Markov chains are combined. The parameter n_i specifies the next node request in the sequences and the corresponding mean is denoted as $\overline{n}$. In regards to the runtime sequence similar definitions apply to r_i and $\overline{r}$. Next, the correlation cor_0 between the two sequences can be calculated as:

$$cor_0 = cor(N,R) = \frac{E((n_i - \overline{n})(r_i - \overline{r}))}{\sqrt{E((r_i - \overline{r})^2)} \cdot \sqrt{E((n_i - \overline{n})^2)}} \tag{7}$$

Here $E()$ denotes the expected value function. The index 0 was used to specify that the original sequences were used without further modifications. Furthermore, the first-order difference correlation cor_1 is used to denote how the changes of one transformation path affect the other. In order to define the first-order correlation precisely some more variables have to be introduced. Therefore, two new sets are built which consist of the changes in the transformation path.

$$\Delta N = \{\Delta n_i = n_{i+1} - n_i | \forall n_i : 0 < n_i \leq |N| - 1\} \tag{8}$$

$$\Delta R = \{\Delta r_i = r_{i+1} - r_i | \forall r_i : 0 < r_i \leq |R| - 1\} \tag{9}$$

Second, as shown in Section 3 the elements in a sequence often do not differ. As a consequence the sequence of first-order differences includes many zero values. As the modeling focuses on the changes of the system behavior all elements have to be removed were the number of nodes or the required runtime is not changing. This procedure leads to the new sets $\Delta N'$ and $\Delta R'$. These two sets can be formulated as follows:

$$\Delta N' = \{\Delta n_i | \forall n_i \in \Delta N : \Delta n_i \neq 0 \land \Delta r_i \neq 0\} \tag{10}$$

$$\Delta R' = \{\Delta r_i | \forall r_i \in \Delta R : \Delta n_i \neq 0 \land \Delta r_i \neq 0\} \tag{11}$$

Now, the correlation cor_1 can be defined as: $cor_1 = cor(\Delta N', \Delta R')$. The actual values in our examinations for cor_0 and cor_1 are also presented in Table 3.

Assume that the Markov chains for the number of nodes and the required runtime have dimensions a and b respectively. Further assume that for the synthetically generated transformation path of the requested number of nodes a connection from the state j to the state k exists as well as the connection from m to n within the synthetic transformation path of the runtime.

The above mentioned procedure can be summarized in the following three rules in order to adjust the state in the Markov chain for the requested number of nodes based on the the Markov chain of the required runtime:

1. If $j = k$, no transformation is applied. As the state in the Markov chain for the number of nodes is not changing ($j = k$), no adjustment by the Markov chain of the required runtime is needed.
2. If $j \neq k; m = n$, the destination state k is adjusted to $k = n \cdot (a/b)$ with probability of correlation cor_0. This means that the resulting number of nodes is changed with the probability of correlation cor_0 if the active state within the Markov chain for the requested number of nodes changed while the state in the runtime chain stayed constant. The factor a/b is used as a normalization between the two matrices. The value of n reflects the fact that the Markov chain of the runtime is used for the adjustment of the Markov chain for the number of nodes. Here a job with a higher runtime should also have a higher demand on the number of nodes as explained earlier.
3. If $j \neq k; m \neq n$, the destination state k is changed to $k = (n - m) \cdot (a/b) \cdot sign(cor_1) + j$ with probability of the correlation $|cor_1|$. This rule is used in situations were in both Markov chains the states are changing. Here the incremental changes can be used for the adjustment. The term $(n - m)$ describes the incremental change in the Markov chain for the runtime requirement, where the $sign(cor_1)$ indicates the direction of the change. Again, the factor of a/b is used for the necessary normalization process. As the first terms only describe the change, the originating state j is used as the basis. Similar to step 2 the changes are only applied with a certain probability. This time cor_1 is used as the calculation is based on the incremental changes.

5 Results

For our evaluation we have examined workloads from the Standard Workload Archive which are presented in Table 1. For all of those workloads the corresponding Markov chains for the requested number of nodes and the required runtime have been created. Using the presented modeling algorithm new synthetical workload traces have been created with these Markov chains. The quality of the presented modeling method is measured by comparing the original with the newly generated traces with the following statistical and temporal criteria.

5.1 Statistical Comparison

A common method of comparing sequences is the Kolmogorov-Smirnov(KS) test [22]. Here a small value indicates a high degree of similarity.

Another criteria in comparing different workloads is the *squashed area* which is the total resource consumptions of all jobs:

$$\text{squashed\_area} = \sum_{j \in \text{Jobs}} \text{req\_processors}_j \cdot \text{run\_time}_j \tag{12}$$

Furthermore, we calculate the difference of squashed area (SA) by

$$d_{\text{SA}} = \frac{\text{synthetic\_SA} - \text{original\_SA}}{\text{original\_SA}}. \tag{13}$$

	KS Test of Nodes	KS Test of Runtime	Squashed Area Difference
NASA	0.08	0.08	−16 %
SDSC 96	0.08	0.06	8 %
CTC	0.02	0.03	38 %
KTH	0.04	0.03	15 %
LANL	0.01	0.04	−1 %
SDSC SP2	0.02	0.04	8 %
SDSC 95	0.09	0.02	−3 %

Table 4. Statistical Comparison of the Modeled and the Original Workloads.

	Real Data	Markov chain Model	Lublin/Feitelson Model
SDSC 95	0.277	0.140	0.105
SDSC 96	0.371	0.155	0.116
KTH	0.011	0.005	0.005
LANL	0.172	0.226	0.29

Table 5. Correlation of Node and Runtime Requirements.

It can be seen in Table 4 that the explained Markov chains model match well the original traces. In addition, Figure 4 shows the distributions of runtime and node requirements for the KTH workload as an example. The results for squashed area as well as for the KS test are quite acceptable. Only for the CTC workload the squashed area criterion shows an inappropriate deviation in the modeled amount of workload. The squashed area or amount of total workload within a trace has significant impact on scheduling performance [13]. However, information about this criteria is usually not provided for most workload models.

5.2 Correlation Between Parallelism and Runtime

We compared the presented model with the model by Lublin and Feitelson [24]. In terms of correlation between the models and the original traces it can be seen from Table 5 that in most of the cases our model is closer to the real correlation value as in the Lublin/Feitelson model. Note, that the results for the Lublin/Feitelson model were taken from [24]. A more comprehensive comparison in terms of the squashed area between the models failed as a first implementation yielded different results from the paper. This will be addressed in our future examinations.

5.3 Temporal Relations

The autocorrelation ρ_1 of the original trace and the modeled workloads has been used to examine the temporal dependencies within each sequence. Table 6 shows

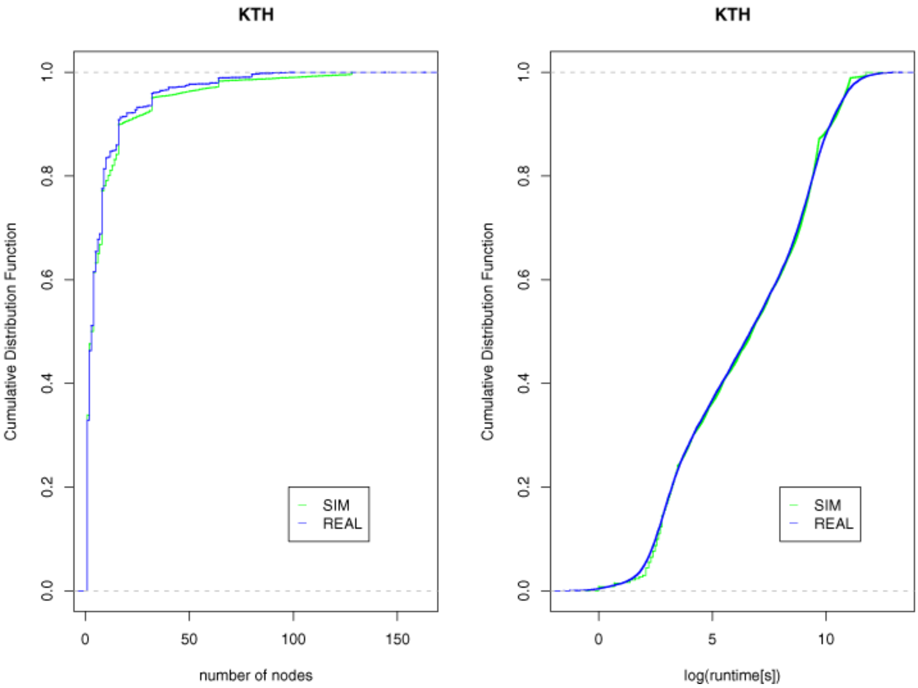


Fig. 4. Comparison of Modeled and Original Distributions of Runtime and Node Requirements.

that the Markov chain model correctly incorporates the temporal dependency since the ρ_1 from the synthetic data are close to the real data. The probability distribution function model does not contain such a dependency.

6 Conclusion

In this paper a workload model based on Markov chains has been presented. This model incorporates temporal dependencies and the correlation between job parameters. To this end, individual Markov chains have been created for runtime and node requirements of jobs. The information have been extracted from real workload traces.

The correlation between job parameters requires the combination of the different Markov chains. To this end, a novel approach of transforming the states in the different Markov chains have been proposed. Note, that these method as presented in this paper have been shown for node and runtime requirements only. However, the approach is very general and can easily be extended to incorporate the modelling of other job parameters.

The quality of the modeling method has been evaluated with existing real workload traces. The presented workload model yielded good results in compari-

	The number of nodes series			The runtime series		
	Real data	MCM	PDF	Real data	MCM	PDF
SDSC 95	0.43	0.31	−0.01	0.28	0.16	0.01
SDSC 96	0.41	0.37	−0.01	0.17	0.20	0.02
KTH	0.29	0.29	0.01	0.29	0.30	−0.01
LANL	0.16	0.20	−0.01	0.18	0.19	−0.03

Table 6. Comparison of the Autocorrelation ρ_1 of the Node Requirements and Runtime Sequences.

son to the real traces. Here, the statistical characteristics as well as the temporal dependencies between jobs are resembled within the model.

The quality criteria used are based on the assumption that the model is used to create a stream of new jobs without further interaction. This can be used to create workload traces for the evaluation of scheduling algorithms as used in [10, 11].

However, as found in [9, 12, 14] new scheduling systems can also benefit by dynamic adaptation according to the current system state. This enables the scheduler to dynamically adjust its parametrization and consequently its behavior. To this end, the workload modeling can also be used to dynamically predict the next job given the last real job submission. This extension is partially already included within our workload model as the parameters of the next created job only depend on the last job. A qualitative evaluation has not yet been done and will be part of future experiments. Currently, no other models are know that incorporates such job predictions.

References

1. Kento Aida. Effect of Job Size Characteristics on Job Scheduling Performance. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1911, pages 1–17. Springer Verlag, 2000. Lecture Notes in Computer Science (LNCS).
2. George E.P. Box and Gwilym M. Jenkins. *Time Series Analysis.* Holden Day Publishing, 1976. ISBN:0-8162-1104-3.
3. Walfredo Cirne and Francine Berman. A Comprehensive Model of the Supercomputer Workload. In *4th Workshop on Workload Characterization*, December 2001.
4. Walfredo Cirne and Francine Berman. A Model for Moldable Supercomputer Jobs. In *Proceedings of the IPDPS 2001 - International Parallel and Distributed Processing Symposium*, April 2001.
5. Jerome T. Connor, R. Douglas Martin, and L.E. Atlas. Recurrent Neural Networks and Robust Time Series Prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254, 1994.
6. Allen B. Downey. A Parallel Workload Model and its Implications for Processor Allocation. Technical Report CSD-96-922, University of California, Berkeley, November 1996.

7. Allen B. Downey and Dror G. Feitelson. The Elusive Goal of Workload Characterization. *ACM SIGMETRICS Performance Evaluation Review*, 26(4):14–29, March 1999.
8. Rakesh Dugad. A Tutorial on Hidden Markov Models and Selected Apllications in Speech Recognition. In A. Waibel and K.-F. Lee, editors, *Readings in Speech Recognition*, pages 267–296. Kaufmann, San Mateo, CA, 1990.
9. Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore*. Springer–Verlag, Lecture Notes in Computer Science LNCS, 2002.
10. Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, May 2002. IEEE Press.
11. Carsten Ernemann, Volker Hamscher, Achim Streit, and Ramin Yahyapour. On Effects of Machine Configurations on Parallel Job Scheduling in Computational Grids. In *International Conference on Architecture of Computing Systems, ARCS*, pages 169–179, Karlsruhe, April 2002. VDE.
12. Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Economic Scheduling in Grid Computing. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537, pages 128–152. Springer Verlag, 2002. Lecture Notes in Computer Science (LNCS).
13. Carsten Ernemann, Baiyi Song, and Ramin Yahyapour. Scaling of Workload Traces. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003 Seattle, WA, USA, June 24, 2003*, volume 2862 of *Lecture Notes in Computer Science (LNCS)*, pages 166–183. Springer-Verlag Heidelberg, October 2003.
14. Carsten Ernemann and Ramin Yahyapour. *"Grid Resource Management - State of the Art and Future Trends"*, chapter "Applying Economic Scheduling Methods to Grid Environments", pages 491–506. Kluwer Academic Publishers, 2003.
15. Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 89–110. Springer-Verlag, 1996. Lecture Notes in Computer Science (LNCS).
16. Dror G. Feitelson. Workload Modeling for Performance Evaluation. In Mariacarla Calzarossa and Sara Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459, pages 114–141. Springer Verlag, 2002. Lecture Notes in Computer Science (LNCS).
17. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science (LNCS)*, pages 1–34. Springer, Berlin, April 1997.
18. Steven Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. In Dror G. Feitelson and Larry Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer, Berlin, Lecture Notes in Computer Science LNCS 1162, 1996.
19. Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

20. 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, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 94–116. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.
21. J. H. Jenkins. Stationary joint distributions arising in the analysis of the M/G/1 queue by the method of the imbedded Markov chain. *Journal of Applied Physics*, 3:512–520, 1966.
22. Hubert Lilliefors. On the Kolmogorov-Smirnov Test for the Exponential Distribution with Mean Unknown. *Journal of the American Statistical Association*, 64:387–389, 1969.
23. Virginia Lo, Jens Mache, and Kurt Windisch. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 25–46. Springer-Verlag, 1998.
24. Uri Lublin and Dror G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, Nov 2003.
25. Jens Mache, Virginia Lo, and Kurt Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems*, 1997.
26. Ara V. Nefian and Monson H. Hayes. Face recognition using an embedded HMM. In *Proceedings of the IEEE Conference on Audio and Video-based Biometric Person Authentication*, pages 19–24, 1999.
27. Bahl Lalit R, Frederick Jelinek, and Robert L. Mercer. A maximum likelihood approach to continuous speech recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligenc PAMI-5*, pages 179–190, 1983. Reprinted in (Waibel and Lee 1990), pp. 308–319.
28. Emilia Rosti, Evgenia Smirni, Lawrence W. Dowdy, Giuseppe Serazzi, and Brian M. Carlson. Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation Journal, Special Issue on Parallel Systems*, 19(2-3):141–165, 1994.
29. Jaspal Subhlok, Thomas Gross, and Takashi Suzuoka. Impact of Job Mix on Optimizations for Space Sharing Schedulers. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, 1996.
30. Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, July 2004.
31. Gautama Temujin and Marc M. Van Hulle. Separation of Acoustic Signals Using Self-organizing Neural Networks. In *Proceedings IEEE Neural Network for Signal Processing Workshop 1999, Madison, WI, USA*, pages 324–332, 1999.

Enhancements to the Decision Process of the Self-Tuning dynP Scheduler

Achim Streit

PC^2- Paderborn Center for Parallel Computing
Paderborn University
33102 Paderborn, Germany
streit@upb.de
http://www.upb.de/pc2

Abstract. The self-tuning dynP scheduler for modern cluster resource management systems switches between different basic scheduling policies dynamically during run time. This allows to react on changing characteristics of the waiting jobs. In this paper we present enhancements to the decision process of the self-tuning dynP scheduler and evaluate their impact on the performance: (i) While doing a self-tuning step a performance metric is needed for ranking the schedules generated by the different basic scheduling policies. This allows different objectives for the self-tuning process, e. g. more user centric by improving the response time, or more owner centric by improving the makespan. (ii) Furthermore, a self-tuning process can be called at different times of the scheduling process: only at times when the characteristics of waiting jobs change (half self-tuning), i. e. new jobs are submitted; or always when the schedule changes (full self-tuning), i. e. when jobs are submitted or running jobs terminate.
We use discrete event simulations to evaluate the achieved performance. As job input for driving the simulations we use original traces from real supercomputer installations. The evaluation of the two enhancements to the decision process of the self-tuning dynP scheduler shows that a good performance is achieved, if the self-tuning metric is the same as the metric used measuring the overall performance at the end of the simulation. Additionally, calling the self-tuning process only when new jobs are submitted, is sufficient in most scenarios and the performance difference to full self-tuning is small.

1 Introduction

Resource management systems (RMS) for modern high performance computing (HPC) clusters consist of many components which are all vital in keeping the machine fully operational. An efficient usage of the machines is important for users and owners, as such systems are rare and high in cost. With regards to performance aspects all components of a modern RMS should perform their assigned tasks efficiently and fast, so that no additional overhead is induced. However, if resource utilization and job response times are addressed, the scheduler plays a major role. A clever scheduling strategy is essential for a high utilization of the

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 63–80, 2005.

machine and short response times for the jobs. However, these two objectives are contradicting. Jobs tend to have to wait for execution on a highly utilized system with space sharing. Short or even no waiting times are only achievable with low utilizations or time-sharing. Typically a scheduling policy that optimizes the utilization prefers jobs needing many resources for a long time. Jobs requesting few resources for a short amount of time may have to wait longer until adequate resources are available. If such small and short jobs are preferred by the scheduler, the average waiting time would be reduced. As jobs typically have different sizes and lengths, fragmentation of the schedule occurs and the utilization drops [1]. The task of the scheduler is to find a good compromise between optimizing these two contrary metrics.

Cluster systems usually have a large user community with different resource requirements and general job characteristics [3]. For example, some users primarily submit parallel and long running jobs, whilst others submit hundreds of short and sequential jobs. Furthermore, the arrival patterns vary between specific user groups. Hundreds of jobs for a parameter study might be submitted in one go via a script. Other users might only submit their massively parallel jobs one after the other. This results in a non-uniform workload and job characteristics that permanently change. The job scheduling policy used in a RMS is chosen in order to achieve a good overall performance for the expected workload. Most commonly used is first come first serve (FCFS) combined with backfilling [7,14,9], as on average a good performance for the utilization and response time is achieved. However, with certain job characteristics other scheduling policies might be superior to FCFS. For example, for mostly long running jobs, longest job first (LJF) is beneficial, whilst shortest job first (SJF) is used with mostly short jobs [1]. Hence, a single policy is not enough for an efficient resource management of clusters. Many modern RMSs have several scheduling policies implemented, or it is even possible to replace the scheduling component.

The remainder of this paper is structured as follows. In Section 2 some related work on self-tuning and dynamic policy switching is given. Section 3 starts with a short history of development, contains the concept of the self-tuning dynP scheduler, and presents the different decider mechanisms and enhancements to them. The used performance metric and the applied workload for the evaluation are presented in Section 4. The evaluation in Section 5 starts with a look on the performance of the three basic policies. Then the evaluation results of the different self-tuning metrics and of the comparison of half vs. full self-tuning are presented. The paper ends with conclusions an a brief outlook on future work.

2 Related Work

In [13] the problem of scheduling a machine room of MPP-systems is described. Users either submit long running batch jobs or they work interactively (typically only for a short time). To accomplish this on a single MPP-system the resource management system has to switch from batch mode (preferring batch jobs) to interactive mode (preferring interactive jobs) and back. Usually this is done manually by the administrative staff, e. g. at fixed times of the day: interactive mode during working hours, batch mode for the rest of the day and over weekends. In general, the overall job throughput is the main objective of batch processing. As batch jobs typically have a long run time, waiting is not very critical. On the other hand, a user that works interactively counts the five minutes until he/she can start working with the requested resources. Other issues like the overall job throughput or the utilization are less important while operating in interactive mode. Which in comparison to batch mode jobs are rather short. The idea [4] is to allow the users to decide in which mode the system should be operating. Hence, the Implicit Voting System (IVS) is introduced, as users should not vote explicitly:

- If most of the waiting jobs are submitted for batch processing, IVS switches to LJF (longest job first). As batch jobs are typically long, they receive a higher priority in the scheduling process. Hence, resources are longer bound to jobs, less resource fragmentation is caused and the utilization and throughput of the system is increased.
- If most of the waiting jobs are submitted for interactive access, IVS switches to SJF (shortest job first). As interactive jobs are usually short in their run time and short jobs are preferred, the average waiting time is reduced.
- If the system is not saturated, the default scheduling strategy FCFS (first come first serve) is used. Note, a threshold for defining when a system is saturated and when not is defined by the administrative staff. For the authors a MPP system becomes saturated, if more than five jobs can not be scheduled immediately.

Unfortunately, the idea of IVS was never realized nor implemented and tested in a real environment.

In [3] a similar approach for the NASA Ames iPSC/860 system is presented. In the prime time during the day only a fraction of the resources is allocated to the batch partition, while most of the resources are available for interactive access. During non prime time all resources are assigned to the batch partition. The re-partitioning is done manually and at fixed times of the day.

The problem of getting the best performance from a modern resource management system is described in [2]. Commonly such software systems are highly parameterized and the administrative staff performs a lot of trial and error testing in order to find a good parameter setting for the current workload. If the workload changes, new parameter settings have to be found. However, they are notoriously overworked and have little or no time for this fine tuning, so the idea is to automate this process. Much information about the current and past workload is available, which is used to run simulations in the idle loop of the system (or on a dedicated machine). Various parameter settings are simulated and the best setting is chosen. The authors call such a system *self-tuning*, as the system itself searches for optimized parameter settings. To create new parameter settings for the simulations, genetic algorithms are used. New parameter settings are generated by randomly combining several potential combinations from the previous step. Chromosomes are the binary representation of a parameter. A parameter setting is called individual and the according parameter values are concatenated in their binary representation. In this example the fitness function is the average utilization of the system achieved by the according parameter setting. All simulated parameter settings (individuals) in one step represent a generation. The chromosomes of the fittest individuals of a generation are used to produce new individuals for the next generation. New generations are continuously created with the latest system workload as input. The process is started with default values. In a case study for scheduling batch jobs of the NASA Ames iPSC/860 system the authors observed that with the self-tuning search for parameter settings the overall system utilization is improved from 88% (with the default parameters) to 91%.

In [8] heuristics for the dynamic mapping of a class of independent tasks onto heterogeneous computing systems are introduced. The mapping problem consist of two parts: matching and scheduling. In the matching phase the assignment of tasks to machines is computed, whilst during scheduling the execution order of the tasks on each machine is computed. In their work a dynamic mapping is needed, as the arrival times of the tasks may be random and the set of available machines is changing. Machines may go off-line and new machines may come on-line. Mapping heuristics can be grouped in two classes: on-line mode and batch-mode heuristics. In the on-line mode tasks are mapped onto a machine immediately after their submission. In the batch-mode, tasks are not immediately mapped when they arrive, instead they are stored and the mapping is invoked at pre-scheduled mapping events. The heuristic MCT (minimum completion time) assigns each task to that machine which results in the task's earliest completion time. Tasks may be assigned to machines, which do not have the minimum execution time. In contrast, MET (minimum execution time) assigns each task to that machine that performs the task in the least amount of execution time. As the machines ready times are not considered by MET, load imbalance across the machines may occur. The new batch-mode SA (switching algorithm) heuristic uses these two heuristics (MCT and MET) in a cyclic fashion depending on the load distribution across the machines. By switching between MCT and MET a new heuristic with the desirable properties of the two single heuristics is generated.

3 Self-Tuning dynP

A single scheduling policy is usually used in a resource management system and it typically generates good schedules only for jobs with specific characteristics (e. g. short jobs). If the job characteristics change, other scheduling policies might perform better and it might be beneficial that the system administrator changes the scheduling policy. However, system administrators are not able to monitor the situation and continuously alter the scheduling policy in response to workload changes.

We developed the dynP scheduler, which automatically switches the active scheduling policy during run time. In general, the set of scheduling policies to choose from can consist of many policies one can think of. We started with a variant of the dynP scheduler, which uses bounds for the average estimated run time of waiting jobs to check, which policy is best suited for the current job characteristics. A major drawback of this version is obvious, as the performance depends on a proper setting of the bounds. And in order to reflect different job characteristics, these bounds need to be changed. We developed the self-tuning dynP scheduler which automatically searches for the best suited policy.

3.1 Concept

At the PC^2 (Paderborn Center for Parallel Computing) the self-developed resource management system CCS (Computing Center Software, [6]) is used for managing the *PSC* Pentium3 cluster [12] and the *pling* cluster [11]. Three scheduling policies are currently implemented: FCFS, SJF, and LJF. According to the classification in [5], CCS is a planning based resource management system. Planning based resource management systems schedule the present and future resource usage, so that newly submitted jobs are placed in the active schedule as soon as possible and they get a start time assigned. With this approach backfilling is done implicitly. By planning the future resource usage, a sophisticated approach is possible for finding a new policy. For all waiting jobs the scheduler computes a full schedule, which contains planned start times for every waiting job in the system. With this information it is possible to measure the schedule by means of a performance metric (e. g. response time, slowdown, or makespan). The concept of self-tuning dynP is:

> The self-tuning dynP scheduler computes full schedules for each available policy (here: FCFS, SJF, and LJF). These schedules are evaluated by means of a performance metric. Thereby, the performance of each policy is expressed by a single value. These values are compared and a decider mechanism chooses the best value, i. e. the smallest value.

In the following, the performance metric used in the self-tuning process is called self-tuning metric for simplicity.

3.2 Decider Mechanisms

For the required decision, several levels of sophistication are thinkable. In [16] we presented the *simple decider* that basically consists of three if-then-else constructs. It chooses that policy which generates the minimum value of the applied self-tuning metric. The simple decider also has drawbacks, as it does not consider the old policy. In particular if two policies are equal and a decision between them is needed, information about the old policy is helpful. Table 5 shows a detailed analysis of the simple decider. FCFS is favored in three and SJF in one case, although staying with the old policy is the correct decision with these cases. This behavior is implemented in the *advanced decider*. At a first glance it does not make any difference which policy among equals is chosen. At this stage the scheduler only knows estimates of the job's run time and usually the job's actual run time is shorter than estimated. When a job ends earlier than estimated, the schedule changes and new planning is necessary. Depending on the chosen policy different jobs might have been started in the meantime. Therefore, even a decision between two equal policies is required. Previously, the fairness among the policies was of major interest. However, it might be interesting to explicitly prefer one of the policies and neglect the others. For that purpose we developed the *preferred decider* [17]. The preferred policy is not switched unless any other policy is better. Whenever any of the other policies are currently used, the preferred policy only has to achieve an equal performance and the decider switches back.

The deciders of the self-tuning dynP scheduler consider only the three policies FCFS, SJF, and LJF for three reasons. First of all, we evaluate the general behavior and performance of self-tuning dynP schedulers for the resource management of HPC systems. We do not evaluate, which combination of policies is best suited for specific job characteristics. Presumably, combinations with other and more scheduling policies exist, which generate even better results. Secondly, FCFS, SJF, and LJF are the most known scheduling policies and many resource management systems have at least these three implemented. And thirdly, these three policies are implemented in the resource management software CCS, which depicts the basis and starting position for our work.

In previous work we already presented the basics of the self-tuning dynP scheduler. It started with the simple decider in [16]. Next, we developed the advanced decider [15] and recently the preferred decider [17]. In this paper we present further enhancements to the decision process, which can be applied to all three mentioned decider mechanisms.

3.3 Enhancements to the Decision Process

As previously mentioned the aim of the self-tuning dynP scheduler is to eliminate input parameters for the scheduler, especially those which depend on the characteristics of the processed jobs and need to be re-adapted continuously. Nevertheless, enhancements that influence the scheduler in a more general way are thinkable. Of course they should be independent of any job characteristics and easy to handle, so that a continuous manual re-adaptation is not needed.

Different Self-Tuning Metrics The concept of the self-tuning dynP scheduler is based on the planning-based scheduling approach, where all waiting jobs are placed in a schedule. With assigning a proposed start time to each job, it is possible to compute the waiting time of the jobs, so that schedules can be compared. For this, different self-tuning metrics can be applied, e. g. user centric metrics like the average response time or the slowdown (both unweighted or weighted), and owner centric metrics like the makespan. By choosing a specific metric, the self-tuning dynP scheduler optimizes its behavior according to this metric. We use the following metrics, which are all defined in the next section: average response time (ART), average response time weighted by area (ARTwA), average response time weighted by width (ARTwW), average slowdown (SLD), average slowdown weighted by area (SLDwA), average slowdown weighted by width (SLDwW) and makespan.

Calling of Self-Tuning Doing self-tuning and potentially switching the active scheduling policy, should be done whenever the schedule or the set of waiting jobs changes, i. e. whenever a new job is submitted or an already running job terminates earlier than estimated. In the following, we call this *full self-tuning*. However, it might also be sufficient to do the self-tuning step only when new jobs are submitted, i. e. the characteristics of waiting jobs in the system change. We call this option *half self-tuning*, as roughly only half as much self-tuning calls are performed. This option is interesting to evaluate in combination with the compute time to do the self-tuning process. It might be beneficial to save up some compute time and make the scheduling behavior more comprehensible for the users.

4 Evaluation

It is common practice to use discrete event simulations for the evaluation of job-scheduling strategies. For this purpose we developed MuPSiE (Multi Purpose Scheduling Simulation Environment). Several policies and the planning-based scheduling approach from [5] are implemented. All presented results are generated with MuPSiE.

4.1 Performance Metrics

We use the user centric slowdown metric for measuring the simulated schedules with all processed jobs. The slowdown of a job is also often called stretch [10] or relative response time, as the jobs response time is divided by the jobs run time. The slowdown comes without a dimension in contrast to e. g. response time. Additionally, we weight each job's slowdown with its area. Thereby, it is circumvented that jobs with the same run time, but with different resource requirements, have the same impact on the overall performance.

With the parameters for a finished job i of a total of m processed jobs:

- t_i^a is the arrival or submission time
- t_i^s is the start time
- t_i^e is the end time
- w_i is the width (number of requested/used resources)

- $l_i = t_i^e - t_i^s$ is the length (run time, duration)
- $t_i^w = t_i^s - t_i^a$ is the waiting time
- $t_i^r = t_i^w + l_i$ is the response time

- $s_i = \frac{t_i^r}{l_i} = 1 + \frac{t_i^w}{l_i}$ is the slowdown
- $a_i = w_i \cdot l_i$ is the area

The average slowdown weighted by job area (SLDwA) for all jobs is defined as:

$$SLDwA = \frac{\sum\limits_{i=1}^{m} a_i \cdot s_i}{\sum\limits_{i=1}^{m} a_i} \tag{1}$$

If the processed jobs are not changed in their width or run time, the average slowdown weighted by job area is equal to the average response time weighted by job width and the equation holds:

$$SLDwA = ARTwW \cdot \frac{\sum\limits_{i=1}^{m} w_i}{\sum\limits_{i=1}^{m} a_i} \tag{2}$$

For completeness, the other metrics used during the self-tuning process are defined as follows:

- the average response time:

$$ART = \frac{1}{m} \cdot \sum_{i=1}^{m} t_i^r \tag{3}$$

- the average response time weighted by job area:

$$ARTwA = \frac{\sum\limits_{i=1}^{m} a_i \cdot t_i^r}{\sum\limits_{i=1}^{m} a_i} \tag{4}$$

- the average response time weighted by job width:

$$ARTwW = \frac{\sum\limits_{i=1}^{m} w_i \cdot t_i^r}{\sum\limits_{i=1}^{m} w_i} \tag{5}$$

- the average slowdown:

$$SLD = \frac{1}{m} \cdot \sum_{i=1}^{m} s_i \tag{6}$$

- the average slowdown weighted by job width:

$$SLDwW = \frac{\sum_{i=1}^{m} w_i \cdot s_i}{\sum_{i=1}^{m} w_i} \tag{7}$$

- the makespan:

$$\max_{i=1,\ldots,m} \; t_i^e \tag{8}$$

4.2 Workload

An evaluation of job scheduling policies requires to have job input. In this work a job is defined by the submission time, the number of requested resources (= width), and the estimated run time (= length). As we model a planning based resource management system [5], run time estimates are mandatory. Additionally, for the simulation the actual run time is needed.

In this paper, we use four traces from the Parallel Workloads Archive [18], as all other traces do not come with information about run time estimates. The characteristics of the four traces are shown in Table 1 (taken from [17]).

- **CTC** (Cornell Theory Center), system: 512-node IBM SP2 (only 430 nodes are available for batch processing), duration: July 1996 - May 1997, jobs: 79,302
- **KTH** (Swedish Royal Institute of Technology), system: 100-node IBM SP2, duration: October 1996 - August 1997, jobs: 28,490
- **LANL** (Los Alamos National Lab), system: 1024-node Connection Machine CM-5 from Thinking Machines, duration: October 1994 - September 1996, jobs: 201,387
- **SDSC** (San Diego Supercomputing Center), system: 128-node IBM SP2, duration: May 1998 - April 2000, jobs: 67,667

	requested			estimated			actual			average	interarrival		
	resources			run time [sec.]			run time [sec.]			overest.	time [sec.]		
trace	min	avg.	max	min	avg.	max	min	avg.	max	factor	min	avg.	max
CTC	1	10.72	336	0	24,324	64,800	0	10,958	64,800	2.220	0	369	164,472
KTH	1	7.66	100	60	13,678	216,000	0	8,858	216,000	1.544	0	1,031	327,952
LANL	32	104.95	1,024	1	3,683	30,000	1	1,659	25,200	2.220	0	509	201,006
SDSC	1	10.54	128	2	14,344	172,800	0	6,077	172,800	2.360	0	934	79,503

Table 1. Basic properties of the used traces (86,400 seconds = 1 day).

5 Results

We start with presenting the results for the three basic policies FCFS, SJF, and LJF in short. This gives a good reference for the subsequent evaluations. At first, the results of the comparison of different self-tuning metrics are presented. A comparison of half and full self-tuning for the mentioned decider mechanisms follows. Finally, an analysis of the switching behavior for the simple and advanced decider is done.

5.1 Basic Policies

As previously stated, the evaluation shows that none of the policies is the best for every job set characteristic. In Table 2 the best basic policy with respect to the average slowdown weighted by area is highlighted with bold font. Particularly for the SDSC trace, the differences in slowdown are large as SJF is worse than FCFS by a factor of almost two and LJF is even worse (twice as bad as FCFS).

Backfilling is done implicitly with the planning-based scheduling approach for all three basic scheduling policies.

	FCFS	SJF	LJF
CTC	2.0455	**1.9277**	2.5212
KTH	3.1015	**2.5488**	5.8118
LANL	**1.6801**	1.7031	2.0507
SDSC	**6.8260**	12.5662	26.8207

Table 2. Overall average slowdown weighted by area (SLDwA) for the three basic policies FCFS, SJF, and LJF.

5.2 Different Self-Tuning Metrics

In the following, the results with different self-tuning metrics are presented. We used the following user centric metrics: average response time (ART), average response time weighted by area (ARTwA), average response time weighted by width (ARTwW), average slowdown (SLD), average slowdown weighted by area (SLDwA), and average slowdown weighted by width (SLDwW). Additionally, the owner centric metric makespan is used.

One can assume that the best performance is achieved by using the same metric during the self-tuning process and after the simulation is finished. As stated earlier, we use the average slowdown weighted by area (SLDwA) metric for measuring all simulated jobs. Hence, using SLDwA during self-tuning should lead to the best performance (i. e. the smallest values). This is seen in the upper part of Table 3. For the CTC, LANL, and SDSC trace the best self-tuning metric is SLDwA and the mentioned expectation is fulfilled. However, it is interesting that for the KTH trace using the job's width as a weight is slightly (0.8%) better than using the job's area as a weight. A possible reason for this is the overestimation factor, which is smaller for the KTH trace (1.5) than for the other three (2.2, cf. Table 1).

Using all other self-tuning metrics results in a significantly worse performance. In particular this is seen for the ARTwA metric and the SDSC trace, as the achieved performance is twice as bad. Observing the numbers for the ARTwW and the SLDwA self-tuning metric shows that the numbers are equal. This is a result of Equation 2 and the fact that the processed jobs are not changed in their width or run time.

The bottom part of Table 3 shows the overall utilization with all simulated jobs. Independent of the applied self-tuning metric the exact same utilizations are generated for the traces CTC, KTH, and LANL. This indicates that those jobs submitted towards the end of the schedule are always scheduled at the same start time and are responsible for the makespan and therefore for the utilization. Only for the SDSC trace different utilizations are achieved, as the differences of the basic policies are large (cf. Table 2). Using the owner centric metric makespan leads to the best utilization. This matches to the results with SLDwA, as makespan and utilization are connected via the total sum of job areas and the totally available resources on the simulated machine.

These observations reflect the different switching behavior of the self-tuning dynP scheduler, if different performance metrics are applied. It is possible to tune the system performance in either way: user or owner centric. Using either owner or user centric metrics in the self-tuning process to generate good overall results for opposing metrics, i. e. user and owner, generally leads to a poor performance and should be avoided.

Comparing the best self-tuning metric with the best basic policy from Table 2 shows that only for the CTC and LANL trace the self-tuning dynP scheduler is better. The performance loss of the self-tuning dynP scheduler is marginal (0.4%) for the KTH trace, but almost 200% for the SDSC trace. This is a result of the overestimation of the job's run time by the users and the large differences of the basic policies. This misleads the self-tuning dynP scheduler in the decision process and results in wrong decisions. Although this also happens with the other traces, the impact is most seen for the SDSC trace.

5.3 Half Vs. Full Self-Tuning

For the comparison of half and full self-tuning one can assume that applying full self-tuning is the best option. With this the self-tuning dynP scheduler chooses the best scheduling policy every time the schedule changes, i. e. when a new job is placed in the schedule and a running job terminates earlier than estimated and a re-scheduling is required. The self-tuning dynP scheduler plans schedules for each available scheduling policy and for all waiting jobs. This results in an increased computational time of the scheduler. As this can be unappropriate in certain scenarios, half self-tuning is an option, as roughly only half as many

	self-tuning metrics						
	ART	ARTwA	ARTwW	SLD	SLDwA	SLDwW	Makespan
SLDwA							
CTC	2.0073	2.2866	**1.8781**	1.9585	**1.8781**	1.9021	2.4582
KTH	3.1459	5.6542	**2.5754**	2.6939	**2.5754**	2.5594	5.3823
LANL	1.7008	1.8328	**1.6177**	1.6626	**1.6177**	1.6179	2.0357
SDSC	14.6495	20.5247	**10.1598**	12.6742	**10.1598**	11.8321	24.8958
Utilization							
CTC	for all self-tuning metrics: 65.701%						
KTH	for all self-tuning metrics: 68.716%						
LANL	for all self-tuning metrics: 55.607%						
SDSC	81.787%	81.812%	81.633%	81.762%	81.633%	81.309%	**82.473%**

Table 3. Overall SLDwA and utilization values for different self-tuning metrics. Advanced decider and full self-tuning applied. Values for ARTwW and SLDwA are equal because of Equation 2.

self-tuning calls are done. Some performance loss might occur with half self-tuning, as with new job submissions the scheduling policy might be changed. In the MuPSiE simulation environment a single self-tuning call for finding a new policy is completed within 6 ms for an average of 22.5 waiting jobs (simulated configuration: advanced decider, full self-tuning, ARTwW as self-tuning metric, CTC trace) and applying half self-tuning might not be necessary.

In Table 4 the slowdown results for the simple, advanced, SJF- and FCFS-preferred decider are presented. One can see that the assumption from above is not always true. In particular for the SDSC trace, half self-tuning is better than full self-tuning. Also with the simple decider full self-tuning is not beneficial. Looking at the performance of the advanced and SJF-preferred decider shows that for the CTC and LANL trace the self-tuning dynP scheduler is always better than the best basic policy. Furthermore, it is interesting to observe that half and full self-tuning have no major impact on the performance of these two deciders. The generated SLDwA values are closer together. Similar to the comparison of the different self-tuning metrics, the SDSC trace is more vulnerable for the switching behavior of the self-tuning dynP scheduler. In particular the simple and FCFS-preferred deciders generate very bad results with full self-tuning applied. In this case doing more self-tuning calls increases the SLDwA by a factor of two. Again this can be a result of overestimating the job run times. By doing self-tuning when jobs terminate and by potentially switching to a disadvantageous scheduling policy, different jobs are immediately started.

One can also see that the advanced decider obviously outperforms the simple decider due to its design. This is independent of whether full or half self-tuning is performed. The performance benefit of the advanced decider is different for the four traces; quite large for the KTH and SDSC trace and smaller for the LANL trace. For the SDSC trace and full self-tuning the difference between the two deciders is almost 70%.

	best basic		decider mechanisms			
	policy	self-tuning	simple	advanced	SJF-preferred	FCFS-preferred
CTC	1.9277	half	2.3036	**1.9085**	**1.8567**	2.3270
	(SJF)	full	2.2812	**1.8781**	**1.8873**	2.2804
KTH	2.5488	half	4.7256	2.5812	2.5734	4.9281
	(SJF)	full	5.7433	2.5754	2.5578	5.7492
LANL	1.6801	half	1.7534	**1.6027**	**1.6330**	1.7605
	(FCFS)	full	1.7610	**1.6177**	**1.6143**	1.7680
SDSC	6.8260	half	13.3353	10.0953	10.2896	16.1766
	(FCFS)	full	32.6934	10.1598	10.5198	32.6965

Table 4. Overall SLDwA comparison of full and half self-tuning with different decider mechanisms.

As for the CTC and KTH trace SJF is the best basic policy and FCFS for the LANL and SDSC trace, a SJF- and FCFS-preferred policy makes sense. The results indeed show that the SJF-preferred decider can improve the performance of the advanced decider for the CTC and KTH trace, but the same does not apply for the FCFS-preferred decider and the LANL and SDSC traces. In fact the FCFS-preferred decider is worse (almost three times for the SDSC trace) than the advanced and SJF-preferred decider. This is surprising, as FCFS proves to be a good basic policy. The poor performance can be based on the fact that some jobs, which are not started by FCFS, alter the schedule in such a way that many subsequent jobs have to wait long and therefore the SLDwA drops. A possible example for this scenario could look like the following: some jobs with a large area (requesting many resources and/or with a long estimated run time) may induce a policy change in order to favor these jobs. However, the estimate of the run time may have been wrong, so that the end after some time. In this case delaying these jobs would be beneficial, as due to their short actual run time their influence on the overall SLDwA performance may only be small.

Detailed Analysis of the Switching Behavior With full self-tuning the difference between the simple and advanced decider is best seen for the SDSC trace, hence a detailed case analysis is done in the following. Table 5 shows the amount each case is reached during the decision process. The numbers show a significant difference in case 6b: the performance of FCFS is equal to SJF, LJF is worse than both, and the old policy is SJF. In 80,419 (75.11%) of 107,066 total self-tuning decisions this situation occurs and the advanced decider stays with SJF. In contrast, the simple decider reaches this case in only 42.56% of all self-tuning decisions and switches to FCFS in this situation.

In case 1 all three policies have the same performance. The correct decision is to stay with the old policy like the advanced decider does, but the simple decider arbitrarily favors FCFS. The other two cases 8c and 10c are not reached by the simple or advanced decider, hence they can not induce the difference in performance. With the large differences in case 6b the number of appearances

case		combinations	simple decider	counted	advanced decider	counted
1		FCFS = SJF = LJF	**FCFS**	**11,135**	**old policy**	**15,861**
2		SJF < FCFS, SJF < LJF	SJF	47,285	SJF	962
3		FCFS < SJF, FCFS < LJF	FCFS	60	FCFS	119
4		LJF < FCFS, LJF < SJF				
	a	FCFS < SJF	LJF	19	LJF	8
	b	FCFS = SJF	LJF	2,007	LJF	7,178
	c	FCFS > SJF	LJF	26	LJF	10
5		FCFS = SJF, LJF < FCFS (⇔ LJF < SJF)	LJF	0	LJF	0
6		FCFS = SJF, FCFS < LJF (⇔ SJF < LJF)				
	a	old policy = FCFS	FCFS	362	FCFS	603
	b	old policy = SJF	**FCFS**	**46,617**	**SJF**	**80,419**
	c	old policy = LJF	FCFS	254	FCFS	1085
7		FCFS = LJF, SJF < FCFS (⇔ SJF < LJF)	SJF	0	SJF	0
8		FCFS = LJF, FCFS < SJF (⇔ LJF < SJF)				
	a	old policy = FCFS	FCFS	1,751	FCFS	820
	b	old policy = SJF	FCFS	0	FCFS	0
	c	old policy = LJF	**FCFS**	**0**	**LJF**	**0**
9		SJF = LJF, FCFS < SJF (⇔ FCFS < LJF)	FCFS	3	FCFS	0
10		SJF = LJF, SJF < FCFS (⇔ LJF < FCFS)				
	a	old policy = FCFS	SJF	2	SJF	1
	b	old policy = SJF	SJF	0	SJF	0
	c	old policy = LJF	**SJF**	**0**	**LJF**	**0**
		totally counted		109,521		107,066

Table 5. Case analysis for the SDSC trace and the simple vs. advanced decider with full self-tuning applied and SLDwA as self-tuning metric.

of the other cases is also influenced. This is best seen for case 4b. However, the other cases have no influence on the different performance of the simple and advanced decider, as both deciders choose the same policy (LJF) as their new policy.

Focusing on the policy usage, Table 6 shows the differences, i. e. how many times the decider switched to each policy and how many jobs were started with each policy. If the advanced decider is applied almost 80% of all jobs are started by SJF and only a minority of 6% by FCFS. About 15% of the jobs are started with LJF. Focusing on the number of switches to each of the policies shows that the advanced decider stays with the current policy and does not switch it in

		simple decider	advanced decider
switches to each policy	FCFS	47,499 (43.37%)	1,086 (1.01%)
	SJF	47,287 (43.17%)	963 (0.90%)
	LJF	395 (0.36%)	1,085 (1.01%)
	no policy switch	14,340 (13.09%)	103,932 (97.07%)
job started with each policy	FCFS	39,936 (59.06%)	4,120 (6.09%)
	SJF	26,737 (39.54%)	53,634 (79.32%)
	LJF	947 (1.40%)	9,866 (14.59%)

Table 6. Comparison of the decision behavior and the usage of policies for the SDSC trace with full self-tuning applied.

most cases (97%). Only in about 1,000 cases the advanced decider switches to one of the policies. This means that once the decider switched to a policy, many jobs are started with this policy. This applies in particular to SJF.

If on the other hand the simple decider is applied, its switching behavior is much more spontaneous. In only 10% of all cases the simple decider does not switch its policy. Most of the time it switches back and forth between FCFS and SJF. This results in an almost equal usage of the two policies over a period of time (43%) and the difference in the number of jobs started with FCFS and SJF is also considerably smaller than with the advanced decider. In only 13% of all self-tuning decisions the simple decider stays with its current policy. Discarding its previous decision leads to a scenario where preceding jobs are started by alternating policies. Compared to the advanced decider about ten times more jobs (60%) are started with FCFS by the simple decider, whereas about only half as many jobs (40%) are started by SJF. Only a minority of all jobs are started with LJF.

The number of self-tuning calls with the simple decider (109,521) is larger than with the advanced decider (107,066). This results from the fact that more than one job ends at the same time. Why? As full self-tuning is applied and the same job trace is used, the amount of self-tuning calls at job submission does not change for one of the deciders. However, if more than one job ends at the same time, a reschedule takes place only once and therefore self-tuning is also called only once. Hence, the advanced decider performs better than the simple decider and at the same time induces less self-tuning calls.

From this fact another question arises: If only half self-tuning is applied, i. e. self-tuning is not done when jobs end, the number of self-tuning calls should almost be the same for both deciders? And yes, if half self-tuning is applied the simple decider is called 56,738 times whereas the advanced decider is called 56,208 times. Both amounts are a slightly less than the number of totally scheduled jobs (67,620). This is based on the fact that if enough resources are free to start all jobs immediately, these jobs do not have to wait. Self-tuning and scheduling policies in general make no sense in this case, as the starting order of jobs does not matter, as long as they are started immediately.

6 Conclusions and Future Work

In this paper we presented two options for the decision process of the self-tuning dynP scheduler. The idea of dynamically switching the scheduling policy (dynP) is based on the fact that usually no single policy generates good schedules for every possible job characteristic. In order to achieve the best possible performance, it becomes necessary to switch the active scheduling policies according to the currently waiting jobs. The scheduler switches the scheduling policies without the need of a permanent intervention of the system administrator. With the planning-based scheduling approach, the self-tuning dynP scheduler generates full schedules for each available basic policy, measures the generated schedules with a performance metric, and finally switches to the best policy. A decider mechanism is in charge of choosing the best policy according to the applied performance metric. In previous papers we presented different decider mechanisms.

In this paper we evaluated two general enhancements, which can be applied to all decider mechanisms. We compared different owner and user centric performance metrics for the self-tuning process and studied their influence. By using different self-tuning metrics the objective of the self-tuning dynP scheduler can be altered. Additionally, we studied the behavior of calling the self-tuning process at different times (half and full self-tuning) of the scheduling process. The evaluation is done by using discrete event simulation with job traces from real supercomputer installations as input.

Studying the different self-tuning metrics one assumes that most likely the best performance is achieved by using the same metric during the self-tuning process and after the simulation is finished to measure all jobs. This is true, the user centric average slowdown weighted by area (SLDwA) is the best self-tuning metric for three traces (CTC, SDSC, and LANL). For the KTH trace the average slowdown weighted by width (SLDwW) slightly improves the performance slightly by 0.8%. If the objective of the self-tuning dynP scheduler is to optimize the owner centric overall utilization of the system, the makespan generates the best results, although only for the SDSC trace. The characteristics of the remaining three traces generate equal overall utilizations for all applied self-tuning metrics.

We also compared half and full self-tuning, i. e. calling the self-tuning process only when new jobs are submitted or additionally when running jobs terminate. Although much less self-tuning calls are done with half self-tuning, the performance different to full self-tuning is small with the advanced and SJF-preferred decider. Similar to the comparison of different self-tuning metrics, the SDSC trace is more vulnerable for the switching behavior of the self-tuning dynP scheduler. In particular the simple and FCFS-preferred decider generate very bad results, which are almost three times as bad as for the other deciders. Therefore, if less self-tuning calls are intended by the system administrators, e. g. to reduce the switching behavior of the self-tuning dynP scheduler, the generated performance is only sightly behind and half self-tuning is a good compromise.

We showed that in general the presented self-tuning scheduler with dynamic policy switching can be beneficial to commonly used static scheduling approaches. Therefore, we think that self-tuning schedulers, which are able to adapt their scheduling behavior according to the job characteristics of the currently waiting jobs, should be implemented in modern cluster resource management systems. From a practical perspective the self-tuning dynP scheduler might cause problems for the users, as the scheduling behavior of the system might become unpredictable. For practical matters a policy switching might only be done at well established times, e. g. only once an hour.

In the future it will be interesting to study, whether a combination of different self-tuning performance metrics is beneficial. For example, the owner centric makespan metric is considered with 20%, and the user centric average response time weighted by job width is considered with 80%.

References

1. D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1995.
2. D. G. Feitelson and M. Naaman. Self-Tuning Systems. In *IEEE Software 16(2)*, pages 52–60, April/Mai 1999.
3. D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer, 1995.
4. J. Gehring and F. Ramme. Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 1996.
5. M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In D. G. Feitelson and L. Rudolph, editor, *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.
6. A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC Clusters. In *Annual Review of Scalable Computing, vol. 3, Singapore University Press*, pages 1–31, 2001.
7. D. A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer, 1995.
8. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.
9. A. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Trans. Parallel & Distributed Systems 12(6)*, pages 529–543. IEEE Computer Society Press, June 2001.
10. S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online Scheduling to Minimize Average Stretch. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 433–442, 1999.
11. The *pling* Itanium2 Cluster at the Paderborn Center for Parallel Computing (PC^2). http://www.upb.de/pc2/services/systems/pling/index.html, April 2004.
12. The *PSC* Pentium3 Cluster at the Paderborn Center for Parallel Computing (PC^2). http://www.upb.de/pc2/services/systems/psc/index.html, April 2004.
13. F. Ramme and K. Kremer. Scheduling a Metacomputer by an Implicit Voting System. In *3^{rd} Int. IEEE Symposium on High-Performance Distributed Computing*, pages 106–113, 1994.
14. J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY — LoadLeveler API Project. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer, 1996.

15. A. Streit. A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. In D. G. Feitelson and L. Rudolph, editor, *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.
16. A. Streit. The Self-Tuning dynP Job-Scheduler. In *Proc. of the 11th International Heterogeneous Computing Workshop (HCW) at IPDPS 2002*, pages 87 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2002.
17. A. Streit. Evaluation of an Unfair Decider Mechanism for the Self-Tuning dynP Job Scheduler. In *Proc. of the 13th International Heterogeneous Computing Workshop (HCW) at IPDPS*, pages 108 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2004.
18. Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, April 2004.

Reconfigurable Gang Scheduling Algorithm

Luís Fabrício Wanderley Góes and Carlos Augusto Paiva da Silva Martins

Pontifical Catholic University of Minas Gerais, Post-Graduation Program in
Electrical Engineering, CEP 30535-610
Av. Dom José Gaspar, 500 Belo Horizonte, Minas Gerais, Brasil
lfwg@uol.com.br, capsm@pucminas.br

Abstract. Using a single traditional gang scheduling algorithm cannot provide the best performance for all workloads and parallel architectures. A solution for this problem is an algorithm that is capable of dynamically changing its form (configuration) into a more appropriate one, according to environment variations and user requirements. In this paper, we propose, implement and analyze the performance of a Reconfigurable Gang Scheduling Algorithm (RGSA) using simulation. A RGSA uses combinations of independent features that are often implemented in GSAs such as: packing and re-packing schemes (alternative scheduling etc.), multiprogramming levels etc. Ideally, the algorithm may assume infinite configurations and it reconfigures itself according to entry parameters such as: performance metrics (mean utilization, mean response time of jobs etc.) and workload characteristics (mean execution time of jobs, mean parallelism degree of jobs etc.). Also ideally, a reconfiguration causes the algorithm to output the best configuration for a particular situation considering the system's state at a given moment. The main contributions of this paper are: the definition, proposal, implementation and performance analysis of RGSA.

1 Introduction

Nowadays, the service quality requirements of users and institutions increased. Thus, computer systems that provide many services (particularly, parallel machines) need to be highly utilized and provide a short response time for users jobs. Parallel job schedulers should match both requirements and workload (jobs) with resource availability (architecture, processors etc.) in order to maximize the system's performance. The main problem is that workload, requirements and resources change continuously. In order to solve this problem, many works have been developed to make job scheduling algorithms more flexible and adaptable [1], [12], [13], [14], [18], [20]. Up to now, a poorly explored solution is the use of reconfigurable computing concepts [3], [4], [13], [14], [16] in parallel job scheduling algorithms (like gang scheduling).

Reconfigurable computing emerged as a paradigm to fill in the gap between hardware and software, reaching better performance than software and more flexibility than hardware [3], [4], [16]. The reconfigurable devices including FPGAs (Field Programmable Gate Arrays) contain an array of computing elements or constructive blocks, whose functionalities are determined through the programming of configuration bits. Thus, an FPGA can implement different behaviors not established at design

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 81-101, 2005.

time. Because of this, reconfigurable devices (hardware) are improving the solutions for problems from different areas [3], [4], [16].

Our basic idea in this paper is to use reconfigurable computing concepts in a parallel job scheduling algorithm (gang scheduling) to maximize system's performance. According to a deep bibliographic revision [3], [4], [13], [14], [16], we found works that apply reconfigurable computing in software, but we did not find a previous work that used it on algorithms. In [13], we used a first approach to build a reconfigurable algorithm of a static parallel job scheduling algorithm. We improved this first approach to reach our present stage.

Ideally, the algorithm may assume infinite configurations and it reconfigures itself according to entry parameters such as: performance metrics (utilization, mean response time of jobs etc.) and workload characteristics (mean execution time of jobs, mean parallelism degree of jobs etc.). Also ideally, a reconfiguration causes the algorithm to output the best configuration for a particular situation considering the system's state at a given moment.

Gang scheduling algorithms have been intensely studied in the last decade. They demonstrated many advantages over other parallel job scheduling algorithms, for instance, they: provide interactive response time for short jobs, through preemption; prevent long jobs from monopolizing processors; maximize the system's utilization etc [1], [2], [5], [6], [11], [12], [14], [18], [19], [20]. In our specific case it presents some interesting characteristics. It is composed of independent and well defined parts (packing and re-packing schemes, multiprogramming level, etc.) and each one has infinite possible solutions (implementations).

The **main objectives** of this paper are: to define, propose, develop and implement the RGSA; to analyze the performance of RGSA using simulation. The **main goal** is the implementation of RGSA in our simulation tool.

In this paper, we introduce the reconfigurable gang scheduling algorithm (RGSA) and relate it to other works in sections 2 and 3. In section 4, we present our experimental method: workload, metrics, configurations and parallel architecture used in our simulations. Section 5 presents the experimental results and the performance analysis comparing RGSA and other traditional gang scheduling algorithms. Finally, in section 6 we highlight our conclusions and future works.

2 Reconfigurable Gang Scheduling Algorithm (RGSA)

Extending the reconfigurable hardware definition, we define a reconfigurable algorithm as an algorithm that is composed of constructive blocks allowing its behavior to be modified through the form of its configuration.

A reconfigurable algorithm is composed of three layers: Configuration Control Layer (CCL), Reconfigurable Layer (RL) and Basic Layer (BL), as shown in Fig.1. The BL consists of a frame set and data structures. A data structure may be a list, a queue, an array or some structure that stores data. For example, in Fig.2 a wait queue (data structure) stores jobs (data).

A frame represents a part or phase of an algorithm. For example, in a gang scheduling algorithm, a frame may represent a packing scheme that fits a job inside the

Ousterhout matrix, which means it is only a part of a gang scheduling algorithm. There are two frame types: control and action frames. A control frame controls a specific characteristic of a data structure. In Fig.2, the Multiprogramming Levels Frame controls the multiprogramming level of the Ousterhout Matrix. An action frame is responsible for process or move data between or inside data structures and frames. In Fig. 2, the Packing Schemes Frame receives a job from the Queue Policies Frame and fits it inside the Ousterhout Matrix.

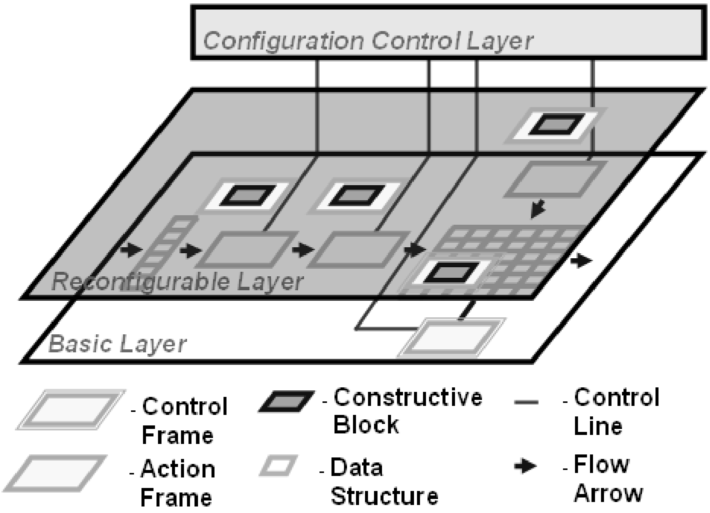


Fig. 1. The general architecture of a reconfigurable algorithm composed of three layers: Configuration Control Layer (CCL), Reconfigurable Layer (RL) and Basic Layer (BL).

The Reconfigurable Layer represents a configuration or an instance of the BL, in which every frame is filled out with one or more compatible constructive blocks at a certain moment. A constructive block is a possible implementation that can fill out with a specific frame. For example, the Re-Packing Schemes Frame, shown in Fig. 2, can be filled out with different re-packing schemes like slot unification and alternative scheduling, one at a time or simultaneously. So, each re-packing scheme implementation is a constructive block. When two or more constructive blocks simultaneously fill out a frame, they are executed in sequence. The maximum number of possible constructive blocks that fill out a frame is the number of different known implementations, for example, the number of known re-packing schemes.

The Configuration Control Layer chooses the constructive blocks that will fill out each frame at a given moment, thus it controls the configuration swapping. The choice is made based on entry parameters. The CCL can be implemented as a static table with pre-defined decisions, an evolutionary algorithm, a learning-based algorithm (neural network) etc. For example, we have a workload composed of long jobs and the most important metric for the user is the reaction time. So, the CCL will set a configuration that reduces the reaction time of the long jobs. In our case, the CCL should fill the Multiprogramming Levels Frame with the Unlimited Constructive Block, allowing a job to start its execution as soon as it was submitted.

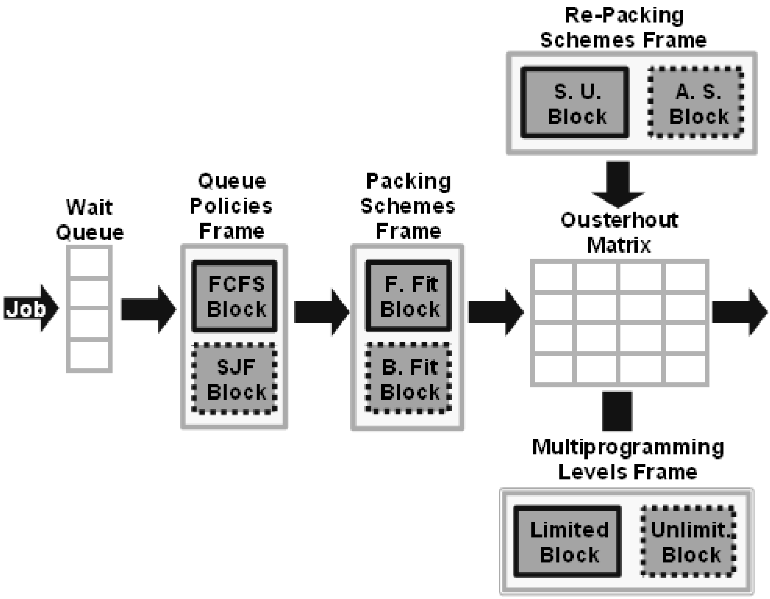


Fig. 2. The Basic Layer of the Reconfigurable Gang Scheduling Algorithm (RGSA) and some possible constructive blocks of the Reconfigurable Layer.

A gang scheduling algorithm may be composed of four parts: a packing scheme, a re-packing scheme, a queue policy and a multiprogramming level. In our Reconfigurable Gang Scheduling Algorithm (RGSA), as show in Fig. 2, each part is a different frame with two constructive blocks, to simplify our study. The first three are action frames and the last one is a control frame.

The Packing Schemes Frame may be filled out with two different packing schemes based on capacity: first fit or best fit. The Re-Packing Schemes Frame may be filled out with the slot unification and/or alternative scheduling re-packing schemes. The Queue Policies Frame can use the First Come First Served (FCFS) or Short Job First (SJF) policies. Finally, the Multiprogramming Levels Frame can be filled out with the Unlimited or Limited Multiprogramming Level Constructive Blocks.

In our RGSA, the CCL is implemented as a table (or switch case structure) that knows the best configuration according to some workload parameters, as shown in Table 1. The workload parameters and possible values are: execution time (high (H) or low (L)), parallelism degree (high (H) or low (L)), predominance degree (60%, 80% or 100%) and the most important metric (utilization (UT), reaction time (ReacT), slowdown (SD), response time (RespT) or simulation time (ST)). Then CCL evaluates these parameters and reconfigures RGSA to the best configuration. The workload parameters chosen and configurations will be better discussed in the Experimental Method section.

The backfilling scheduling algorithm needs an estimated execution time for all submitted jobs as an input parameter [17]. As described before, the RGSA also needs input parameters, but these ones don't need to be introduced by each user (per job). Using past information (log files etc.), depending on the day and time, we can classify

Table 1. The actual CCL implementation that chooses the best configuration according to some workload parameters.

Case	**Workload Parameters**				Configuration
	Metric	Execution Time	Parallelism Degree	Predominance Level	
1	UT or ST	High	Low	100	Conf 2
2	UT or ST	Low	High	80	Conf 2
3	RespT	High	High	80	Conf 2
4	UT or ST	Low	Low	100	Conf 4
5	RespT	Low	Low	80	Conf 4
6	UT or ST	Low	High	100	Conf 5
7	ReacT	High	High	100	Conf 5
8	ReacT or RespT or SD	High	Low	80	Conf 5
9	RespT	Low	High	60	Conf 5
10	ReacT or RespT	Low	High	80	Conf 5
11	ReacT	Low	Low	60 or 80 or 100	Conf 5
12	SD	Low	Low	100	Conf 5
13	SD or ReacT	High	High	60	Conf 6
14	SD or ReacT	High	Low	100	Conf 6
15	SD	Low	Low	60 or 80	Conf 6
16	SD	High	Low	60	Conf 6
17	RespT	High	High	100	Conf 7
18	RespT	High	Low	100	Conf 7
19	RespT	Low	Low	100	Conf 7
20	UT or ST	High	Low	80	Conf 8
21	UT or ST	Low	High	60	Conf 8
22	RespT	High	High	60	Conf 8
23	RespT	Low	High	100	Conf 8
24	RespT	Low	Low	60	Conf 8
25	UT or ST	High	Low	60	Conf 10
26	UT or ST	High	High	60 or 80 or 100	Conf 11
27	UT or ST	Low	Low	60 or 80	Conf 11
28	ReacT	High	High	80	Conf 11
29	ReacT or RespT	High	Low	60	Conf 11
30	SD	Low	High	60	Conf 11
31	ReacT or SD	Low	High	100	Conf 12
32	SD	High	High	80 or 100	Conf 12
33	SD	Low	High	80	Conf 12

or divide workloads in groups (sub-workloads) in a time interval by the predominance level of a job type. For example, in Fig. 3, on Mondays between 0 a.m. to 6 a.m., based on a hypothetical log file, we noted that all executed jobs (predominance level equal to 100%) have a high execution time and high parallelism degree (HH100%). And in this period (night), the most important metric could be utilization. So, according to our CCL implementation, the RGSA reconfigures to the configuration 11.

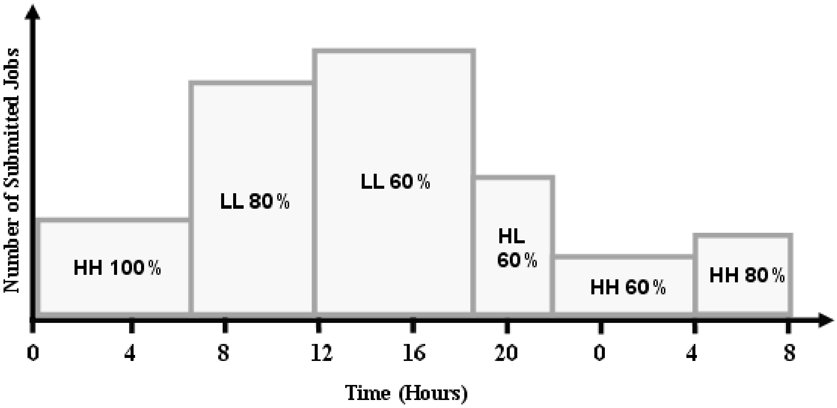


Fig. 3. The classification of a log file in sub-workloads, along the time, by predominance level of a job type.

This classification process can be done by a system administrator or an automated system that examines log files and classifies the sub-workloads. Along the time, the CCL table can be updated. As suggested in [9], the RGSA can use idle cycles to simulate the last executed sub-workload with all possible different configurations and update the table with the best configuration for this sub-workload. As we know, some system's behaviors repeat over the time. For example, if on last Monday at night, the RGSA found that configuration 11 was the best one, probably this configuration will achieve a good performance if RGSA uses it in the next Monday at night.

The selection of the most important metric can be done according to the predominance level of interactive and batch jobs in a workload. For interactive jobs, reaction time and response time are generally most important, because users want a quick answer. And for batch jobs, utilization is the most important, because the system administrator needs to use the maximum of the system resources. The definition of thresholds between high and low execution time and parallelism degree must be determined according to each system.

3 Related Work

This paper presents the main results of a master's thesis [14]. In this research, we found many works about gang scheduling [1], [2], [5], [6], [11], [12], [14], [18], [19], [20], few works about reconfigurable software [13], [14], [16] and algorithms, and none about reconfigurable parallel job scheduling algorithms. Even so, all related works are deeply discussed in [14] and really helped us to reach our objectives and goals. In this paper, we will discuss only four papers that are more relevant and close to our work [6], [9], [11], [17].

In [12], a flexible co-scheduling algorithm is proposed and implemented. As well as our proposal, it uses a different algorithm depending on the workload. The gang scheduling is only used with jobs that really need it, while other jobs can be scheduled with no restrictions. This approach is limited to a couple of scheduling options. Moreover, the used gang scheduling algorithm is the traditional one.

Regarding the experimental results, [6] is the work that presents the closest experimental results to our research. By simulation, Feitelson compares many different packing schemes and few re-packing schemes, looking for the one which best performs on average for the used workload. Thus he does not present the idea that the same algorithm can assume different configurations, by changing its packing schemes, for example. Moreover he does not vary others parameters like the multiprogramming level and queue policies. Even so, it is very important to compare that work with some results that were achieved in our simulations.

In [9], Feiltelson presents the idea of self-tuning systems, in which the process to tune the system is automated. It uses genetic algorithms and log files as input for simulations. These simulations are performed during idle cycles, increasing the utilization of the system, with no cost.

Finally, in [17], a self-tuning job scheduler with dynamic policy switching is simulated and analyzed using trace information from some computing centers. Like backfilling schedulers it needs information about the job's estimated execution time. It is limited to three policies and conservative backfilling. It presents a fine idea of self-tuning that can be used in our Configuration Control Layer to change configurations.

4 Experimental Method

In this section, we first describe the metrics, parallel architecture and workload used in our simulations. Afterwards, we describe the experimental design in which we highlight the used configurations.

4.1 Metrics, Parallel Architecture, and Workloads

In order to analyze a parallel job scheduling algorithm, we can use different metrics. The most common are: utilization, response time, reaction time and slowdown [7], [8], [15].

4.1.1 Metrics

The mean utilization of a parallel architecture may be calculated through Eq.1, where *CPUBusyTime* is the time in which a processor was busy and *TotalTime* is the total time involved in the execution of all the workload. The utilization value is always between 0 and 1. The utilization depends directly on the input load. To compare different job scheduling algorithms under the same load and workload, the relative difference of the obtained utilizations is an important parameter to evaluate the performance gained of the parallel architecture for use a certain scheduling algorithm.

$$\mathit{MeanUtilization} = \frac{\sum \mathit{CPUBusyTime}}{\mathit{NumberOfProcessors} \times \mathit{TotalTime}} \tag{1}$$

The mean response time of a job (in seconds), defined in Eq.2, is the mean time interval between the submission and end of a job.

$$\mathit{MeanResponseTime} = \frac{\sum \left(\mathit{JobEndTime} - \mathit{JobSubmissionTime}\right)}{\mathit{NumberOfJobs}} \tag{2}$$

The mean reaction time of a job (in seconds), defined in Eq.3, is the mean time interval between the submission and the start of a job.

$$\mathit{MeanReactionTime} = \frac{\sum \left(\mathit{JobStartTime} - \mathit{JobSubmissionTime}\right)}{\mathit{NumberOfJobs}} \tag{3}$$

As shown in Eq.4, the mean slowdown of jobs is the sum of jobs response times (reaction time + execution time) divided by the jobs execution times (dedicated time). This metric emerges as a solution to normalize the high variation of the response time of jobs. The nearest the value is from 1, the better is the slowdown.

$$\mathit{MeanSlowdown} = \frac{\sum \frac{\mathit{JobResponseTime}}{\mathit{JobExecutionTime}}}{\mathit{NumberOfJobs}} \tag{4}$$

We decided to use the mean simulation time of the workload as a metric too, which is the time interval between the beginning and the end of the simulation (when the last job ends).

4.1.2 Parallel Architecture

The selected parallel architecture is a cluster composed of 16 nodes and a front-end node interconnected by a Fast Ethernet switch. Each node has a Pentium III 1 Ghz (real frequency = 0.938Ghz) processor. In Table 1, we see the main values of the cluster's characteristics, obtained from benchmarks and performance libraries (Sandra 2003, PAPI 2.3 etc.). These values are essential as input parameters to our simulation tool called ClusterSim, developed by our group.

The ClusterSim is a Java-based parallel discrete-event simulation tool for cluster computing. It supports visual modeling and simulation of clusters and their workloads for performance analysis. In the simulation model, a cluster is composed of single or multi-processed nodes, parallel job schedulers, network topologies and technologies. A workload is represented by users that submit jobs composed of tasks described by probability distributions and their internal structure (CPU, I/O and communications instructions). The simulation model supports a lot of events: job arrival, end of job, unblock task, end of task, message arrival etc. For that reason, depending on cluster size and especially on the number of jobs, the execution of a simulation can be too long and the simulation tool can become out of memory.

Table 2. Cluster characteristics and respective values.

Characteristic	Value	Characteristic	Value
Number of Processors	16 + 1	**Network**	Fast Ethernet
Processor Frequency	0.938 Ghz	**Network Latency**	0.000179 s
Cycles per Instruction	0.9997105	**Max. Segment Size**	1460 bytes
Primary Memory Transfer Rate	11.146 MB/s	**Network Bandwidth (Max. Throughput)**	11.0516 MB/s
Secondary Memory Transfer Rate	23.0 MB/s	**Protocol Overhead**	58 bytes

4.1.3 Workload

As described before, in our simulation tool, a workload is composed of a set of jobs featured by: their types, internal structures, submission probabilities and inter-arrival distributions. Due to the lack of information about the internal structure of the jobs, we decided to create a synthetic set of jobs [8], [10], [15].

In the related works [2], [5], [6], [10], [19], we found only information about the execution time of the jobs, but our simulation tool simulates a job execution based on its number of instructions. So we performed some pilot tests to define some of the values (number of instructions, granularity etc.) for our synthetic jobs. In order to simplify our jobs internal structures, we fixed some of the values and characteristics (Table 3).

Table 3. Workload characteristics and their values.

Characteristic	Value
Granularity	Low – 1 million instructions High – 10 million instructions
Number of Instructions	Low – 100 million instructions High – 1 billion instructions
Parallelism Degree	Low – uniform distribution (1,4) High – uniform distribution (5,16)
Parallel Algorithm Model	Process Farm (Master Slave)
Message Size	16 Kbytes

In the workload jobs, at each one of the iterations, the master task sends a different message to each slave task. On their turn, they process a certain number of instructions, according to the previously defined granularity, and then they return a message to the master task. The total number of instructions that is to be processed by the job and the size of the messages are divided among the slave tasks, that is, the greater is the number of tasks (high parallelism degree) the least is the number of instructions that a single task has to process.

With regard to the parallelism level, which is represented by a probability distribution, we considered jobs between 1 and 4 tasks as low parallelism degree and between 5 and 16 as high parallelism degree. As we know, real workload analyses show that for large parallel machines (bigger than 64 processors), there are more small jobs. In our case, we did a relative equivalence. For example, in a 128-processors machine, short jobs are less than 32 tasks (one quarter). So, for a 16-processor machine, we considered a short job as less than 4 tasks (one quarter). As usual, we used a uniform distribution to represent the parallelism level, another more realistic way could be the use of a uniform distribution that samples power of 2 numbers. Combining the parallelism level, number of instructions and granularity characteristics, we had 8 different basic job types.

There are two main aspects through which a job can influence in a gang scheduling: space and time [7]. In our case, space is related with the parallelism degree and time with the: number of instructions, granularity and the other factors. Combining space (parallelism degree) and time (execution time), we can cover the majority of possible workloads. So, after the simulation, we can identify, in a log file, subworkloads that fit into any of these combinations. Thus we combine these orthogonal aspects to form 4 workload types.

In the first type, the most predominant are the jobs with a high parallelism degree and a structure that leads to a high execution time. In the second type, jobs with a high parallelism level and a low execution time predominate. The third one has the majority of jobs with a low parallelism degree and a high execution time. In the last workload, jobs with a low parallelism degree and a low execution time prevail. For each workload we varied the predominance level between 60%, 80% and 100% (homogeneous). For example, a workload HH60 is a workload composed of 60% of jobs with a high execution time and a high parallelism degree, and the other 40% is composed of the opposite workload (low execution time and parallelism degree). So, we created 12 workloads to test the gang scheduling algorithms: HH60, HH80 and HH100; HL60, HL80 and HL100; LH60, LH80 and LH100; LL60, LL80 and LL100.

In all workloads we use a total number of jobs equal to 100 (due to the ClusterSim simulation time and memory limitations) and the inter-arrival represented by an Erlang hyper-exponential distribution. To simulate a heavy load, we divided the inter-arrival time by a load factor equal to 100. This value was obtained through experimental tests.

4.2 Experimental Design

It is important to note that each RGSA configuration is a traditional gang scheduling algorithm (TGSA). Because in a TGSA, its parts are fixed and cannot be changed over time. For example, Conf01 has the first fit, alternative scheduling, limited multiprogramming level and FCFS, and it cannot changes over time. Through the rest of this paper, TGSA and configuration will be treated as synonyms.

In order to test and analyze the performance of the RGSA, we used a full factorial model. A configuration of RGSA or a traditional gang scheduling algorithm is composed of a packing scheme, a re-packing scheme, a multiprogramming level and a queue policy. In Table 4, we observe the possible configurations of RGSA. The multiprogramming level was limited in 3. When the multiprogramming level is unlimited, it does not make sense to use a wait queue. Because, as soon as a job arrives, it will always fit to the matrix.

Each one of the 12 configurations was tested with each workload, using 10 different simulation seeds. The selected seeds were: 51, 173, 19, 531, 211, 739, 413, 967, 733 and 13. So we made a total of 1440 simulations.

Table 4. RGSA configurations composed of packing and re-packing schemes, mulitprogramming levels and queue policies.

Configurations	Multiprogramming Level	Queue Policy	Packing Scheme	Re-Packing Scheme
Conf 01	Limited	FCFS	First Fit	Alternative Scheduling
Conf 02	Limited	SJF	First Fit	Alternative Scheduling
Conf 03	Limited	FCFS	First Fit	Slot Unification
Conf 04	Limited	SJF	First Fit	Slot Unification
Conf 05	Unlimited	X	First Fit	Alternative Scheduling
Conf 06	Unlimited	X	First Fit	Slot Unification
Conf 07	Limited	FCFS	Best Fit	Alternative Scheduling
Conf 08	Limited	SJF	Best Fit	Alternative Scheduling
Conf 09	Limited	FCFS	Best Fit	Slot Unification
Conf 10	Limited	SJF	Best Fit	Slot Unification
Conf 11	Unlimited	X	Best Fit	Alternative Scheduling
Conf 12	Unlimited	X	Best Fit	Slot Unification

5 Experimental Results

In this section, we present and analyze the performance of RGSA. First, for each metric, we present the results obtained by simulation and analyze the performance and influence of every frame. To do it, we compare sets of configurations in which the analyzed frame is filled out with different blocks and the other frames have a fixed block. At the end of this section, we compare between the performance of RGSA and every configuration individually.

5.1 Utilization

In Fig. 4, we present the relative mean utilization of the cluster among each configuration for all workloads. Considering the packing schemes (Fig. 5(a)), when the multiprogramming level is unlimited, the first fit provides higher utilization for HL and LH workloads.

Initially, the best fit scheme finds the best slot for a job, but at long term, this decision may prevent new jobs from entering in more appropriate positions. In the case of HL and LH workloads, this chance increases, because the long jobs (with a low parallelism degree) that remain after the execution of short jobs (with a high parallelism degree) will probably occupy columns in common, thus, making it difficult to defragment the matrix. On the other hand, the first fit initially makes the matrix more fragmented. Besides, it increases the multiprogramming level. But at long term, it will make it easier to defragment the matrix, because the jobs will have fewer time slot columns in common. In the other cases, the best fit scheme presents a slightly better performance. In general, both packing schemes have an equivalent performance. The same happens to the re-packing schemes (Fig. 5 (b)).

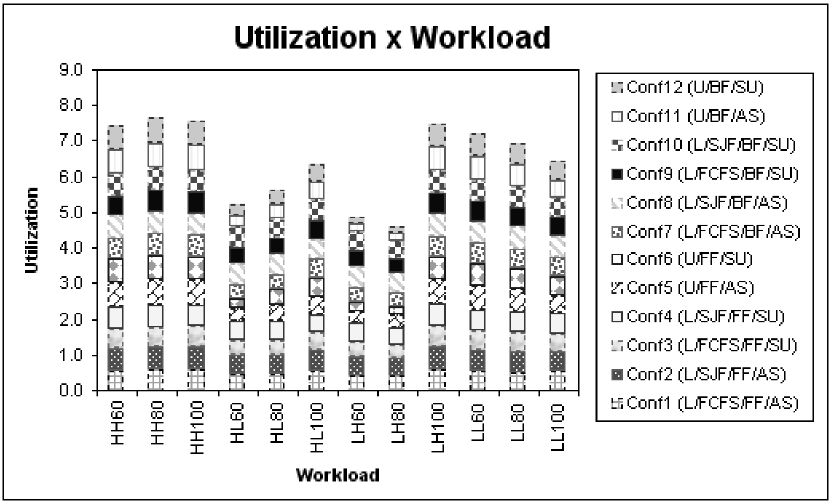


Fig. 4. The relative mean utilization among each configuration for all workloads.

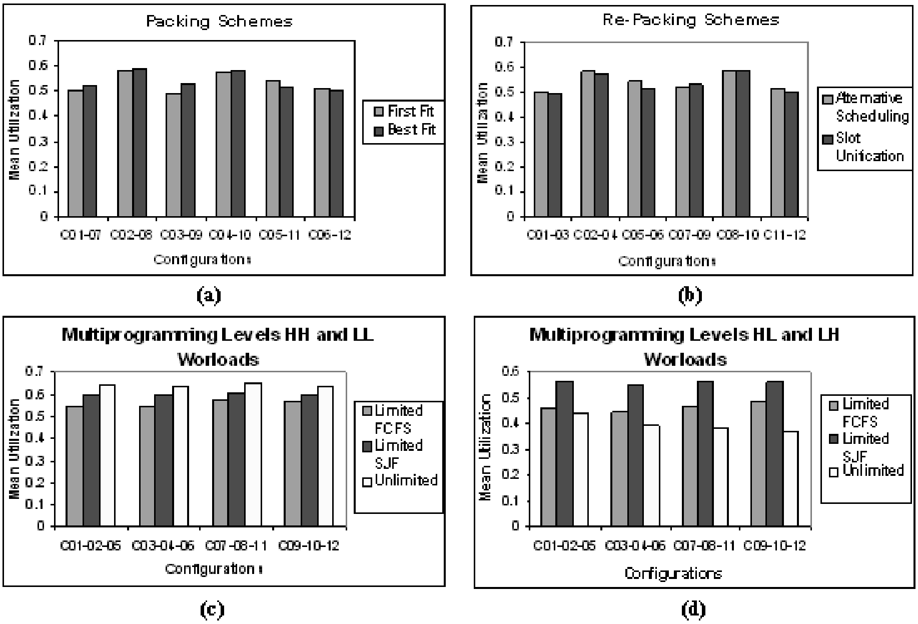


Fig. 5. Mean utilization considering the (a) packing schemes; (b) re-packing schemes; multiprogramming level for (c) HH and LL workloads; and (d) HL and LH workloads.

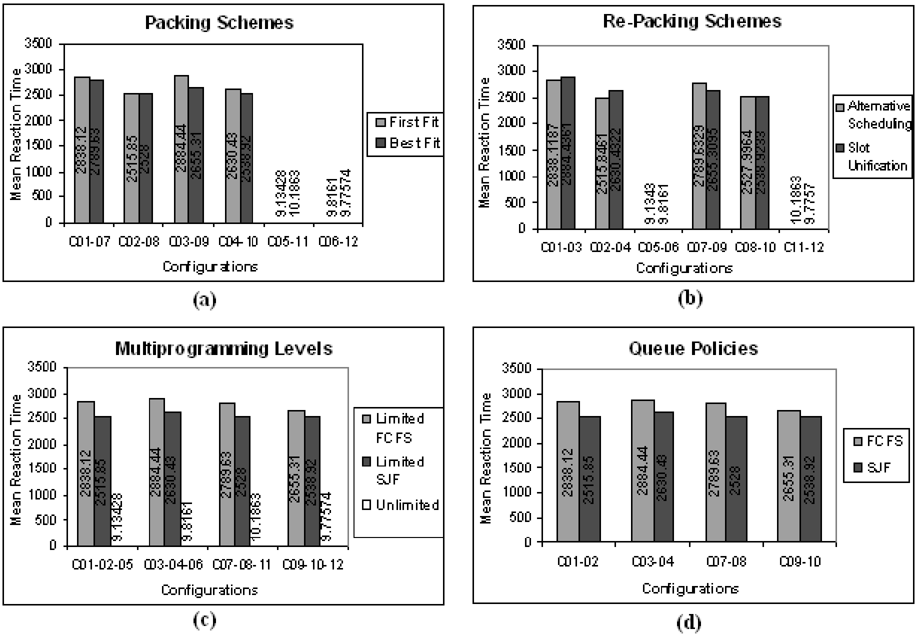
Regarding the multiprogramming level, we reached two conclusions: the unlimited is better for HH and LL workloads (Fig. 5 (c)), but it is very bad for HL and LH workloads (Fig. 5 (d)). With an unlimited multiprogramming level, for each new job that does not fit into the matrix, a new time slot is created. At the end of the simulation, as the load is high, a large number of time slots existed. In this case, the big jobs (high parallelism level) are the long ones. So when the small jobs terminate, the idle space is significantly smaller than the space occupied by the big jobs, that is, the fragmentation is low and the utilization is maximized.

When we use LH and HL workloads, each matrix slot will be occupied by long and short jobs. As time goes by, the short jobs will end, leaving idle spaces on the matrix. In this case, the big jobs can not be the long ones, so a big space can become idle. Even if we use re-packing schemes, the fragmentation becomes high.

With reference to the queue policies, the SJF policy presented a high utilization in all cases. When we remove the short jobs first, there is a higher probability that short idle slots exist where they can fit. Using the FCFS policy, if the first job is a big one, it can not fit into the matrix, thus, preventing other short jobs from being executed. So some slots become idle and the utilization low.

5.2 Reaction Time

In Fig. 7 we present the relative mean reaction time of jobs among each configuration for all workloads. Packing schemes have a very small influence on reaction time, because they depend on the new job that came from the wait queue. According to Fig. 6 (a), we can say that the both packing schemes are quite similar. The same happens to the re-packing schemes, because the defragmentation occurs after the beginning of the job's execution (Fig. 6 (b)).



(a)

(b)

(c)

(d)

Fig. 6. Mean jobs reaction time considering the (a) packing schemes; (b) re-packing schemes; (c) multiprogramming levels; (d) queue policies.

The multiprogramming level has a direct influence on the reaction time of jobs, because with an unlimited number of slots, a job can always fit to the matrix without waiting in the queue. In the worst case, the reaction time of a job will be equal to the number of slots multiplied by the slot quantum. Configurations with an unlimited multiprogramming level present an insignificant reaction time in comparison with those with a limited multiprogramming level, as shown in Fig. 6 (c).

With reference to the queue policies, on average, the SJF is better than FCFS, because the jobs in the queue waste less time waiting to be removed to the matrix and start their execution (Fig. 6 (d)).

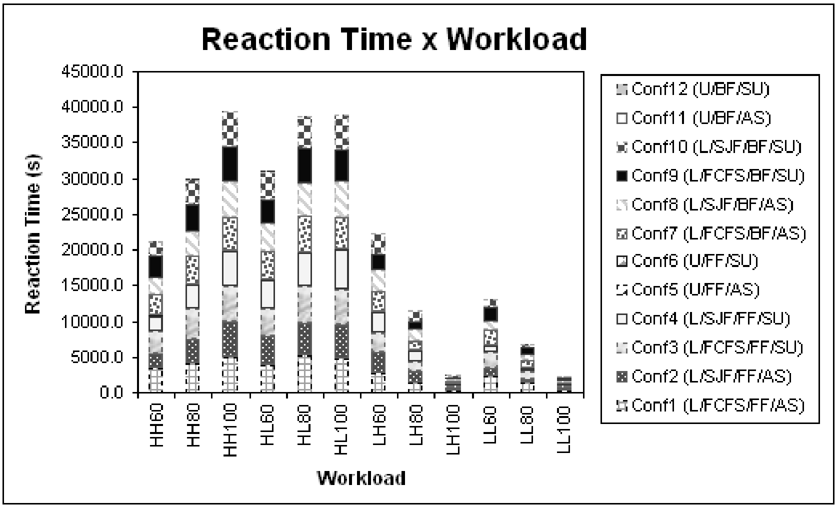


Fig. 7. The relative mean jobs reaction time among each configuration for all workloads.

5.3 Response Time

In Fig. 8 we present the relative mean response time of jobs among each configuration for all workloads. According Fig. 9 (a) and (b), the results showed that both packing and re-packing schemes are equivalent. The multiprogramming level has a direct influence on the response time.

When the multiprogramming level is unlimited, the jobs have a short reaction time, but the execution time tends to be higher, because there are more available time slots (providing more concurrency). The execution time of a job is increased by the reaction time if the multiprogramming level was limited. On average, configurations with an unlimited multiprogramming level are worse than those a with limited one, that is, more jobs concurring in the matrix is worse than more jobs waiting in the queue, but there are some exceptions.

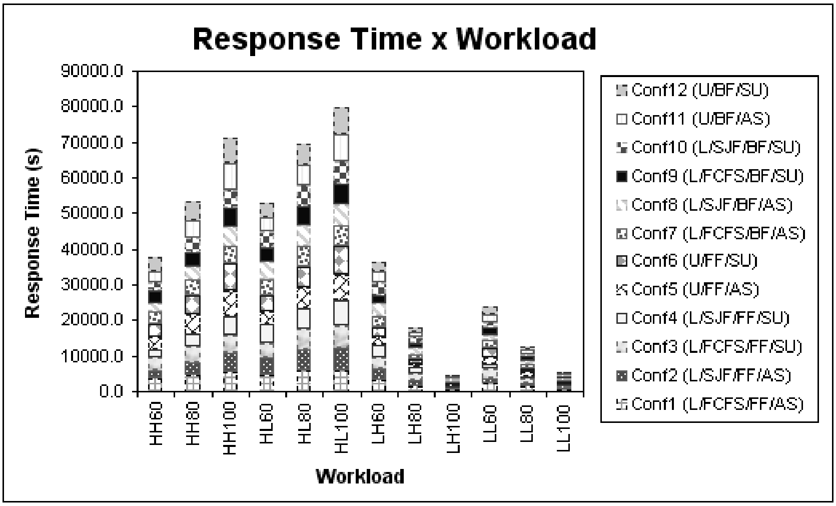
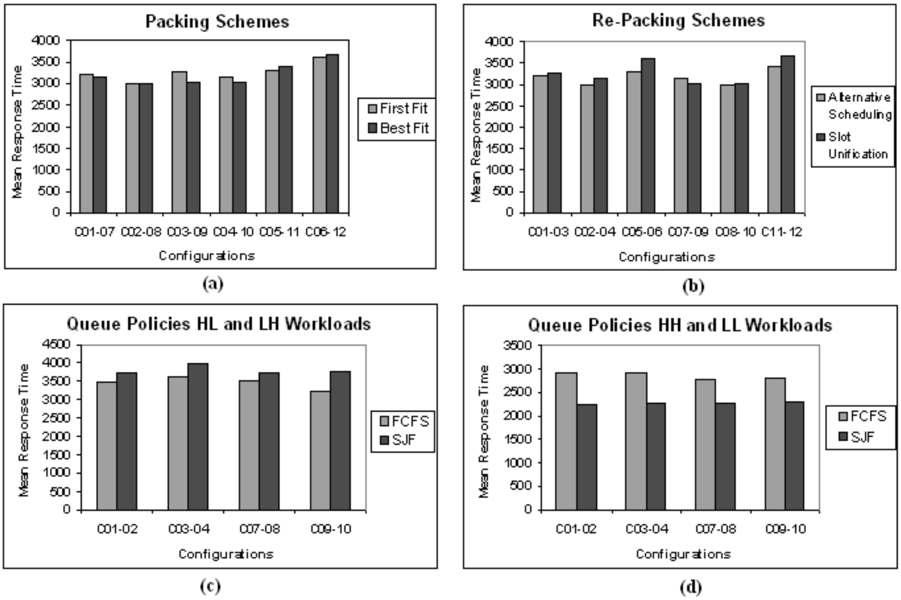


Fig. 8. The relative mean jobs response time among each configuration for and all workloads.



(a)

(b)

(c)

(d)

Fig. 9. Mean response time of jobs considering the (a) packing schemes; (b) re-packing schemes; queue policies for (c) HL and LH workloads and (d) HH and LL workloads.

Generally, we believe that unlimited multiprogramming is always better if we are not considering memory paging, but in Fig 10, we see a simple example in which the mean jobs response time is better (smaller) for a limited multiprogramming level. Suppose a workload composed of three jobs with 2 tasks (each one) and an execution time equal to 2.1 seconds; and an Ousterhout matrix with two columns, a time slice equal to 1 and a limited multiprogramming level equal to 2. In this example, when a job finishes before the time slice ends, a new time slice starts.

In Fig. 10, we observe the following response times for limited multiprogramming level: Job1 = 4.1s; Job2 = 4.2s; Job3 = 6.3s; mean = 4.86s. And for unlimited multiprogramming level we observe the following response times: Job1 = 6.1s; Job2 = 6.2s; Job3 = 6.3s; mean = 6.2s. So, we note that the use of a limited multiprogramming level can achieve better response times for a certain workload, even not considering the memory paging.

With reference to the queue policies, we reached two conclusions: the SJF policy is better for HH and LL workloads (Fig. 9 (c)) and the FCFS policy is better for HL and LH workloads (Fig. 9 (d)). In the first case, the LL jobs are initially executed and terminated quickly. Thus, HH jobs wait less time in the wait queue, reducing their reaction time and consequently their response time.

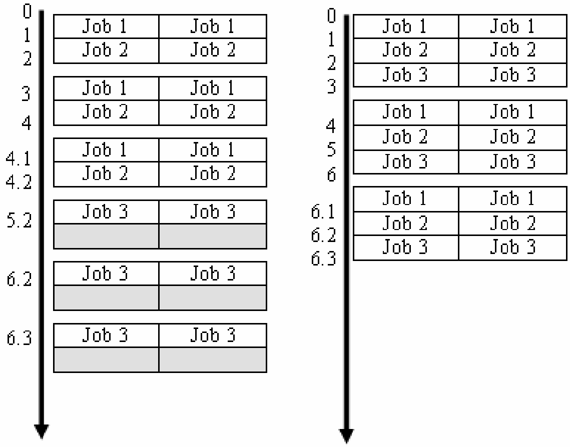


Fig. 10. The simple workload execution using limited and unlimited multiprogramming levels.

In the last case, when we use the SJF policy, the HL jobs are executed first. So LH jobs have to wait so much time in the queue, which increases their reaction time and consequently their response time.

5.4 Slowdown

In Fig. 11 we present the relative mean slowdown of jobs among each configuration for all workloads. Based on past analysis, we conclude that both the packing and repacking schemes are equivalent. The slowdown is based on response time and consequently on reaction time. When the multiprogramming level is unlimited, the response time is almost equal to the execution time. Thus, the slowdown value tends to 1.

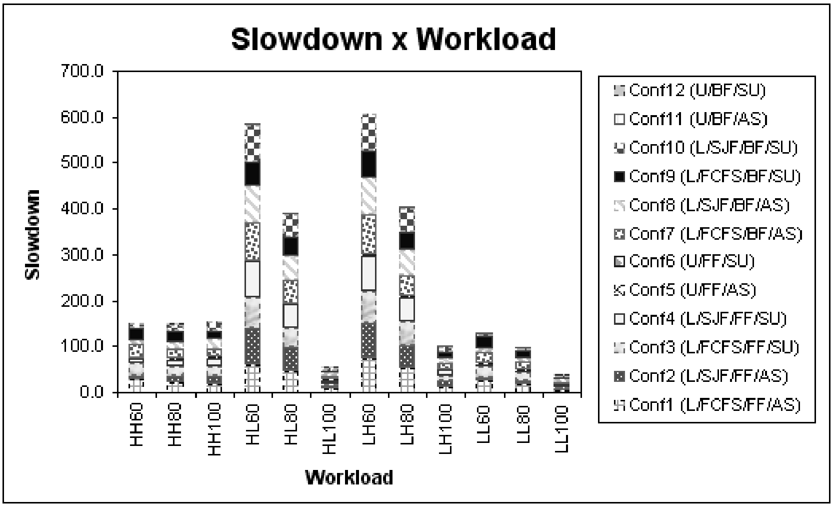


Fig. 11. The relative mean jobs slowdown among each configuration for all workloads.

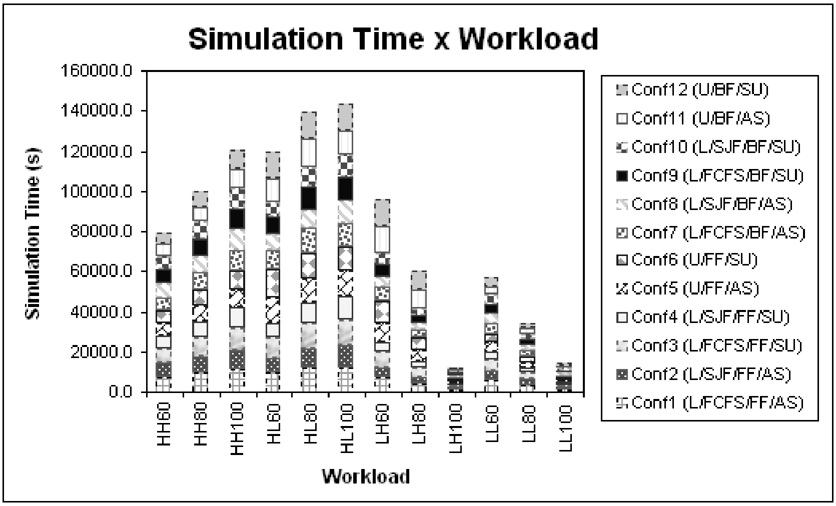


Fig. 12. The relative mean simulation time among each configuration for all workloads.

5.5 Simulation Time

In Fig. 12 we present the relative mean simulation time among each configuration for all workloads. The simulation time depends directly on the utilization. So, all observations and analyses of the utilization metric may be extended to the simulation time.

5.6 RGSA Analysis

In order to analyze the performance of RGSA, we need to compare it to each configuration or traditional gang scheduling algorithm individually. As we said in the proposal of RGSA, based on log files, the workload is divided in sub-workloads that fit into one of the workload classes (ex: LL100%). Then the CCL evaluates the entry parameters, reconfiguring RGSA to the best configuration. As we are not using a trace, we analyze RGSA for all proposed workloads.

Table 5. Speedup, in percentage (%), of the RGSA performace when compared to each configuration for a workload composed of 12 described workloads.

Metrics	**Utilization**	**Reaction Time**	**Response Time**	**Slowdown**	**Simulation Time**	**Mean**
Configurations						
Conf 01	18.8153	99.6937	18.1988	96.6667	19.4711	**50.5691**
Conf 02	5.5793	99.6545	12.7879	96.3408	5.3772	**43.9479**
Conf 03	20.2381	99.6986	20.1357	96.6585	21.5047	**51.6471**
Conf 04	7.1421	99.6695	16.6365	96.4005	8.1059	**45.5909**
Conf 05	12.2866	4.8307	21.1611	0.0566	18.9160	**11.4502**
Conf 06	17.0448	11.4411	27.6418	0.1415	25.2068	**16.2952**
Conf 07	16.0706	99.6884	16.7549	96.9840	17.8183	**49.4632**
Conf 08	5.1443	99.6561	12.9742	96.4882	5.2786	**43.9083**
Conf 09	14.6709	99.6726	13.7521	96.3212	15.1375	**47.9109**
Conf 10	6.0941	99.6576	13.6044	96.4374	5.8391	**44.3265**
Conf 11	16.6605	14.6595	23.2316	0.2179	27.7065	**16.4952**
Conf 12	19.0915	11.0755	28.8892	0.0591	29.9556	**17.8142**
Mean	**13.2365**	**69.9498**	**18.8140**	**64.3977**	**16.6931**	**36.6182**

In Table 4, we observe that on average, considering all metrics, RGSA is 36.61% better than the 12 traditional gang scheduling algorithms. Note that if we had chosen Conf5 (the best configuration on average), RGSA would still be 11.45% better.

Now, we analyze another example, in which the workload is composed of HL60 and LH60 workloads. According to Table 5, on average, the speedup of RGSA increases to 41.53%, and with reference to Conf5, this speedup increases to 18.83 considering all 5 metrics. If we consider only the utilization metric, the speedup of RGSA over Conf5 increases from 18.83% to 42.32%. In the last case, we note that Conf5 (the best on average) would be worse than Conf3, which was previously considered the worst configuration. With these examples, we show that the use of reconfiguration concepts in gang scheduling algorithms may provide a high speedup over traditional gang scheduling algorithms.

In these examples, we considered that there weren't reconfiguration overheads, neither wrong workload classifications nor classification overheads. The reconfiguration overhead is insignificant, it is just the time spent to execute a switch case structure (to select the more appropriated configuration) and fit some specific blocks in the frames to change the configuration. The classification of the workload based on log files in sub-workloads and the CCL table's update can be done using idle cycles. But how to classify the workload and the consequences of wrong classifications are open

Table 6. Speedup of RGSA, in percentage (%), when compared to each configuration for a workload composed of HL60 and LH60 workloads.

Metrics Configurations	Utilization	Reaction Time	Response Time	Slowdown	Simulation Time	Mean
Conf 01	26.7836	99.7605	22.5123	98.4916	27.1878	**54.9471**
Conf 02	0.8016	99.7739	31.9511	98.6920	0.7671	**46.3972**
Conf 03	**29.0247**	99.7647	24.4469	98.4859	29.0702	**56.1585**
Conf 04	4.2309	99.7809	34.1388	98.7164	4.3209	**48.2376**
Conf 05	**42.3255**	4.0597	3.8764	0.0324	43.8949	**18.8378**
Conf 06	50.3595	5.2884	19.6931	0.0742	51.3580	**25.3546**
Conf 07	28.2399	99.7709	25.0613	98.7869	28.8798	**56.1478**
Conf 08	0.7319	99.7739	31.6864	98.7537	0.8703	**46.3632**
Conf 09	19.0918	99.7198	12.7145	98.1925	18.9861	**49.7409**
Conf 10	1.0836	99.7729	31.9171	98.7256	0.8443	**46.4687**
Conf 11	52.1549	0.0000	10.7741	0.0984	53.2991	**23.2653**
Conf 12	54.4356	0.7533	21.7874	0.1247	55.1504	**26.4503**
Mean	**25.7720**	**67.3516**	**22.5466**	**65.7645**	**26.2191**	**41.5307**

and interesting topics to a more detailed research. In despite of these overheads and costs, the speedup may be great enough to make RGSA a good alternative.

6 Conclusion

In this paper, we defined, proposed, developed, implemented (in a simulation tool) and analyzed the performance of RGSA by simulation. As general conclusions about the RGSA frames, we can highlight:

1. Packing Schemes Frame. Considering all metrics, on average, both packing schemes (first fit and best fit) presented an equivalent performance, as found in [6]. It suggests that other constructive blocks may be used.

2. Re-Packing Schemes Frame. Considering all metrics, on average, both re-packing schemes (slot unification and alternative scheduling) presented an equivalent performance. It suggests that other constructive blocks may be used.

3. Multiprogramming Levels Frame. Considering the metrics: utilization and simulation time, the unlimited multiprogramming level presented a better performance to HH and LL workloads, and the limited one for the HL and LH workloads. For reaction time and slowdown metrics, the unlimited multiprogramming level presented a best performance in all cases. Finally, considering the response time metric, on average, the limited multiprogramming level was the best.

4. Queue Policies Frame. Considering the utilization and simulation time metrics, the SJF policy was always better than the FCFS. For reaction time and slowdown metrics, on average, the SJF policy presented a better performance, but in some specific cases FCFS was better than the other. Finally, considering the response time metric, the SJF policy presented a better performance for the HH and LL workloads and the FCFS policy for the HL and LH workloads.

On average, the performance of RGSA was around 40% better than the other traditional gang scheduling algorithms for all tested workloads. One of the most important results was to show that depending on the selected metric and workload, the best algorithm on average for all situations (Conf5) may be worse than the worst algorithm on average (Conf3). In our simulations, the performance of RGSA was 42.32% better than the one of Conf5 in a specific situation. So, the use of a reconfigurable algorithm may largely improve the system's performance.

In our specific case, the longest simulation took about 13000 seconds (3 hours and 36 minutes). So we got to reduce the simulation time in 40% (1 hour and 26 minutes) using RGSA. But in real systems, a workload may execute for a week. In that case, a reduction of 40% would mean to reduce the workload execution time in 2.8 days

In this paper, we proposed a model or architecture of a reconfigurable algorithm that was applied in gang scheduling. But this model can be applied on any other scheduling algorithm. Using a reconfigurable algorithm, developers don't need to create a monolithic algorithm based on behavior description that works well for all situations. They may design a set of structural algorithms or parts of an algorithm that every one is optimized for a different situation.

The **main contributions** of this paper are: the definition, proposal, implementation and performance analysis of RGSA, comparing it with other traditional gang scheduling algorithms for different workloads. As **future works** and **open researches** we can highlight: the inclusion of new frames and blocks in RGSA; an adaptive CCL; compare RGSA with backfilling schemes; study on how to classify the workload found in log files into sub-workloads; tests with other workloads, simulation with a bigger number of jobs, different loads, other jobs algorithm models and real tests.

References

1. Batat, A., Feitelson, D.: Gang Scheduling with Memory Considerations. IEEE International Parallel and Distributed Processing Symposium. (2000) 109-114

2. Chapin, S.J. et all: Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. Job Scheduling Strategies for Parallel Processing. (1999) 67-90

3. Compton, K., Hauck, S.: Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Survey. (2002)

4. Dehon, A.: The Density Advantage of Configurable Computing. IEEE Computer, Vol. 33, No. 4. (2000)

5. Feitelson, D., Rudolph, L.: Evaluation of Design Choices for Gang Scheduling using Distributed Hierarchical Control. Journal of Parallel & Distributed Computing (1996) 18-34

6. Feitelson, D. G.: Packing Schemes for Gang Scheduling. Job Scheduling Strategies for Parallel Processing. (1996) 89-110

7. Feitelson, D.G.: A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657). IBM T. J. Watson Research Center (1997)

8. Feitelson, D., Rudolph, L.: Metrics and Benchmarking for Parallel Job Scheduling. Job Scheduling Strategies for Parallel Processing. (1998) 1-24

9. Feitelson, D. G. and Naaman,, M.: Self-tuning Systems. IEEE Software. (1999) 52-60

10. Feitelson, D.: Metric and Workload Effects on Computer Systems Evaluation. IEEE Computer. (2003) 18-25

11. Franke, H., Jann, J, Moreira, J., Pattnaik, P., Jette, M.: An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. ACM/IEEE Conference on Supercomputing. (1999)

12. Frachtenberg, E., Feitelson, D.G., Petrini, F. and Fernandez, J.: Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. 17th International Parallel and Distributed Processing Symposium. (2003)

13. Góes, L. F. W., Martins, C. A. P. S.: RJSSim: A Reconfigurable Job Scheduling Simulator for Parallel Processing Learning. 33rd ASEE/IEEE Frontiers in Education Conference. Colorado (2003)

14. Góes, L. F. W., Martins, C. A. P. S.: Proposal and Development of a Reconfigurable Parallel Job Scheduling Algorithm. Master's Thesis. Belo Horizonte, Brazil (2004) (in portuguese)

15. Jann, J., Pattnaik, P. and Franke, H.: Modeling of Workload in MPP's. Job Scheduling Strategies for Parallel Processing. (1997) 95-116

16. Martins, C. A. P. S., Ordonez, E. D. M., Corrêa, J. B. T., Carvalho, M. B.: Reconfigurable Computing: Concepts, Tendencies and Applications. SBC JAI - Journey of Actualization in Informatics. (2003) (in portuguese)

17. Streit, A.: A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. 8th WJSSPP, Springer Verlag LNCS Vol 2537. (2002) 1-23

18. Wiseman, Y., Feitelson, D.: Paired Gang Scheduling. IEEE Transactions Parallel and Distributed Systems. (2003) 581-592

19. Zhang, Y., H. Franke, Moreira, E.J., Sivasubramaniam, A.: Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. IEEE International Parallel and Distributed Processing Symposium. (2000)

20. Zhou, B. B., Brent, R. P.: Gang Scheduling with a Queue for Large Jobs. IEEE International Parallel and Distributed Processing Symposium. (2001)

Time-Critical Scheduling on a Well Utilised HPC System at ECMWF Using Loadleveler with Resource Reservation

Graham Holt

ECMWF*
Shinfield Park
Shinfield Road
Reading
Berkshire
UK
RG2 9AX
G.Holt@ecmwf.int

Abstract. This article is written in the context of running a suite of time-critical operational numerical weather prediction batch jobs, along with a substantial number of research batch jobs on a large IBM Cluster 1600 system. The batch subsystem used is IBM's LoadLeveler incorporating a little known feature called Resource Reservation.

The article describes how the mixture of operational and research parallel batch jobs are scheduled to run on the 117 nodes provided, and how Resource Reservation for operational jobs is performed without reference to job class. Where research parallel batch jobs are jobs requesting more than 1 CPU and must run consistently to ensure resources are released predictably. Note - information is given explaining how consistent runtimes are achieved.

1. Background Information

Before 2001, ECMWF had no experience of Loadleveler, having previously used systems from CDC, CRAY (NQE) and Fujitsu (NQS). So ECMWF's experience of Loadleveler is limited to the needs of the system described, and the scheduling strategy, devised in early 2002, was kept simple to minimise the learning curve/time. More recently additional IBM Server systems again using Loadleveler have been installed, but experience has shown there is little common ground between the philosophy of scheduling batch and interactive work on Servers and the philosophy of scheduling high-performance parallel batch jobs on a Cluster. So my Loadleveler/scheduling experience is therefore fine-tuned in a blinkered way to the

* *ECMWF (European Centre for Medium-Range Weather Forecasts) was founded in 1973 and is funded by 25 European Countries (18 original participating Member States and 7 cooperating Member States) where Medium-Range Weather Forecasts concentrate on the period 4 – 10 days ahead. The European Weather Centre, as it is more commonly known, is located 35 miles west of London, England on the outskirts of a town called Reading. See http://www.ecmwf.int/about/overview/ for more information.*

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 102-124, 2005.

needs of well balanced, highly parallelised batch jobs on a large Cluster to ensure good performance and consistent runtimes. Importantly the over-subscription of processes per node is not allowed, shared memory use is only permitted by up to 4 jobs per node having a maximum 8 processes, as long as the total real memory requested by all the jobs does not exceed the total real memory available, and memory paging when detected is reported as very undesirable, even more so if job performance (CPU utilisation) appears to be compromised.

In 2002 the Loadleveler feature Resource Reservation was not known to ECMWF. At that time when operational jobs were about to be run nodes were 'reserved' by draining 'user' batch job classes so that no new 'user' jobs could start. But this manual, time consuming and often complicated method was frequently found to be wasteful, with nodes being left idle unnecessarily. Then in 2003 a decision was taken to introduce a tighter operational schedule and as a result plans were made to develop an automated resource based scheduling scheme that would also overcome the known weaknesses in the manual system in use.

It was clear even in 2002 that predictable runtimes were essential for backfill to maximise node utilisation and for predictable node release. So it was agreed the scheduling scheme should be knowledge based and use predicted wall_clock_limits derived from historical run-time data. By good fortune a lot of work had already gone into creating tools and displays that enabled runtime data to be captured, visualised, and made available to enhance job selection and empower backfill. Importantly, having no previous understanding of IBM backfill, tools had already been created to monitor the results of backfill so that the way it worked could be studied and understood. So plans were made to enhance these displays for scheduling purposes during operational periods. But most importantly a dynamic reservation based scheduling scheme was sought, a scheme independent of physical nodes and physical classes. For this IBM suggested using Resource Reservation.

2. Overview of Operational Needs at ECMWF

At ECMWF, many types of operational forecasts are run on a daily, weekly and monthly basis, and hundreds of thousands of products are sent (disseminated) to the Member States each day. Twice a day for about 90 minutes all 117 nodes (936 CPUs) are reserved for and used by operational batch jobs (see Figure 1 below).

Please note – when all 936 CPUs are reserved for operational jobs, some CPUs become unallocated for a few seconds as jobs complete and new jobs start, and at times due to the job-mix at least 1 node (8 CPUs) may not be used for some minutes. So the average maximum use of around 920 CPUs out of 936 is seen as a very good achievement.

However when operational batch jobs are not being run or do not require all of the 'parallel' nodes, every attempt is made to fully utilise 'resources' by running research user's batch jobs

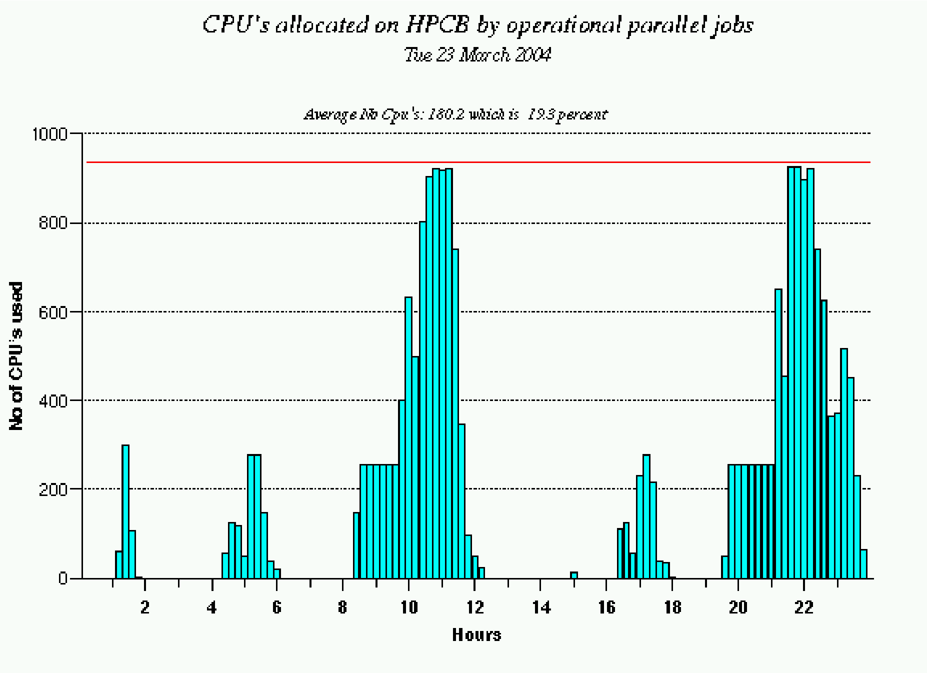


Figure 1 - the average number of CPU's allocated to parallel operational jobs, plotted over 12 minute intervals, where for parallel jobs 936 CPUs (the red line) is the maximum possible.

3. A Mixture of Parallel Operational and Research Batch Jobs

Please note there is little difference between the computational needs and performance characteristics of research batch jobs and their operational equivalent jobs. The codes used are almost identical. The big difference is operational suites of jobs are run usually twice or four times a day and each time use as input the most recent world-wide observations acquired in the preceding few hours. But research experiments differ in that they use historical data (not real-time data acquisition), have no need to create end-user products and over a period of many days submit 28 data assimilations and forecasts, using alternately 00z and 12z data over a 14 date period. Normally there are from 10 to 15 research experiments running simultaneously and these plus other smaller jobs easily utilise the 117 nodes available (see Figure 2 below) whenever resources are not being used operationally. Please note – 00z data is collected globally and simultaneously at 00z GMT, likewise 12z data at 12z GMT.

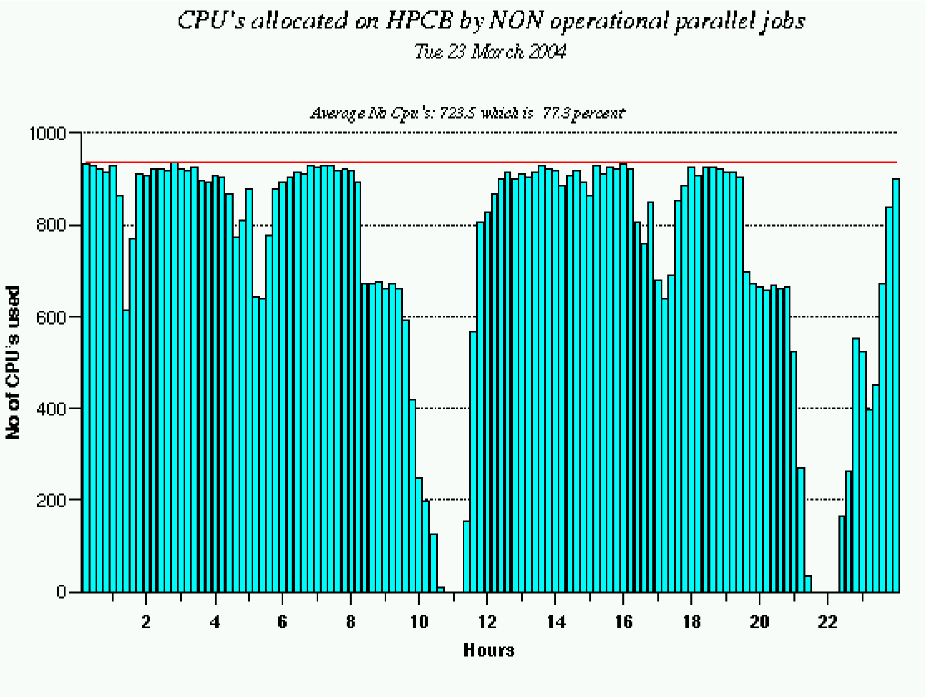


Figure 2 – the average number of CPUs allocated to non-operational jobs, before Resource Reservation was introduced, plotted over 12-minute intervals, where 936 CPUs (the red line) is the maximum possible.

Please note in Figure 2 above at 02:48, a 12-minute period when all 936 CPUs are allocated.

4. Scheduling Requirements – Predictable Node Release

One additional function of pre-operational testing is to detect research jobs that do not perform well or do not exhibit consistent runtimes in keeping with the existing operational schedule. Quite simply, jobs (new code) from research experiments cannot progress beyond the research stage to become operational if they do not perform well enough. So a lot of effort has been put into ensuring all parallel jobs have consistent runtimes, and as a result each job confirms by running in the given time that there are no I/O bottlenecks, the GPFS filesystems used are performing consistently, the high-performance internal switch network is performing consistently, that the jobs have not been slowed by paging to local disk and lastly that the jobs (the new code) perform as expected too.

As a result, in a research experiment of jobs run 28 times, at least 95% of the forecast jobs are not expected to vary by more than ±1%, with the preceding sets of 28 data assimilation jobs varying by no more than ±3%. The big bonus being, once 3 or 4 sets of jobs for an experiment have completed, the subsequent 24 sets of jobs become health checks (each a diagnostic) for good system performance. It is true when a job has run less than twice it has an unknown runtime (shown on displays as a wall_clock_limit of *1-day) but other jobs with similar job-names (for example ifstraj_uptraj_0) are likely to have a known run-time, giving an indication when new jobs might complete.

Importantly the golden rule before submitting an experiment or individual parallel job is – if the data needed by a parallel job is not on a GPFS filesystem, a serial job must first be run to obtain the initial data, and the data obtained must be written to a high-performance GPFS filesystem.

As a result all parallel jobs in the Cluster can be scheduled independent of all other systems and will end as predicted.

If at any time a parallel job runs for longer than expected or the job is flagged as idle, the job is displayed in red on an operator display and immediately investigation begins. What has to be determined is – is the problem a function of the job, the environment or the system?

5. The Mission – To Keep the System Fully Utilised Yet Run Operational Work to a Tight Schedule

5.1 Fully Utilised

Keeping the system lightly loaded to make it easy to run operational work on time is not our style (is for wimps). Users in the Research Department have always been able to expand their experimentation to fully utilise the system, and giving them the service they want by fully utilising the system is very rewarding.

5.2 Keeping to Operational Schedules

The plans made in 2003 to run the operational suites to a more demanding schedule made the introduction of an automatic and class-independent resource reservation system essential. The previous ‘on demand’ scheduling scheme could not guarantee consistent start-times, the variability often exceeded the 15 minute requirement, and as a result there was often insufficient recovery time should jobs fail and need to be rerun.

6. The Main Objective

The main objective of the reservation system was then defined like this: Resource reservation is needed to ensure that the day-to-day variation in the end-times of operational forecasts and the generation of products does not exceed 15 minutes compared with the predicted optimum time, whilst keeping node occupancy close to 96% (which is what had been achieved beforehand).

Fortunately the basic building blocks were all in place based on the predicted run-time for most jobs, the standard design of most experiments and the repetitive consistently running job mix, giving the ability to accurately assess node release and node availability times.

6.1 The Complexity of Operational Suites Should Not Be Underestimated

Until now I have trivialised operational suites at ECMWF by talking only about the more substantial time-critical computational jobs. The reality is each suite contains 1000's of jobs that run on multiple systems, have complex inter-dependencies, multiple-event triggers, time triggers and late flags. Amongst other things, operational jobs acquire observations (data), analyse and check the validity of this data, compute the relevant initial datasets used by the forecast, execute the forecasts, save data at many stages throughout the process, create end-user products, verify the results, archive the results, plot data, send products to users with many operations taking place in parallel. One concern with such a complex set-up, was tighter schedules would lead to new bottlenecks and a complete loss of flexibility, which would reduce efficiency and system utilisation. So the brief was altered subtly to request an automatic system that used the 15 minutes of flexibility if this increased system utilisation.

7. Resource Reservation

7.1 Defining the Need for Resources and Benefits of Resource Reservation

The CPU allocation averages shown previously in Figure 1 indicate that most of the time none of the Cluster resources are used by operational jobs, and show during the main periods (08:20 to 11:40 and 19:45 to 23:35) the need for the resources provided to increase in 3 stages from 0% to 30%, from 30% to 60%, and finally from 60% to100% for about 90 minutes. But what Figure 1 does not show are the 2 small periods when CPU use drops for just a few minutes from 30% to 0% before rising to 60%, and later from 60% down to 30% before rising up to 100%. If resources are not reserved or classes drained, user jobs flood the system.

Past experience showed it is possible, but by no means easy, to drain classes on some nodes to limit the use of resources by non-operational jobs to 40%, whilst the operational use dips briefly from 60% to 30% before rising to 100%. By comparison dynamic reservation independent of class using Resource Reservation provides a much needed simplicity. One might even say Resource Reservation brings elegance to the solution.

And by the same token with one operation, as soon as all operational jobs to be run are submitted, and because operational jobs are selected first by virtue of highest SYSPRIO, it is possible as soon as all operational jobs are submitted, to cancel (reset) reservation and make all non-operational jobs candidates for selection before the final operational jobs have started, which enables backfill to take place.

7.2 Describing Resource Reservation

Resource Reservation for the timely running of operational jobs combined with optimum system utilisation has as a core function the ability to request in advance a reduction to the resources available to non-operational jobs, irrespective of the class the jobs are in, whilst taking into account

- the requirements of operational jobs (scheduled start-time, CPUs needed, elapsed time),
- the timely release of resources (expected elapsed/end times of research jobs in execution)
- the possibility of late starts (if research jobs do not appear to end soon enough)

Additionally the scheme should take into account

- the flexibility of the operational schedule
- how close to or far behind the optimum schedule today's operational runs are
- how consistent the historical runtime data is for the jobs that are running,
- that it is possible to force some jobs to write checkpoint files then kill them without loss.

Finally the scheme should release resources for use by non-operational jobs whenever possible to optimise the use of all resources. The success of this is shown in Figure 3 below.

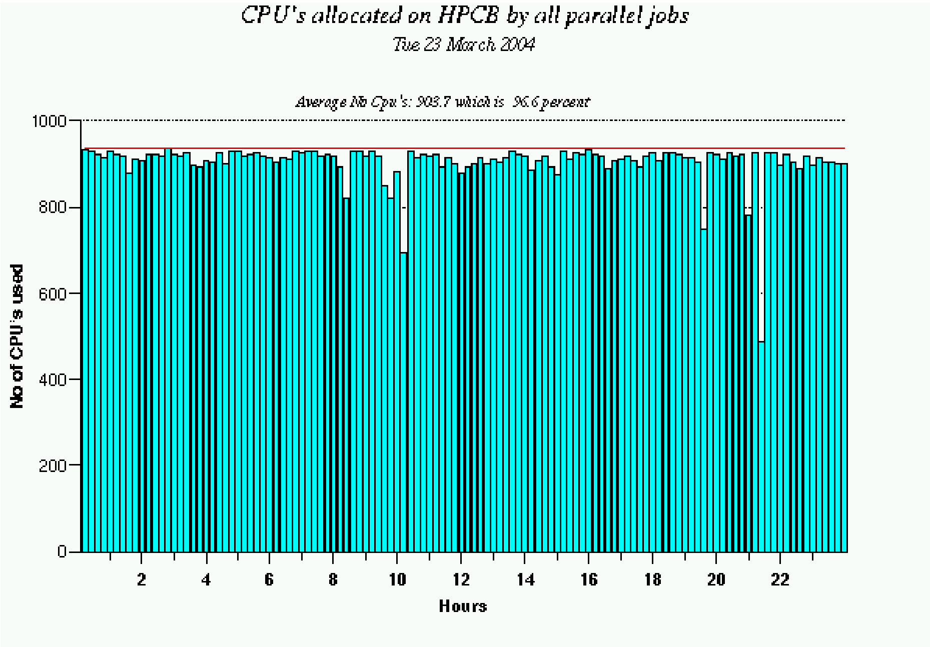


Figure 3 - the allocation of CPU's to all jobs before Resource Reservation was introduced, plotted over 12-minute intervals, where the red line (936 CPUs) is the maximum number that parallel jobs can use.

8. Summary So Far

The key to everything is predictability. ECMWF's business relies on the timely production of weather forecast products (our weather predictions) so not surprisingly

ECMWF has a strong desire for all jobs to run predictably. Otherwise how do you keep to a schedule. And once the causes of variable job runtimes are eliminated (as they have been) and jobs run with little run-time variation, the data available enables

- backfill to work most effectively.
- the runtime and end-time of user batch jobs to be monitored.
- most batch jobs to act as diagnostic checks on system performance as well as on job performance.
- resources to be reserved sufficiently in advance to ensure operational jobs start optimally.

9. Defining the Core Elements Required for Effective Resource Reservation

Analysis of the associated issues have identified 4 main elements, which are listed below so that they may be checked and ticked off. In doing so one can be sure all problems have been identified and solutions put in place. It is true there is some duplication of what has gone before, but this is necessary. The 4 elements being

A. The need to know in advance the resources needed by operational jobs and the time at which these jobs should start.

B. The need to ensure that non-operational jobs in execution can be guaranteed to complete and release the resources in a predictable way (there is little point doing this if, in order to achieve it, many jobs have to be cancelled prematurely).

C. The need to have processes that take the information about operational jobs and running jobs and use this along with other information to dynamically reserve resources, the purpose being to restrict or stop non-operational jobs from starting, then release resources.

D. The need to be able to visualise what is happening and verify that it is working as designed.

10. Expanding on the Core Elements for Effective Resource Reservation

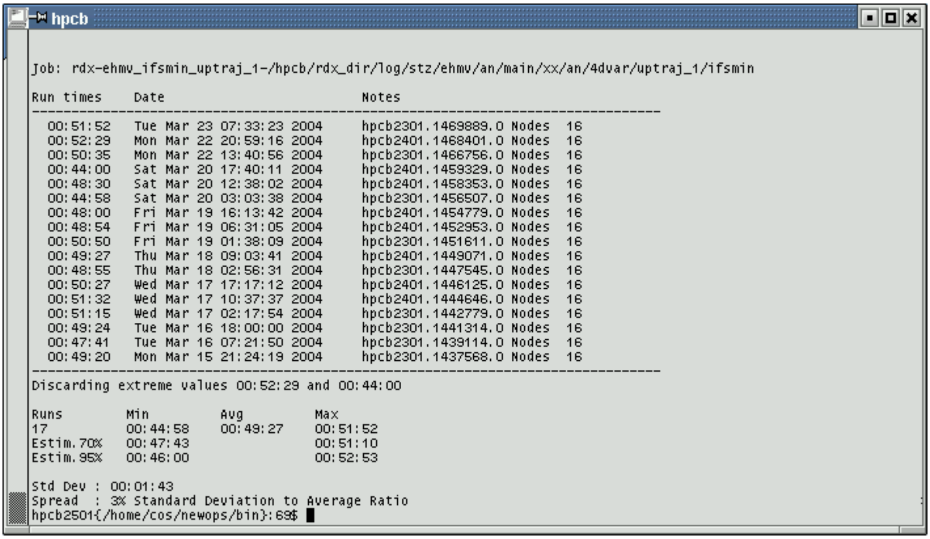
Taking the headings from 9. (and covering A. and B. very quickly) we have:-

A. The need to know in advance what resources are required by operational jobs and the time at which these jobs should start.

The requirements of operational jobs, in terms of resources, start time, elapsed time and dependencies on other factors are well known. Nothing more needs to be added.

B. The need to ensure that non-operational jobs in execution can be guaranteed to complete and release the resources in a predictable way.

As mentioned before, predictability is paramount. There is no point reserving resources if job end-times are unreliable. Good utilisation is compatible with tight schedules only if you have a predictable set of non-operational jobs. The key elements of ECMWF's success are the independence of parallel jobs, consistent job performance and consistent system performance. And to illustrate the point Figure 4 , which follows, shows historical run-time data for an assimilation job and the predicted Min and Max runtimes. Where the predicted 95% Max value is used for scheduling and backfill purposes. The runtime data for the job shown in Fig. 4 below does not quite have the quoted runtime range of ±3%, and this is commented on later.



```
Job: rdx-ehmv_ifsmin_uptraj_1-/hpcb/rdx_dir/log/stz/ehmv/an/main/xx/an/4dvar/uptraj_1/ifsmin

Run times    Date                          Notes
------------------------------------------------------------------------------
  00:51:52   Tue Mar 23 07:33:23 2004      hpcb2301.1469889.0 Nodes  16
  00:52:29   Mon Mar 22 20:59:16 2004      hpcb2401.1468401.0 Nodes  16
  00:50:35   Mon Mar 22 13:40:56 2004      hpcb2301.1466756.0 Nodes  16
  00:44:00   Sat Mar 20 17:40:11 2004      hpcb2401.1459329.0 Nodes  16
  00:48:30   Sat Mar 20 12:38:02 2004      hpcb2401.1458353.0 Nodes  16
  00:44:58   Sat Mar 20 03:03:38 2004      hpcb2301.1456507.0 Nodes  16
  00:48:00   Fri Mar 19 16:13:42 2004      hpcb2401.1454779.0 Nodes  16
  00:48:54   Fri Mar 19 06:31:05 2004      hpcb2401.1452953.0 Nodes  16
  00:50:50   Fri Mar 19 01:38:09 2004      hpcb2301.1451611.0 Nodes  16
  00:49:27   Thu Mar 18 09:03:41 2004      hpcb2401.1449071.0 Nodes  16
  00:48:55   Thu Mar 18 02:56:31 2004      hpcb2301.1447545.0 Nodes  16
  00:50:27   Wed Mar 17 17:17:12 2004      hpcb2401.1446125.0 Nodes  16
  00:51:32   Wed Mar 17 10:37:37 2004      hpcb2401.1444646.0 Nodes  16
  00:51:15   Wed Mar 17 02:17:54 2004      hpcb2301.1442779.0 Nodes  16
  00:49:24   Tue Mar 16 18:00:00 2004      hpcb2301.1441314.0 Nodes  16
  00:47:41   Tue Mar 16 07:21:50 2004      hpcb2301.1439114.0 Nodes  16
  00:49:20   Mon Mar 15 21:24:19 2004      hpcb2301.1437568.0 Nodes  16
------------------------------------------------------------------------------
Discarding extreme values 00:52:29 and 00:44:00

Runs        Min         Avg         Max
17          00:44:58    00:49:27    00:51:52
Estim.70%   00:47:43                00:51:10
Estim.95%   00:46:00                00:52:53

Std Dev : 00:01:43
Spread  : 3% Standard Deviation to Average Ratio
hpcb2501{/home/cos/newops/bin}:69$ █
```

Figure 4 – historical run-time data and predicted run-time estimate (95%/Max) for job ehmv_ifsmin_uptraj_1

Figure 4 shows 17 runtimes.

Discarding the shortest and longest, the average runtime (under Avg) is shown to be 00:49:27.

The expected ±3% would give Min 00:47:59 and Max 00:50:55. Sadly 6 runtimes fall outside this range.

The larger than expected spread of 00:44:58 to 00:51:52 means the predicted 95% Max runtime of 00:52:53 exceeds by 24 seconds the known longest run of 00:52:29, and exceeds the average by almost 3.5 minutes. But this is no disaster.

It means successive jobs will end about 3.5 minutes ahead of the predicted run-time, and as a result 16 nodes out of 117 may be idle for 3.5 minutes longer than expected before operational jobs run. Less than ideal but never-the-less still very acceptable.

What is clear however is the basic requirements exist for the introduction of a resource reservation system. We know in advance the resources needed by operational jobs and the time these jobs should start. We have confidence that the non-operational jobs have well predicted run-times and will finish as required/expected.

C. The need to have processes that take the information about operational jobs and running jobs and use this along with other information to dynamically reserve resources, the purpose being to restrict or stop non-operational jobs from starting, then release resources.

To reserve resources IBM suggested ECMWF use a little-used Loadleveler feature called "**floating resources**", where "**floating resources**" are in effect cluster-wide licenses to use CPUs. The floating resources we use are defined as "**FloatingCpus**", the unit of which is a physical CPU. These are requested by non-operational jobs in the same way as other resources, such as ConsumableCpus and ConsumableMemory via the "resources" LoadLeveler directive. e.g.

```
# @ resources = ConsumableCpus(4) ConsumableMemory(3600Mb) FloatingCpus(4)
```

And as it is imperative that all non-operational jobs contain such a directive and request the correct number of FloatingCpus, all jobs are passed through a "job filter" and the job filter inserts the request. Then by reducing the number available (as specified in the 'config' file), it is possible to restrict all non-operational jobs to a limited number of CPUs, and ensure that only operational jobs, which by design do not request floating resources, may access the remaining CPUs.

With 117 nodes, each node an 8-way SMP system, making 936 CPUs in total, the following directive in LoadLeveler's configuration file:

```
FLOATING_RESOURCES = FloatingCpus(936)
```

This means that outside operational periods all 936 CPUs are available to non-operational work.

So at some point ahead of the time, when CPU's will be needed by operational jobs, the value of cluster-wide FloatingCpus limit is reduced in LoadLeveler's configuration file, and by signalling the relevant LoadLeveler daemons the change is introduced. As non-operational jobs finish and release nodes, the number of FloatingCpus in use is reduced but no new non-operational jobs will start until the number of FloatingCpus in use becomes less than the value of FloatingCpus set in the LoadLeveler configuration file. Of course, as this is a reservation scheme, the value is reduced some time before the operational jobs are submitted, taking into account the time needed for user jobs to end (which will be covered later).

Ignoring the time in advance calculation for now, to reserve 256 CPUs for the first operational job, the value of FloatingCpus in the LoadLeveler configuration file is reduced from:

```
FLOATING_RESOURCES = FloatingCpus(936)
```

to

```
FLOATING_RESOURCES = FloatingCpus(680)
```

and once 256 CPUs are released, they will remain unused until the operational job starts.

However, the 'skill' is in working out when to reduce the value of FloatingCpus. Too early and resources will lie idle, waiting for the operational jobs to be submitted. Too late and the resources will not be released in time. However as the operational schedule has a degree of flexibility a little late is better than a little early.

Clearly much depends on the number of nodes needed (to give time for a running job or running jobs to end) and taking into account the known flexibility in the running of operational jobs. To maximise the use of the system it would be beneficial if shorter non-operational jobs with the most consistent historical run-times were to be selected just before operational periods. Additionally if the operational resources required are needed in stages rather than all at once, a more flexible approach can be made.

First ECMWF talked to IBM about this. IBM mentioned their plans for a resource reservation scheme that, like LoadLeveler's backfill scheduler, works by using a job's wall_clock_limit. Sadly on a basic IBM system wall_clock_limit is rarely set accurately, and if it is the kill on wall_clock_limit exceeded is a rigid scheme that is not acceptable to ECMWF. So a local fix was needed.

11. The Backfill Problem

11.1 First Identify the Problem

IBM ask users to set the wall_clock_limit and then use the value given for backfill. Yet users are known to have a poor understanding of the length of time their jobs will take to run and even on the 'best' systems runtimes will vary. So expecting users to keep tabs on the spread in unrealistic. Then add to this the basic philosophy behind wall_clock_limit, which is to kill jobs which over-run, and it is no wonder that users never specify an "accurate" value. You do not have to be a rocket scientist to realise I have no faith in scheduling and backfill using user-specified wall_clock_limits. The system forces users to be inaccurate.

11.2 The ECMWF Solution

- Ask users to not specify a job wall_clock_limit.
- When each job completes, to collect data (including actual run-time) and store it in a database.
- To predict wall_clock_limits using the data collected.

If a job (same name, same user) is submitted more than twice, the job filter uses the historical run-time data to create a predicted run-time, see Figure 4 using the Estim.95%/Max value 00:52:53 for this purpose. Then add one day to this and set the wall_clock_limit to "1+00:52:53". The time added ensures the job is not killed should it over-run its' expected runtime and the 1+ is used because it is very easy to code. Additionally it is very easy to mask out in displays leaving the true predicted run-time for all to see. Please note - the backfill scheduler works by comparing time differences. So provided all wall_clock_limits use the 24-hour baseline, which they do, the backfill scheduler functions as designed. The 1+ is only an offset used to stop backfill from killing jobs, it can otherwise be ignored.

As pointed out earlier, job runtimes are quite consistent, and the predicted run-times are always a slight over-estimate using the "Estim.95%/ Max" value, so it is quite rare for jobs to over-run. It is much more common for non-operational jobs to end about 1 to 2 minutes ahead of the predicted run-time. But should a job run for more than 1 minute longer than the predicted run-time it will be highlighted in red on the operator display, to ensure investigation starts as soon as possible. An example of this comes later with Figure 7.

Up to now I have described all of the elements that enable a system to predict and control the reservation, use and release of CPUs (nodes). I have shown that the data exists to enable nodes to be filled and to ensure nodes are released in line with tight operational schedules. Theory is fine, but how is it done in practise?

12. When Does the Value of FloatingCpus Get Modified in the LoadLeveler Configuration File?

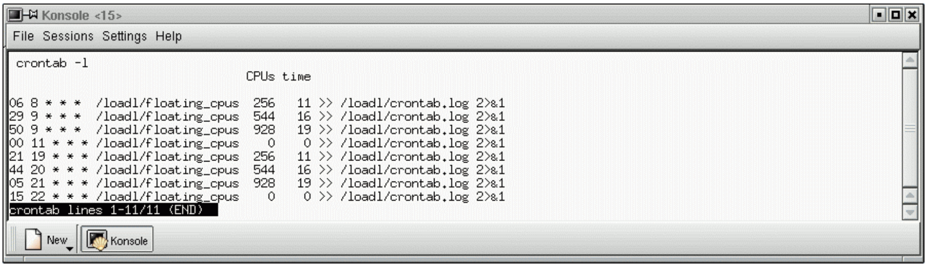
Analysis of the workload is used to determine the average time for nodes to be released. And from this we have determined the time in advance (in minutes) for each reduction of FloatingCpus. The values chosen can be seen under the heading 'time' in Figure 6. A more sophisticated scheme could be used but this simple scheme works and has not yet been improved on.

13. How Does the Value of FloatingCpus Get Modified in the LoadLeveler Configuration File?

A utility was written to modify the LoadLeveler configuration file "/loadl/config". This utility "floating_cpus" takes care of file locking to ensure that the configuration file is not being manipulated by other utilities concurrently. At present "floating_cpus" is executed via "cron" at the times shown in Figure 6. The call to run "floating_cpus" comes with 2 parameters, the number of **CPUs** to be reserved and **time** in advance the new value is being applied.

Taking the first cron entry from Figure 6

```
                                  CPUs   time

06  8  *  *  *  /loadl/floating_cpus   256     11
```



```
 crontab -l
                                CPUs time

06 8 * * *  /loadl/floating_cpus  256   11 >> /loadl/crontab.log 2>&1
29 9 * * *  /loadl/floating_cpus  544   16 >> /loadl/crontab.log 2>&1
50 9 * * *  /loadl/floating_cpus  928   19 >> /loadl/crontab.log 2>&1
00 11 * * * /loadl/floating_cpus    0    0 >> /loadl/crontab.log 2>&1
21 19 * * * /loadl/floating_cpus  256   11 >> /loadl/crontab.log 2>&1
44 20 * * * /loadl/floating_cpus  544   16 >> /loadl/crontab.log 2>&1
05 21 * * * /loadl/floating_cpus  928   19 >> /loadl/crontab.log 2>&1
15 22 * * * /loadl/floating_cpus    0    0 >> /loadl/crontab.log 2>&1
crontab lines 1-11/11 (END)
```

Figure 6 - the crontab runtimes where the first parameter is CPUs reserved and the second value is time (minutes), that is minutes in advance of the operational need.

At 08:06 the crontab job /loadl/floating_cpus is run to reserve 256 **CPUs**. This is done by altering the value of FloatingCpus in the /loadl/config file, reducing the value from 936 to 680 with the knowledge that the 256 CPUs requested will be needed. The value of **time** (11 minutes) means the first operational job is expected to be submitted at 08:17.

In the future, the script 'floating_cpus' will be executed by an event that is an integral part of the operational suite of jobs. Then if operational schedules are changed, the reservation times will automatically change too. However the "cron" mechanism will be retained as a safety net. If the start of operational activity is delayed significantly and the event-linked reservation of nodes does not run, a "cron" run a little later will ensure that resources are reserved so that when the operational activity eventually starts it is not be delayed further. Although potentially wasteful, on the few occasions when forecasts run very late it is essential that the resources needed are already available.

14. When Does the Value of FloatingCpus Get Modified in the /loadl/config File?

The utility "floating_cpus" that sets the limit of FloatingCpus in the configuration file, runs ahead of the time reserving resources based on our experience of the normal average release of nodes. We plan to change this "rule of thumb" mechanism to one that analyses running jobs, and pre-selects the 'right' jobs. But currently this is not done.

It is obvious that schemes to alter job selection before operational periods will influence the use of nodes, the release of nodes and the optimum time in advance that resource reservation should be made. The opportunity for complexity is large. However we keep this simple by using average node release patterns and checking to see what actually happens. There is some waste, but to start with if there is less than a 15-minute variation to the runtimes of the operational suites we have achieved our primary aim. Slightly better node utilisation could be obtained but not a lot. Integrating Resource Reservation into the operational suite comes first.

15. Summarising A. B. and C.

The components of Resource Reservation are

- CPU resources called FloatingCpus defined in the /loadl/config file
- the job-filter ensures all non-operational jobs request FloatingCpus
- known requirements for all operational jobs
- historical run-time data for most user jobs
- A predicted run-time for most user jobs
- Confidence factors (see Figures 4 on page 6, and later Figure 7 on page 10) are provided as Std.dev and Spread in the predicted run-times.
- User job wall_clock_limits set to 1+ predicted run-times.
- Efficient backfill but with kill on wall_clock_limit disabled.
- A utility called floating_cpus that sets new values of FloatingCpus in the config file
- No need to drain or resume job class globally or node by node.
- A utility that gives information about the average frequency of released nodes
- cron jobs that ensure the value of FloatingCpus is set sufficiently in advance

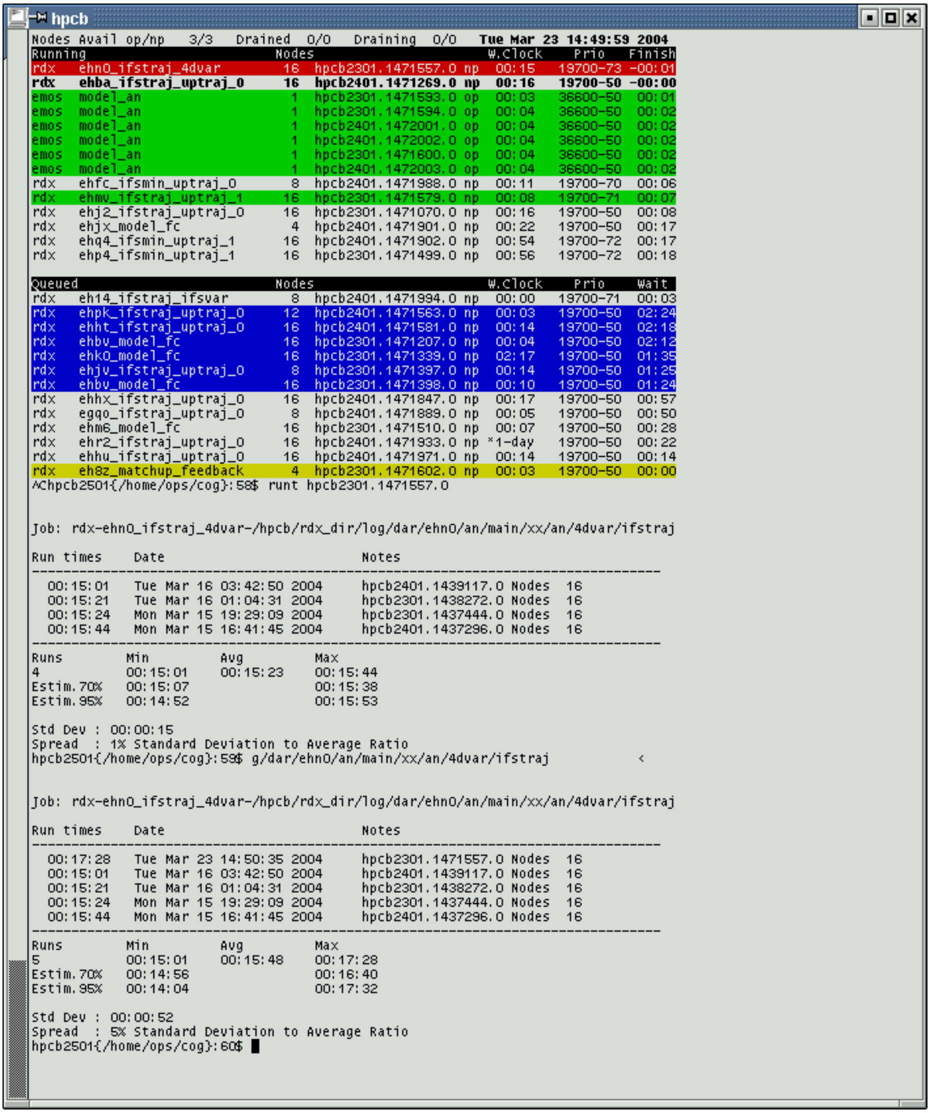
16. Visualisation and Human Intervention

So finally we get to **D. The need to be able to visualise what is happening and verify that it is working as designed.**

Visualising job selection and backfill activity is done with a single display called ll_jobs. Please note - ll_jobs is a display program and does no data gathering. A perl program called nll_sched provides the information displayed. nll_sched runs every 10 seconds and uses the Loadleveler API to get the data needed then processes the data.

ll_jobs has 2 modes - non-operational and operational, where operational is triggered by FloatingCpus <936. By watching the ll_jobs display over time, it is possible to be confident that what is supposed to happen actually happens. Importantly when resources have been reserved, but the display shows CPUs are not going to be released because the active jobs will run for too long, detailed information is displayed from which action can be taken. And when action needs to be taken, it is shift staff taking the action. They make sure the nodes get released before a delay of more than 15 minutes occurs.

So next a display of ll_jobs in non-operational mode and other related information.



```
Nodes Avail op/np    3/3   Drained  0/0   Draining  0/0   Tue Mar 23 14:49:59 2004
Running                         Nodes                          W.Clock    Prio    Finish
rdx   ehn0_ifstraj_4dvar          16  hpcb2301.1471557.0 np   00:15   19700-73 -00:01
rdx   ehba_ifstraj_uptraj_0       16  hpcb2401.1471269.0 np   00:16   19700-50 -00:00
emos  model_an                     1  hpcb2301.1471593.0 op   00:03   36600-50  00:01
emos  model_an                     1  hpcb2301.1471594.0 op   00:04   36600-50  00:02
emos  model_an                     1  hpcb2401.1472001.0 op   00:04   36600-50  00:02
emos  model_an                     1  hpcb2401.1472002.0 op   00:04   36600-50  00:02
emos  model_an                     1  hpcb2301.1471600.0 op   00:04   36600-50  00:02
emos  model_an                     1  hpcb2401.1472003.0 op   00:04   36600-50  00:02
rdx   ehfc_ifsmin_uptraj_0         8  hpcb2401.1471988.0 np   00:11   19700-70  00:06
rdx   ehmv_ifstraj_uptraj_1       16  hpcb2301.1471579.0 np   00:08   19700-71  00:07
rdx   ehj2_ifstraj_uptraj_0       16  hpcb2301.1471070.0 np   00:16   19700-50  00:08
rdx   ehjx_model_fc                4  hpcb2401.1471901.0 np   00:22   19700-50  00:17
rdx   ehq4_ifsmin_uptraj_1        16  hpcb2401.1471902.0 np   00:54   19700-72  00:17
rdx   ehp4_ifsmin_uptraj_1        16  hpcb2301.1471499.0 np   00:56   19700-72  00:18

Queued                          Nodes                          W.Clock    Prio    Wait
rdx   eh14_ifstraj_ifsvar          8  hpcb2401.1471994.0 np   00:00   19700-71  00:03
rdx   ehpk_ifstraj_uptraj_0       12  hpcb2401.1471563.0 np   00:03   19700-50  02:24
rdx   ehht_ifstraj_uptraj_0       16  hpcb2401.1471581.0 np   00:14   19700-50  02:18
rdx   ehbv_model_fc               16  hpcb2301.1471207.0 np   00:04   19700-50  02:12
rdx   ehk0_model_fc               16  hpcb2301.1471339.0 np   02:17   19700-50  01:35
rdx   ehjv_ifstraj_uptraj_0        8  hpcb2301.1471397.0 np   00:14   19700-50  01:25
rdx   ehbv_model_fc               16  hpcb2301.1471398.0 np   00:10   19700-50  01:24
rdx   ehhx_ifstraj_uptraj_0       16  hpcb2401.1471847.0 np   00:17   19700-50  00:57
rdx   eggo_ifstraj_uptraj_0        8  hpcb2401.1471889.0 np   00:05   19700-50  00:50
rdx   ehm6_model_fc               16  hpcb2301.1471510.0 np   00:07   19700-50  00:28
rdx   ehr2_ifstraj_uptraj_0       16  hpcb2401.1471933.0 np  *1-day   19700-50  00:22
rdx   ehhu_ifstraj_uptraj_0       16  hpcb2401.1471971.0 np   00:14   19700-50  00:14
rdx   eh8z_matchup_feedback        4  hpcb2301.1471602.0 np   00:03   19700-50  00:00
^Chpcb2501{/home/ops/cog}:58$ runt hpcb2301.1471557.0


Job: rdx-ehn0_ifstraj_4dvar-/hpcb/rdx_dir/log/dar/ehn0/an/main/xx/an/4dvar/ifstraj

Run times     Date                             Notes
-------------------------------------------------------------------------------
  00:15:01    Tue Mar 16 03:42:50 2004         hpcb2401.1439117.0 Nodes  16
  00:15:21    Tue Mar 16 01:04:31 2004         hpcb2301.1438272.0 Nodes  16
  00:15:24    Mon Mar 15 19:29:09 2004         hpcb2301.1437444.0 Nodes  16
  00:15:44    Mon Mar 15 16:41:45 2004         hpcb2401.1437296.0 Nodes  16
-------------------------------------------------------------------------------
Runs          Min           Avg           Max
4             00:15:01      00:15:23      00:15:44
Estim.70%     00:15:07                    00:15:38
Estim.95%     00:14:52                    00:15:53

Std Dev :  00:00:15
Spread  :  1% Standard Deviation to Average Ratio
hpcb2501{/home/ops/cog}:59$ g/dar/ehn0/an/main/xx/an/4dvar/ifstraj          <


Job: rdx-ehn0_ifstraj_4dvar-/hpcb/rdx_dir/log/dar/ehn0/an/main/xx/an/4dvar/ifstraj

Run times     Date                             Notes
-------------------------------------------------------------------------------
  00:17:28    Tue Mar 23 14:50:35 2004         hpcb2301.1471557.0 Nodes  16
  00:15:01    Tue Mar 16 03:42:50 2004         hpcb2401.1439117.0 Nodes  16
  00:15:21    Tue Mar 16 01:04:31 2004         hpcb2301.1438272.0 Nodes  16
  00:15:24    Mon Mar 15 19:29:09 2004         hpcb2301.1437444.0 Nodes  16
  00:15:44    Mon Mar 15 16:41:45 2004         hpcb2401.1437296.0 Nodes  16
-------------------------------------------------------------------------------
Runs          Min           Avg           Max
5             00:15:01      00:15:48      00:17:28
Estim.70%     00:14:56                    00:16:40
Estim.95%     00:14:04                    00:17:32

Std Dev :  00:00:52
Spread  :  5% Standard Deviation to Average Ratio
hpcb2501{/home/ops/cog}:60$ █
```

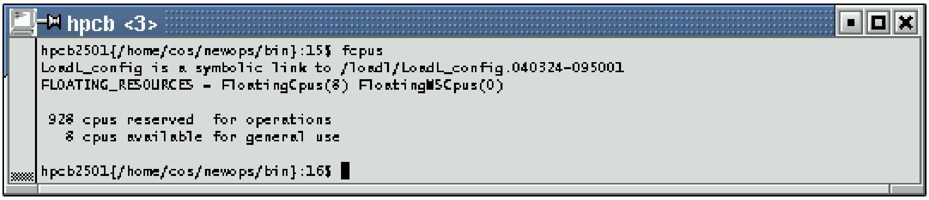
Figure 7 – ll_jobs in non-operational mode (with no mention of nodes reserved), where Running jobs, Queued jobs and 2 sets of predicted run-times for job ehn0_ifstraj_4dvar are shown.

Figure 7 (the top half) shows ll_jobs in a non-operational period – Running jobs first, below this Queued jobs. No mention is made of nodes being reserved. Running jobs are listed top down in the order that they are expected to complete. Some jobs are highlighted in colour. The job in red has run for longer than predicted. Jobs in green started in the previous 3 minutes. Queued jobs are listed in the order they are expected to be dispatched according to SYSPRIO and UserPriority. Jobs in blue have been waiting for more than 1 hour, jobs in yellow are new and were submitted in the previous 2 minutes.

As there is a job that has run for longer than predicted I have extracted the original 4 run-times from which a predicted run-time of 00:15:53 was calculated and the 5 run-times, from which a new predicted run-time of 00:17:32 is calculated.

17. How Many FloatingCpus Have Been Reserved?

As mentioned before during operational periods, the value of FloatingCpus will be less than 936. The scripts fcpus can be used to check the value of FloatingCpus and the relationship with CPUs reserved.

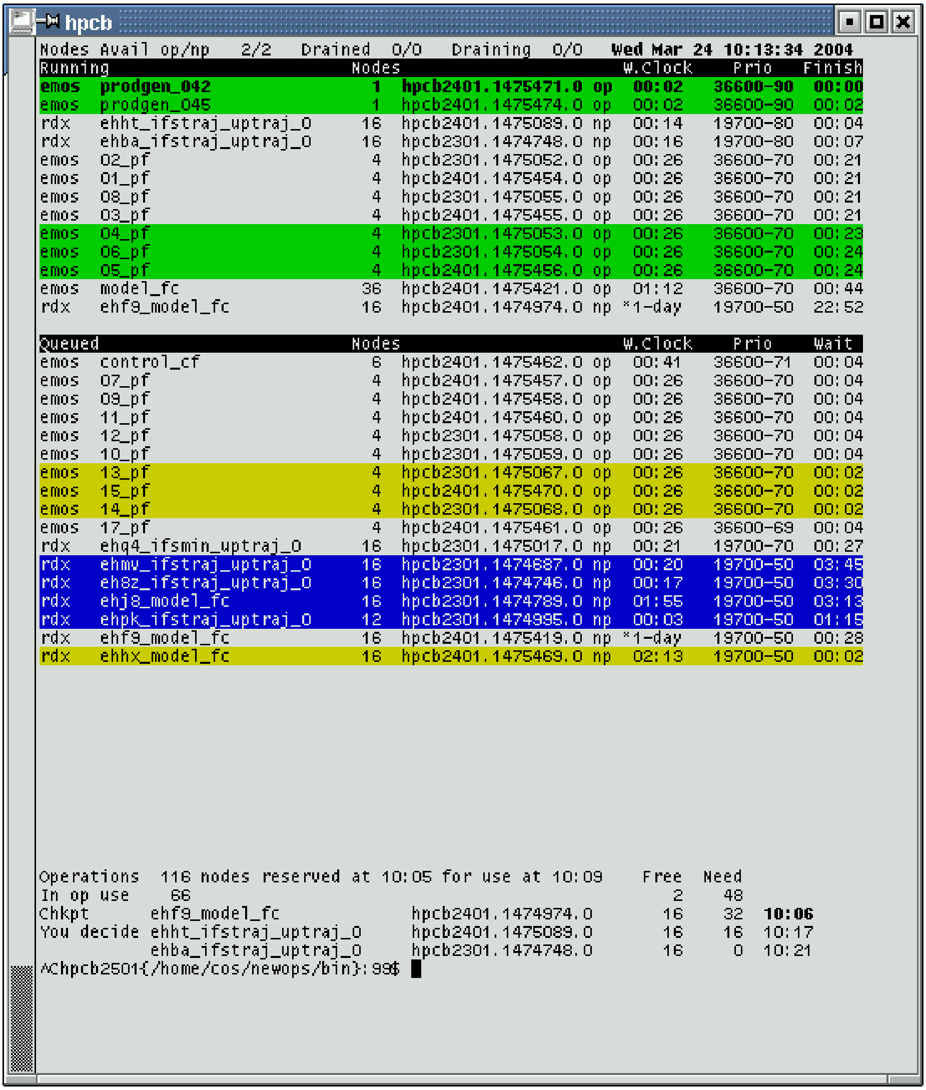


```
hpcb2501{/home/cos/newops/bin}:15$ fcpus
LoadL_config is a symbolic link to /loadl/LoadL_config.040324-095001
FLOATING_RESOURCES = FloatingCpus(8) Floating█SCpus(0)

 928 cpus reserved  for operations
   8 cpus available for general use

hpcb2501{/home/cos/newops/bin}:16$ █
```

Figure 8 – a display showing the relationship between the value of FloatingCpus set a CPUs reserved.

When FloatingCpus is set to 8, 928 CPUS (116 nodes) are reserved (ask me later why FloatingCpus is not set to 0), and as the value of FloatingCpus is 8 (<936) ll_jobs will show more information -see Figure 9 – below.

Please note the situation shown in Figure 9 was created artificially. Resource Reservation at 09:50 was delayed until 10:05. ll_jobs shows the time is 10:13:34, 116 nodes reserved, where 10:09 is the time the nodes were needed operationally. Unusually 48 nodes remain in use by non-operational jobs. So some jobs need to be cleared out of the system manually. One decision is easy. Job hpcb2401.1474974.0 (ehf9_model_fc) can be check-pointed and rerun. The other 2 jobs will complete soon, one at 10:17 (in 4 minutes time) and the other at 10:21 (in 8 minutes time), so waiting a few minutes for these 2 jobs to end will not cause a problem, both being well within the 15-minute flexibility.



```
Nodes Avail op/np   2/2   Drained  0/0   Draining  0/0   Wed Mar 24 10:13:34 2004
Running                                Nodes                         W.Clock    Prio    Finish
emos  prodgen_042                          1  hpcb2401.1475471.0 op  00:02  36600-90  00:00
emos  prodgen_045                          1  hpcb2401.1475474.0 op  00:02  36600-90  00:02
rdx   ehht_ifstraj_uptraj_0               16  hpcb2401.1475089.0 np  00:14  19700-80  00:04
rdx   ehba_ifstraj_uptraj_0               16  hpcb2301.1474748.0 np  00:16  19700-80  00:07
emos  02_pf                                4  hpcb2301.1475052.0 op  00:26  36600-70  00:21
emos  01_pf                                4  hpcb2401.1475454.0 op  00:26  36600-70  00:21
emos  08_pf                                4  hpcb2301.1475055.0 op  00:26  36600-70  00:21
emos  03_pf                                4  hpcb2401.1475455.0 op  00:26  36600-70  00:21
emos  04_pf                                4  hpcb2301.1475053.0 op  00:26  36600-70  00:23
emos  06_pf                                4  hpcb2301.1475054.0 op  00:26  36600-70  00:24
emos  05_pf                                4  hpcb2401.1475456.0 op  00:26  36600-70  00:24
emos  model_fc                            36  hpcb2401.1475421.0 op  01:12  36600-70  00:44
rdx   ehf9_model_fc                       16  hpcb2401.1474974.0 np *1-day  19700-50  22:52

Queued                                 Nodes                         W.Clock    Prio    Wait
emos  control_cf                           6  hpcb2401.1475462.0 op  00:41  36600-71  00:04
emos  07_pf                                4  hpcb2401.1475457.0 op  00:26  36600-70  00:04
emos  09_pf                                4  hpcb2401.1475458.0 op  00:26  36600-70  00:04
emos  11_pf                                4  hpcb2401.1475460.0 op  00:26  36600-70  00:04
emos  12_pf                                4  hpcb2301.1475058.0 op  00:26  36600-70  00:04
emos  10_pf                                4  hpcb2301.1475059.0 op  00:26  36600-70  00:04
emos  13_pf                                4  hpcb2301.1475067.0 op  00:26  36600-70  00:02
emos  15_pf                                4  hpcb2401.1475470.0 op  00:26  36600-70  00:02
emos  14_pf                                4  hpcb2301.1475068.0 op  00:26  36600-70  00:02
emos  17_pf                                4  hpcb2401.1475461.0 op  00:26  36600-69  00:04
rdx   ehq4_ifsmin_uptraj_0                16  hpcb2301.1475017.0 np  00:21  19700-70  00:27
rdx   ehmv_ifstraj_uptraj_0               16  hpcb2301.1474687.0 np  00:20  19700-50  03:45
rdx   eh8z_ifstraj_uptraj_0               16  hpcb2301.1474746.0 np  00:17  19700-50  03:30
rdx   ehj8_model_fc                       16  hpcb2301.1474789.0 np  01:55  19700-50  03:13
rdx   ehpk_ifstraj_uptraj_0               12  hpcb2301.1474995.0 np  00:03  19700-50  01:15
rdx   ehf9_model_fc                       16  hpcb2401.1475419.0 np *1-day  19700-50  00:28
rdx   ehhx_model_fc                       16  hpcb2401.1475469.0 np  02:13  19700-50  00:02









Operations  116 nodes reserved at 10:05 for use at 10:09    Free  Need
In op use    66                                                2    48
Chkpt       ehf9_model_fc            hpcb2401.1474974.0       16    32  10:06
You decide ehht_ifstraj_uptraj_0     hpcb2401.1475089.0       16    16  10:17
           ehba_ifstraj_uptraj_0     hpcb2301.1474748.0       16     0  10:21
^Chpcb2501{/home/cos/newops/bin}:99$ █
```

Figure 9 - ll_jobs during an operational period. FloatingCpus = 8, so additional information is provided particularly as resources will not be released at the time specified.

18. What Has Been Achieved So Far?

Has the variability in the start-times and end-times of the operational runs been reduced? Do the daily runs of the operational forecasts complete within 15 minutes of the optimum time?

The variability of the start-time of the operational runs, before FloatingCpus were reserved, is shown in Figure 10 and the variability of the start-time of the operational run after FloatingCpus were reserved is shown in Figure 11.

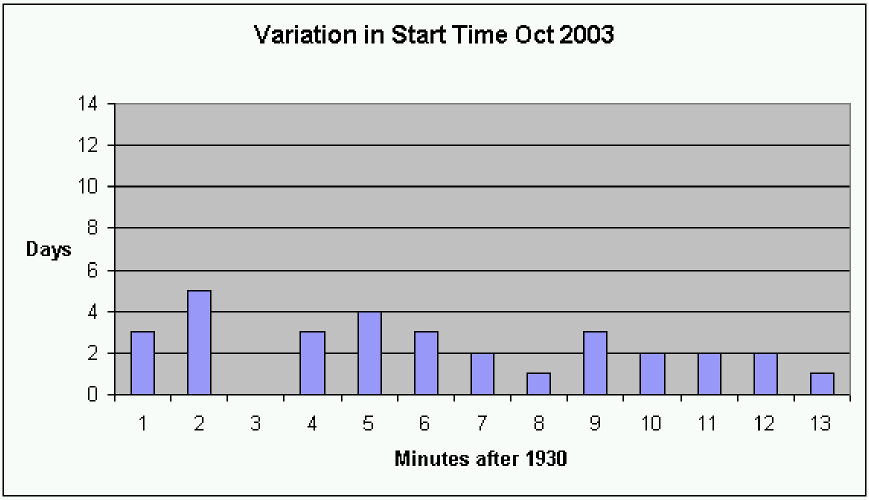


Figure 10 – the variability in the start-time of operational runs, before Resource Reservation was introduced.

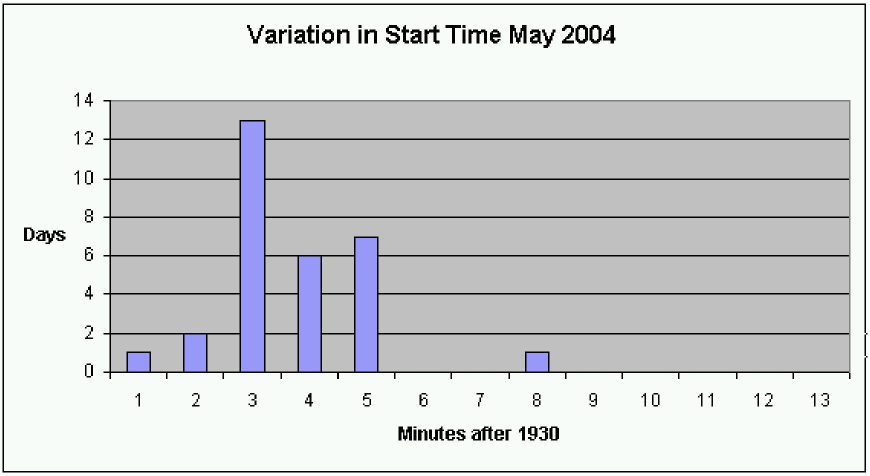


Figure 11 – the variability in the start-time of operational runs after Resource Reservation was introduced.

Comparing Figures 10 and 11 you can see the reservation scheme has significantly reduced the variability in the scheduling of the first operational job. Only once in May 2004 was more than 5 minutes of the flexibility used.

In Figures 12 and 13 below you can see the variability of the end-times of the operational runs before FloatingCpus were reserved and after FloatingCpus were reserved.

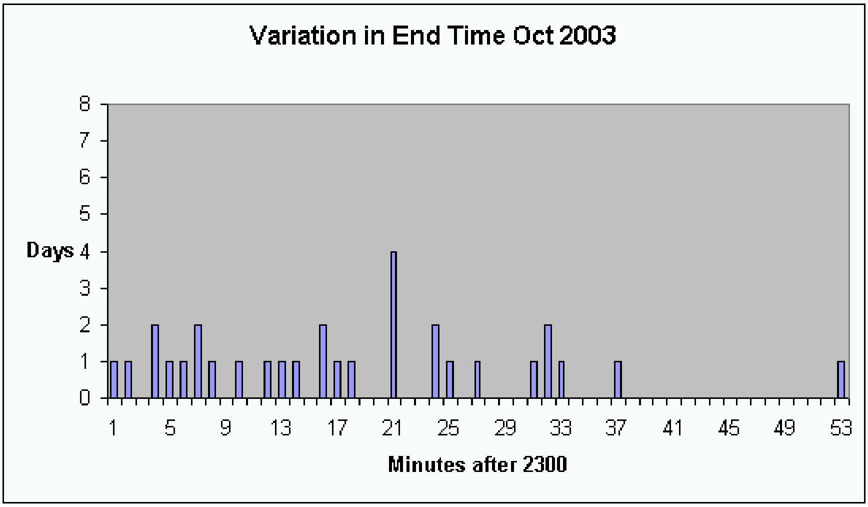


Figure 12 – the variability in the end-times of operational jobs, before Resource Reservation was introduced.

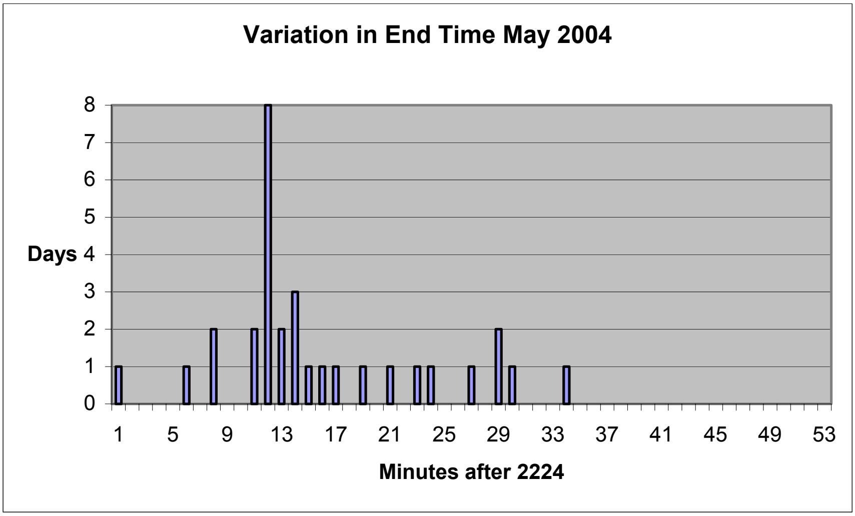


Figure 13 - the variability in the end-time of operational jobs, after Resource Reservation was introduced.

Comparing figures 12 and 13 you can see the variability in the end-times of all operational jobs have been reduced too.

Note – since October 2003 significant improvements have been made to the efficiency of operational jobs and they run faster than before. As a result, the optimal end time of 23:00 achieved in October 2003 has been replaced by an end-time of 22:24 in May 2004. This means the need to keep operational jobs to a 15-minute variation is less acute at present, but not for long. Plans for higher quality forecasts (with greater computational needs) will soon push the end-time back to 23:00.

So clearly the variation exceeds 15 minutes and more work is needed to limit the variation to 15 minutes. 2 solutions are already planned, to optimise the pre-selection of jobs and to ensure when human intervention is needed that action is taken quickly.

Finally Figure 14 shows the allocation of CPUs to all jobs on a typical day since Resource Reservation was introduced.

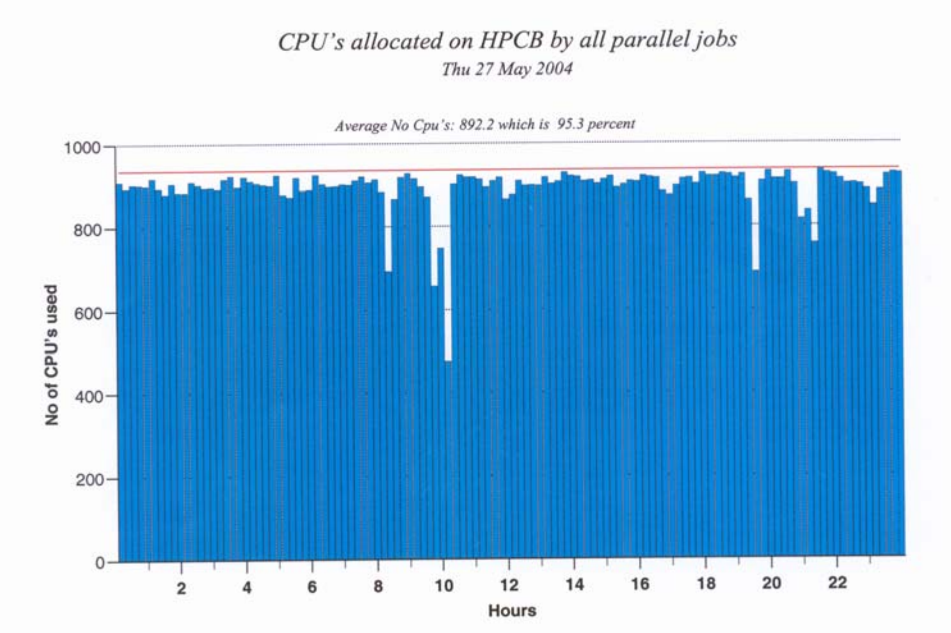


Figure 14 – the allocation of CPU's to all jobs on a typical day since Resource Reservation was introduced, plotted over 12-minute intervals, where the red line (936 CPUs) is the maximum number that parallel jobs can use.

Comparing Figure 14 with Figure 3 it is clear the allocation of CPU's to all jobs since Resource Reservation was introduced has reduced by a little over 1%. This is greater than was hoped for. But as mentioned earlier there are plans to optimise the selection of jobs before operational periods, which is when the greatest waste occurs. So importantly not only will optimising job selection minimise runtime variation it can be made to increase node utilisation too.

19. The Conclusion

The basic elements of the Reservation Scheme are very sound. The scheme and has been very effective in keeping the start-time to a tighter schedule but less so the more important end-time. So some fine tuning is needed. In addition there has been a reduction in the overall system utilisation seen by comparing Figure 3 and Figure 14, but ways have been described to both minimise the variation and maximise the allocation of nodes.

20. Main Points

It is essential that parallel jobs can be scheduled independent of other systems, that the jobs are designed well (suit the system) and system performance (GPFS, I/O nodes, Loadleveler, network, paging) does not vary. Thus (using historical data) predicted run-times will be accurate.

The Loadleveler feature FloatingCpus enables resources to be controlled globally, dynamically and logically, independent of job class where jobs that request FloatingCpus are members of one logical class and jobs that do not request FloatingCpus are members of another logical class. Managing the resources to suit these 2 logical classes could not be easier.

Setting values of FloatingCpus a fixed time in advance based on averages is very straightforward and reasonably effective but the variation in runtimes and better system utilisation could be obtained by optimising job selection prior to operational periods.

Reducing the value of FloatingCpus without turning classes off means that the underlying scheduling of non-operational jobs continues as normal in the reduced set of nodes, and there is complete flexibility as to which nodes are used by operational jobs and which nodes are used by non-operational jobs.

Having the system set wall_clock_limit with offset 1+ overcomes the kill on wall_clock_limit exceeded.

Operational jobs have their wall_clock_limit set automatically too which means operators need not remember how long the multitude of operational jobs should take to run and any over-run is flagged immediately.

It is essential that monitoring tools (in our case operator displays) are created so that it is possible to confirm both backfill and scheduling are performing as expected or when events do not go to plan a mechanism (again operator displays) is in place to make operators, administrators and analysts aware of the problem(s).

There is no need to have lots of user batch classes (we had many NQS queues on previous systems) for jobs that are short, long, slow, fast, big, small etc so that in pre-operational periods the right jobs can be selected by draining classes, reducing job-limits per class, per user etc.

21. Other Thoughts

The ability to schedule jobs as described stems from the understanding that the system used is a High Performance Cluster (HPC) System managed as a Super-Computer and is not a configuration of loosely coupled Server systems. All those involved in providing elements of the service on the HPC system; analysts, support staff, users, administrators, managers, have a fundamental desire for the system to be configured optimally and for jobs to run efficiently. I cannot stress enough that jobs must not use (need) swap space. The processes in each node of a multi-node, high performance, well parallelised, cpu-bound job, must not exceed physical memory as it only takes 1 process to use too much memory and all nodes will perform badly as the rogue process swaps.

Ideally jobs will parallelise well, scale reasonable well and use a whole node or multiples of nodes. Memory sharing is only permitted if job performance and known-runtimes are not altered. If users have CPU-bound jobs that use less than 1 node, particularly serial jobs with 1 process, the users are advised to run the jobs on Server systems.

Jobs that perform badly must be detected and users must know to call support staff for help (or eventually get caught) if they realise new jobs (for which there is no known run-time data) perform badly. Users and more importantly application programmers, who make it possible for suites of jobs (in the form of an experiment) to be submitted automatically, must be willing to help sort out job related problems and identify solutions.

As the release of all nodes for use by operational jobs occurs twice a day, all user jobs have to finish within 1 hour. Jobs that run for more than 1 hour should either create restart files so that they can be killed without substantial losses, or should have the ability to trap a signal and then create restart files before killing themselves. In Figure 9 such a job was labelled Chkpt. This job will accept and trap such a signal, write a restart file and kill itself.

There is no room for sloppy philosophy. The idea that it is OK for a job to be inefficient and run for a long time as long as the user pays for it, is just not acceptable.

22. Acknowledgements

I would like to thank the following for giving me the opportunity to write this paper and for their support and assistance in this work. Without them there would be no paper.

My managers

Walter Zwieflhofer, Head Operations Department, ECMWF.
Neil Storer, Acting Head Computer Division, ECMWF.
Sylvia Baylis, Head Computer Operations, ECMWF.

My team member

Peter Towers, Consultant, ECMWF

Then last but not least, the IBM LL support analyst for ECMWF

Peter Mayes, of IBM UK.

Peter is the IBM LL contact person. It was he who suggested we try Resource Reservation. It is he who has helped unravel (some of) the mysteries of LL backfill scheduling.

Inferring the Topology and Traffic Load of Parallel Programs Running in a Virtual Machine Environment

Ashish Gupta and Peter A. Dinda

Department of Computer Science, Northwestern University
{ashish,pdinda}@cs.northwestern.edu

Abstract. We are developing a distributed computing environment based on virtual machines featuring application monitoring, network monitoring, and an adaptive virtual network. In this paper, we describe our initial results in monitoring the communication traffic of parallel applications, and inferring its spatial communication properties. The ultimate goal is to be able to exploit such knowledge to maximize the parallel efficiency of the running parallel application by using VM migration, virtual overlay network configuration and network reservation techniques, which are a part of the distributed computing environment. Specifically, we demonstrate that: (1) we can monitor the parallel application network traffic in our layer 2 virtual network system with very low overhead, (2) we can aggregate the monitoring information captured on each host machine to form a global picture of the parallel application's traffic load matrix, and (3) we can infer from the traffic load matrix the application topology. In earlier work, we have demonstrated that we can capture the time dynamics of the applications. We begin here by considering offline traffic monitoring and inference as a proof of concept, testing it with a variety of synthetic and actual workloads. Next, we describe the design and implementation of our online system, the Virtual Topology and Traffic Inference Framework (VTTIF), and evaluate it using a NAS benchmark.

1 Introduction

Virtual machines have the potential to simplify the use of distributed resources in a way unlike any other technology available today, making it possible to run diverse applications with high performance after only minimal or no programmer and administrator effort. Network and host bottlenecks, difficult placement decisions, and firewall obstacles are routinely encountered, making effective use of distributed resources an obstacle to innovative science. Such problems, and the human effort needed to work around them, limit the development, deployment, and scalability of distributed parallel applications.

We have presented a detailed case for virtual machine-based distributed and parallel computing [4], and we are now developing a system, Virtuoso, which has the following model:

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 125–143, 2005.

- The user receives what appears to be a new computer or computers on his network at very low cost. The user can install, use, and customize the operating system, environment, and applications with full administrative control.
- The user chooses where to execute the virtual machines. Checkpointing and migration is handled efficiently through Virtuoso. The user can delegate these decisions to the system.
- A service provider need only install the VM management software to support a diverse set of users and applications.
- Monitoring, adaptation, resource reservation, and other services are retrofitted to existing applications at the VM level with no modification of the application code, resulting in broad application of these technologies with minimal application programmer involvement.

An important element of our system is a layer 2 virtual network, VNET, which we initially developed to create the "networking illusion" needed for the first element of the model. It can "move" a set of virtual machines in a WAN environment to the user's local layer 2 domain. We are now expanding VNET into a tool that supports arbitrary overlay topologies and routing rules, passive application and network monitoring, adaptation (based on VM migration and topology/routing changes), and resource reservation. VNET is described in detail in a previous paper [12].

This paper reports on one of our first steps toward achieving the last element of the model. The question we address in particular is: can we monitor, with low overhead and no application or operating system modifications, the communication traffic of a parallel application running in a set of virtual machines interconnected with a virtual network, and compute from it the traffic load matrix and application communication topology? Our initial results demonstrate that this is possible. We are integrating the online implementation of our ideas, VTTIF (Virtual Topology and Traffic Inference Framework), into the evolving VNET system.

We consider here Bulk-Synchronous Parallel [6] (BSP) style applications. Specifically, we consider parallel programs whose execution alternates between one or more computing phases and one or more communication phases, including metaphases. We are testing whether our results hold for more general applications. In earlier work, we have demonstrated that the network traffic of compiler-parallelized BSP applications, when measured using techniques similar to those used here, exhibits clear time dynamical structure (periodicity with harmonics) [3]. Our results here show that we can quickly and efficiently recover its spatial structure, its topology and traffic load, as well.

The ultimate motivation behind recovering the spatial and temporal properties of a parallel application running in a virtual environment is to be able to maximize the parallel efficiency of the running application by migrating its VMs, changing the topology and routing rules of the communication network, and taking advantage of underlying network reservations on the application's behalf.

A parallel program may employ various communication patterns for its execution. A communication pattern consists of a list of all the message exchanges of a representative processor during a communication phase. The result of each processor executing its communication pattern gives us the application topology, such as a mesh, toroid, hypercube, tree, etc, which is in turn mapped to the underlying network topology [8]. In this paper, we attempt to infer the application topology and the costs of its edges, the traffic load matrix, by observing the low-level traffic entering and leaving each node of the parallel application, which is running inside of a virtual machine.

It is important to note that application topologies may be arbitrarily complex. Although our initial results are for BSP-style applications, our techniques can be used with arbitrary applications, indeed, any application or OS that the virtual machine monitor (we use VMWare GSX server in this work) can support. However, we do not yet know the effectiveness of our load matrix and topology inference algorithms for arbitrary applications.

In general, it is difficult for an application developer, or, for that matter, the user of a “dusty deck” application, to analyze and describe his application at the level of detail needed in order for a virtual machine distributed computing system to make adaptation decisions on its behalf. Furthermore, the description may well be time or data dependent or react to the conditions of the underlying network.

The goal of VTTIF is to provide these descriptions automatically, as the unmodified application runs on an unmodified operating system. In conjunction with information from other monitoring tools, and on the policy constraints, VTTIF information will then be used to schedule the VMs, migrate them to appropriate hosts, and change the virtual network connecting them. The adaptation control mechanisms will query VTTIF to understand what, from a communication perspective, the parallel application is attempting to accomplish.

We began by offline analysis, using traffic logs of parallel applications to develop our three step monitoring and analysis process. Although this initial work was carried out without the use of VMs, using PVM applications whose traffic was captured using tcpdump techniques, it is directly applicable for two reasons. First, VNET interacts with the virtual interfaces of virtual machines in a manner identical (packet filter on the virtual interface) to how tcpdump interacts with physical interfaces (packet filter on a physical interface). Second, the physical machines generate considerably more “noise” than the virtual machines, thus making the problem harder. In Section 2, we describe our three step process and how it is implemented for physical monitoring. In Section 3 we describe a set of synthetic applications and benchmarks we will use to evaluate VTTIF. In Section 4, we show the performance results of applying the process to a wide variety of application topologies and parallel benchmarks.

The results for the offline, physical machine-based were extremely positive, so we designed and implemented an online process that is integrated with our VNET virtual networking tool. Section 5 describes the design of the online VTTIF tool and provides an initial evaluation of it. We are able to recover application topologies online for a NAS benchmark running in VMs and communicating via VNET. The performance overhead of the VTTIF implementation in VNET is negligible.

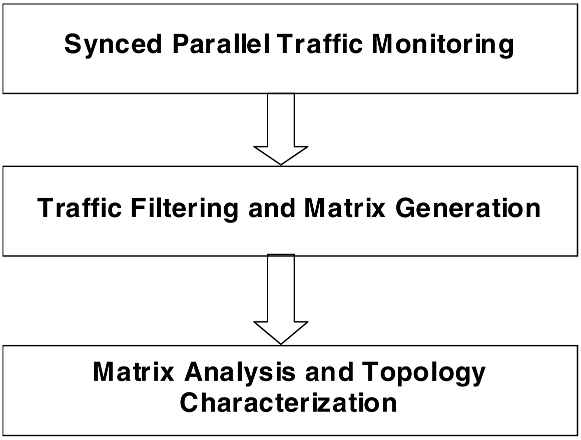

Synced Parallel Traffic Monitoring
Traffic Filtering and Matrix Generation
Matrix Analysis and Topology Characterization

Fig. 1. The three stages involved in inferring the topology and traffic load matrix of a parallel application

In Section 6, we conclude by describing our plans for using the VTTIF and other monitoring information for heuristic adaptive control of the VMs and VNET to maximize application performance.

2 VTTIF and Its Offline Implementation

The inference of parallel application communication is based on the analysis of low level traffic. We first wanted to test whether this approach was practical at all, and, if so, to develop an initial framework for traffic monitoring, analysis and inference, enabling us to test our ideas and algorithms. This initial step resulted in an offline process that focused on parallel programs running on physical hosts. In Section 5, we describe how these results have been extended to an online process that focuses on parallel programs running in virtual machines.

In both our online and offline work, we study PVM [5] applications. Note that the techniques described here are general and are also applicable to other parallel applications such as MPI programs. We run programs on the nodes of our Virtuoso cluster, which is an IBM e1350 with 32 compute nodes, each of which is a dual 2.2 GHz Intel HT Xeon Processors, 1.5 GB RAM, and 40 GB of disk. Each node runs Red Hat Linux 9, PVM 3.4.4, and VMWare GSX Server 2.5. Each VM runs Red Hat Linux 7.3 and PVM 3.4.4. The communication measured here is via a 100 mbit switched network, specifically a Cisco 3550 48 port switch. The nodes speak NFS and NIS back to a separate management machine via a separate network.

The VTTIF framework has three stages as shown in Figure 1. In the first stage, we monitor the traffic being sourced and sinked by each process in the parallel program.

In the offline analysis, this is accomplished by using tcpdump on the physical interface with a packet filter that rejects all but PVM traffic. In the online analysis, we integrate monitoring into our virtual network tool VNET. VNET does the equivalent of running tcpdump on the virtual interface of the virtual machine, capturing all traffic. The Virtuoso cluster uses a switched LAN, so the interface of each node must be monitored separately and the data aggregated. A challenge in the online system is that it must decide when to start and stop this monitoring.

The second stage of the framework eliminates irrelevant traffic from the aggregated traffic and integrates the packet header traces captured by tcpdump to produce a traffic matrix, T. Element $T_{i,j}$ represents the amount of traffic sent from node i to node j. A challenge in the online system is to decide when to recompute this matrix.

The final stage of the framework applies inference algorithms to eliminate noise from the traffic matrix to infer from it the likely application topology. Both the original matrix and the inferred topology are then returned. The topology is displayed graphically. A challenge in the online system is to decide when to recompute the topology.

The current offline framework is designed to automate all of the above steps, allowing the user to run a single command, `infer [parallel PVM program]` This runs the PVM program mentioned in the argument, monitors it for its entire execution, completes the remaining steps of the framework, and prints the matrix and displays the topology. The framework is implemented as a set of Perl scripts, as described below.

Monitor This script is responsible for synchronized traffic monitoring on all the physical hosts, running the parallel program, and storing the packet header traces to files. The script also reads a configuration file that describes the set of hosts on which monitoring is to be done. It runs tcpdump on each of the hosts. It then executes the parallel program and waits for it to finish. Each tcpdump stores its packet header trace output into a file named by the hostname, the date, and the time of execution. Hence, each execution produces a group of related packet header trace files.

Generate This script parses, filters and analyzes the packet header traces to generate a traffic matrix for the given hosts. It sums the packets sizes between each pair of hosts, filtering out irrelevant packets. Filtering is done according to the following criteria:

- Type of packet. Packets which are known not to be a part of the parallel program communication, like ARP, X11, ssh, etc, are discarded. This filtering has only a modest effect in the Virtuoso cluster because there is little extra traffic. However, in the future, we may want to run parallel programs in a wide area environment or one shared with many other network applications, where filtering and extracting the relevant traffic may pose extra challenges.
- The source and destination hosts involved in the packet transmission. We are only interested traffic among a specific group of hosts.

The matrix is emitted in a single file.

Infer This script infers the application topology from the traffic matrix file. In effect, topology inference amounts to taking the potentially "noisy" graph described by the traffic matrix and eliminating edges that are unlikely to be significant. The script also outputs a version of the topology that is designed to be viewed by the algorithm animation system Samba [11].

For inferring the topology, various algorithms are possible. One method is to prune all matrix entries below a certain threshold. More complex algorithms could employ pattern detection techniques to choose an archetype topology that the traffic matrix is most similar to. For the results we show, topology inference is done using a matrix normalization and simple edge pruning technique. The pseudo-code description of the algorithm is:

```
InferTopology(traffic_matrix T,pruning_threshold $b_{min}$)
{
    $b_{max} \leftarrow max(T_{i,j}) \forall_{i,j}$
    $G \leftarrow \emptyset$
    foreach($T_{i,j}$)
    {
        $r_{i,j} \leftarrow T_{i,j}/b_{max}$
        if ($r_{i,j} \geq b_{min}$)
        {
            add edge(i,j) to G
        }
    }
    return G
}
```

In effect, if the maximum bandwidth entry in T is b_{max}, then if ratio of any edge value ($T_{i,j}$) to b_{max} is below a certain threshold b_{min}, then the edge is pruned. The value of b_{min} determines the sensitivity of topology inference.

Visualization makes it very convenient to quickly understand the topology used by the parallel program. By default, we have Samba draw each topology with the nodes laid out in a circle, as this is versatile for a variety of different topologies. However, there is an option to pass a custom graph layout. An automated layout tool such as Dot could also be used.

Figure 2 shows an example of the final output for the program PVM POV, a parallel ray tracer, running on four hosts. The thickness of an edge indicates the amount of traffic for that particular run. Each host is represented by a different color and color of the edge represents the source host for the edge traffic.

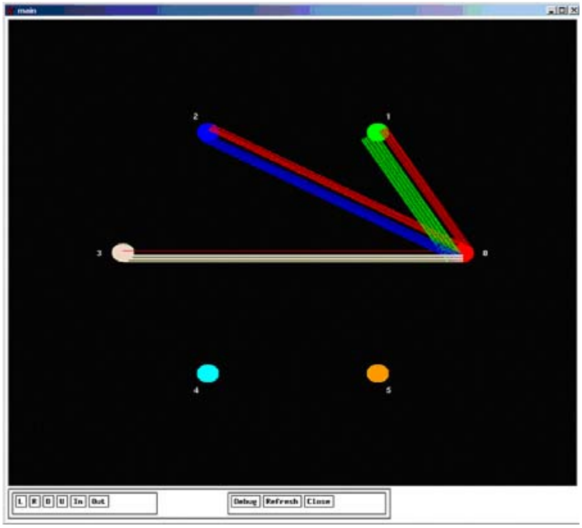


Fig. 2. An example of the final output of the Topology Inference Framework for the PVM-POV application. The PVM-POV application runs on four hosts.

3 Workloads for VTTIF

To test our ideas, we first needed some actual parallel applications to measure. We created and collected the following applications.

- Patterns: This is a synthetic workload generator, which we describe below. It can execute many different kinds of topologies common in BSP parallel programs. We use this extensively to test our framework.
- NAS Parallel Benchmarks: We use the PVM implementation of the NAS benchmarks [1] IS, MG, FT, and EP as developed by Sundaram, et al [13].
- PVM POV: PVM version of the popular ray tracer POVRAY. The PVM version gives it the ability to distribute a rendering across multiple heterogeneous systems. [2].

Except for patterns, these are all well known benchmark programs.

Patterns does message exchanges according to a topology provided at the command line. Patterns emulates a BSP program with alternating dummy compute phases and communication phases according to the chosen topology. It takes the following arguments:

- pattern: The particular topology to be used for communication.
- numprocs: The number of processors to use. The processors are determined by a special hostfile.

- messagesize: The size of the message to exchange.
- numiters: The number of compute and communicate phases
- flopsperelement: The number of multiply-add steps
- readsperelement: The number of main memory reads
- writesperelement: The number of main memory writes

Patterns generates a deadlock free and efficient communication schedule at startup time for the given topology and number of processors to be used. The following topologies are supported:

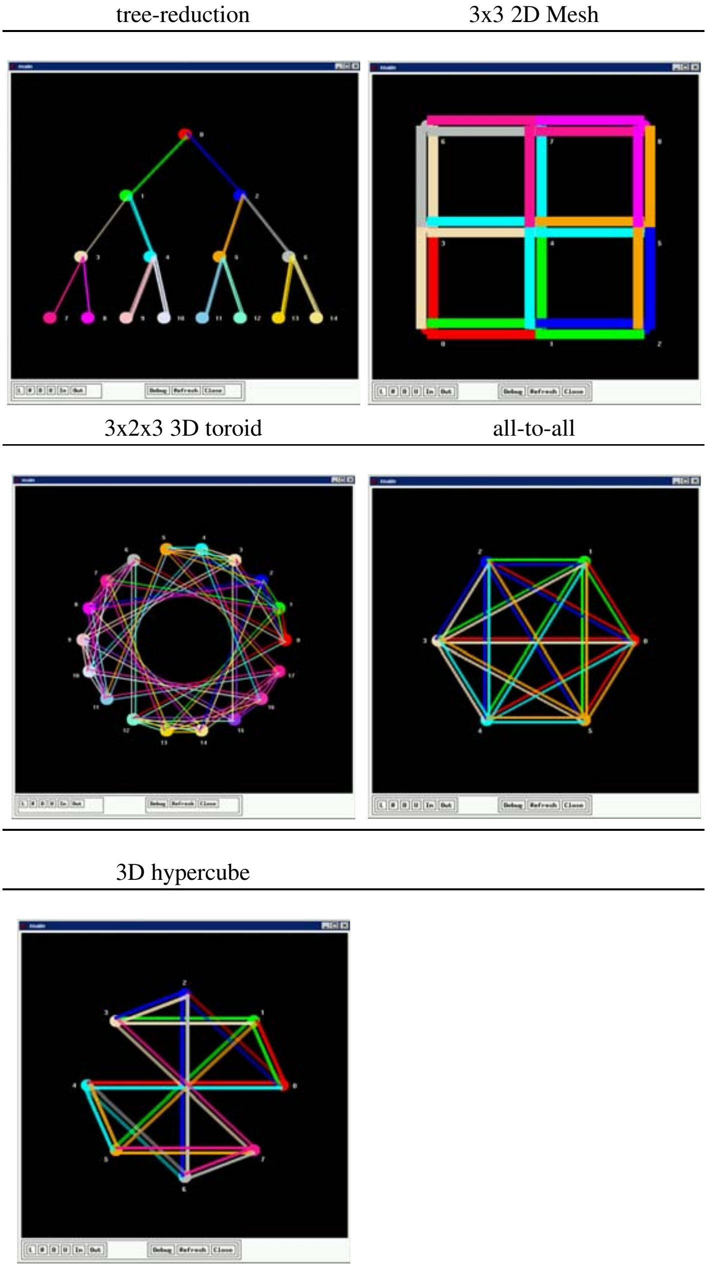
- n-dimensional mesh, neighbor communication pattern
- n-dimensional torus, neighbor communication pattern
- n-dimensional hypercube, neighbor communication pattern
- Binary reduction tree
- All-to-all communication

4 Evaluation of Offline VTTIF

We evaluated our offline inference framework with the various parallel benchmarks described in the previous section. Figure 3 shows the inferred application topologies of various patterns benchmark runs, as detected by our offline framework. These results suggest that there is indeed considerable promise in traffic-based topology inference: parallel program communication behavior can be inferred without any knowledge of the parallel application itself. Of course, more complex filtering processes may need to be used for more complex applications and complex network environments where parallel application traffic is just a part of the network traffic.

We also ran the application benchmarks described earlier. These results are also promising. Figure 4 shows a representative, the traffic matrix for an execution of the Integer Sort (IS) NAS kernel benchmark on 8 physical hosts, with the corresponding topology shown in Figure 5. The topology resembles an all-to-all communication, but the thickness of the edges vary indicating that the bandwidth requirements vary depending on the host pairs. A closer look at the traffic matrix reveals that $host_1$ receives data in the range of 20 MB from each of the other hosts, indicating that this is a communication intensive benchmark. Other hosts $host_2$ to $host_8$ transfer data of $\simeq 10 - 11$ MB with each other, almost half of that exchanged with $host_1$.

Notice that this information could be used to boost the performance of the IS benchmark if it were running in our VM computing model. Ideally, we would move the VM $host_1$ to a host with relatively high bandwidth links and reconfigure the virtual network with appropriate virtual routes over the physical network [10]. Such decisions need to be dynamic, as the properties of a physical network vary [14]. Without any intervention by the application developer or knowledge of the parallel application itself, it is feasible to infer the spatial and temporal [3] properties of the parallel application. Equipped with this knowledge, we can use VM checkpointing and migration along with VNET's virtual networking capabilities to create a efficient network and host environment for the application.



tree-reduction

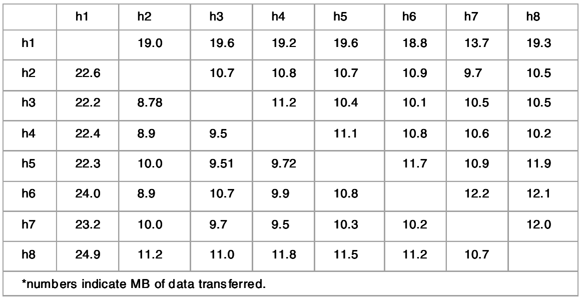
3x3 2D Mesh

3x2x3 3D toroid

all-to-all

3D hypercube

Fig. 3. The communication topologies inferred by the framework from the patterns benchmark. It shows the inferred tree-reduction, 3x3 2D Mesh, 3x2x3 3D toroid, all-to-all for 6 hosts and 3D hypercube topologies.



	h1	h2	h3	h4	h5	h6	h7	h8
h1		19.0	19.6	19.2	19.6	18.8	13.7	19.3
h2	22.6		10.7	10.8	10.7	10.9	9.7	10.5
h3	22.2	8.78		11.2	10.4	10.1	10.5	10.5
h4	22.4	8.9	9.5		11.1	10.8	10.6	10.2
h5	22.3	10.0	9.51	9.72		11.7	10.9	11.9
h6	24.0	8.9	10.7	9.9	10.8		12.2	12.1
h7	23.2	10.0	9.7	9.5	10.3	10.2		12.0
h8	24.9	11.2	11.0	11.8	11.5	11.2	10.7	

*numbers indicate MB of data transferred.

Fig. 4. The traffic matrix for the NAS IS kernel benchmark on 8 hosts.

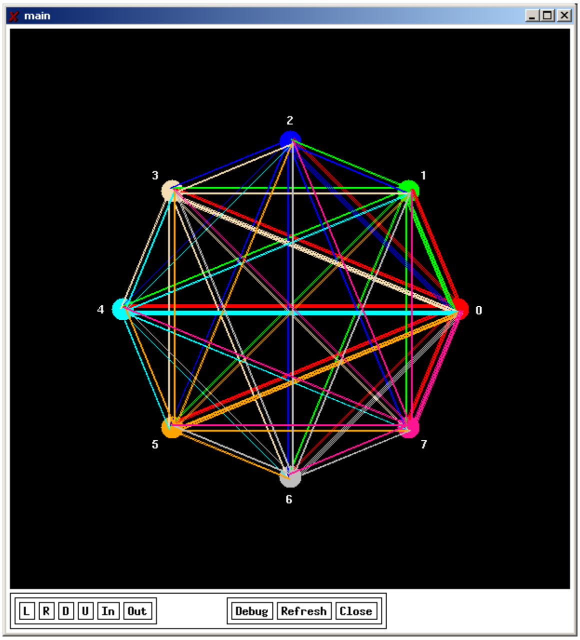


Fig. 5. The inferred topology for the NAS IS kernel benchmark

5 Online VTTIF

After working with offline parallel program topology inference on the physical hosts, the next step was to develop an online framework for a virtual machine environment. We extended VNET [12], our virtual networking tool, to include support for traffic analysis and topology inference. VNET allows the creation of layer 2 virtual networks interconnecting VMs distributed over an underlying TCP/IP networking infrastructure.

A VNET daemon manages all the network traffic of the VMs running on its host, and thus is an excellent place to observe the application's network traffic. All traffic monitoring is done at layer 2, providing flexibility in analyzing and filtering the traffic at many layers.

Due to a networking issue with the Virtuoso cluster, the work in the section was done on a slower cluster consisting of dual 1 GHz Pentium III processors with 1 GB of RAM and 30 GB hard disks. We used a switched 100 mbit network connecting the machines. As before VMWare GSX Server 2.5 was used, except here it was run on Red Hat Linux 7.3. The VMs were identical. A Dell PowerEdge 4400 (dual 1 GHz Xeon, 2 GB, 240 GB RAID) running Red Hat 7.1 was used as the VNET proxy machine.

5.1 Observing Traffic Phenomena of Interest: Reactive and Proactive Mechanisms

VMs can run for long periods of time, but their traffic may change dramatically over time as they run multiple applications in parallel or serially. In the offline VTTIF, monitoring and aggregation are triggered manually while running the parallel application. This is not possible in an online design. The online VTTIF needs a mechanism to detect and capture traffic patterns of interest, reacting automatically to interesting changes in the communication behavior of the VMs. It must switch between active states, when it is accumulating data and computing topologies, and passive states, when it is waiting for traffic to intensify or otherwise become relevant. Ideally, VTTIF would have appropriate information available whenever a scheduling agent requests it.

We have implemented two mechanisms for detecting interesting dynamic changes in communication behavior: reactive and proactive. In the reactive mechanism, VTTIF itself alerts the scheduling agent when it detects certain pre-specified changes in communication. For example, in the current implementation, VTTIF monitors the rate of traffic for all flows passing through it and starts aggregating traffic information whenever the rate crosses a threshold. If this rate is sustained, then VTTIF can alert the scheduling agent about this interesting behavior along with conveying its local traffic matrix.

In the proactive mechanism, VTTIF allows an external agent to make traffic-related queries such as: *what is traffic matrix for the last 512 seconds?* VTTIF stores sufficient history to answer various queries of interest, but it does not alert the scheduling agent, unlike the reactive mechanism. The agent querying traffic information can determine its own policy, for example polling periodically to detect any traffic phenomena of interest and thus making appropriate scheduling, migration and network routing decisions to boost parallel application performance. Figure 6 shows the high level view of the VNET-VTTIF architecture. The VM and overlay network scheduling agent may be located outside the VM-side VNET daemon, and all relevant information can conveyed to it so that it can make appropriate scheduling decisions.

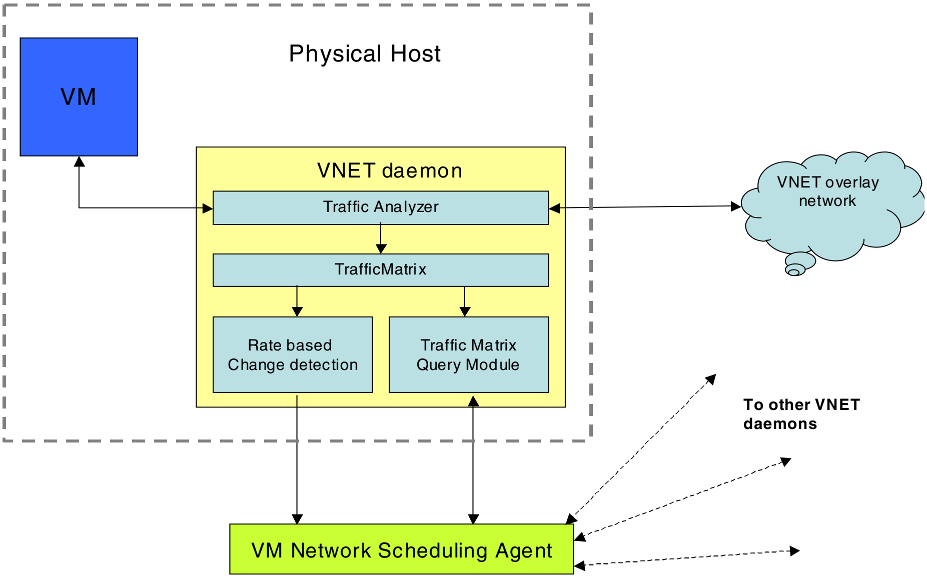


Fig. 6. The VNET-VTTIF topology inference architecture. VTTIF provides both reactive and proactive services for the scheduling agent.

5.2 Implementation

We extended VNET so that each incoming and outgoing Ethernet packet passes through a packet analyzer module. This function parses the packet into protocol headers (Ethernet, IP, TCP) and can filter it if it is irrelevant. Currently all non-IP packets are filtered out—additional filtering mechanisms can be installed here. Packets that are accepted are aggregated into a local traffic matrix. Specifically, for each flow, a row and column of the matrix are determined in this way. The matrix is stored in a specialized module TrafficMatrix. TrafficMatrix is invoked on every packet arrival.

Reactive mechanism The TrafficMatrix module does non-uniform discrete event sampling for each source/destination VM pair to infer the traffic rate between the pair. The functioning of rate_threshold mechanism is illustrated in Figure 7. It takes two parameters: byte_threshold and time_bound. Traffic is said to cross the rate_threshold, if for a particular VM pair, byte_threshold bytes of traffic is transmitted in a time less than time_bound. This is detected by time-stamping the packet arrival event whenever the number of transmitted bytes for a pair exceeds a integral multiple of byte_threshold. If two successive time-stamps are less than time_bound, this indicates our rate_threshold requirement has been met. Once a pair crosses the rate_threshold, TrafficMatrix starts accumulating traffic information for all the pairs. Before the rate_threshold is crossed, TrafficMatrix doesn't accumulate any information, i.e. it is a memoryless system. After the

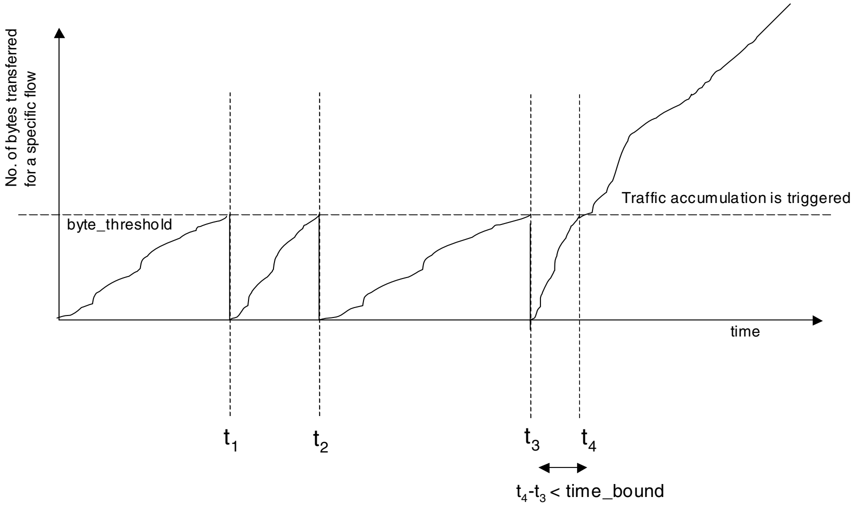


Fig. 7. The rate-threshold detection based reactive mechanism in VNET-VTTIF. Whenever two successive byte thresholds are exceeded within a time bound, the accumulation of traffic is triggered.

`rate_threshold` is crossed, TrafficMatrix alerts the scheduling agent in two situations. First, if the high traffic rate is sustained up to time t_{max}, then it sends all its traffic matrix information to the scheduling agent. In other words, TrafficMatrix informs the scheduling agent if an interesting communication behavior persists for a long enough period of time. The second situation is if the rate falls below the threshold and remains there for more than t_{wait} seconds, in which case TrafficMatrix alerts the scheduling agent that the application has gone quiet.

Figure 8 illustrates the operation of the reactive mechanism in flowchart form.

Proactive mechanism The proactive mechanism allows an external agent to pose queries to VTTIF and then take decisions based on its own policy. VTTIF is responsible solely for providing the answers to useful queries. TrafficMatrix maintains a history for all pairs it is aware in order to answer queries of the following form: What is the traffic matrix over the last n seconds? To do so, it maintains a circular buffer for all pairs in which each entry corresponds to the number of bytes transferred in a particular second. As every packet transmission is reported to TrafficMatrix, it updates the circular buffer for the particular pair. To answer the query, the last n entries are summed up, up to the size of the buffer.

The space requirements for storing the state history needs some consideration. The space requirements depends on the maximum value of n. For each pair, $4n$ bytes are needed for the circular buffer. If there are m VMs, then the total space allocation is $4nm^2$. For $n = 3600$ (1 hour) and $m = 16$ VMs, the worst case total space requirement

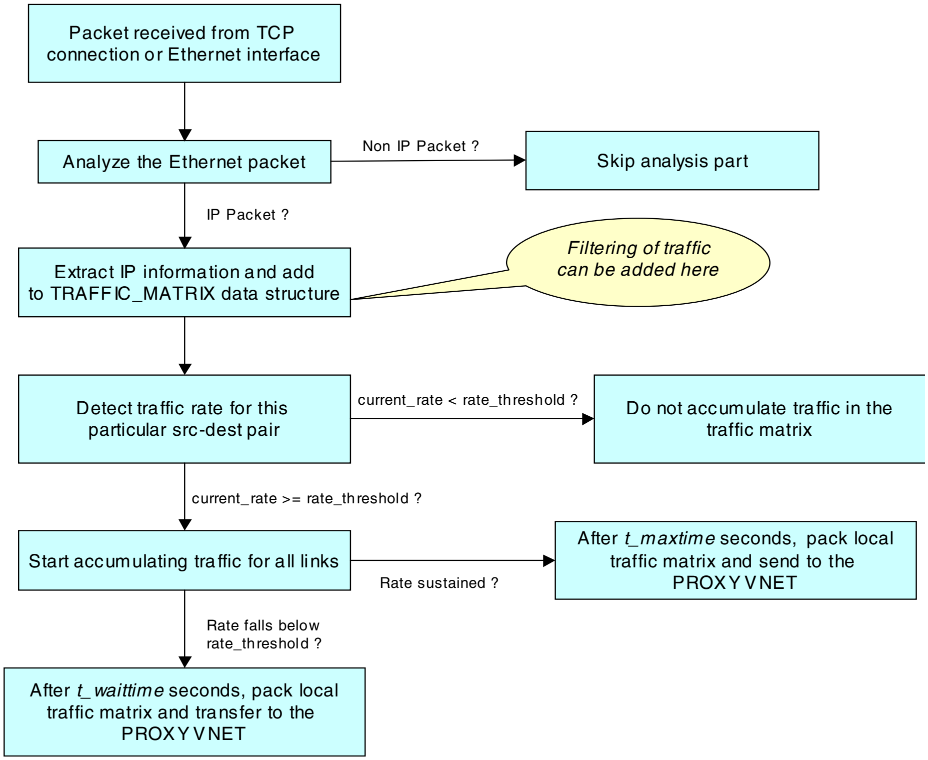

Packet received from TCP connection or Ethernet interface
Analyze the Ethernet packet
Non IP Packet ?
Skip analysis part
IP Packet ?
Extract IP information and add to TRAFFIC_MATRIX data structure
Filtering of traffic can be added here
Detect traffic rate for this particular src-dest pair
current_rate < rate_threshold ?
Do not accumulate traffic in the traffic matrix
current_rate >= rate_threshold ?
Start accumulating traffic for all links
Rate sustained ?
After t_maxtime seconds, pack local traffic matrix and send to the PROXY VNET
Rate falls below rate_threshold ?
After t_waittime seconds, pack local traffic matrix and transfer to the PROXY VNET

Fig. 8. The steps taken in the VM-side VNET daemon for the reactive mechanism.

is 3.7 Mbytes. A sparse matrix representation could considerably reduce this cost and thus the communication cost in answering the queries.

5.3 Aggregation

Aggregation of traffic matrices from the various VNET daemons provides a global view of the communication behavior exhibited by the VMs. Currently, we aggregate the locally collected traffic matrices to a global, centralized matrix that is stored on the VNET proxy daemon, which is responsible for managing the virtual overlay network in VNET. We use a push mechanism—the VNET daemons decide when to send their traffic matrix based on their reactive mechanism. A pull mechanism could also be provided, in which the proxy would request traffic matrices when they are needed based on queries.

The storage analysis of the previous sections assumes that we will collect a complete copy of the global traffic matrix on each VNET daemon—in other words, that we will follow the reduction to the proxy VNET daemon with a broadcast to the other VNET daemons. This is desirable so that the daemons can make independent decisions.

Method	Average	STDEV	Min	Max
Direct	0.529 ms	0.026 ms	0.483 ms	0.666 ms
VNET	1.563 ms	0.222 ms	1.277 ms	2.177 ms
VNET-VTTIF	1.492 ms	0.198 ms	1.269 ms	2.218 ms

Fig. 9. Latency comparison between VTTIF and other cases

However, if we desire only a single global copy of the whole matrix, or a distributed matrix, the storage and computation costs will scale with the number of VMs hosted on the VNET daemon.

Scalability is an issue in larger instances of the VM-based distributed environment. Many possibilities exist for decreasing the computation, communication and storage costs of VTTIF. One optimization would be to maintain a distributed traffic matrix. Another would be to implement reduction and broadcast using a hierarchical structure, tuned to the performance of the underlying network as in ECO [9]. Fault tolerance is also a concern that needs to be addressed.

5.4 Performance Overhead

Based on our measurements, VTTIF has minimal impact on bandwidth and latency. We considered communication between two VMs in our cluster, measuring round-trip latency with ping and bandwidth with ttcp. Figure 9 compares the latency between the VMs for three cases:

- Direct communication. Here VNET is not involved. The machines communicate locally using VMWare's bridged networking. This measures the maximum performance achievable between the hosts, without any network virtualization.
- VNET. Here we use VNET to proxy the VMs to a different network through the PowerEdge 4400. This shows the overhead of network virtualization. Note that we are using an initial version of VNET here without any performance enhancements running on a stock kernel. We continue to work to make VNET itself faster.
- VNET-VTTIF. This case is identical to VNET except that we are monitoring the traffic using VTTIF.

There is no significant difference between the latency of VNET and VNET-VTTIF.

Figure 10 shows the effect on throughput for the three cases enumerated above. These tests were run using ttcp with a 200K socket buffer, and 8K writes. The overhead of VNET-VTTIF compared to VNET is a mere 4.1%.

5.5 Online VTTIF in Action

Here we show results of running a parallel program in the online VNET-VTTIF system. We use the NAS Integer Sort (IS) benchmark for illustration because of its interesting

Method	Throughput
Direct	11485.75 KB/sec
VNET	8231.82 KB/sec
VNET-VTTIF	7895.06 KB/sec

Fig. 10. Throughput comparison between VTTIF and other cases

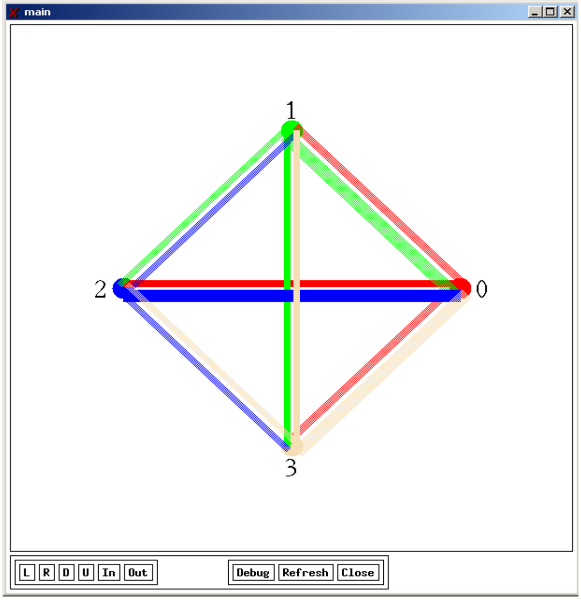
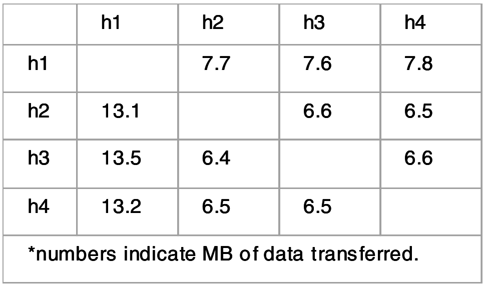


Fig. 11. The PVM IS benchmark running on 4 VM hosts as inferred by VNET-VTTIF

communication pattern and traffic matrices. We executed NAS IS on 4 VMs interconnected with VNET-VTTIF. Here, the Virtuoso cluster, as used in the offline work, was employed. The rate-based reactive mechanism was used to intelligently trigger aggregation mechanisms on detecting traffic flow from the benchmark. When the benchmark finished executing, the traffic matrix was automatically aggregated at the VNET proxy. For comparison, we also executed the same benchmark with on 4 physical hosts and analyzed the traffic using the offline method.

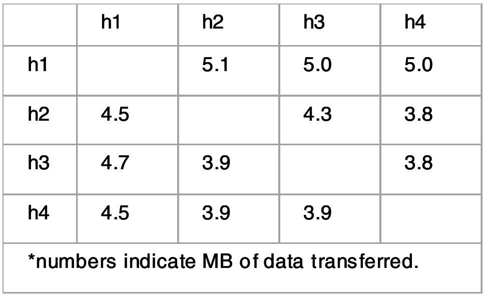
Figures 11 and 12 show the topology and traffic matrix as inferred by the online system. Figure 13 shows the matrix inferred from the physical hosts using the offline method. The topology for the offline method is identical to that for the offline method and is not shown. There are some differences between the online and offline traffic matrices. This can be attributed to two factors. First, the byte count in VNET-VTTIF includes the size of the entire ethernet packet whereas in the offline method, only the TCP payload size is taken into account. Second, tcpdump, as used in the offline method,



	h1	h2	h3	h4
h1		7.7	7.6	7.8
h2	13.1		6.6	6.5
h3	13.5	6.4		6.6
h4	13.2	6.5	6.5	

*numbers indicate MB of data transferred.

Fig. 12. The PVM IS benchmark traffic matrix as inferred by VNET-VTTIF



	h1	h2	h3	h4
h1		5.1	5.0	5.0
h2	4.5		4.3	3.8
h3	4.7	3.9		3.8
h4	4.5	3.9	3.9	

*numbers indicate MB of data transferred.

Fig. 13. The PVM IS benchmark traffic matrix running on physical hosts and inferred using the offline method.

is configured to allow packet drops by the kernel packet filter. In the online method, VNET's packet filter is configured not to allow this. Hence, the offline method is seeing a random sampling of packets while the online method is seeing all of the packets.

The main point here is that the online method (VNET-VTTIF) can effectively infer the application topology and traffic matrix for a BSP parallel program running in a collection of VMs.

6 Conclusions and Future Work

We have demonstrated that it is feasible to infer the topology and traffic matrix of a bulk synchronous parallel application running in a virtual machine-based distributed computing environment by observing the network traffic each VM sends and receives. We have also designed and implemented an online framework (VTTIF) for automated inference in such an environment. This monitoring can be piggy-backed, with very low overhead, on existing, necessary infrastructure that establishes and optimizes network connectivity for the VMs. We are now focusing on expanding this work in the following ways:

- We plan to generalize our results to other forms of applications and to determine the limits of network behavior that can be inferred.
- We are implementing a general query interface for querying traffic matrix information from our system.
- We plan to evaluate our system in more complex network environments, possibly revealing more filtering and topology inference based issues.
- We plan to improve the scalability and resilience of the system by adopting a distributed information aggregation approach.
- We intend to exploit the topological information provided by VNET-VTTIF to do optical call path setup on behalf of applications in networks that support it.
- We are working on leveraging VNET to do passive network measurement as a side effect of inter-VM data transfers.
- Finally, we are working on adaptation algorithms that will make use of VNET-VTTIF and network information to guide VM placement and migration, and VNET overlay topology construction and routing in order to maximize the performance of unmodified applications [7].

References

1. Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, D., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (Fall 1991), 63–73.
2. Dilger, A., Flierl, J., Begg, L., Grove, M., and Dispot, F. The PVM patch for POV-Ray. Available at http://pvmpov.sourceforge.net.
3. Dinda, P. A., Garcia, B., and Leung, K. S. The measured network traffic of compiler-parallelized programs. In *Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001)* (September 2001), pp. 175–184.
4. Figueiredo, R., Dinda, P. A., and Fortes, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)* (May 2003).
5. Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Mancheck, R., and Sunderam, V. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.
6. Gerbessiotis, A. V., and Valiant, L. G. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing 22*, 2 (1994), 251–267.
7. Jerry Rolia, Jim Pruyne, X. Z., and Arlitt, M. Grids for Enterprise Applications. In *Proceedings of the 9th Workshops on Job Scheduling Strategies for Parallel Processing (JSSPS 2003)* (June 2003).
8. Leighton, F. T. *Introductio to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman, 1992.
9. Lowekamp, B., and Beguelin, A. ECO: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the International Parallel Processing Symposium (IPPS 1996)* (1996), pp. 399–405.

10. SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. E. The end-to-end effects of internet path selection. In *SIGCOMM* (1999), pp. 289–299.

11. STASKO, J. Samba Algorithm Animation System. Available at http://www. cc. gatech,edu/gvu/ softviz/algoanim/samba, html.

12. SUNDARARAJ, A., AND DINDA, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). To Appear. Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

13. WHITE, S., ALUND, A., AND SUNDERAM, V. S. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing 26*, 1 (1995), 61–71.

14. ZHANG, Y., DU, N., PAXSON, E., AND SHENKER, S. the constancy of internet path properties, 2001.

Multi-toroidal Interconnects: Using Additional Communication Links to Improve Utilization of Parallel Computers

Yariv Aridor[1], Tamar Domany[1], Oleg Goldshmidt[1], Edi Shmueli[1],
Jose Moreira[2], and Larry Stockmeier[3]

[1] IBM Haifa Research Labs, Haifa, Israel
{yariv,tamar,olegg,edi}@il.ibm.com
[2] IBM Watson Research Center, Yorktown, NY
jmoreira@us.ibm.com
[3] IBM Almaden Research Center
stock@acm.org

Abstract. Three-dimensional torus is a common topology of network interconnects of multicomputers due to its simplicity and high scalability. A parallel job submitted to a three-dimensional toroidal machine typically requires an isolated, contiguous, rectangular partition connected as a mesh or a torus. Such partitioning leads to fragmentation and thus reduces resource utilization of the machines. In particular, toroidal partitions often require allocation of additional communication links to close the torus. If the links are treated as dedicated resources (due to the partition isolation requirement) this may prevent allocation of other partitions that could, otherwise, use those links. Overall, on toroidal machines, the likelihood of successful allocation of a new partition decreases as the number of toroidal partitions increases.

This paper presents a novel "multi-toroidal" interconnect topology that is able to accommodate multiple adjacent meshed and toroidal partitions at the same time. We prove that this topology allows connecting every free partition of the machine as a torus without affecting existing partitions. We also show that for toroidal jobs this interconnect topology increases machine utilization by a factor of 2 to 4 (depending on the workload) compared with three-dimensional toroidal machines. This effect exists for different scheduling policies. The BlueGene/L supercomputer being developed by IBM Research is an example of a multi-toroidal interconnect architecture.

1 Introduction

Tightly coupled multicomputers provide a natural way to build large-scale parallel systems. A tightly coupled multicomputer consists of a collection of *nodes*. Each node has one or several CPUs, memory, and network connections. Such systems are intended to run massively parallel computational jobs. A typical job requires a set of nodes, called a *partition*, connected in a particular fashion, e.g.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 144–159, 2005.

a three-dimensional rectangular block wired as mesh or torus. The job management system has to allocate a partition to each job and to schedule the waiting jobs optimally in order to maximize the machine utilization and to reduce the jobs' response times.

The high performance network that connects the nodes is designed with the jobs' topology requirements in mind. A frequently used interconnect topology is a three-dimensional mesh or torus where every node is connected to six neighbors, two in each dimension (a torus differs from a mesh in that the six edges are connected in a wrap-around fashion). This interconnect topology is simple, scalable (the number of links grows linearly with the machine size), and fits many types of real-world computations. Examples of three-dimensional toroidal parallel systems are the Cray T3D and T3E machines [2,3,4].

Job partitions are usually allocated contiguously, i.e. the constituent nodes are geometrically adjacent. Contiguous allocation simplifies allocation algorithms and facilitates partition isolation, i.e. localization of the intra-job communications within the partition. The latter is required both for security reasons and/or to reduce message congestion on shared network links. Under the isolation requirement, the nodes that form a partition and the network links that connect them are dedicated resources used by at most one job at a time.

Efficient partition allocation is of critical importance to the system performance in terms of resource utilization and job response times. As shown in the next section, a toroidal partition often requires allocation of additional links to close the torus. If the links are treated as dedicated resources, this prevents allocation of other partitions that could, otherwise, use those links. Overall, on toroidal machines, the likelihood of successful allocation of a new partition decreases as the number of toroidal partitions increases. The problem is particularly acute when a first-come-first-served (FCFS) scheduling is used. Backfilling [7,8] is an improvement over FCFS, but we show below that the adverse effect of isolated toroidal partitions on utilization exists independently of the scheduling policy used.

In this paper we present a novel approach, hereafter called *multi-toroidal topology*, that augments the traditional toroidal interconnect with additional links to improve machine utilization while allocating isolated rectangular partitions connected as mesh or a torus. Unlike other existing solutions, such as full crossbar interconnects [5], the number of additional links is small, linear in the number of allocation units in the machine (see Section 2 for more details), and thus is inexpensive and highly scalable. A variant of multi-toroidal interconnect is implemented in the upcoming BlueGene/L supercomputer developed by IBM Research [16].

Multi-toroidal topology suggests a practical compromise between additional hardware complexity (due to additional communication links) and better system performance (in terms of utilization) gained by introducing it. Utilization is increased due to the ability to allocate multiple adjacent meshed and toroidal partitions thanks to the additional connectivity options and to the possibility of non-contiguous allocation of isolated partitions leading to less machine fragmentation.

In this paper we shall restrict ourselves to isolated contiguous rectangular partitions only. We thus focus on the ability of multi-toroidal interconnect to accommodate multiple adjacent meshed and toroidal partitions at the same time. We will show that this property alone gives the proposed interconnect a significant advantage over the traditional three-dimensional torus machines. We will defer the discussion of non-contiguous allocation to a future work.

We experimented with partition allocation on a machine with a regular multi-toroidal interconnect (see Section 2). We measured the machine utilization for two job logs of real supercomputing centers, simulating allocation of mesh and toroidal partitions on 512-unit machine, and varying the offered load. We explored two scheduling policies: FCFS and aggressive backfilling. We compared the results with those of a basic three-dimensional toroidal architecture with the same workloads, and found a significant improvement in utilization (by a factor of 2 to 4 for purely toroidal workloads) of the multi-toroidal machine compared to the traditional one.

The rest of the paper is organized as follows. In section 2 we discuss multi-toroidal topology in detail. Section 3 describes our simulation environment. Section 4 presents the main quantitative results. Section 5 concludes the paper and discusses directions of future research.

2 The Multi-toroidal Topology

We shall use a model of the architecture of a machine that provides a topologically correct representation of the traditional three-dimensional toroidal machines and, at the same time, can be generalized to more complex topologies. This architecture separates the network connections from the processing components of the machine. The model takes into account that for scalability reasons large systems may operate in terms of *allocation units* that contain a number of nodes (as a special case, an allocation unit may contain just one node). We shall assume that an allocation unit is composed of a three-dimensional mesh of nodes and has three network switches, one for each dimension[1]

Fig. 1 illustrates an allocation unit with its switches. Two of the ports of each switch are connected to the opposing sides of the allocation unit. Remaining ports can be connected to ports of other switches by communication links. Only one link can be connected to each port. Some ports may be left unused. A switch has the capability of making internal connections between any two of its ports, thus connecting allocation units with each other.

The links connecting the switches determine the interconnect topology of the machine. We will assume that the link topology is identical for all the dimensions of the computer. We also assume that no link connects switches that belong to different dimensions, i.e. different dimensions are independent of each other. This

[1] It is easy to see that this guarantees that the topology of a rectangular network of allocation units (mesh or torus) will be identical to that of the constituent nodes.

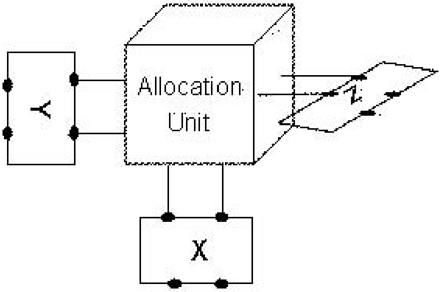


Fig. 1. An allocated unit and three switches

is the case, for example, with the BlueGene/L machine [16]. This separation of dimensions permits a view of the three-dimensional machine as a collection of independent one-dimensional "lines" in each dimension (hereafter, X-line, Y-line and Z-line). Using dimension X as an example, links exist only between switches that belong to the same X-line, and all X-lines have the same link configuration. This will allow us to focus on a single line rather than consider the full three-dimensional machine throughout the rest of the paper.

Fig. 2 shows an X-line of a mesh-connected machine. The switches in an X-line are connected to their nearest neighbors in a linear fashion and are enumerated in ascending order from left to right.

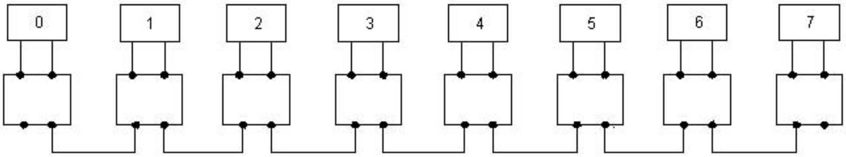


Fig. 2. An X-line of a mesh-connected machine.

Fig. 3 shows an X-line of a toroidal machine. The switches are connected in a cyclic fashion, namely 0, 2, 4, 6, 7, 5, 3, 1 and back to 0. The torus could be obtained from the mesh of Fig. 2 by adding a link between switch 0 and switch 7. However, there may be a physical limitation on the length of the cables, and the torus is often wired as shown in Fig. 3. A three-dimensional torus architecture is defined by replicating the links of a single line to all the lines in the same dimension.

Multiple meshed partitions can co-exist in a single line since each only needs links between the switches that form it. Toroidal partitions that consist of one

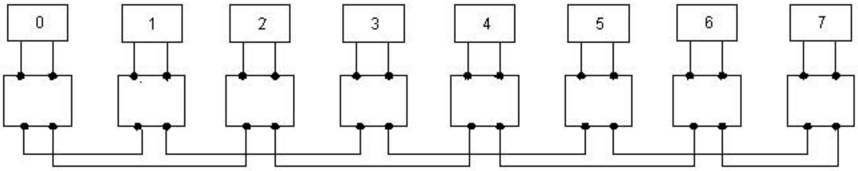


Fig. 3. An X-line of a toroidal machine.

allocation unit can live together as well: to create one it is enough to use an internal link between the two switch ports that are connected to the allocation unit. However, only one toroidal partition containing two or more allocation units can exist in a single line of a toroidal machine at a time. For example, the partition {0,1} in Fig. 4 can be connected as a torus only by using *all* of the links in the line. Therefore, this partition cannot co-exist with other partitions of more than one allocation unit, such as {6,7}, in the same line.

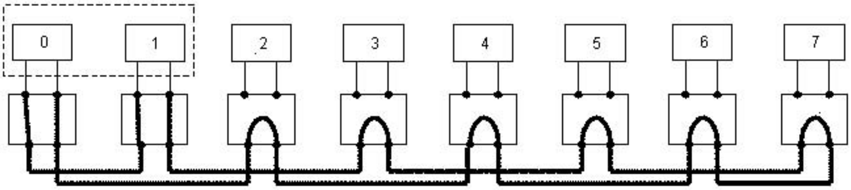


Fig. 4. A toroidal partition containing two allocation units {0,1}.

We propose augmenting the toroidal interconnect with additional links to overcome this limitation. This assumes, of course, that the switches have free ports that can be connected. Fig. 5 shows additional links (in bold) on top of the toroidal line of Fig. 3. Each switch has now six ports instead of four ports. Note that the total number of links we use is linear with the size of the machine in allocation units. Specifically, for each machine line of N allocation units we now have $2N-1$ links in that line, for $N > 2$. This number may be much smaller than the total number of links in the machine. Thus, the cost of such augmentation is low and, unlike the full crossbar topology, this interconnect is still practical even for a very large machine.

This connection scheme has a very important property that conventional toroidal topology lacks; multiple toroidal partitions can co-exist in a single line. This property is the central motivation for augmenting the traditional toroidal interconnect with additional links (as well as for the "multi-toroidal" moniker).

In the following section, we will formulate and prove a central theorem that is strictly valid for the particular wiring scheme shown in Fig. 5. We do not lose generality, however: other schemes that allow wiring toroidal partitions that do not occupy the whole line will have similar advantages. The described topology is regular, scalable, efficient, and, as shown below, greatly simplifies the algorithms of partition allocation.

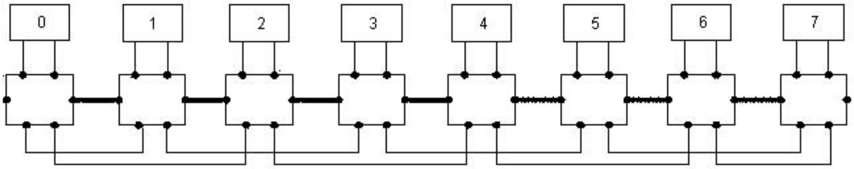


Fig. 5. Additional links (in bold) added to a toroidal line.

2.1 Properties of the Multi-toroidal Interconnect

We describe several properties of the multi-toroidal interconnect. We focus on a single line in one dimension. The generalization to a three-dimensional machine is straightforward due to the independence of the machine dimensions and the fact that all the lines of the dimensions are identical.

Property 1: Any partition of size $N = 1$ in a particular dimension can be wired without using any links between switches in that dimension. To close a torus we will use the internal connection between the two ports that are connected to the allocation unit.

Property 2: Any mesh partition of size $N > 1$ in a particular dimension can be wired using exactly $N - 1$ links in each line it spans in that dimension. We simply connect the switches in a linear order: $i, i+1, ., i+N-1$. Such wiring uses exactly $N - 1$ links (cf. Fig. 6).

Property 3: Any toroidal partition of size $N > 2$ in a particular dimension can be wired using exactly N links in each line it spans in that dimension. We distinguish between two cases: If N is odd, we connect the switches in the following order: $i, i+2, , i+N-1, i+N-2, i+N-4, , i+1, i$ (cf. Fig. 7). If N is even the order is $i, i+2, , i+N-2, i+N-1, i+N-3, , i+1, i$ (cf. Fig. 8). With the above ordering, each switch is connected to each of its neighbors by a single link, so a total of N links are used.

Property 4: A toroidal partition of size $N = 2$ at either the left or the right boundary of the machine in a particular dimension can be wired using exactly

2 links in each line it spans in that dimension. This is obvious from Fig.5: there are two links that connect allocation units 0 and 1 (or 6 and 7).

Property 5: A toroidal partition of size $N = 2$ which is not next to either the left boundary or the right boundary of the machine in a particular dimension can be wired using exactly 3 links in each line it spans in that dimension. It is the only case where links and switches lying beyond the geometrical boundaries of the partition are used.

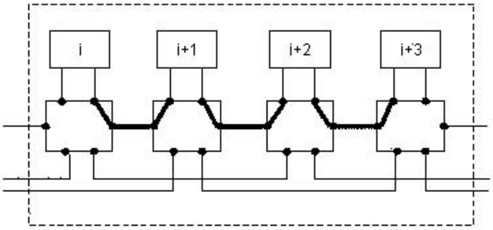


Fig. 6. A mesh partition wired in a linear manner.

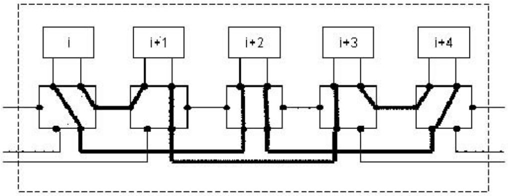


Fig. 7. A toroidal partition of odd size.

There are two ways to connect such a partition: by using an additional link to the right or to the left of the partition (*right-oriented* or *left-oriented* wiring, respectively). Fig. 9 shows an example of two such partitions. The first uses allocation units 1 and 2 and is right-oriented. The second uses allocation units 5 and 6 and is left-oriented. Fig. 9 also illustrates a fragmentation problem with a mix of co-allocated right and left-oriented partitions. Suppose we now need to connect allocation units 3 and 4 as a torus. Obviously, we lack links to close the torus, and the allocation will fail.

The solution is to use a uniform orientation for all internal toroidal partitions of size two in a particular dimension. Without loss of generality, we chose to use right-oriented wiring for all partitions. As seen in Fig. 10, allocation units 3 and 4 can now be connected as a torus. We have described a set of connection rules

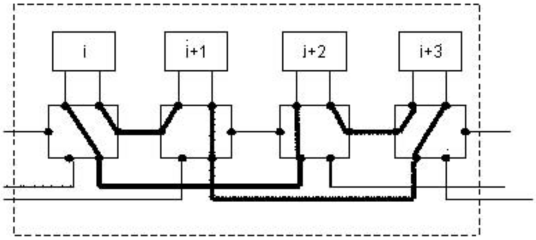


Fig. 8. A toroidal partition of even size.

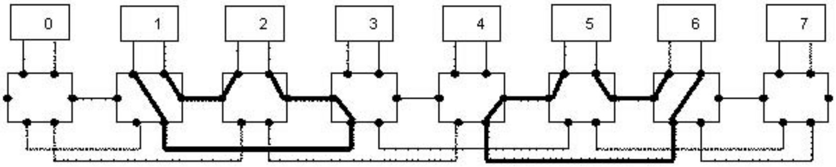


Fig. 9. Two toroidal partitions (1,2 and 5,6) of size two.

that satisfy the above properties. We now proceed to prove the main theorem behind the multi-toroidal architecture.

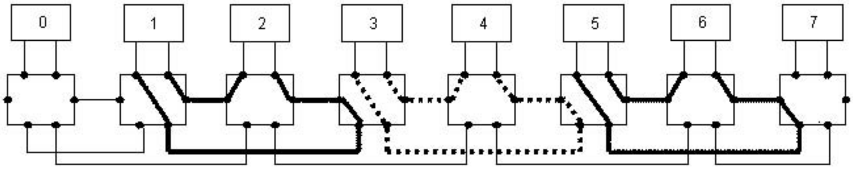


Fig. 10. Right-oriented wiring solves allocation problems.

Theorem:

Let $P_1, \ldots, P_k$, be co-allocated meshed or toroidal contiguous partitions that are wired according to the above connection rules. Let P be a new contiguous partition that can fit into the free space of a multi-toroidal machine. P can always be connected according to the same rules without changing the wiring of $P_1, \ldots, P_k$.

Proof:

Allocation of P can fail only if one (or more) of its adjacent partitions uses one or more links that P needs in one of the lines. By assumption, we always connect partitions according to the rules described above. Assume in addition that we always use right-oriented wiring in this line. The only way P may need a link that is allocated to one of its neighbors in the line is if P and the neighbor are both of size 2 in this dimension, neither is at the edge of the machine, and if they are wired using different orientation (cf. Fig. 9). The last statement contradicts the assumption that we always choose right-oriented wiring, thus proving the theorem for a single line. Since all the lines are independent, the theorem holds for a three-dimensional machine as well.

Note that we could have equally chosen left-oriented wiring, or different orientation in different dimensions, or even different orientations in different lines in the same dimension — the proof will still hold.

The significance of the theorem lies in the proof that any new partition can be wired without modifying the existing ones, and in the following corollary that stems directly from it.

Corollary:

Allocating meshed or toroidal partitions according to the connection rules described above allows us to dispense with searching for a suitable set of links to connect the partition as requested. The above theorem guarantees that such a set exists, and the connection rules define the link set to be used.

As noted above, the theorem is strictly valid for the particular topology we have chosen. In general, however, any alternative scheme that allows allocation of multiple tori in a single line will be similarly beneficial. It may be not possible to define a set of fixed connectivity rules for any given variant of multi-toroidal topology. Accordingly, the allocation algorithms may have to be augmented with a search for a suitable link set. Multiple valid link sets may exist for the same partition, and an algorithm may be needed to determine the optimal one. Our connectivity rules are, in fact, such an algorithm: they determine a valid link set with the minimal number of wires and satisfy the above theorem. We defer discussion of more general cases to a future work.

3 The Simulation Environment

We simulated a three-dimensional machine of 512 ($8 \times 8 \times 8$) allocation units, connected as shown in Fig. 5 above. We simulated a batch system in which arriving jobs are placed in a queue in the order of arrival. The scheduler is invoked upon every job arrival and job termination to schedule queued jobs (if any) for execution according to a specified policy. We experimented with both FCFS and aggressive backfilling scheduling policies. With the former, if no partition is found for the first job in the queue the system will wait until one or more running jobs terminate before attempting to allocate space for the first waiting job again. With backfilling, if we cannot start the first waiting job, we traverse the queue trying to schedule other jobs out of order.

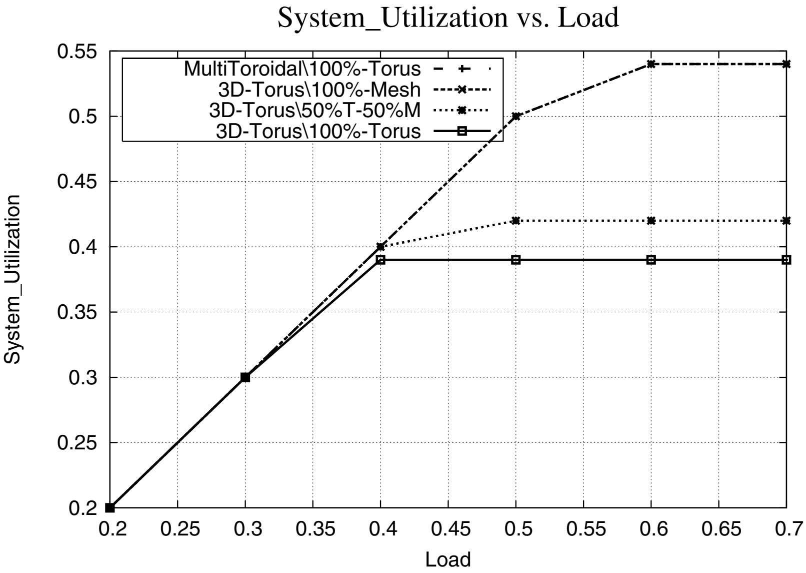
We used two job logs of real parallel systems as inputs to the simulator: the Cornell Theory Center (CTC) SP2 and the San Diego Supercomputer Center (SDSC) SP2. Both logs are publicly available from [1]. Each job entry in these logs contained the job's size, arrival time, actual and estimated runtimes, and other descriptive fields. The CTC log represents a 512 node machine, and the SDSC log represents a 128 node machine. For the latter we multiplied the job sizes by 4, to scale to our 512 node machine.

Neither of these systems is a 3-dimensional toroidal machine. Nor do the logs contain the jobs' shapes, only scalar sizes. We used them because of the scarcity of publicly available realistic workloads that provide useful statistics. To adjust for the shortcomings, we had to transform the scalar sizes to 3-dimensional shapes and to specify the topology (mesh or torus) of each job. We used a simple algorithm for the size transformation: we calculated three integers, a, b and c, so that each of these integers was in the range of $1 \ldots 8$, and $a \times b \times c$ was equal to the job's size. These integers can easily be found in a first-match manner u sing three nested loops running from 1 to 8. We then set the job shape to be $a \times b \times c$. If all combinations of $a \times b \times c$ have been tried and none is found equal to the job size, we use the first combination at which $a \times b \times c$ is minimal but still larger than the job size. Accordingly, a job of size 6 will request a partition's shape of $1 \times 1 \times 6$ and a job of size 27 will request a partition with shape of $3 \times 3 \times 3$. To determine the topology we used a simple probabilistic model that outputs "torus" with a probability of P_t and "mesh" with a probability of $1 - P_t$.

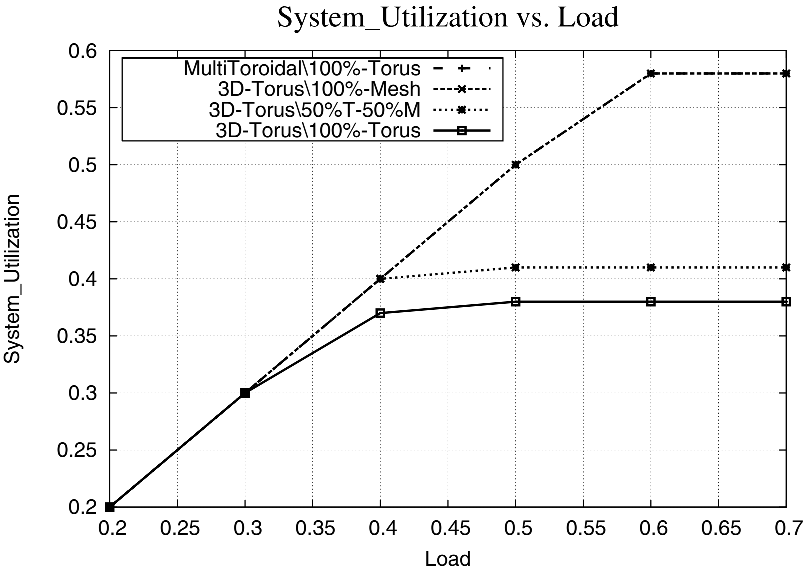
To simulate different offered loads we multiplied the jobs' arrival times in the logs by different constant factors leaving the rest of the logs' characteristics unchanged. For each simulation, we calculated the average system utilization (see [6] for details) to evaluate the system performance as a function of offered load.

For partition allocation, we used a brute force first-fit algorithm. Its input is the shape of a rectangular partition. We search for a free area with this exact shape, allowing for rotation. The search always starts from a specific node (in our case, the node in location (0,0,0) of the machine) and proceeds independently in every dimension.

We rely on the theorem of Section 2.1 to guarantee that any free partition in the multi-toroidal machine can be wired appropriately, as a mesh or a torus. The rules described in Section 2.1 prescribe link allocation. Therefore, any free partition of the required shape can be used to run a job. Note that the theorem guarantees that any space allocation algorithm usable on traditional toroidal machines (and more generally, on hypercubes) can be used instead of ours. Any improvements in spatial allocation, including non-contiguous partition allocation, are left for future research.

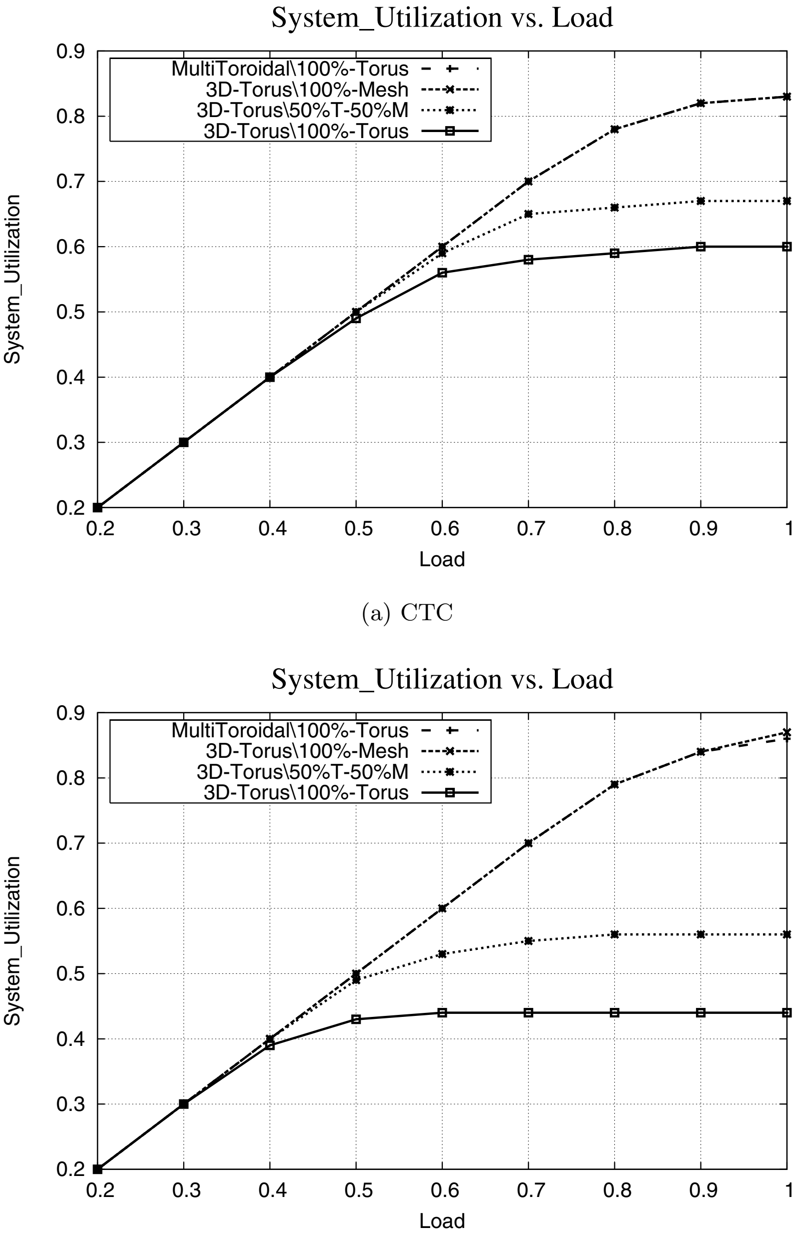


(a) CTC



(b) SDSC

Fig. 11. System utilization of a simulated multi-toroidal machine versus a simulated three-dimensional toroidal machine — scheduling of different mixtures of torus and mesh requests with FCFS: The graphs for the multi-toroidal machine with toroidal jobs only and toroidal machine with mesh jobs only completely overlap.



(a) CTC

(b) SDSC

Fig. 12. System utilization of a simulated multi-toroidal machine versus a simulated three-dimensional torus — scheduling of mixtures of torus and mesh requests with aggressive backfilling: The graphs for the multi-toroidal machine with toroidal jobs only and toroidal machine with mesh jobs only completely overlap.

4 Simulation Results

We compared the performance of a multi-toroidal machine of Fig. 5 with that of a 3-dimensional toroidal machine of equal size. For each log, we ran three sets of simulations on the toroidal machine. These simulation sets used different mixtures of toroidal and mesh jobs. In the first set, all jobs requested a mesh. In the second, 50% of the jobs requested a mesh and the rest requested a torus. In the third, all jobs requested a torus. Each set consisted of simulations with offered loads ranging between 20% and 100%, as described in Section 3. Then, we ran a single set of simulations on the multi-toroidal machine of Fig.5. In these simulations, all the jobs requested toroidal partitions.

The results for the system utilization are shown in Fig. 11 and Fig. 12 for FCFS and backfilling scheduling, respectively. Overall, utilization is lower for FCFS scheduling, which is consistent with numerous other results in the literature [9,10,11,12,13,14,15]. Torus-heavy workloads result in lower utilization of toroidal machines than workloads that consist of meshes only, and the utilization decreases as the percentage of toroidal jobs in the workload increases. The reason was explained in Section 2: toroidal partitions that contain more than one allocation unit can consume many links, thus preventing allocation of other partitions in a large fraction of the remaining free space.

The utilization of the multi-toroidal machine is higher compared to the toroidal machine, even though all the jobs are toroidal. Specifically, it is equal to the utilization of the toroidal machine with mesh jobs only. If we think of allocation of only toroidal partitions as a worst case scenario, then the worst case performance of the multi-toroidal machine is equal to the best-case performance (with allocation of only mesh jobs) of the toroidal machine. This readily follows from the central theorem of Section 2.1.

This effect is qualitatively independent of the scheduling strategy; the advantage of the multi-toroidal topology is manifested clearly for both FCFS and backfilling scheduling schemes. To assess the difference quantitatively, one should focus on the points where each graph first deviates from a straight line: even though utilization can be improved further with higher loads, in online systems this deviation means that more jobs keep arriving than the system can accommodate, and the system will saturate unless some jobs are rejected. For fully toroidal workloads derived from the SDSC logs, the multi-toroidal machine may improve utilization by as much as a factor of 2 over the toroidal machine: 60% vs. 30% with FCFS scheduling (Fig. 11b), 80% vs. 40% with aggressive backfilling (Fig. 12b).

For fully toroidal workloads derived from the CTC logs, the improvement is more modest: 50% vs. 40% with FCFS scheduling (Fig. 11a), 80% vs. 60% with aggressive backfilling. This is partly due to the particular algorithm we used to generate the jobs' shapes from their scalar sizes (cf. Section 3 above). It eases allocation of tori on traditional toroidal machines by generating partition shapes that are "slim" in one or two dimensions. For example, a toroidal partition of size 8 will be assigned a shape of $8 \times 1 \times 1$ and will be wired as a torus without consuming extra links. Similarly, a partition of size 64 will be shaped as $8 \times 8 \times 1$, and there will be no extra links involved in wiring it as a torus, either. "Fatter" partitions such as 2 × 2 × 2 and 4 × 4 × 4 will consume many more links when connected as tori (cf. Section 2). In fact, only two 4 × 4 × 4 partitions can be fit into an 8 × 8 × 8 torus simultaneously, for the overall utilization of 25%.

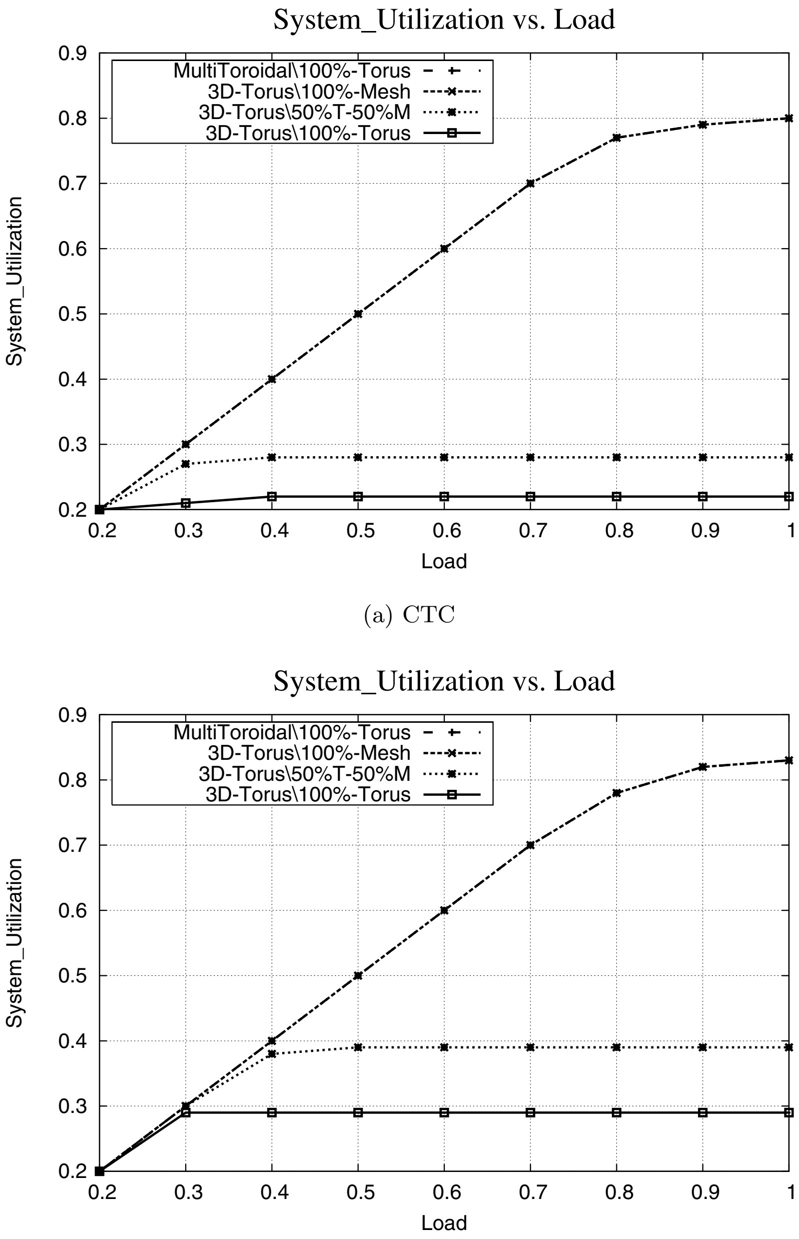
We experimented with "fat" shapes. To this end, we modified our shape-generating algorithm to start from a minimal size of 2 in each dimension. Thus, the smallest job will request a 2 × 2 × 2 partition. The results of these the simulations with backfilling are shown in Fig. 13. It is obvious that the utilization of a three-dimensional torus is simply dismal in the case of torus-heavy workloads, while the results for the multi-toroidal machine (and for the purely mesh workloads) are the same as in Fig. 12, as could be expected. The multi-toroidal machine improves the resource utilization by toroidal jobs by a factor that lies in the range of 2.5 (Fig. 13b) to 4 (Fig. 13a).

Note that the spatial allocation algorithm we used in all our experiments is not optimized in any way — it is a first fit brute force search, as described in Section 3. A significant body of research comparing different allocation schemes for mesh partitions on three-dimensional toroidal machines: first fit, best fit, worst fit, lookahead allocation, etc [9,10,11,12,13,14,15] show that the difference in machine utilization between the different allocation schemes in not significant — at most of the order of a few per cent. Part of our future research agenda is to experiment with various algorithms for different kinds of multi-toroidal machines and to validate that the advantage of the multi-toroidal interconnect is maintained.

5 Concluding Remarks

We have presented a new connectivity scheme that augments the topology of toroidal parallel machines with additional links. The new "multi-toroidal" topology is d designed to allow connecting a new partition as a mesh or a torus without modifying any existing partition, as long as the new partition can fit into the free space of the multicomputer. We have shown that for isolated contiguous rectangular partitions the new topology can improve machine utilization by a large factor of 2 to 4 (depending on the workload) compared with the traditional toroidal interconnect, for different scheduling policies (FCFS or aggressive backfilling). This improvement is due to the ability to co-allocate multiple toroidal partitions in the same line of any dimension of the machine.

The multi-toroidal topology suggests other possible advantages such as non-contiguous allocations of partitions by leveraging the additional links to connect non-adjacent nodes. Non-contiguous allocation will require management (including discovery, selection and allocation) of the communication links as dedicated resources, since one will be generally able to connect a partition using one of multiple valid link sets. Another advantage is the degree of redundancy offered by the new topology that leads to increased fault tolerance. These are some of the topics of our ongoing research.



(a) CTC

(b) SDSC

Fig. 13. System utilization of a simulated multi-toroidal machine versus a simulated 3-dimensional torus for different mixtures of torus and mesh requests — scheduling of "fat" jobs with aggressive backfilling: The graphs for the multi-toroidal machine with toroidal jobs only and toroidal machine with mesh jobs only completely overlaps.

6 Acknowledgments

We would like to thank Yoav Gal and Eitan Frachtenberg for fruitful discussions on the ideas in this paper.

References

1. Parallel workload archive. http://www.cs.huji.ac.il/labs/parallel/workload.
2. Cray Research, Inc. Cray T3D System Architecture Overview, Technical Report, Sept. 1993.
3. D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling, Job Scheduling Strategies for Parallel Processing workshop, Lecture Notes in Computer Science, v. 1291, pp. 238-261. Springer-Verlag, 1997.
4. R. Kessler and J. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research, COMPCON, pp. 176-182. 1993.
5. Earth Simulator : http://www.es.jamstec.go.jp
6. E. Krevat, J. G. Castanos, and J. E. Moreira. Job Scheduling for the BlueGene/L System, Job Scheduling Strategies for Parallel Processing workshop, Lecture Notes in Computer Science v. 2537, pp. 38-54, Springer-Verlag, 2002.
7. A. Mualem Weil and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling, IEEE Trans. Parallel & Distributed Syst., 12(6), 2001.
8. D. Lifka. The ANL/IBM SP Scheduling System, Job Scheduling Strategies for Parallel Processing workshop, Lecture Notes in Computer Science, v. 949, pp. 295-303, Springer-Verlag, 1995.
9. D. Das Sharma and D. K.Pradhan. A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers, IEEE Symposium on Parallel and Distributed Processing, pp 682-689, 1993.
10. P. J.Chuang and N. F. Tzeng. An Efficient Submesh Allocation Strategy for Mesh Connected Systems, International Conference. on Distributed Computing Systems., pp. 256-263, 1991.
11. J. Ding and L. N. Bhuyan. An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems, International Conference on Parallel Processing, pp. 193-200, 1993.
12. W. Qiao and L. M. Ni., Efficient Processor Allocation for 3D Tori., Technical Report, Michigan State University, East Lansing, MI, 48824-1027, 1994.
13. S.M. Yoo, H. Choo, and H.Y.Youn. Processor Scheduling and Allocation for 3D Torus Multicomputer Systems, IEEE Trans. on Parallel and Dist. Syst., pp. 475-484, 2000.
14. S. Yoo and C. R. Das. Processor Management Techniques for Mesh-Connected Multiprocessors, International Conference on Parallel Processing, pp. 105-112, 1995.
15. Y. Zhu., Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers, Journal of Parallel and Distributed Computing, v. 16, pp. 328-337, 1992.
16. N. R. Adiga et al. An Overview of the BlueGene/L Supercomputer, Supercomputing 2002.

Costs and Benefits of Load Sharing in the Computational Grid

Darin England and Jon B. Weissman

Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{england,jon}@cs.umn.edu

Abstract. We present an analysis of the costs and benefits of load sharing of parallel jobs in the computational grid. We begin with a workload generation model that captures the essential properties of parallel jobs and use it as input to a grid simulation model. Our experiments are performed for both homogeneous and heterogeneous grids. We measured average job slowdown with respect to both local and remote jobs and we show that, with some reasonable assumptions concerning the migration policy, load sharing proves to be beneficial when the grid is homogeneous, and that load sharing can adversely affect job slowdown for lightly-loaded machines in a heterogeneous grid. With respect to the number of sites in a grid, we find that the benefits obtained by load sharing do not scale well. Small to modest-size grids can employ load sharing as effectively as large-scale grids. We also present and evaluate an effective scheduling heuristic for migrating a job within the grid.

1 Introduction

An emerging trend in high-performance computing is to build interconnected networks of super-computing centers known as computational grids. Individually, these centers house computing resources and instruments needed for large-scale collaborative applications. As these applications place increasing demands on existing resources, increased efficiency in scheduling jobs onto the grid is becoming more important. Already proven in the LAN environment, load sharing is becoming feasible in WANs and grids. The emergence of test-beds like TeraGrid[15] promises remarkable network bandwidth between distant sites, enabling load sharing with minimal network penalties.

In this work we investigated the costs and benefits of load sharing of parallel jobs in a simulated computational grid. First we present a detailed model of a supercomputer workload. The nature of our workload model makes it easy to use as input to the grid simulation experiments. We performed experiments for both homogeneous and heterogeneous grids. The results indicate that load sharing among sites is indeed worthwhile. We find that in a homogeneous grid any amount of load sharing results in decreased wait times for users. In a heterogeneous grid with differing workloads and machine capacities, we find that the processing of remote jobs on a (previously) lightly-loaded machine can cause

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 160–175, 2005.

delay to local jobs. We also investigate how well the benefits of load sharing scale as the number of sites in a grid increases. The benefits are limited in that most of the opportunities for load sharing are exploited in small and medium-sized grids. Finally we present a simple heuristic for determining the target machine of a migrated job. This heuristic, which we call *Weighted Queue*, is easy to compute and does not require estimates of job run time. The paper is organized as follows: In Section 2 we discuss related research. We present our workload model in Section 3. Section 4 represents the bulk of the paper. It includes a description of the simulation model, the homogeneous and heterogeneous grid results, the scaling results, and the evaluation of the scheduling heuristic. Section 5 concludes the work.

2 Related Work

The quality of the input data is paramount to any simulation model. Cirne and Berman [1] developed a comprehensive model of workloads for space-shared parallel supercomputers. They modeled the variation in job arrival rates throughout the work day and they examined the differences between estimated and actual job run times. Our workload model differs from theirs in that we provide an alternative method for generating the job arrivals and for modeling the job run times and job run lengths. We describe our workload model in the next section. In considering the importance of workload traces in simulation experiments, Lo et. al. [10] investigated the effects on job scheduling algorithms due to the use of real workload traces vs. synthetic workload models. They found that the use of either real or synthetic workloads did not affect the overall performance of job scheduling algorithms. However, we note that the use of a real workload trace necessarily limits the simulation to a single run. By using a workload model and its generated job traces, a large number of simulation runs may be conducted, thereby producing enough data to make statistically significant comparisons among alternative scenarios. Lo et. al. did find that other workload characteristics such as the proportion of *power-of-two* job sizes and the correlation between job size and job run time did affect scheduler performance. In the next section we discuss these two characteristics as they relate to our experiments.

Hollingsworth and Maneewongvatana [7] propose a novel approach to scheduling parallel jobs in a computational grid. They present the idea of an imprecise calendar where jobs are scheduled into time slots by a hierarchical system of manager nodes. Time slots that are further into the future are scheduled at a coarse level. As the time for a slot nears, it is scheduled at a finer level. Like the imprecise calendar approach, we wish to efficiently distribute parallel jobs in a grid. However, we employed simple scheduling methods that do not require job information such as run length[1]. Eager et. al. [2] examined the relative benefits of

[1] That is, no estimate of run length is required for job migration to another machine in the grid. However, we employ backfilling at the local machine level which requires run time estimates.

simple vs. complex load sharing policies. Using an analytical model for a homogeneous network, they concluded that simple policies that require only a small amount of state information perform as well as complex policies. We also examined simple policies and we extended their work by comparing relative amounts of load sharing in both homogeneous and in heterogeneous networks. More recently Subramani et. al. [14] used simulation to evaluate distributed scheduling algorithms in a grid environment. They use a scheme in which jobs are placed in queues at multiple sites. The system tracks which copy of the job is first to begin execution and all other copies are cancelled. Our work differs from theirs in that we employ a scheduling algorithm that is easy to implement and we have a more complete workload model with which we are able to make multiple simulation runs and hence reduce the variance in our performance measures.

3 Modeling the Workload

Our workload generation model takes an actual job trace as input. It produces a synthetic workload in standard workload format[12] that captures the following job characteristics:

1. Job inter-arrival times
2. Job sizes
3. Job run times

In this section we discuss how we model each of these characteristics. Our model was created from careful examination of the traces shown in Table 1. Since the SDSC (San Diego Supercomputing Center) and the CTC (Cornell Theory Center) traces are more recent and come from a machine that is more widely used, these two traces were used as the basis for our simulation experiments.

Table 1. Actual Workloads Examined

Center	Machine	Nodes	Time Period
LANL	CM5	1024	Oct 1994 to Sep 1996
SDSC	IBM SP2	128	Apr 1998 to Apr 2000
CTC	IBM SP2	512	Jun 1996 to May 1997

In production systems it is likely that some jobs will be unable to begin execution until other jobs have finished due to precedence constraints. The standard workload format allows for the specification of precedence constraints; however, none of the workloads that we examined contained this information. For this reason and for simplicity, we assume that all jobs arrive to the system independently. Figure 1 shows the average arrival rates of jobs to the IBM SP2 supercomputer at the San Diego Supercomputing Center. From the figure it is clear that more jobs arrive to the system during the working hours than during the night. This phenomenon, which was also observed by Cirne and Berman, occurs in all of the workload traces that we examined. We capture the variation in job arrival rates by using a Non-stationary Poisson Process to model the job arrivals. In a Poisson Process the inter-arrival times (the times between job arrivals) follow an Exponential probability distribution. Thus in our model the job inter-arrival times are generated from six Exponential distributions, one for each period of the day as shown in Fig. 1. Modeling the job arrivals in this manner helps to produce a realistic workload which is more intense during the middle of the day.

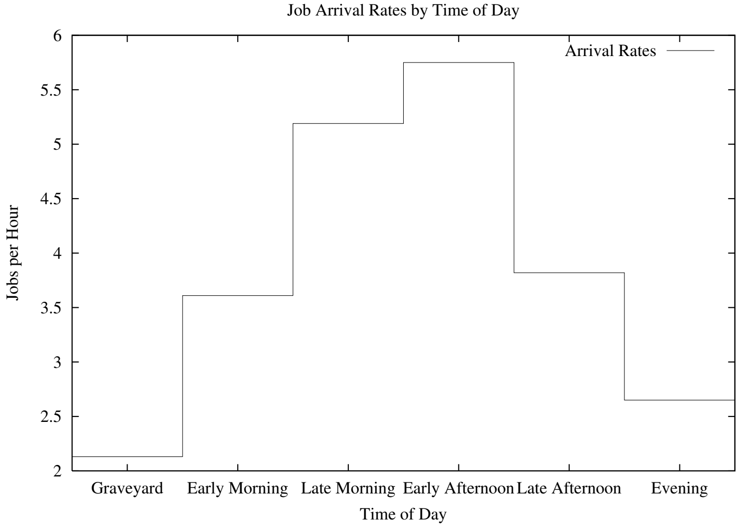


Fig. 1. SDSC Job Arrival Rates by Time of Day

For our workload model the size of a job is characterized by the number of CPUs it requests. The workload traces that we examined are dominated by *power-of-two* sizes, i.e. 2, 4, 8, 16, 32, and 64. All other job sizes occur infrequently. Cirne and Berman [1] model the size of a job with a uniform-log distribution. In order to capture the prevalence of the power-of-two job sizes, they added a direct probability for turning a job size into its nearest power-of-two neighbor. In contrast, we chose to use a discrete probability distribution that reflects the frequency with which the job sizes appear in an actual workload. The discrete probabilities are computed directly from the ratios in the real workload. Figure 2 shows the job size probabilities for the SDSC data up to 64 CPUs.

We made several attempts at modeling job run times as a function of job size. However, we found no correlation between these two characteristics. At each center we assume an independent work model in which there is no correlation between job size and job run time. After experimenting with several probability distributions we found that the job run times fit the Weibull distribution quite well. The quality of the fit is not surprising since Weibull random variables are

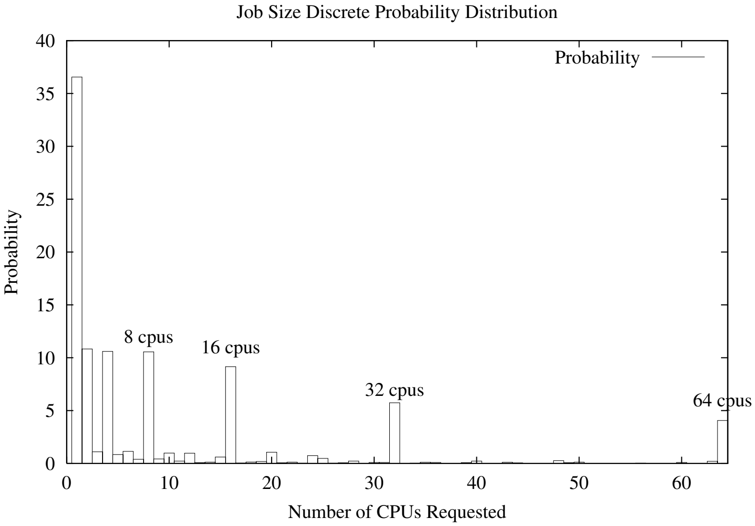


Fig. 2. Discrete Probability Distribution for SDSC Job Sizes

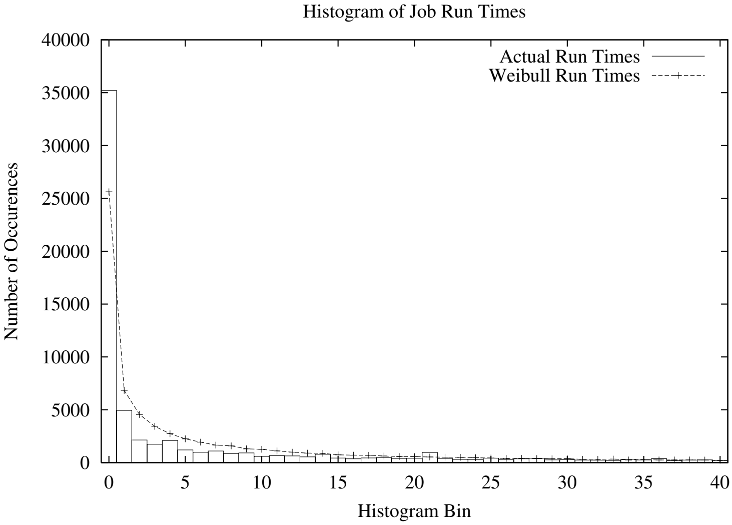


Fig. 3. SDSC Actual Job Run Times vs. Estimated Weibull Distribution

commonly used to model task completion times. Figure 3 shows the actual job run times from the SDSC workload plotted against run times from a Weibull distribution. The plot is a histogram with a bin size of 500 seconds. It is clear from the figure that most jobs run for a short period of time, while a few jobs run for very long periods of time. Again, the size of the job is not a good predictor of job run time. We only included jobs that ran to completion in order to avoid jobs that were killed or that died.

4 Simulation

4.1 Model Description

Our simulation model of a computational grid consists of four supercomputer centers. Traditionally, each center would operate in an autonomous fashion with no job migration to other sites. The model shown in Fig. 4 illustrates the idea of cooperation among the centers by allowing some jobs to be migrated to remote sites. Each center has a local workload that is representative of the actual workload for its machine type. Some percentage of the job arrivals are flagged as migratable. We envision this occurring as users indicating via a job submission script that they are willing to allow a particular job to be migrated. Of course not all jobs are migratable due to various reasons: locality of data, parallel architecture-specific code, security concerns, etc. Therefore, our experiments were conducted with varying percentages of the workload being migratable. The choice of which jobs are flagged as such is completely random. In this way we are certain to simulate the migration of both large jobs and small jobs.

We say that a *local* job is a job that executes on the machine at which the job originally arrived. A *remote* job is one that has been migrated and executes on a remote machine. We make this distinction because a job that is flagged as migratable might actually execute on its local machine if it appears more favorable. There are two requirements for a job to be transferred to a remote machine:

1. The originating machine's job queue must be nonempty.
2. There must be a remote machine with a more favorable queue status.

In other words, if a machine is currently lightly loaded, i.e. its queue is empty, then it will not attempt to migrate an arriving job (even though the job may be flagged as migratable.) In addition, when a machine's queue is nonempty and it attempts to migrate a job that just arrived, then it will transfer the job to the machine whose queue size is smallest. Thus we employ the Shortest Queue scheduling policy for the experiments in this section. Later, we will introduce a new scheduling policy called Weighted Queue. These requirements are common sense attempts to create a reasonable migration policy. This means that before a machine attempts to migrate a job, it must poll the other machines in the grid to obtain their load information.

The cost of job migration includes estimates for network bandwidth and the amount of data to be transferred. As part of their work in predicting data transfer

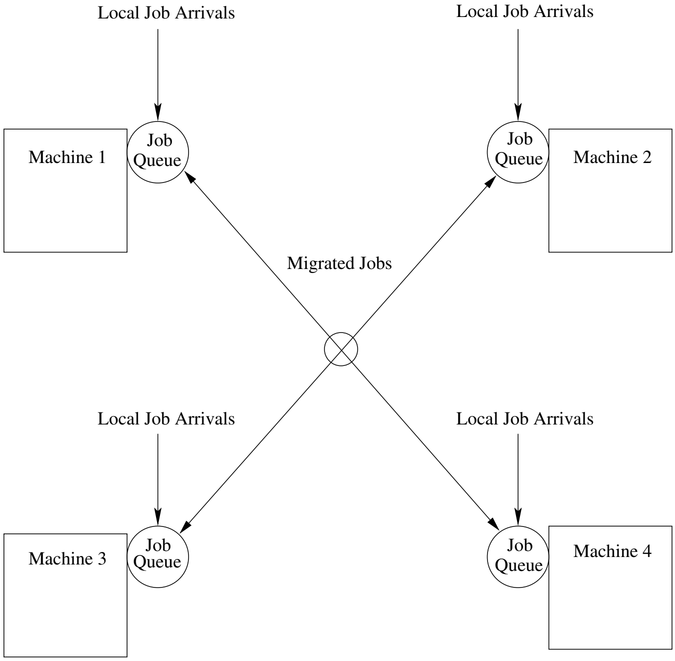


Fig. 4. Supercomputing Grid Simulation Model

costs, Vazhkudai et. al. [17] measured the end-to-end bandwidth between two remote super-computing centers. Their measurements were made using GridFTP, the file transfer service of the Globus Toolkit[5]. They found the network bandwidth to vary from 1.5 to 10.2 MB/sec (megabits). For our experiments we use a constant network bandwidth of 5 MB/sec. Based on the work of Vazhkudai et. al., this represents an achievable bandwidth for current systems. In the future, advances in network infrastructure will help to reduce the cost of job migration. For example, the TeraGrid project [15] will have the ability to transfer data at the rate of 40GB/sec. The actual workload traces that we examined did not contain information about data sizes. In the absence of this information, we used a Triangular distribution as an approximation. The range of the distribution is from 1MB to 1GB, with a mode of 100MB (megabytes)[2].

For scheduling jobs at each local machine we employ backfilling, a technique by which a job is allowed to move ahead of other jobs in the queue and begin execution as long it does not cause the first job in the queue to be delayed. The version of backfilling that we use is known as aggressive backfilling. It is employed in the EASY scheduler on the IBM SP2. Our implementation is exactly the one described in Mu'alem and Feitelson [11]. For an excellent description of

[2] Data sizes were estimated based on a survey by Cirne.

backfilling and its sensitivity to user run time estimates, we refer the reader to their work.

4.2 Experimental Design

Table 2. Network Configurations for Simulation Runs

	Homogeneous Network	Heterogeneous Network
Machine 1	SDSC1	SDSC1
Machine 2	SDSC2	SDSC2
Machine 3	SDSC3	CTC1
Machine 4	SDSC4	CTC2

Table 2 shows the homogeneous and the heterogeneous grid makeup for our simulation experiments. We made 20 independent replications of the simulation for each type of network and for each level of load sharing, 0, 25, 50, 75, and 100%. A level of load sharing indicates the percentage of jobs that are able to be migrated. Each replication of an experiment was performed with a different (but statistically similar) workload that was generated in accordance with our workload model. Our performance measure of interest is job slowdown, which is defined as follows.

$$\text{Slowdown} = \begin{cases} \frac{\text{Queue Time}+\text{Run Time}}{\text{Run Time}} & \text{for a local job,} \\ \frac{\text{Migration Time}+\text{Queue Time}+\text{Run Time}}{\text{Run Time}} & \text{for a remote job.} \end{cases}$$

Job slowdown captures the notion that users are more willing to accept long queue times for long-running jobs than for short-running jobs. For each measurement shown in the Results section we present an average of the 20 replications for an experiment. In order to be certain that performance differences among the different levels of load sharing are not due to randomness in the synthetic workloads, we used the same sets of synthetic workloads as input to each experiment.

4.3 Results

Homogeneous Grid Simulation. In the homogeneous grid simulation all machines have statistically identical workloads. Therefore, all machines get roughly the same intensity of workload regardless of the amount of load sharing performed. We present the average job slowdown for each level of load sharing in Fig. 5. Since the results for all machines in the homogeneous network are similar, only the results for one SDSC-type machine are presented. Each level of load sharing corresponds to an experiment and the average job slowdown is presented. The results are broken out by *local* and *remote* jobs. From the figure we see that

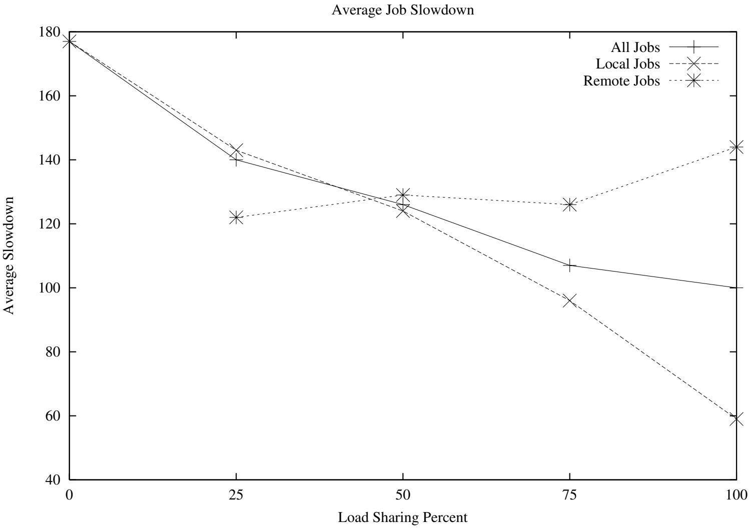


Fig. 5. Machine SDSC1 Homogeneous Grid Simulation

as the amount of load sharing is increased, the average slowdown for local jobs decreases. This is because as more jobs are allowed to be migrated, there is more opportunity to exploit the benefits of load sharing, i.e. machines are able to off-load more work to less heavily-loaded machines. Also, by the nature of our migration policy, a machine will not attempt to migrate a job if its own job queue is empty. Therefore, local jobs arriving to an empty queue (which is common when backfilling is employed) are guaranteed to execute on the lightly-loaded local machine. At the 25% load sharing level remote jobs have shorter average slowdown than local jobs. This is because migrated jobs get sent to machines with more favorable queue statuses. At 25% load sharing, the majority of jobs (75%) are not allowed to be migrated and so they must execute locally, regardless of the load on the local machine. Compared to the slowdown for local jobs, the slowdown for remote jobs remains relatively unchanged as the amount of load sharing is increased, although there is a slight increase at the 100% level. We note that this increase is possible because the Shortest Queue scheduling policy is not optimal. Although not presented, we also collected average and median job queue times and average queue sizes for each experiment. These statistics exhibit the same general trends as job slowdown. We conclude that for a homogeneous grid even a small amount of load sharing produces benefits. In addition, by the use of a reasonable migration policy, local jobs can greatly benefit from large amounts of load sharing, while remote jobs still experience lower slowdown than when there is no load sharing.

A Confidence Interval for Improvement in Average Job Slowdown. Here we statistically compare the improvement in average job slowdown for local jobs when the amount of load sharing is increased from zero to 25%. We present a paired-t confidence interval. Since different sets of workloads were used for each replication of an experiment, our observations of average slowdown are IID (Independent and Identically Distributed.) Let our observations of slowdown be labeled as X_{ij} for $i = 1, 2$ (for no load sharing and for 25% load sharing respectively), and for $j = 1, \ldots, n$ (where n is 20 because there are 20 replications.) Let $Z_j = X_{1j} - X_{2j}$. We construct a 90% confidence interval for $E(Z_j)$, i.e. for the expected value of the difference in average job slowdown. If this confidence interval does not contain zero, then we can state with approximately 90% confidence that a small amount of load sharing (25%) decreases the average job slowdown (assuming the accurateness of our workload and simulation models.) The confidence interval is constructed as follows. We first compute the average and an estimate of the variance of the Z_j's.

$$\bar{Z}(n) = \frac{\sum_{j=1}^{n} Z_j}{n}$$

and

$$\widehat{var}[\bar{Z}(n)] = \frac{\sum_{j=1}^{n} [Z_j - \bar{Z}(n)]^2}{n(n-1)} \quad .$$

The 90% confidence interval is

$$\bar{Z}(n) \pm t_{n-1,0.95} \sqrt{\widehat{var}[\bar{Z}(n)]} \quad .$$

We computed $\bar{Z}(20) = 34.2$ and $\widehat{var}[\bar{Z}(20)] = 169.8$, which leads to a 90% confidence interval of $[11.7, 56.7]$. Therefore, we can state (with approximately 90% confidence) that under our workload and simulation assumptions, allowing 25% load sharing results in a decrease in slowdown for local jobs of between 11.7 and 56.7.

Heterogeneous Grid Simulation. Grids consist of machines that have different capacities, speeds, and workload characteristics. Our simulation of a heterogeneous grid captures those differences in capacities and workload characteristics. We did make the simplification that remote jobs, although generated from different distributions for different machine types, will execute at the same speed on any machine in the network, given the same number of processors. This is a reasonable assumption for our simulations since all of the workloads in the model are based on job traces from IBM SP2 supercomputers.

The model for the heterogeneous grid consists of two SDSC machines and two CTC machines. Each CTC machine has 512 processors and each SDSC machine has 128 processors. Although the CTC machines have more computing capacity, their workloads are more intense than those at the SDSC machines. In fact, the CTC machines handle more than twice the number of jobs than SDSC machines; and the average and the median run times for CTC jobs are more than twice those for SDSC jobs[12].

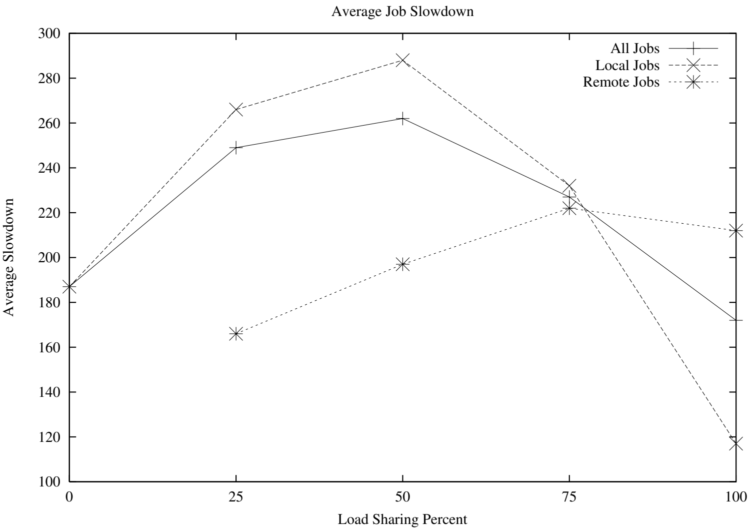


Fig. 6. Machine SDSC1 Heterogeneous Grid Simulation

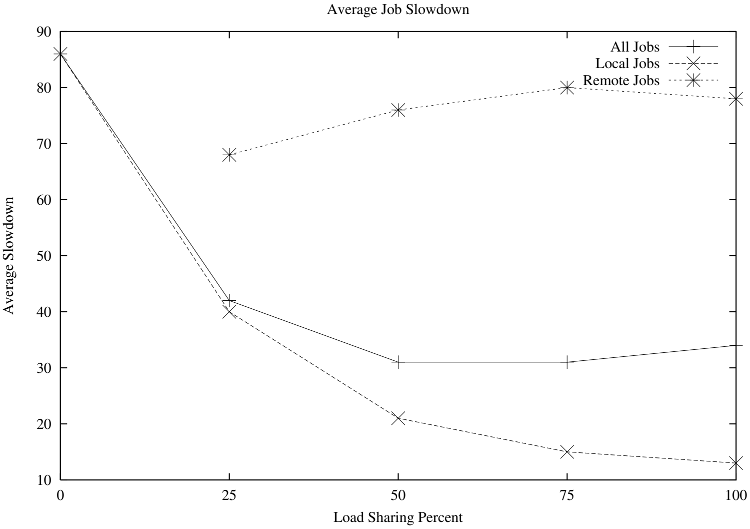


Fig. 7. Machine CTC1 Heterogeneous Grid Simulation

Figures 6 and 7 show the average slowdown for SDSC-type and CTC-type machines respectively. Again, we only present the results for one machine of each type since the results for other two machines are similar. We can see from the ordinate scale in the two figures that the CTC machines have lower job slowdown. The computing capacity of these machines is able to handle their heavy workloads. An interesting result is that the slowdown for SDSC local jobs increases at the 25, 50, and 75% load sharing levels when compared to no load sharing. This is because the remote jobs that get processed by the SDSC machines are in general of a longer duration than the normal local SDSC jobs. Hence, the long-running remote jobs tend to interfere with the processing of local jobs. Not until we have 100% load sharing do the SDSC local jobs actually experience lower average slowdown than under no load sharing (0%.) The slowdown for remote jobs processed at the SDSC machines increases with the amount of load sharing since the queue time increases as more long-running CTC jobs are processed.

It is easy to see that load sharing has greater benefits for users of machines that are more heavily loaded. The slowdown measurements for CTC local jobs become more favorable as the amount of load sharing is increased. The slowdown for remote jobs processed at the CTC machines remains relatively unchanged, although there is a slight increase with the increase in the amount of load sharing. We conclude that load sharing in a heterogeneous grid can adversely affect local jobs on (previously) lightly-loaded machines. Machines that were previously heavily-loaded receive the most benefit. In this type of environment, our results indicate that as much load sharing as possible should be permitted so that the workload can be evenly distributed.

Scaling the Number of Sites. Large-scale projects that include the administration of a computational grid may need to consider expansion of the grid to new sites. An example is the addition of the Pittsburgh Supercomputer Center (PSC) to the TeraGrid project in October 2002. If load sharing is employed, then the effect of the new site will be an important consideration. In this section we test the performance of load sharing with respect to the number of sites in a grid. In addition to the runs with 4 sites as described in the previous sections, we made runs with 2, 6, 8, and 10 sites for both homogeneous and heterogeneous grids. The homogeneous grid consists entirely of SDSC-type machines. For the heterogeneous grid we split the number of machines evenly between SDSC-type machines and CTC-type machines. For example, in the run with with 10 total machines, the grid consists of 5 SDSC machines and 5 CTC machines. All runs for this experiment were performed at the 50% load sharing level. We present the average job queue times in Fig. 8. In this figure the average queue times for the heterogeneous networks are higher due to the heavy workloads at the CTC machines. The results for both types of grids are presented in the same figure in order to save space. We are not implying that all homogeneous grids perform better than heterogeneous grids. The figure shows that the average job queue time decreases as the number of sites increases; however, the improvements come

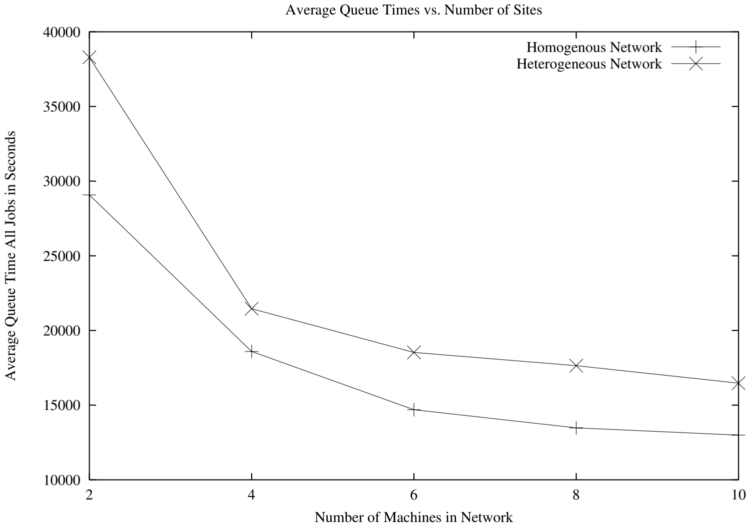


Fig. 8. Scaling the Number of Sites in a Grid

at a decreasing rate. In moving from a small number of sites (2 or 4) to a larger number of sites, the benefits of load sharing are readily apparent. As the number of sites increases, the benefits of load sharing still exist, but there seems to be a saturation point where all of the opportunities for load sharing have been exploited. This suggests that small to modest-sized grids can be as effective as large-scale grids with respect to load sharing.

A Proposed Scheduling Heuristic. In this section we present a new heuristic for choosing the target machine for job migration. In the absence of detailed job information, or when low scheduling overhead is desired, one simple measure is the number of jobs in the remote machine's job queue. By itself this criterion does not always yield the best migration decisions because it does not take into account the job runtime. Nevertheless, schedulers only have estimates of job run time a priori to job execution and these estimates are notoriously inaccurate [9]. Also, backfilling has a significant effect on a machine's queue size. A simple and natural extension to using shortest queue size is to compute the ratio of the total number of CPUs being requested by jobs currently in the queue to the number of CPUs in the machine. We call this criterion *Weighted Queue*. It measures the percentage of a machine's capacity that has already been requested, which could be greater than 100%. The appeal of this heuristic is that it is easy to compute and it does not require estimates of job run time. In a homogeneous grid the Weighted Queue heuristic performs exactly the same as Shortest Queue because all machines have the same workload characteristics and the same capacity. However, in a heterogeneous grid this heuristic can exploit the differences in workloads and machine capacities. We compare the performance of Weighted

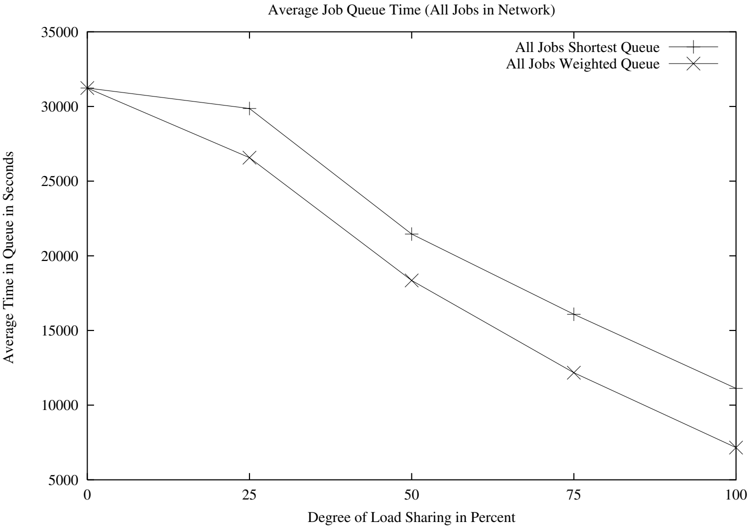


Fig. 9. Weighted Queue vs. Shortest Queue

Queue vs. Shortest Queue in a simulation experiment for a heterogeneous grid of two SDSC-type machines and two CTC-type machines. The average job queue times are presented in Fig. 9. In this figure we are directly comparing the two measures. For both heuristics the average queue time decreases as the amount of load sharing increases. Depending on the level of load sharing, the reductions in queue times range between 4% and 64% for Shortest Queue, and between 15% and 77% for Weighted Queue. Thus Weighted Queue performs better in the heterogeneous environment.

5 Conclusions

In this work we investigated the benefits of load sharing of parallel jobs among supercomputer centers in a computational grid. By closely examining actual job traces, we were able to create a model that generates accurate synthetic workloads. Using these workloads as input, we employed a discrete-event simulation model to explore the effects of load sharing in both homogeneous and heterogeneous grids. For homogeneous grids our results demonstrate that cooperation among sites in the form of load sharing leads to overall reduced job slowdown. By the use of a migration policy that only allows migration from a nonempty queue to a queue that is more favorable, local jobs receive the most benefit from load sharing. For heterogeneous grids, where there are large differences in workload characteristics among the sites, a small amount of load sharing results in increased job slowdown for local jobs on lightly-loaded machines. Local jobs in the heavily-loaded machines receive the most benefit. In this case the migration policy should be carefully considered and simulation is one tool that can help in this evaluation. We also see that the benefits of load sharing do not scale particularly well. There is a point of diminishing returns as the number of sites in a grid increases. Thus we conclude that modest-sized grids can provide as much benefit with respect to load sharing as large-scale grids. Finally, we presented a simple heuristic for selecting the target machine of migrated job. The Weighted Queue measure, which considers the number of CPUs being requested relative to a machine's capacity, is effective, easy to compute, and does not require estimates of job run time.

Acknowledgment

The authors would like to acknowledge support from the Department of Energy through contract DE-FG02-03ER25554 and the National Science Foundation 0305641.

References

1. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: 4th Workshop on Workload Characterization. (2001)
2. Eager, D.L., Lazowska, E.D., Zahorjan, J.: Adaptive load sharing in homogenous distributed systems. IEEE Transactions on Software Engineering **SE-12** (1986)
3. Feitelson, D.G.: Packing schemes for gang scheduling. In Feitelson, D.G., Rudolph, L., eds.: Job Scheduling Strategies for Parallel Processing. Volume 1162. Springer Verlag (1996) 89–110 Lecture Notes in Computer Science.
4. Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (1998)
5. The Globus Alliance: Project Website. `http://www.globus.org` (2003)
6. Gross, D.M., Harris, C.M.: Fundamentals of Queueing Theory. Second edn. John Wiley and Sons (1985)
7. Hollingsworth, J.K., Maneewongvatana, S.: Imprecise calendars: an approach to scheduling computational grids. In: 19th IEEE International Conference on Distributed Computing Systems. (1999)
8. Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis. Second edn. McGraw Hill (1991)
9. Lee, C.B., et al.: Are user runtime estimates inherently inaccurate? In Feitelson, D.G., Rudolph, L., eds.: Job Scheduling Strategies for Parallel Processing. Springer Verlag (2004) Lecture Notes in Computer Science.
10. Lo, V., Mache, J., Windisch, K.: A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In Feitelson, D.G., Rudolph, L., eds.: Job Scheduling Strategies for Parallel Processing. Volume 1459. Springer Verlag (1998) 25–46 Lecture Notes in Computer Science.
11. Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user run time estimates in scheduling the IBM SP2 with backfilling. IEEE Transactions on Parallel and Distributed Systems **12** (2001)
12. Parallel Workload Archive: The Hebrew university of Jerusalem, school of computer science and engineering. `www.cs.huji.ac.il/labs/parallel/workload` (2002)

13. Smith, W., Foster, I., Taylor, V.: Predicting application run times using historical information. In Feitelson, D.G., Rudolph, L., eds.: Job Scheduling Strategies for Parallel Processing. Volume 1459. Springer Verlag (1998) 122–142 Lecture Notes in Computer Science.
14. Subramani, V., et al.: Distributed job scheduling on computational grids using multiple simultaneous requests. In: 11th IEEE International Symposium on High Performance Distributed Computing. (2002)
15. The TeraGrid Project: A distributed computing infrastructure for scientific research. `www.teragrid.org` (2003)
16. Trivedi, K.S.: Probability and Statistics with Reliability, Queueing and Computer Science Applications. Second edn. John Wiley and Sons, Inc. (2002)
17. Vazhkudai, S., et al.: Predicting the performance of wide area data transfers. In: Proceedings of the International Parallel and Distributed Processing Symposium. (2002)

Workload Characteristics of a Multi-cluster Supercomputer

Hui Li[1], David Groep[2], and Lex Wolters[1]

[1] Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
hli@liacs.nl, llexx@liacs.nl

[2] National Institute for Nuclear and High Energy Physics (NIKHEF),
The Netherlands
davidg@nikhef.nl

Abstract. This paper presents a comprehensive characterization of a multi-cluster supercomputer[3] workload using twelve-month scientific research traces. Metrics that we characterize include system utilization, job arrival rate and interarrival time, job cancellation rate, job size (degree of parallelism), job runtime, memory usage, and user/group behavior. Correlations between metrics (job runtime and memory usage, requested and actual runtime, etc) are identified and extensively studied. Differences with previously reported workloads are recognized and statistical distributions are fitted for generating synthetic workloads with the same characteristics. This study provides a realistic basis for experiments in resource management and evaluations of different scheduling strategies in a multi-cluster research environment.

1 Introduction

Workload characterization of parallel supercomputers is important to understand the system performance and develop workload models for evaluating different system designs and scheduling strategies [1,2]. During the past several years, lots of workload data has been collected [3], analyzed [4,5,6], and modeled [7,8,9]. Benchmarks and standards are also proposed for job scheduling on parallel computers [10].

In previously studied workloads [4,5,6,7], some characteristics are similar. For example, most of the workloads are collected from large custom-made production facilities (IBM SP2, SGI Origin, etc) in supercomputing centers. Jobs typically request "power-of-two" number of processors and have different arrival patterns in different periods (e.g. peak and none-peak hours in a daily cycle). Some characteristics, such as job attribute distributions and correlations, vary across different workloads [4,5,11]. Other characteristics are studied and reported separately, such as job cancellation rate [9] and conditional distributions (e.g. actual runtime distributions conditioned on requested runtime [4]). In this paper we compare our workload with previous reported ones on a per characteristics basis.

[3] Distributed ASCI Supercomputer-2 (DAS-2). ASCI stands for Advanced School for Computing and Imaging in the Netherlands.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 176–193, 2005.

This paper presents a comprehensive workload characterization of the DAS-2 [12] supercomputer. The DAS-2 system is interesting in that it is built using the popular COTS (Commodity Off The Shelf) components (e.g. Intel Pentium processors and Ethernet networks) and consists of multiple distributed clusters serving the participating universities. Not like other production machines, DAS-2 is dedicated to parallel and distributed computing research thus has much lower system utilization. We analyze twelve-month workloads on DAS-2 clusters in year 2003. Characteristics include system utilization, job arrival rate and interarrival time, job cancellation rate, job size (degree of parallelism), job runtime, memory usage, and user/group behavior. Correlations between metrics are also identified and studied.

The contributions of this paper reside in the following. Firstly, our study is based on cluster workloads. Cluster computing is a popular alternative in the HPC community and to our knowledge, not much work has been done in characterizing cluster workloads. Secondly, the system we study is a research facility. This provides an interesting comparison point to the well studied production workloads. Thirdly, we present a comprehensive characterization of the DAS-2 workloads. We not only analyze most of the metrics appeared in previous work, but also extensively study the correlations between different characteristics. Moreover, we fit the observed data with statistical distributions to facilitate synthetic workload generation. This research serves as a realistic basis in modeling cluster workloads, which contributes as input for evaluations of different scheduling strategies in a multi-cluster research environment [13].

The rest of the paper is organized as follows. Section 2 provides an overview of the DAS-2 system and workload traces used in our study. Section 3 analyzes the overall system utilization. Section 4 describes the job arrival characteristics, including job arrival rate, job interarrival time and job cancellation rate. Distributions are fitted for job interarrival times and job cancellation lags. Section 5 describes job execution characteristics. This includes job size, job actual runtime, memory usage, and correlations between them. Distributions and/or conditional distributions are also provided. Section 6 describes user/group behavior and its implications in modeling and predictions. In section 7 conclusions are presented and future work is discussed.

2 The DAS-2 Supercomputer and Workload Traces

The DAS-2 supercomputer consists of five clusters located at five Dutch universities and is primarily used for computing and scientific research. The largest cluster (Vrije Uni-

Cluster	Location	#CPUs	Period	#Job entries
fs0	Vrije Univ. (VU)	144	01-12/2003	219618
fs1	Leiden Univ.	64	01-12/2003	39356
fs2	Univ. of A'dam (UvA)	64	01-12/2003	65382
fs3	Delft Univ. of Tech.	64	01-12/2003	66112
fs4	Utrecht Univ.	64	02-12/2003	32953

Table 1. DAS-2 clusters and workload traces.

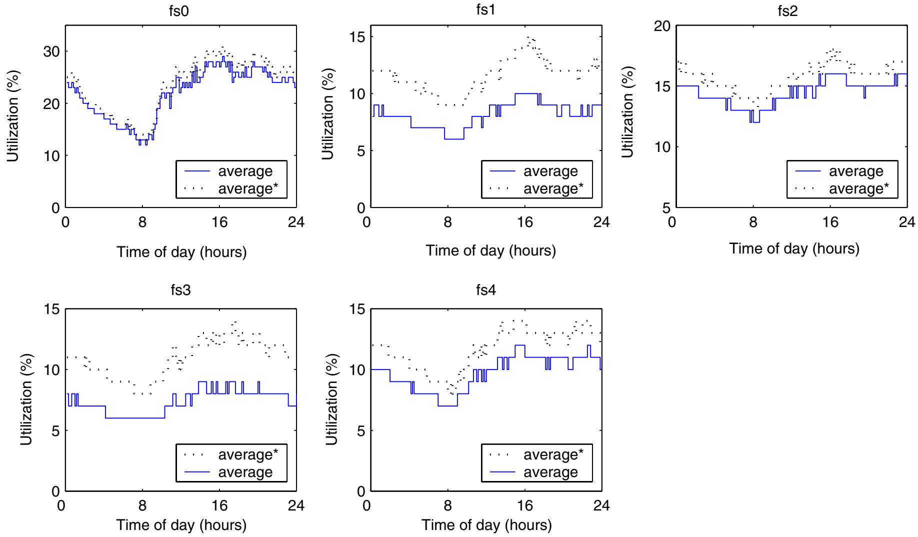


Fig. 1. System utilization of DAS-2 clusters. “Average” stands for the average utilization of all days in the year. “Average*” stands for the average utilization of all active days in the year, excluding system downtime and days without job arrivals.

versiteit) contains 72 nodes and the other four clusters have 32 nodes each. Every node contains two 1GHz Pentium III processors, 1GB RAM and 20GB local storage. The clusters are interconnected by the Dutch university internet backbone and the nodes within a local cluster are connected by high speed Myrinet as well as Fast Ethernet LANs. All clusters use openPBS [14] as local batch system. Maui [15] (FCFS with backfilling) is used as the local scheduler. Jobs that require multi-clusters can be submitted using toolkits such as Globus [16]. DAS-2 runs RedHat Linux as the operating system.

We use job traces recorded in the PBS accounting logs for twelve months in year 2003 on the five clusters[4]. All jobs in the traces are *rigid* (jobs that do not change parallelism at runtime) batch jobs. An overview of the DAS-2 system and workload traces is provided in Table 1. As we can see, fs0 (VU) is the most active cluster, with more than two hundred thousand job entries. Next we have clusters at UvA (fs2) and Delft (fs3), each with more than sixty thousand entries. Leiden (fs1) and Utrecht (fs4) are relatively less active among the DAS-2 clusters. Next section gives a more detailed analysis on the overall system utilization.

3 System Utilization

Figure 1 shows the DAS-2 system utilization as function of time of day. Two plots are shown for each cluster. One is average utilization of all days and the other is average uti-

[4] Logs of January on fs4 are not available.

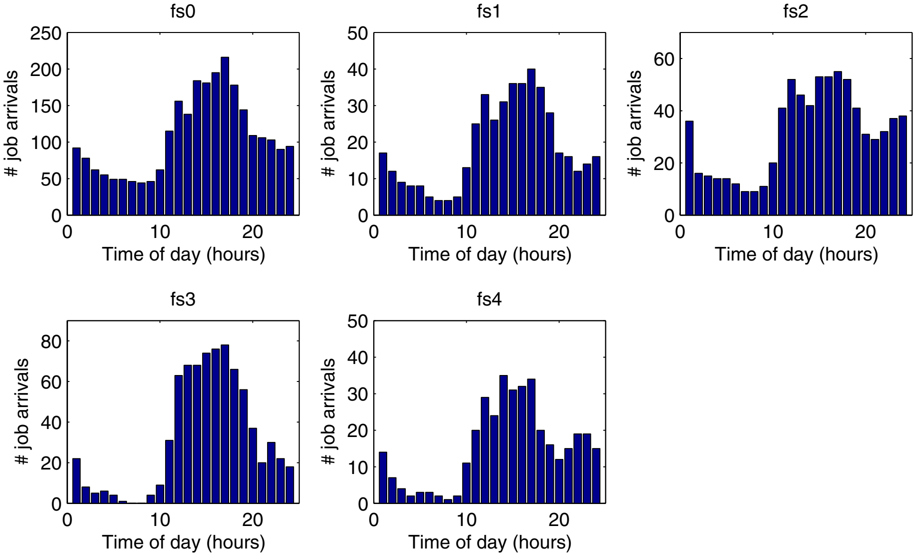


Fig. 2. Daily cycle of job arrivals during weekdays on DAS-2 clusters.

lization of all active days in the year (excluding system down time and days without job arrivals[5]). In average, fs0 has the highest (22%) and fs3 has the lowest system utilization (7.3%) among DAS-2 clusters. The utilization (7.3% to 22%) is substantially lower than previously reported workloads (e.g. 50% in average excluding downtime [5]). This is because DAS-2 system is designed for scientific research and production jobs are precluded from it. The goal of DAS-2 is not on high utilization, but rather on provide fast response time and more available processors for university researchers. Moreover, DAS-2 schedulers define one special policy, which forbids jobs to be scheduled on nodes (SMP dual processor) of which one processor is already used by another job. This policy also has certain negative impact on the overall system utilization.

We can see that the utilization approximately follows the daily job arrival rate (see Figure 2), although the differences between day and night are generally smaller. It is because nightly jobs often require more processors and run longer than daily jobs, despite substantially fewer job arrivals. This is particularly evident on cluster fs3 and fs4.

4 Job Arrival Characteristics

In this section we analyze the job arrival characteristics. We first describe the job arrival rate, focusing mainly on daily cycles. Daily peak and non-peak hours are identified. Secondly, we characterize the job interarrival times during daily peak hours. Several statistical distributions are examined to fit the job interarrival times. Finally, job cancel-

[5] Since we calculate the system utilization based on traces, we could not distinguish whether it is system down time or time without job arrivals.

Cluster	Period	M (s)	CV	Best Fitted distribution	KS
fs0	2003/12/02	17	1.6	gamma (a = 0.44, b = 39)	0.10
fs1	2003/11/25	26	2.4	gamma (a = 0.30, b = 86)	0.13
fs2	2003/12/29	14	1.3	hyperexp2 ($c1$=0.92, $\lambda1$=0.07, $c2$=0.08, $\lambda2$=100)	0.07
fs3	2003/05/26	10	1.8	hyperexp2 ($c1$=0.55, $\lambda1$=0.06, $c2$=0.45, $\lambda2$=0.42)	0.10
fs4	2003/08/13	62	3.0	hyperexp2 ($c1$=0.09, $\lambda1$=0.003, $c2$=0.91, $\lambda2$=0.03)	0.10

Table 2. High load distributions of job interarrival time during daily peak hours (M - Mean, CV - Coefficient of Variation, KS - maximal distance between the cumulative distribution function of the theoretical distribution and the sample's empirical distribution).

Cluster	Period	M (s)	CV	Best Fitted distribution	KS
fs0	Dec	27	4.5	hyperexp2 ($c1$=0.04, $\lambda1$=0.003, $c2$=0.96, $\lambda2$=0.06)	0.15
fs1	Aug, Dec	66	3.6	Weibull (a = 22.6, b = 0.44)	0.10
fs2	Dec	44	5.0	Weibull (a = 26.1, b = 0.58)	0.08
fs3	May, Dec	23	6.0	Weibull (a = 11.6, b = 0.53)	0.14
fs4	Aug, Nov	86	5.1	Weibull (a = 33.2, b = 0.5)	0.09

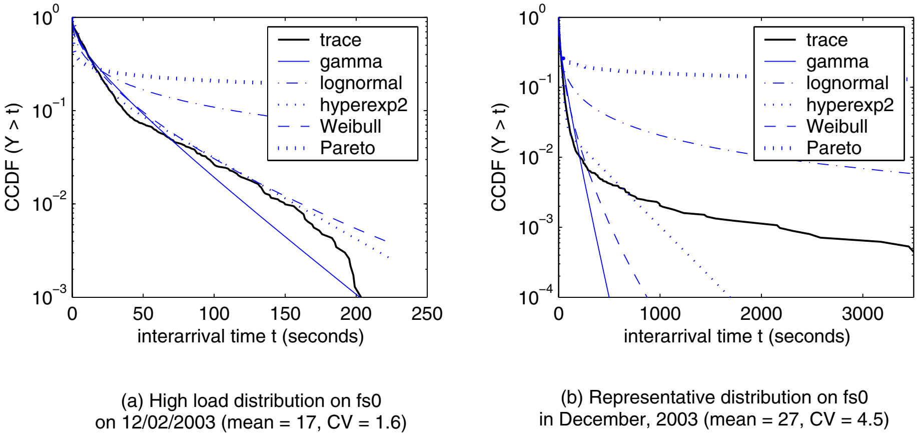
Table 3. Representative distributions of job interarrival time during daily peak hours (M - Mean, CV - Coefficient of Variation, KS - maximal distance between the cumulative distribution function of the theoretical distribution and the sample's empirical distribution).

lation rate and cancellation lags are analyzed and modeled, since it may also affect the scheduling process.

4.1 Job Arrival Rate

As is studied in [7], job arrivals are expected to have cycles at three levels: daily, weekly, and yearly. In a yearly cycle, we find that workloads are not distributed evenly throughout the year. Instead, workloads concentrate on specific months and job entries in these months are around two or more times above average. We call them "job-intensive" months (October, November and December on fs0, August, November on fs1, November, December on fs2, May, December on fs3, and August, November on fs4). This is because of the different active users/groups on different clusters and they are active in specific periods during the year (see Section 6). In a weekly cycle, all clusters share similar characteristics. Wednesday has the highest average job arrival rate and decreases alongside, with Sunday and Saturday have the lowest arrival rate. This is natural since people generally work more during weekdays (Monday - Friday) than weekends (Saturday and Sunday).

The most important cycle is the daily cycle. As is shown in Figure 2, clusters share similar daily workload distributions during weekdays. We identify the daily peak hours as from 9am to 7pm on all five clusters. This is in accordance with normal "working hours" at Dutch universities. Similar job arrival distributions are reported on other



(a) High load distribution on fs0 on 12/02/2003 (mean = 17, CV = 1.6)

(b) Representative distribution on fs0 in December, 2003 (mean = 27, CV = 4.5)

Fig. 3. Fitting distributions of interarrival time during peak hours on fs0.

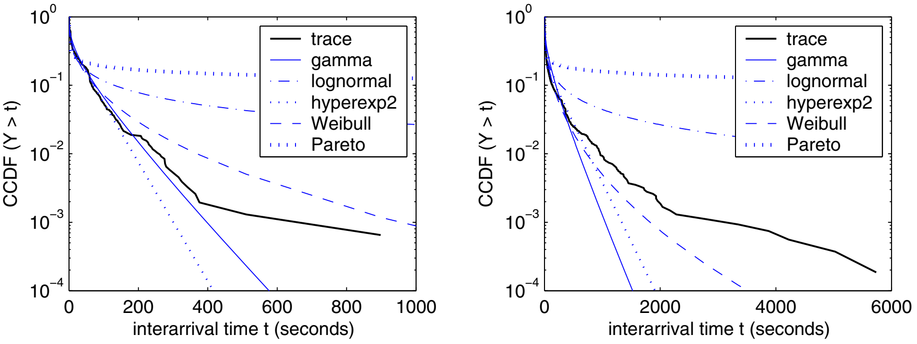
workloads with different peak hour periods (e.g. 8am to 6pm in [4], 8am to 7pm in [7]). Additionally, an intermediate period is reported from 6pm to 11pm in [4]. We observed similar characteristics on DAS-2 clusters, with an intermediate arrival period from 8pm to 1am and a low arrival period from 1am to 8am. The arrival rate per hour can be divided into three scales. The fs0 cluster has the highest one, with an average arrival rate of 108 jobs per hour and peak arrival rate exceeding 200 jobs per hour. In the middle there are fs2 and fs3, with average arrival rates of 31 and 32 jobs per hour each. Clusters fs1 and fs4 have average arrival rates of 19 and 15 jobs per hour, respectively.

4.2 Job Interarrival Time

Based on the observed job interarrival patterns, we choose to characterize “representative” and “high load” period of job interarrival times. The representative period is defined as the peak hours during weekdays in job-intensive months. The high load period is the peak hours of the most heavily loaded days in the year. As is shown in Table 2, during high load period the *mean* ranges from 14 to 62 seconds and the *coefficient of variation* (CV) varies from 1.3 to 3.0 on DAS-2 clusters. The mean and CV are considerably larger in the representative period (see Table 3). Both small (1-2) and large CVs (3-6) have been reported in other workloads [4,6].

We have selected several statistical models to fit the interarrival times of representative and high load period, including hyperexponential, gamma, Weibull, and heavy-tailed distributions like lognormal and Pareto [17]. We fit the above mentioned distributions (except hyperexponential) using *Maximum Likelihood Estimation* (MLE) method, and a two-phase hyperexponential distribution using *Expectation Maximization* (EM) algorithm[6] [18]. The goodness of fit is assessed using the Kolmogorov-Smirnov test.

[6] Matlab [19] and Dataplot [20] are used to calculate means, CVs, do MLE fitting and goodness of fit test. EMpht [21] is used to fit the hyperexponential distribution.



(a) High load distribution on fs1 on 11/25/2003 (mean = 26, CV = 2.4)

(b) Representative distribution on fs1 in Aug, Dec, 2003 (mean = 66, CV = 3.6)

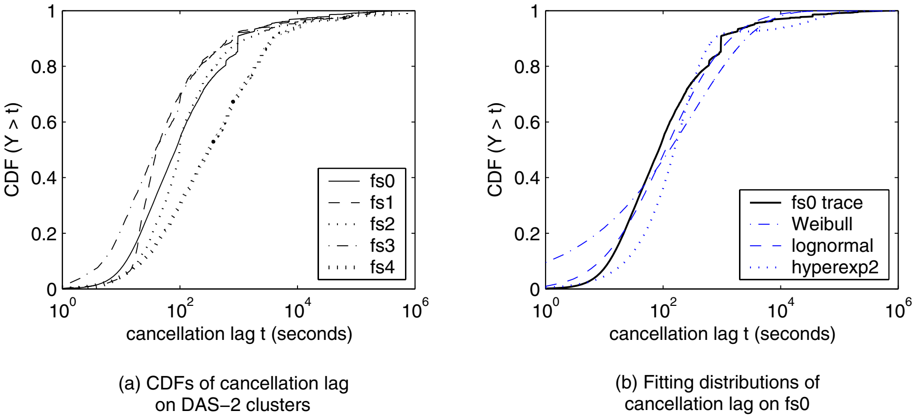
Fig. 4. Fitting distributions of interarrival time during peak hours on fs1.

Results of distribution fitting are shown in Table 2 and 3. Figure 3 and 4 further illustrate how well the different distributions fit the trace data on fs0 and fs1. Generally speaking, none of the chosen distributions pass the goodness of fit test. Some distributions, such as gamma and hyperexponential, fit the head of the sample distribution well but fail to fit the tail. Others like lognormal and Pareto, fit the tail but not the head. It seems not likely to find a model that fits all parts of the empirical distribution well. However, we provide the best fitted distributions for high load and representative period on DAS-2 clusters. For the high load period (see Table 2), gamma and two-phase hyperexponential give the best results among the distributions. One is slightly better than the other depending on the clusters. For the representative period where longer tails and larger CV are observed, Weibull distribution has the best Kolmogorov-Smirnov test results. The only exception occurs on fs0, where a two-phase hyperexponential distribution fits the sample tail better than Weibull. Parameters of fitted distributions are provided in Table 2 and 3.

4.3 Cancelled Jobs

Cancelled jobs may also affect the scheduling process and should be taken into account during workload modeling. In [9], reported job cancellation rates range from 12% to 23% and cancelled jobs are modeled separately. On DAS-2 clusters, as is shown in Table 4, lower cancellation rate are observed. The average percentage of cancelled jobs are 6.8 % (range from 3.3% on fs3 to 10.6% on fs0).

The *cancellation lag* (CL) is defined as the time between job arrival and cancellation. On DAS-2 clusters, the average cancellation lag is 6429 seconds (Table 4). Plots of cancellation lag distributions (CDF) on a log scale are shown in Figure 5 (a). In [9], log-uniform distribution is used to fit the cancellation lag. We examined three distributions (two-phase hyperexponential, lognormal and weibull). Figure 5 (b) illustrates the fitting results on fs0. In general, lognormal provides the best fit for the observed data.



(a) CDFs of cancellation lag on DAS−2 clusters

(b) Fitting distributions of cancellation lag on fs0

Fig. 5. Distributions of cancellation lags on DAS-2 clusters.

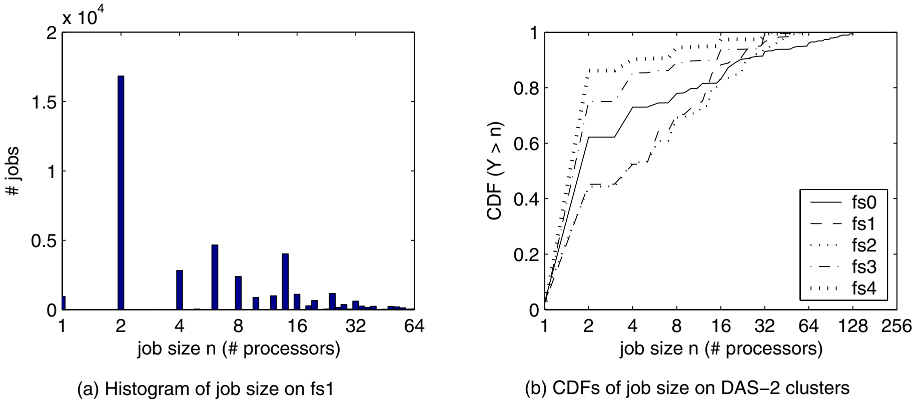
However, only on fs4 it passes the goodness of fit test. Fitted lognormal parameters are provided in Table 4.

cluster	job cancelled (%)	M (s)	CV	lognormal parameters	KS
fs0	10.6	3528	8.7	$\mu = 4.7, \sigma = 2.0$	0.06
fs1	7.7	4749	6.4	$\mu = 4.4, \sigma = 2.0$	0.16
fs2	3.6	13480	6.6	$\mu = 5.0, \sigma = 2.1$	0.14
fs3	3.3	3931	6.5	$\mu = 4.0, \sigma = 2.3$	0.09
fs4	8.6	6458	6.3	$\mu = 5.8, \sigma = 2.1$	0.02
Average	6.8	6429	6.9	$\mu = 4.8, \sigma = 2.1$	0.09

Table 4. Job cancellation rates and cancellation lags (CL) on DAS-2 clusters (M - CL Mean, CV - CL Coefficient of Variation, KS - maximal distance between the cumulative distribution function of the theoretical distribution and the sample's empirical distribution).

5 Job Execution Characteristics

In this section we describe the job execution characteristics. Firstly we characterize job size (number of processors requested), job actual runtime, and memory usage. Secondly the correlations between these metrics are extensively studied and conditional distributions are defined for the job actual runtime.



(a) Histogram of job size on fs1

(b) CDFs of job size on DAS–2 clusters

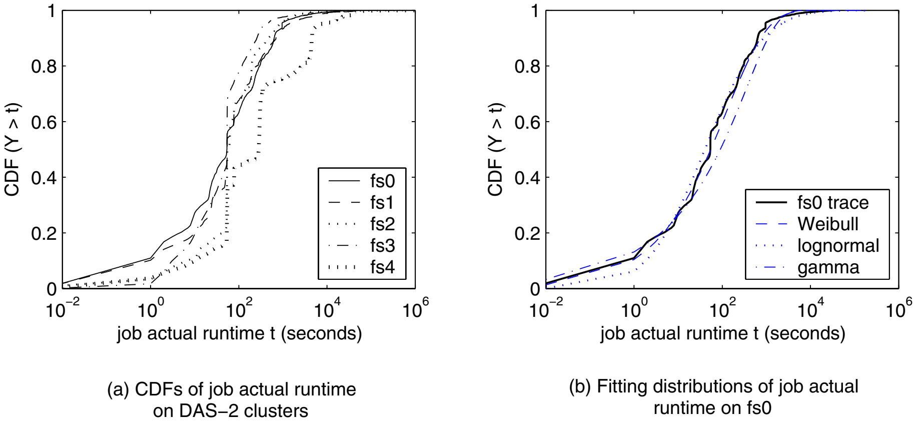
Fig. 6. Distributions of job sizes on DAS-2 clusters.

5.1 Job Size

Table 5 shows the job size characteristics on DAS-2 clusters. The “power-of-two” phenomenon (78.8% in average) is clearly observed, as is found in many other workloads [4,7,9,11]. However, the “power-of-two” sizes on cluster fs0, fs1, and fs2 are not as dominant as on fs3 and fs4. Instead, some multiple-2 sizes also contribute to a significant portion of the total number of jobs (e.g. 6 and 14 processors on fs1, shown in Figure 6 (a)). The fractions of serial (0.9-4.7%) and odd numbers (1% in average) are significantly lower compared to previously reported workloads (30-40%). One possible explanation could be the special policy mentioned in Section 3 , which forbids jobs to be scheduled on nodes (SMP dual processor) with one processor busy. Researchers are not encouraged to submit multi-processor jobs with odd numbers.

As we all noticed in Table 5, job size of *two* processors is surprisingly popular on DAS-2 clusters and it is chosen by a major fraction of jobs (range from 39.6% on fs2 to 85.3% on fs4). To find a proper explanation for this phenomenon, we analyze the internal structure of the workloads. On fs0, for instance, there are ten very active users (out of 130 users in total). The most active user submitted more than 40,000 jobs (18% of the total number of jobs on fs0) in consecutive seven weeks during October and November 2003, which is his/her only active period throughout the year. All of these jobs have the same name and request two processors. For the second most active user on fs0, around 90% of his/her jobs have job sizes of two. On other DAS-2 clusters similar user behavior are observed, resulting in the popularity of job size two and power-of-two. We discuss more on user behavior and its impacts on workload modeling in Section 6.

In [7], the best results for fitting job sizes are obtained by gamma and two-stage uniform distributions. On DAS-2 clusters, we find that two-stage loguniform distribution provides the best fit for job sizes. Plots of the job size distributions on a log scale are shown in Figure 6 (b).



(a) CDFs of job actual runtime on DAS−2 clusters

(b) Fitting distributions of job actual runtime on fs0

Fig. 7. Distributions of job actual runtimes on DAS-2 clusters.

5.2 Job Actual Runtime

Job actual runtime has been extensively studied in previous reported workloads. Table 6 shows the characteristics of job actual runtimes on DAS-2 clusters. The actual runtimes range from 374 to 2427 seconds, which is lower then previously reported workloads (e.g. 3479 seconds on SDSC SP2 [6]). However, the CV (5.3 - 16) is substantially higher than other production systems (2 - 5) [4,5,6]. This is in accordance with the scientific and experimental nature of the DAS-2 usage: the majority of jobs have small execution times and they vary a lot. Plots of the actual runtime distributions on a log scale are shown in Figure 7 (a).

Different kinds of distributions have been used to model the actual runtime, for instance, loguniform in [22], hypergamma in [7] and Weibull in [4]. We evaluate gamma, lognormal and Weibull distributions for actual runtimes on DAS-2 clusters. Figure 7 (b) shows the distribution fitting on fs0. Weibull and lognormal have similar goodness of fit test results, and they both fit better than gamma. Lognormal is a better model for samples that have a lower head and a longer tail (fs2, fs3, and fs4, see Figure 7 (a)). Parameters of fitted distributions are listed in Table 6.

cluster	serial(%)	two(%)	power-of-two(%)	others(%)	odd (except serial) (%)
fs0	2.8	59.4	78.1	19.1	4.2
fs1	2.4	42.8	60.5	37.1	0.2
fs2	4.7	39.6	61.9	33.4	0.4
fs3	1.4	73.6	96.1	2.5	0.03
fs4	0.9	85.3	97.6	1.5	0.05
average	2.4	60.1	78.8	18.7	1.0

Table 5. Job size characteristics on DAS-2 clusters.

cluster	mean (s)	CV	fitted distributions	KS
fs0	374	5.3	Weibull ($a = 121.7$, $b = 0.46$)	0.08
fs1	648	7.9	Weibull ($a = 142.2$, $b = 0.45$)	0.12
fs2	531	16	lognormal ($\mu = 4.2$, $\sigma = 1.8$)	0.22
fs3	466	12	lognormal ($\mu = 3.7$, $\sigma = 1.7$)	0.12
fs4	2427	6.4	lognormal ($\mu = 5.3$, $\sigma = 2.5$)	0.13

Table 6. Job actual runtimes on DAS-2 clusters.

5.3 Memory Usage

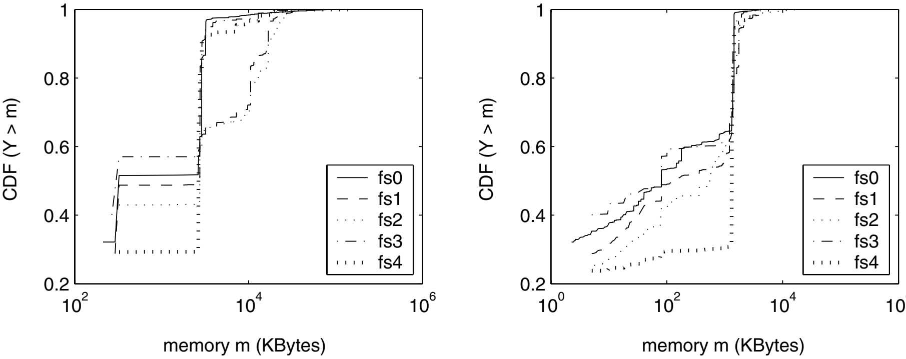
The PBS [14] accounting logs record the maximum amount of physical memory used by the job. Hereafter we refer to memory usage as the maximum used physical memory. Memory usage per processor is defined as the maximum used memory divided by the number of processors requested.

cluster	0KB (%)	324KB (%)	2600-3000KB (%)
fs0	32	19	34
fs1	29	20	16
fs2	25	18	21
fs3	40	17	34
fs4	24	6	62
Average	30	16	33

Table 7. Three special memory usage values and their corresponding job percentages.

Figure 8 (a) shows the distributions of memory usage on DAS-2 clusters. It is clearly observed that three special values are chosen by a major fraction of jobs. These special values are 0KB, 324KB and 2600-3000KB (slightly different values in this range depending on the clusters), and their corresponding job percentages are listed in Table 7. We can see that a large fraction (30% in average) of jobs have very small memory usage[7]. 324KB and 2600-3000KB, on the other hand, contributes nearly one-sixth and one-third (in average) to the total number of jobs, respectively. The reason why memory usage concentrates on these special values might be that jobs typically have to load certain shared libraries (e.g. C, MPI, Globus), and these shared libraries normally require a fixed amount of memory. To verify this claim, we run MPI jobs (fractal computation) with different requested number of processors (4, 8, 16 and 32) on DAS-2 clusters. We found that memory usage for these jobs is almost the same (324KB, for job size 4, 8 and 16). The exception occurs for job size 32, of which memory usage jumps to 52,620KB. Other MPI programs also appears to use memory size of 324KB. Therefore, we might

[7] 0KB is recorded in the PBS accounting logs. It means that the job uses very small memory (rounded to zero) instead of saying that the job does not use memory at all.



(a) CDFs of memory usage

(b) CDFs of memory usage per processor

Fig. 8. Distributions of memory usage and memory usage per processor on DAS-2 clusters.

cluster	memory versus job size	memory/proc versus job size	actual runtime versus job size	actual runtime versus memory	actual versus requested runtime
fs0	0.34	-0.02	0.01	0.72	0.44
fs1	0.59	0.22	0.27	0.71	0.61
fs2	0.64	0.13	0.46	0.68	0.45
fs3	0.25	0.08	-0.25	0.54	0.02
fs4	0.13	-0.08	-0.21	0.51	0.62

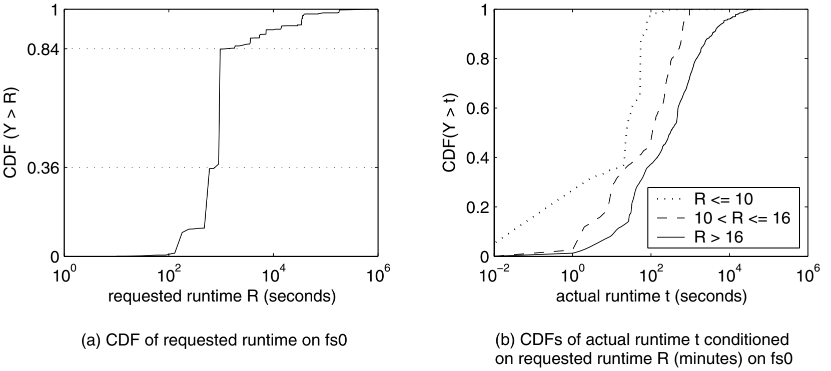
Table 8. Spearman's rank correlation coefficients between job execution characteristics.

say that jobs which use 324KB memory most likely have to load certain libraries like MPI. Memory usage of 2600-3000KB could be other shared libraries or objects.

Distributions of memory usage per processor on a log scale are shown in Figure 8 (b). As we can see, most of the jobs uses less than 10MB memory per processor (only 2% of the available amount). Correlations between memory usage and job sizes are discussed in next section.

5.4 Correlations Between Job Execution Characteristics

A simple way to check the correlations between job execution characteristics is to calculate the *Pearson's R correlation coefficients* between these variables. However, Pearson's R is very weak and misleading in our case since the variables we study are not normally distributed. Instead, we use *Spearman's rank correlation coefficients* to assess the relationship between job execution characteristics, as it makes no assumptions about the variable's distributions. Correlations that we studied are: memory usage versus job size, memory usage per processor versus job size, actual runtime versus job size, memory usage, and requested runtime. Spearman's r coefficients are listed in Table 8.



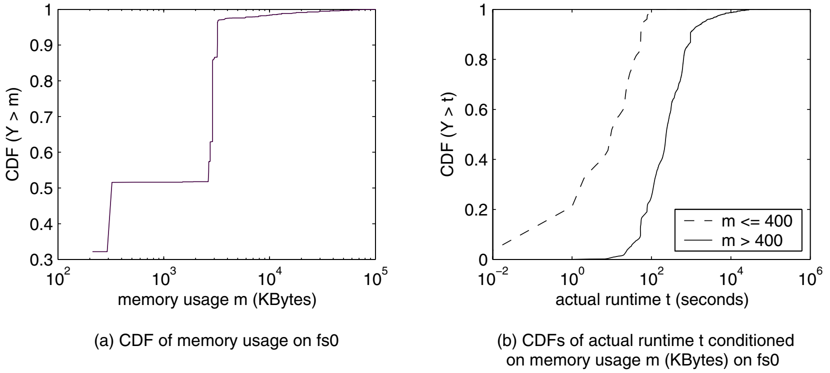
(a) CDF of requested runtime on fs0

(b) CDFs of actual runtime t conditioned on requested runtime R (minutes) on fs0

Fig. 9. CDF of requested runtime and conditional distributions of actual job runtime on fs0.

Firstly we examine the correlations between memory usage and job size. The Spearman's r coefficients show positive correlations. This indicates that larger size jobs (using more processors) tend to use more memory than smaller jobs. Similar characteristics are reported in [23]. Correlations between memory usage per processor and job size have two folds on DAS-2 clusters. On fs1-3 small positive correlations are observed, while on fs0 and fs4, weak inverse correlations are shown. We would expect that memory usage per processor would increase as the job size increases. However, as is discussed in Section 5.3, memory usage is concentrated on special values. Following the same example in Section 5.3, MPI programs with different job sizes (e.g. 4, 8, 16) use the same amount of memory (324KB). This will result an inverse correlation between memory usage per processor and job size. As the job size increases to a certain extent (e.g. 32), the maximum used memory jumps to another level (e.g. 52,620KB). Correspondingly the memory usage per processor grows rapidly and exceeds those of smaller job sizes. This explains why the correlations between memory usage per processor and job size are weak and two-fold.

Correlations between job actual runtime and other characteristics (e.g. job size, requested runtime, etc) are also extensively studied in previous workloads [4,7,9]. For job runtime and size, small positive correlation coefficients are reported in [7], meaning that in general larger jobs runs longer than smaller jobs. On DAS-2 clusters, however, both positive and negative correlations are observed and it is hard to said in general how the actual runtime is related with size. The correlations between actual and requested runtime appear to be strong (except fs3). Naturally jobs with larger requested runtimes generally run longer. This is clearly observed in Figure 9, which illustrates the requested and actual runtime distributions on fs0. In Figure 9 (a), we can see that requested runtimes can be divided into three ranges and each range contains a significant portion of jobs. Actual runtime distributions conditioned on these ranges are shown in Figure 9 (b). Jobs with larger requested runtimes run longer is evident by the fact that their CDFs are below those of jobs with smaller requested runtimes.



(a) CDF of memory usage on fs0

(b) CDFs of actual runtime t conditioned on memory usage m (KBytes) on fs0

Fig. 10. CDF of memory usage and conditional distributions of actual job runtime on fs0.

The most significant correlation is obtained between actual runtime and memory usage. This is also illustrated in Figure 10. However, as our observed memory usage is very special compared with other workloads [23], we choose to generate actual runtimes in a synthetic workload based on the requested runtimes. The fitted conditional actual runtime distributions for the five DAS-2 clusters are given is Table 9. Generally speaking, two-phase log-uniform, Weibull, and lognormal are the best fitted distributions for small, medium, and large requested runtimes, respectively. Exception occurs on fs3, where requested runtimes are only divided into medium and large ranges. Above all, distributions conditioned on requested runtimes are more realistic and accurate in modeling job actual runtimes.

6 User/Group Behavior

User behavior has been discussed in [2,11] as an important structure in the workloads. Workloads typically contain a pool of users with different activity levels and periods. A few users and applications tend to dominate the workload. This special structure results in uniformity and predictability on short time scales, allowing better predictions to be made for improving the scheduler performance [11]. Similar structures are observed on the DAS-2 clusters. In Figure 11 (a), we can see that there are twelve groups on fs0 in total. Six of them are dominant, contributing to the major fraction of the workload. Among the six groups two of them are the most active. They are local groups[8] at VU (CS staff/group 3 and student/group 7). On other clusters similar behavior is observed: local groups are the most active in their cluster workloads. Group Leiden and Delft are of special interest and they are active on most of the DAS-2 clusters. This is partially because Leiden students have to accomplish grid tasks utilizing more than one clusters, and Delft researchers are experimenting processor co-allocation on multi-clusters.

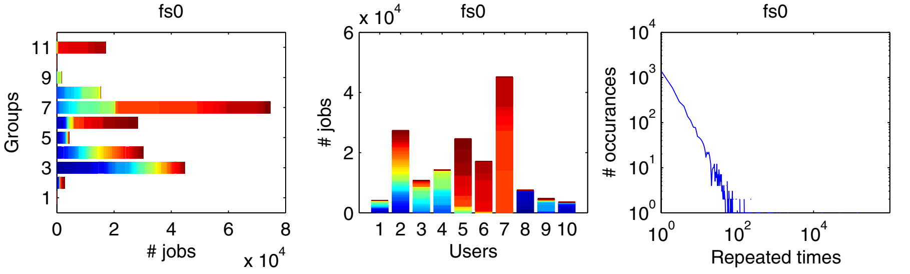
[8] The common group and user accounts are mapped onto all five clusters.

Cluster	Small requested runtime (R - minutes)	Middle requested runtime (R - minutes)	Large requested runtime (R - minutes)
fs0	$0<R\leq 10$, m = 34s, CV = 1.2, loguniform-2 (l=-2.5,m=1.2,h=2.1,p=0.1)	$10<R\leq 16$, m=206s, CV = 1.2, Weibull (a=150, b=0.6)	$R>16$, m = 1624s, CV = 2.9, lognormal (μ=5.4, σ=2.2)
fs1	$0<R\leq 10$, m = 40s, CV = 0.9, loguniform-2 (l=-2.5,m=1.2,h=2, p=0.08)	$10<R\leq 60$, m=250s, CV = 1.5, Weibull (a=184, b=0.7)	$R>60$, m = 6022s, CV = 2.8, lognormal (μ=6.4, σ=2.9)
fs2	$0<R\leq 10$, m = 69s, CV = 0.8, loguniform-2 (l=-2.6,m=1.6,h=2.1,p=0.03)	$10<R\leq 60$, m=301s, CV = 1.5, Weibull (a=229, b=0.7)	$R>60$, m = 7473s, CV = 4.9, lognormal (μ=6, σ=2.7)
fs3	none	$0<R\leq 61$, m=85s, CV = 1.8, Weibull (a=71, b=0.8)	$R>61$, m = 10060s,CV = 2.8, lognormal (μ=6.9, σ=2.6)
fs4	$0<R\leq 16$, m = 72s, CV = 1.5, loguniform-2 (l=-2.5,m=1.7,h=2.3, p=0.04)	$16<R\leq 600$, m=3131s, CV = 10.5, Weibull (a=1369, b=0.5)	$R>600$, m = 4270s,CV = 3.1, lognormal (μ=6.6, σ=2.1)

Table 9. Distributions of job actual runtimes conditioned on requested runtimes (loguniform-2 stands for two-stage log-uniform distribution).

As to the users, 10 out of 130 are the most active on fs0 (see Figure 11 (b)). We further analyze two users with the largest portion of jobs. User 7 submitted more than 40,000 jobs in consecutive seven weeks during October and November 2003, which is his/her only active period throughout the year. Moreover, these jobs all have the same name and request two processors. Jobs from user 2 are distributed evenly throughout the year, but 70% of them have the same name and 90% request two processors. This structure explains some of our main observations before - a majority of DAS-2 workloads have a job size of two processors, and certain applications appear many more times than others. Figure 11 (c) shows the application repeated times and their number of occurrences on fs0. We can see that while lots of applications run only once or a small number of times, there are highly repeated applications that contribute to the heavy tail in the distribution. Similar phenomena are reported on other workloads [11].

Since the user/group structure have an significant impact on the workload modeling, techniques and models have been proposed to capture the user behavior in the workloads [24]. We are also investigating multi-class models on other cluster workloads which are strongly group/VO (Virtual Organization) oriented.



(a) Jobs submitted by different groups in consecutive weeks

(b) Jobs submitted by 10 most active users in consecutive weeks

(c) Application repeat times and their number of occurances

Fig. 11. Activity of groups, users and applications on the cluster fs0. From left to right the color changes (gray scale on a none-color printer) of bars symbolize the consecutive weeks in year 2003.

7 Conclusions and Future Work

In this paper, we present a comprehensive characterization of a multi-cluster supercomputer (DAS-2) workload. We characterized system utilization, job arrival process (arrival rate, interarrival time, and cancellation rate), job execution characteristics (job size, runtime, and memory usage), correlations between different metrics, and user/group behavior. Differences of DAS-2 workloads compared with previously reported workloads include the following:

1. A substantially lower system utilization (from 7.3% to 22%) is observed.
2. Lower job cancellation rates (3.3%-10.6%) are observed than in previously reported workloads (12%-23%).
3. Power-of-two phenomenon of job sizes is clearly observed, with an extreme popularity of job size *two*. The fraction of serial jobs (0.9%-4.7%) is much lower than other workloads (30%-40%).
4. The job actual runtimes are strongly correlated with memory usage as well as job requested runtimes. Conditional distributions based on requested runtime ranges are well fitted for actual runtimes.
5. A large portion of jobs has very small memory usage and several special values are used by a major fraction of jobs.

To facilitate generating synthetic workloads, we provide distributions and conditional distributions of the main characteristics. The distributions are summarized as follows:

1. Interarrival time: in high load period, gamma or two phase hyperexponential are the most suitable distributions; in representative period, Weibull gives the best fit.
2. Cancellation lag: lognormal is the best fitted distribution.
3. Job size: two-stage loguniform is the suitable distribution.

4. Actual runtime: Weibull or lognormal is the best fitted distribution.
5. Actual runtime conditioned on requested time ranges (R): for small R, two-stage loguniform is the most suitable distribution; for medium R, Weibull is the best fitted distribution; for large R, lognormal gives the best fit.

In future work, we plan to generate workload models based on the results in this paper and evaluate several scheduling strategies for DAS-2 clusters. Since the goal of DAS-2 system is to provide fast response time to researchers, load balancing techniques and higher level resource brokering are to be investigated. Another interesting point in a multi-cluster environment is co-allocation. Currently multi-cluster job information is not logged on the DAS-2 clusters. We plan to instrument the Globus gatekeeper to collect the necessary traces and identify the key characteristics for multi-cluster jobs.

8 Acknowledgments

The DAS-2 supercomputer is funded by NWO (Netherlands Organization for Scientific Research) and the participating universities. We thank Dr. Dick Epema (Delft University of Technology) and the referees for their many valuable suggestions that improved the quality of this paper.

References

1. M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81(8): 1136–1150, 1993.
2. D. G. Feitelson. Workload modeling for performance evaluation. *Lecture Notes in Computer Science*, 2459:114–141, 2002.
3. Parallel Workload Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.
4. S.-H. Chiang and M. K. Vernon. Characteristics of a large shared memory production workload. *Lecture Notes in Computer Science*, 2221: 159–187, 2001.
5. D. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 337–360. Springer, 1995.
6. K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A comparison of workload traces from two production parallel machines. In *6th Symp. Frontiers Massively Parallel Comput.*, pages 319–326, 1996.
7. U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel and Distributed Comput.*, 63(11): 1105–1122, 2003.
8. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan. Modeling of workload in MPPs. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 95–116. Springer Verlag, 1997.
9. W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
10. 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 D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer-Verlag, 1999.

11. A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *Perf. Eval. Rev.*, 26(4): 14–29, 1999.
12. The DAS-2 Supercomputer. http://www.cs.vu.nl/das2.
13. S. Banen, A. Bucur and D. H. J. Epema. A Measurement-Based Simulation Study of Processor Co-Allocation in Multicluster Systems. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 105–128. Springer-Verlag, 2003.
14. Portable Batch System. http://www.openpbs.org.
15. The Maui Scheduler. http://www.supercluster.org.
16. The Globus project. http://www.globus.org.
17. O. Allen. Probability, Statistics, and Queueing Theory with Computer Science Applications. Acdemic Press, 1978.
18. R. E. A. Khayari, R. Sadre, B. R. Haverkort. Fitting world-wide web request traces with the EM-algorithm. Performance Evaluation 52, pp 175–191, Elsevier, 2003.
19. Matlab. http://www.mathworks.com.
20. Dataplot. http://www.itl.nist.gov/div898/software/dataplot/.
21. The EMpht programme. http://www.maths.lth.se/matstat/staff/asmus/pspapers.html.
22. Allen B. Downey. Using Queue Time Predictions for Processor Allocation. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 35–57. Springer-Verlag, 1997.
23. D. G. Feitelson Memory usage in the LANL CM-5 Workload. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 78–94. Springer-Verlag, 1997.
24. M. Calzarossa and G. Serazzi. Construction and use of multiclass workload models. *Performance Evaluation*, 19(4): 341–352, 1994.

A Dynamic Co-allocation Service in Multicluster Systems

Jove M.P. Sinaga, Hashim H. Mohamed, and Dick H.J. Epema

Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
D.H.J.Epema@ewi.tudelft.nl
www.pds.ewi.tudelft.nl/˜epema

Abstract. In multicluster systems, and more generally in grids, jobs may require *co-allocation*, i.e., the simultaneous allocation of resources such as processors in multiple clusters to improve their performance. In previous work, we have studied processor co-allocation through simulations. Here, we extend this work with the design and implementation of a dynamic processor co-allocation service in multicluster systems. While an implementation of basic co-allocation mechanisms has existed for some years in the form of the DUROC component of the Globus Toolkit, DUROC does not provide resource-brokering functionality or fault tolerance in the face of job submission or completion failures. Our design adds these two elements in the form of a software layer on top of DUROC. We have performed experiments that show that our co-allocation service works reliably.

1 Introduction

Computer systems consisting of multiple clusters, and more generally grids, offer the promise of transparent access to large collections of resources for very demanding applications. In fact, the needs of a single application may exceed the capacity available in any subsystem making up such a system, and so *co-allocation*, i.e., the simultaneous access to resources of possibly multiple types (processors, data, network bandwidth) in multiple locations, managed by different resource managers [1], may be required. Then, the jobs executing such applications consist of multiple components, with each component using resources in a different subsystem. When co-allocation is not used in multiclusters and grids, such systems only act as large load-balancing devices with higher-level schedulers trying to find good single locations for jobs to run. The real challenge of using such systems is in trying to achieve good mechanisms and policies for co-allocation.

Among the simplest types of applications that need co-allocation are parallel applications that require the simultaneous allocation of processors managed by different schedulers. The feasibility of running such applications in multicluster systems with their relatively slow wide-area connections has been demonstrated for instance in [2]. One of the main problems of processor co-allocation is to achieve the simultaneous availability of the processors managed by different local schedulers. The basic mechanisms for processor co-allocation have existed for a number of years in the form of

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 194–209, 2005.

the Dynamically Updated Request Online Co-allocator (DUROC) [1] component of the Globus Toolkit [3]. However, even though DUROC serves its basic purpose of submitting multicomponent jobs, it cannot be regarded as a co-allocating scheduler for general use in multiclusters or in grids. First, it lacks resource allocation policies by requiring jobs to specify exactly the sites where their components should run. Secondly, DUROC does not provide good fault tolerance in that it may wait for enough available resources for an unspecified amount of time, and in that it requires the user's intervention when the submission or completion of a job fails.

In previous work we have studied processor co-allocation by means of simulations [2,4,5]. In this paper, we extend this work by the design and implementation of a prototype called the Dynamic Co-Allocation Service (DCS) on our wide-area Distributed ASCI[1] Supercomputer (DAS, see Section 4.1) that implements mechanisms and policies for processor co-allocation in multicluster systems. Our design is built on top of DUROC and consists of a Scheduler that implements policies such as FCFS and a form of backfilling, a Resource Monitor that reports on the available resources, a Resource Broker that maps jobs onto the most suitable clusters, and a Co-allocator that submits jobs to DUROC. In particular, our DCS solves the two problems with DUROC mentioned above. The results of the experiments with our prototype show that it works reliably.

2 The Problem of Processor Co-allocation

In this section we formulate the problem of co-allocation in multiclusters, we discuss the DUROC component of Globus, and we describe the structure of multicomponent jobs and the placement and scheduling policies used in our design.

2.1 Processor Co-allocation

In itself, processor co-allocation is a simple notion: Assign processors in different systems in a multicluster or a grid to single jobs simultaneously. The potential advantage to a job is that it may employ more processors than available in a single cluster and so may experience a shorter runtime, and the potential advantage to the system is that the system load may be increased. Of course, due to the relatively slow wide-area communications, not all applicatons will benefit from using processors in clusters connected by a wide-area network, but some definitely will [2].

An important issue in processor co-allocation is that the processors in the different clusters have to be available at the same time. What would be very helpful to guarantee the simultaneous allocation of processors is a reservation mechanism of the local resource managers. However, hardly any of the popular local resource managers such as PBS [6] supports such a mechanism. A processor co-allocation mechanism built on top of resource managers without reservation capabilities cannot do anything else than repeatedly try to assess whether sufficient numbers of processors are available, and then claim these.

[1] ASCI is the acronym of the Advanced School for Computing and Imaging in the Netherlands.

2.2 Motivation

Currently, the only available standard tool for processor co-allocation in grids is the DUROC component of the Globus Toolkit. In general, Globus can accept job descriptions written in the Globus Resource Specification Language (RSL), in which such things as the name of the executable and the input and output files can be specified. When a job consists of a single component, one of the things that also has to be specified is the name of the resource manager of the system where the job has to run. After having parsed the RSL specification of a job, Globus sends the job to the Globus Resource Allocation Manager (GRAM) running on the specified resource, which in turn hands over the job to the local resource manager. DUROC [1] is the component of the Globus Toolkit that contains the basic mechanisms for co-allocation. It accepts what are called multirequests, which consist of multiple simple requests, each written in RSL and each specifying a component of a multicomponent job. DUROC supports two approaches for co-allocating resources to jobs. In the atomic trannsaction approach, all resources specified by a job have to be available otherwise the job's submission fails. In the interactive transaction approach, some resources may be specified as nonessential or optional, in order to tolerate resource failures.

DUROC cannot be regarded as a full-blown co-allocating scheduler in multiclusters or grids, as it lacks certain functionality. First, it does not do any resource brokering by picking suitable resources for a job, and the RSL specification of a job request must be complete in that the subsystems (clusters) to be used by a job must be exactly specified in advance. In other words, DUROC implements what we call a *static* co-allocation mechanism, and it can only deal with what we call ordered jobs (see Section 2.3). Secondly, a job submission may fail because the resources required by the job are not immediately available, or because a job may not complete successfully. When the first happens, DUROC cannot do anything except sending an error message to the user telling that the submission has failed or just waiting until sufficient resources do become available. In the latter case, there is no time-out mechanism for removing jobs that are waiting too long. When the second happens, the user simply has to resubmit the job.

This situation has motivated us to design our Dynamic Co-Allocation Service (DCS) for multicluster systems on top of DUROC. The DCS detects the states of the clusters and *dynamically* allocates resources according to those states. It gives users a more flexible way of specifying multicomponent jobs in that they do not have to tell in advance the locations where the the jobs' components have to run. In addition, the DCS will repeatedly submit a job that experiences submission failures (for a maximum number of times), and it will repeatedly resubmit a job that experiences completion failures (again for a maximum number of times).

2.3 The Structure of the System and of Job Requests

Our model of multicluster systems is very simple: We assume a system of say C clusters consisting of possibly different numbers of identical processors.

Jobs submitted to our DCS that consist of multiple components and require co-allocation, have to specify the number and the sizes of their components, i.e., of the numbers of processors needed in the separate clusters. We assume jobs to be rigid, which means that they do not change size over their lifetime. So a job is represented by a tuple of C values (some of which may be zero), indicating its component sizes. We will consider two cases for the structure of job requests:

1. In an *ordered request* the positions of the request components in the tuple specify the clusters from which the processors must be allocated.
2. For an *unordered request*, by the components of the tuple the job only specifies the numbers of processors it needs in the separate clusters, allowing the scheduler to choose the clusters for the components. Here, we do allow different components of unordered jobs to go to the same cluster.

Ordered requests are used in practice when a user has enough information about the complete system to take full advantage of the characteristics of the different clusters. Unordered requests model applications like FFT, which needs few data, and in which tasks in the same job component share data and need intensive communication, while tasks from different components exchange little or no information.

The RSL descriptions of unordered jobs submitted to the system are incomplete in that the locations where the components should run have not been filled out; the RSL descriptions of the ordered jobs submitted are complete.

2.4 The Placement and Scheduling Policies

For ordered requests it is clear when a job fits on the system or not, given the current numbers of idle processors. To determine whether an unordered request fits, we use the Worst-Fit (WF) placement policy avoiding as much as possible reusing the same clusters. When placing a job, we first order the job components according to decreasing size, and then assign the job components in that order. When assigning a job, we keep two lists of clusters, both ordered according to decreasing numbers of idle processors. The first is the list N of clusters that do not yet have a job component assigned (initially all clusters), and the second is the (initially empty) list Y of clusters that already have at least one job component assigned to them. For every job component, if it fits on the cluster at the head of list N, it is assigned to that cluster and that cluster is removed from N and inserted into the appropriate place into list Y. If it does not fit on the cluster at the head of list N, the job component is assigned to the cluster at the head of list Y if it fits there, and then that list is reordered if necessary. If the component also does not fit on the cluster at the head of list Y, then the whole job cannot be placed. Our motivation for using WF is that it balances the load, leaving roughly equal numbers of idle processors in all clusters. However, WF can easily be replaced by any other placement policy that better suits a multicluster's objectives.

As we will see in Section 3, the DCS maintains a single global queue. As the scheduling policy we use First Come First Served (FCFS) or Fit Processors First Served (FPFS). In FPFS, when the job at the head of the queue does not fit, the queue is scanned from head to tail for any jobs that may fit. FPFS may cause starvation, which may for instance be repaired by putting a maximum to the number of times a job can be overtaken by other jobs, but we have not implemented this. FPFS is a variation of backfilling

[7], for which it is usually assumed that (estimates of) the service times are available before jobs start, and in which the job at the head of the queue may not be delayed by jobs overtaking it. However, we assume that we do not have runtime estimates, and so we cannot implement this type of backfilling.

3 The Design of the Dynamic Co-allocation Service

In this section we will present the design of our Dynamic Co-allocation Service (DCS). We will first give an overview of the architecture of the DCS and the flow of a multi-component job through it. Then, we will focus on each component of the DCS in more detail.

3.1 An Overview of the DCS

The basic idea underlying our design is to employ DUROC and to add several higher-level components to it to implement fault-tolerant dynamic co-allocation. The software components of the DCS are the Scheduler, the Resource Broker, the Resource Monitor, and the Co-allocator. In addition, we maintain as data structures a wait queue and a run list. Figure 1 depicts all of these components.

When a user submits a job to the system, the Scheduler appends it to the tail of the *wait queue*, which contains all jobs submitted but not yet allocated. As long as the wait queue is not empty, the Scheduler tries to schedule jobs from it by contacting the Resource Broker.

When the Resource Broker receives a job request from the Scheduler, it attempts to fit the job on the system taking into account the job type and the available resources. If there are sufficient resources, the Resource Broker decides on an allocation and sends the job request back to the Scheduler; otherwise it sends a failure message back to the Scheduler. In order to fit a job to the resources, the Resource Broker needs to know about the current resource status, which it gets from the Resource Monitor.

When the Scheduler gets a failure message from the Resource Broker, it will simply keep the job in its current location in the wait queue, and it will later attempt to reschedule it. When the Scheduler gets a completed RSL file from the Resource Broker, it will send the job request to the Co-allocator, which in turn will forward it to DUROC. DUROC will use its co-allocation mechanism to submit all subjobs to their destination clusters.

The success or failure of a job submission to DUROC is noted by the Co-allocator in a so-called submission status, which it returns to the Scheduler. If the job submission is successful, the Scheduler removes the corresponding element from the wait queue, and the Co-allocator puts the job request into the *run list*, which contains a record of all currently running jobs so that the Co-allocator can monitor their progress.

However, even if the Resource Broker has found a suitable allocation for a job request, it is possible that the job submission fails. For instance, there may have been a change in the resource availability while the Resource Broker is working to fit the job on the system so that the allocated resources are not available anymore, the executable file cannot be found, etc. If this happens, the Co-allocator will cancel the job submission

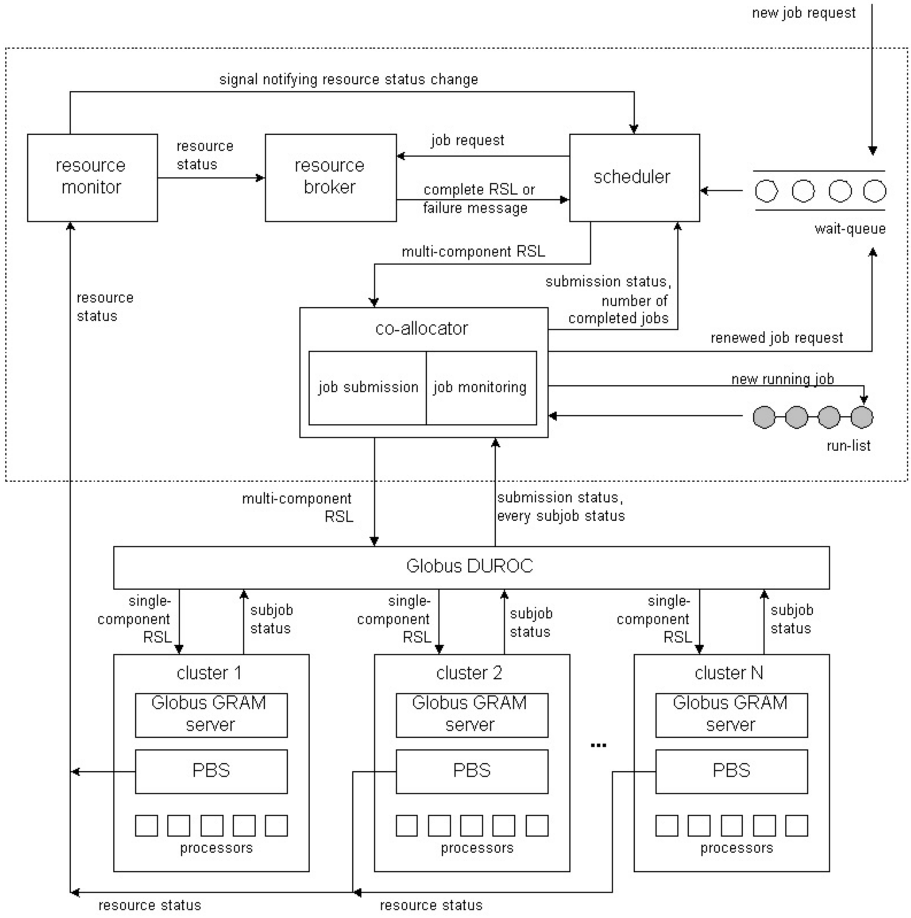


Fig. 1. The architecture of the Dynamic Co-allocation Service.

and tell the Scheduler about the failure. The job request is then moved to the tail of the wait queue.

The Co-allocator also keeps track of the completion status of jobs, which indicates whether or not a runnning job has completed its execution successfully. If a running job fails to complete its execution, the Co-allocator will put the job request back at the tail of the wait queue so that it can later be rescheduled. When a running job finishes successfully, the Co-allocator removes the job from the run list, and sends a message to the Scheduler that contains the number of jobs that have successfully finished so far.

Now, we will see in more detail how each main component is designed.

3.2 The Scheduler and the Wait Queue

The Scheduler is the central component in our design. It manages the wait queue as the place for the requests of jobs that have not yet been allocated, it gets allocations of jobs from the Resource Broker, and it calls the Co-allocator to actually submit jobs to DUROC.

An element of the wait queue includes the following fields:

- The type of the job request (ordered or unordered).
- The text of the original job request in RSL that is still incomplete in case of unordered jobs.
- The number of times the job has suffered a submission failure.
- The number of times the job has suffered a completion failure.

As long as the wait queue is not empty, the Scheduler tries to schedule jobs, based on the scheduling policy, which is FCFS or FPFS. The Scheduler does this whenever it gets a signal from the Resource Monitor that a change in the resource availability has occurred. With FCFS, the Scheduler then invokes the Resource Broker for the job at the head of the wait queue. Only when that job fits does the Scheduler invoke the Resource Broker for the next job, etc. With FPFS, the Resource Broker is invoked once for every job request in the queue. Nevertheless, with FPFS, the Resource Monitor will be invoked only once during the activity of the Resource Broker in a single scheduling action.

If the Resource Broker finds that an ordered job fits, or is able to find a suitable allocation of an unordered job and can fill out its RSL specification, the Scheduler sends the job to the Co-allocator, and waits until the Co-allocator notifies it whether or not the job has been successfully submitted.

If there is a submission failure, the Scheduler increments the relevant field of the job request in the wait queue. If the number of submission failures is then still less than a configurable maximum number, the job request will be moved from its current position to the tail of the queue. If the number of submission failures has reached the maximum number, the Scheduler will remove the job from the queue.

3.3 The Resource Monitor

The Resource Monitor is responsible for collecting information about the resource status of all the clusters and for providing this to the Resource Broker. In our case, the only such information is the processor availability in the clusters. We considered two options for the Resource Monitor to retrieve the processor availability, namely using the Globus Toolkit's MDS component, and directly contacting the local resource managers, all of which in our case are PBS [6]. The MDS command `grid-info-search` in principle provides the information we need, but unfortunately, the MDS information is often not up-to-date since the GRAM reporter is not activated all the time to collect the resource status and report it to the MDS. Therefore, we rejected the MDS as the basis for the Resource Monitor.

PBS provides the `qstat` command to retrieve status information from a cluster, which gives us the number of jobs running in the cluster, the number of compute nodes that are used by each job, and the identifier of each processor executing job processes, etc. After straightening out some little problems in the output of `qstat`, we found that we get accurate and timely information on processor availability, and so this is the option we used. The Resource Monitor stores the information in an output file, which is read by the Resource Broker.

3.4 The Resource Broker

When the Resource Broker gets a job request from the Scheduler, it depends on the request type how it operates. For ordered requests it simply checks whether the requested numbers of processors the job wants to use are available in the specified clusters. After it has done so, the Resource Broker sends a message back to the Scheduler whether the job fits or not.

For unordered jobs the Resource Broker employs the WF algorithm avoiding reusing clusters as much as possible (see Section 2.4). When the job can indeed be assigned to the system according to this algorithm, the Resource Broker completes the RSL specification of the job with the identifications of the clusters to which it has assigned the job's components, and sends it back to the Scheduler. Otherwise it sends a failure message back.

Note that compared to the situation with only plain DUROC, when a job does not fit, in our design DUROC is not called unnecessarily.

3.5 The Co-allocator and the Run List

The Co-allocator is responsible for the submission to DUROC of job requests it gets from the Scheduler. It is also responsible for monitoring the progress of all subjobs in every job while they are being executed. Therefore, it needs a so-called run list which stores the elements representing the running jobs. Each element of this list includes:

- The ID given by DUROC to the job during its execution.
- The number of subjobs in the job.
- A set of states describing the status of every subjob in the job.

The Co-allocator continuously waits for the Scheduler to send it a job request. When it receives such a request, the Co-allocator calls DUROC's job request function to submit the job through DUROC to the system. This function is synchronous (blocking) so the Co-allocator must wait until the function returns. When it does, the Co-allocator gets the information of whether each subjob has been able to get to its destination cluster. If any subjob fails to do so, the Co-allocator will call the DUROC job cancel function to remove all subjobs associated to the job from their clusters, and it will send a submission failure message to the Scheduler. We have DCS use the Global Access to Secondary Storage (GASS) component of the Globus toolkit to automatically move the executable of the job to all clusters where a job component is going to run.

If all subjobs do get to their destination clusters, the Co-allocator must guarantee the job submission success by calling the DUROC barrier release function. This function will hold until all subjobs have entered their own barriers. It may happen that there is a subjob that fails to enter the barrier; after a time-out, the function then returns a failure message to the Co-allocator. However, if the function returns correctly, all the subjobs have been released from their barriers, and the Co-allocator will send a success message to the Scheduler. If the job submission succeeds, the Co-allocator creates a run list element for the job.

The monitoring component of the Co-allocator is active as long as the run list is not empty. For each element in this list, the Co-allocator will call DUROC's `globus-duroc-control-subjob-states` function which has the `Subjob_States` array as its output parameter. The Co-allocator can get the status of every subjob of the corresponding job from this array. If all subjobs have completed their execution successfully, the Co-allocator will remove the element of the completed job from the run list, and record the time when the job completes. When a job experiences a completion failure, the job's number of completion failures is incremented, and when this number does not exceed a maximum, the job is appended to the tail of the wait queue and its number of submission failures is reset to zero. All the progress of the job will be recorded in a log file.

As a note on the implementation, the whole of the DCS consists of four threads, one for the Scheduler and Resource Broker together, one for Resource Monitor, one for the submission function of the Co-Allocator, and one for the monitoring function of the Co-Allocator.

4 Experiments with the DCS

In this section we present some experiments with our Dynamic Co-allocation Service on the DAS. The purpose of these experiments is to show that indeed this service works correctly and reliably, we do not pretend to do a complete performance analysis of it here. Before we present the results of our experiments, we describe the DAS and the application we submit to it in our experiments.

4.1 The Distributed ASCI Supercomputer

The DAS [8] is a wide-area computer system consisting of five clusters (one at each of five universities in the Netherlands, amongst which Delft) of dual-processor Pentium-based nodes, one with 72, the other four with 32 nodes each. The clusters are interconnected by the Dutch university backbone (100 Mbit/s), while for local communications inside the clusters Myrinet LANs are used (1200 Mbit/s). The system was designed for research on parallel and distributed computing. On single DAS clusters the scheduler is PBS [6]. Before the DCS was implemented, jobs spanning multiple clusters could only be submitted with plain DUROC [3].

4.2 The Application

The application that we repeatedly submit to the DAS to test the DCS implements a parallel iterative algorithm to find a discrete approximation to the solution of the two-dimensional Poisson equation (a second-order differential equation governing steady-state heat flow in a two-dimensional domain) on the unit square. For the discretization, a uniform grid of points in the unit square with a constant step in both directions is considered. The application uses a red-black Gauss-Seidel scheme (see for instance [9], pp. 429–433), for which the grid is split up into ”black” and ”red” points, with every red point having only black neighbours and vice versa. In every iteration, each grid point has its value updated as a function of its previous value and the values of its four neighbours, and all points of one colour are visited first followed by the ones of the other colour.

The domain of the problem is split up into a two-dimensional pattern of rectangles of equal size among the participating processes; in our experiments, only one process is assigned to each processor. Every process communicates with each of its neighbours in order to exchange the values of the grid points on the borders and to compute a global stopping criterion. When we execute the Poisson application on multiple clusters, the process grid is split up into adjacent vertical strips of equal width, with each cluster running an equal consecutive number of processes (we assume processes to be numbered in column-major order).

In [2] we have reported extensive measurements on the multicluster performance of this application, showing that for this type of applications, co-allocating them across wide-area systems is a viable option.

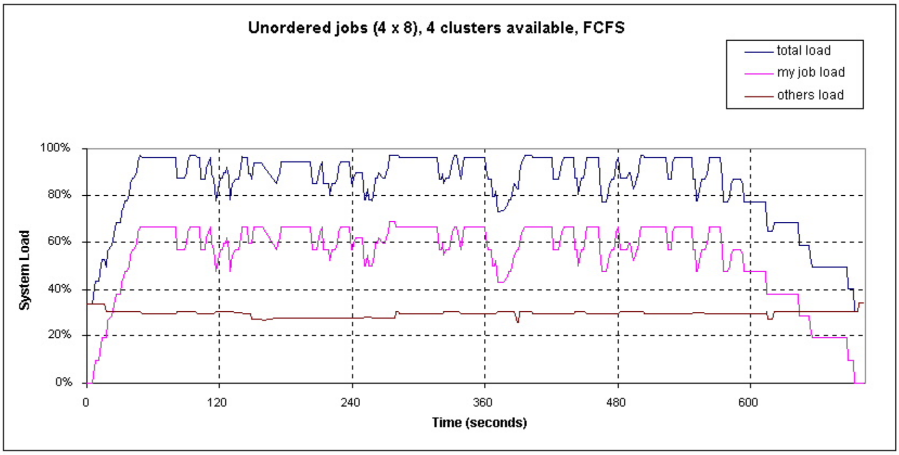
4.3 The Experimental Setup

In all of our experiments, we submit a batch of 40 jobs to the system, all of which run the application explained in Section 4.2. That is, rather than have the jobs arrive over some period of time, they arrive simultaneously. We note that this strains our DCS more than when the jobs would not arrive together, as now many jobs will initially not fit and the wait queue will be long. In all but one experiment we submit only ordered or unordered jobs; in the remaining experiment we submit an even mix of these types. All jobs always have 4 components of equal size, which is either 4 or 8 (indicated by 4x4 and 4x8). In none of our experiments was any job removed from the system because it reached the maximum number of submission or completion failures, which were both set to 3.

Only one of our experiments uses 4 clusters of the DAS, namely the largest cluster with 144 processors and three clusters with 64 processors each (indicated by 144+3x64). In all the other experiments, we could only employ two clusters, one of 144 and one of 64 processors (indicated by 144+64). In our experiments we use both the FCFS and the FPFS policies.

4.4 Experimental Results

We will present the results of five experiments. For each experiment we show a graph that plots the system loads due to our own jobs and due to the jobs of other users, and the sum of these over the time period from the jobs’ submission until the last one of our jobs completes. These system loads are normalized with respect to the total capacity of the clusters that are actually used. In all of our five experiments the system loads due to our jobs is (much) higher than the load due to the jobs of other users. In addition, for all experiments we report the average and the standard deviation of the job response



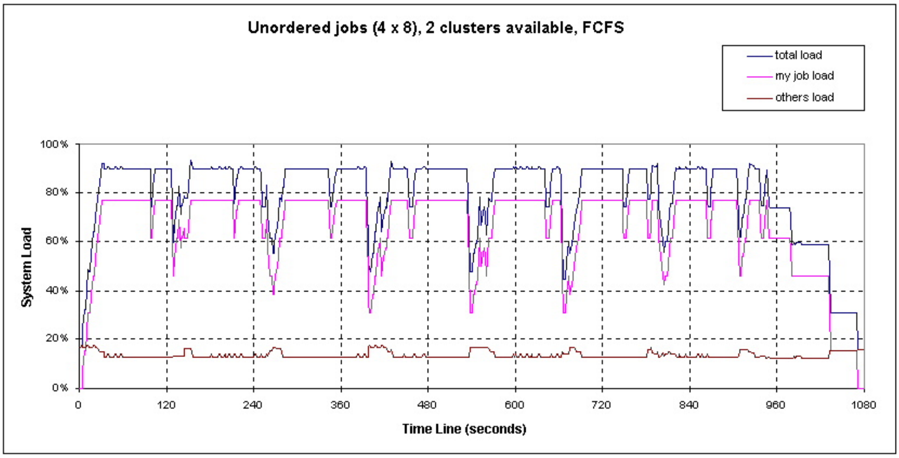
	response time	runtime	overhead
	(seconds)		
avg	377.2	79.4	32.9
stdev	177.2	29.8	17.8

Fig. 2. The system loads and the job response time for the experiment with 4 clusters (144+3x64), unordered jobs of size 4x8, and FCFS.

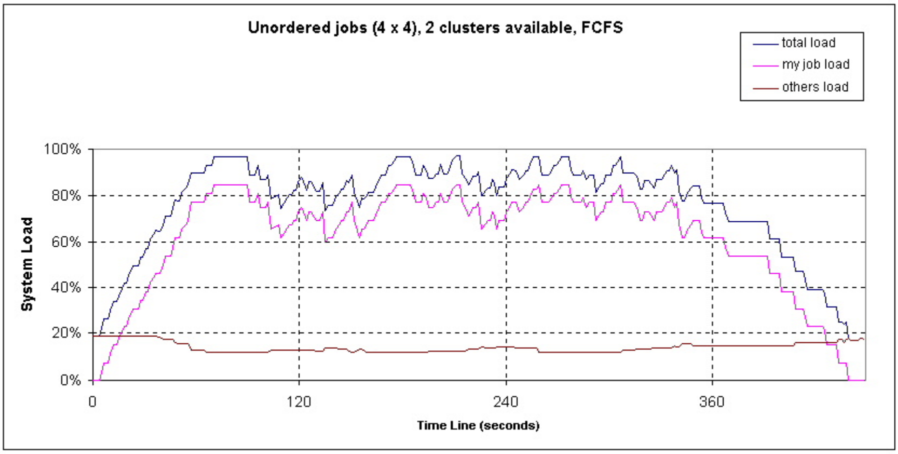
time (total time in the system), of the run time, and of the time due to the overhead caused by DUROC. As it turns out, the standard deviation of the response time is rather large, which is caused by all jobs being submitted at the same time. The overhead due to DUROC has two components, one at job initialization and one at the job completion—the former is by far the largest.

In the first experiment, 4 clusters are employed, all jobs are unordered, and the scheduling policy is FCFS; the results are in Figure 2. We find that the DCS is able to drive the system load to very high levels. This is not very surprising as the jobs are unordered and the job component sizes are relatively small. The sudden drops in system load due to our jobs and the subsequent increase occur at job departures. This phenomenon is caused by the overhead of DUROC and of the DCS. When a job departs from the system, it takes at least a few seconds before the Resource Monitor notices a change in resource status, and then the Scheduler and Resource Broker have to do their work before the Co-Allocator can submit another job to DUROC. Note that here FPFS would have exhibited the same performance as FCFS because all jobs are of equal size.

In Figure 3 we compare two situations with 2 clusters, unordered jobs, and FCFS, where the only difference is the job size. The graphs show the same high total system load and spiky behavior as in Figure 2. Note that the total duration of the experiment with the large job size is much longer, which is due to the larger job size but also to the longer average job runtime. Similarly, a comparison of the graph in Figure 2 and the top graph in Figure 3 shows that with identical workloads, the experiment in the 2-cluster



	response time	runtime	overhead
	(seconds)		
avg	581.3	102.1	25.4
stdev	295.2	12.5	3.5



	response time	runtime	overhead
	(seconds)		
avg	267.2	76.8	26.9
stdev	108.2	17.0	3.2

Fig. 3. The system loads and the job response time for the experiments with 2 clusters (144+64), unordered jobs of size 4x8 (top) and 4x4 (bottom), and FCFS.

case takes much longer (although not quite twice as long because the background load is lower).

In Figure 4 again we compare two situations with 2 clusters, one with ordered jobs and FCFS, and one with an even mix of ordered and unordered jobs and FPFS. Here, the ordered jobs consist of more components than there are clusters, and we specify two components of those jobs to go to either cluster (so in fact, we would achieve the same situation with ordered jobs of size 2x16). With only ordered jobs and FCFS (again we would have the same behavior with FPFS) we find that the total system load achieved is quite low. The reason is that the cluster with 64 processors is quite heavily used while the cluster with 144 processors is not so, but still for every job we need equal numbers of processors in either cluster. In the case of a mix of jobs and FPFS, the total system load is again quite high (and the duration of the experiment is much lower). This is caused by the presence of unordered jobs (which can use the capacity in the large cluster) and the use of FPFS which can schedule unordered jobs even when an ordered job is stuck at the head of the queue.

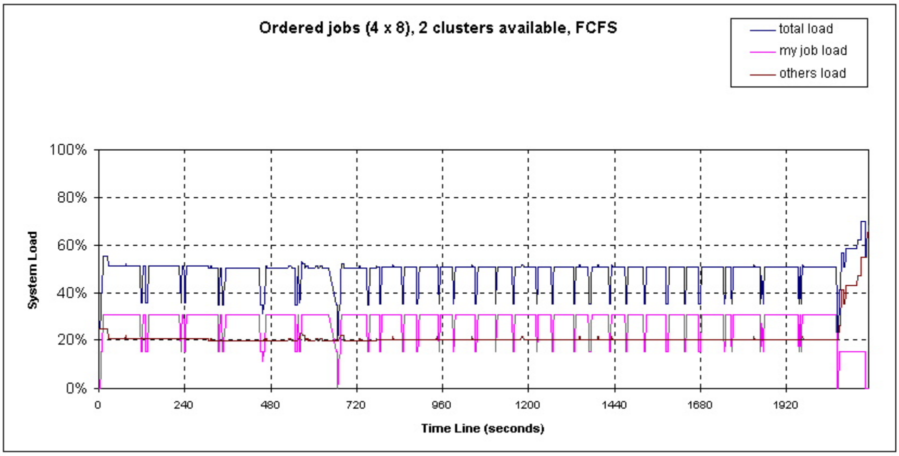
We conclude from our experiments first that our prototype works reliably. Furthermore, we can conclude from our sketchy experiments that ordered jobs may be an obstacle to achieving high utilizations, and that when there are both unordered and ordered jobs in the system, FPFS is definitely to be preferred over FCFS as the scheduling policy.

5 Related Work

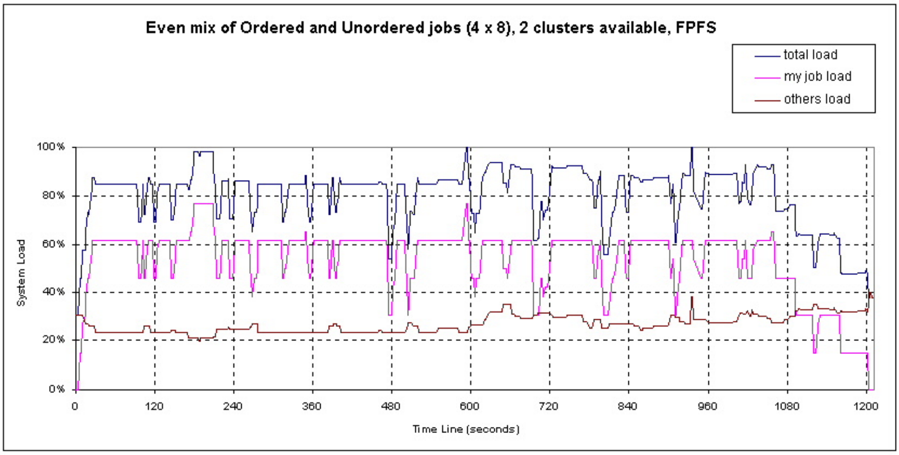
Not very much work has been done on the design, implementation, and performance analysis of co-allocation in multicluster systems and in grids. In terms of designs and implementations, a system that is able to perform allocation of resources in different administrative domains to a single job is Condor with its DAG-manager [10]. Condor's DAGMan takes as input job descriptions in the form of Directed Acyclic Graphs (DAGs), and schedules a task in such a graph when it is enabled (i.e., when all its precedence constraints have been resolved). However, no simultaneous resource possession implemented by a co-allocation mechanism is implemented. In [11], the Condor class-ad matchmaking mechanism for matching single jobs with single machines is extended to "gangmatching" for co-allocation. The running example in [11] is the inclusion of a software license in a match of a job and a machine, but it seems that the gangmatching mechanism might be extended to the co-allocation of processors and data.

In [12], the creation of abstract workflows consisting of application components, their translation into concrete workflows, and the mapping of the latter onto grid resources is considered. These operations have been implemented using the Pegasus [13] planning tool and the Chimera [14] data definition tool. The workflows are represented by DAGs, which are assigned to resources using the Condor DAGMan and Condor-G [10].

In our previous work [2,4,5] we have studied the performance of processor co-allocation in multiclusters through simulations for a wide range of such parameters as the number and sizes of the job components, the number of clusters, the service-time distributions, and the number of queues in the system. There, we considered both syn-



	response time	runtime	overhead
	(seconds)		
avg	1135.2	87.8	21.9
stdev	611.1	10.0	3.4



	response time	runtime	overhead
	(seconds)		
avg	634.7	92.4	25.3
stdev	335.3	15.6	6.5

Fig. 4. The system loads and the job response time for the experiments with 2 clusters (144+64), ordered jobs (top) and an even mix of ordered and unordered jobs (bottom) of size 4x8, and FCFS (top) and FPFS (bottom).

thetics workloads, and workloads derived from the logs of the DAS and from application runtimes on the DAS. In [15,16], co-allocation (called multi-site computing there) is studied with simulations, with as performance metric the average weighted response time. One of the most important findings is that when the slowdown of jobs due to the wide-area communication is less than or equal to 1.25, it pays to use co-allocation. In [17], we consider the maximal utilization, i.e., the utilization at which the system becomes saturated, as a metric for assessing the performance of processor co-allocation.

6 Conclusions and Future Work

In this paper we have presented the design of a Dynamic Co-Allocation Service (DCS) for processor co-allocation in multicluster systems, which has been implemented on our DAS multicluster system. We have also shown the results of experiments that indeed show that this DCS works reliably, and that it is able to achieve a quite high total system load, although the jobs submitted in our experiments were not very large. As far as the authors know this is the first implementation of processor co-allocation with proper resource-brokering functionality and fault tolerance.

We are only at the beginning of our design and implementation efforts of co-allocation in grids. In particular, we are planning to extend the current design of the DCS to more types of resources, to more heterogeneous systems both with repect to the hardware and the local resource managers, and to more complicated job types (e.g., work flows). We note that we have been experimenting with a design of mechanisms for the co-allocation of both processors and information resources which does away with DUROC altogether, but which does use components of the Globus toolkit. Finally, we would like to do a better performance analysis. One of the complicating factors here is the lack of reproducibility of experiments in systems that have a background load submitted by other users that we cannot control.

References

1. Czajkowski, K., Foster, I., Kesselman, C.: Resource Co-Allocation in Computational Grids. In: Proc. of the 8th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-8). (1999) 219–228
2. Banen, S., Bucur, A., Epema, D.: A Measurement-Based Simulation Study of Processor Co-Allocation in Multicluster Systems. In Feitelson, D., Rudolph, L., Schwiegelshohn, U., eds.: Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing. Volume 2862 of LNCS. Springer-Verlag (2003) 105–128
3. The Globus Toolkit, `www.globus.org`
4. Bucur, A., Epema, D.: The Performance of Processor Co-Allocation in Multicluster Systems. In: Proc. of the 3rd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2003), IEEE Computer Society Press (2003) 302–309
5. Bucur, A., Epema, D.: Trace-Based Simulations of Processor Co-Allocation Policies in Multiclusters. In: Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12), IEEE Computer Society Press (2003) 70–79
6. The Portable Batch System, `www.openpbs.org`

7. Lifka, D.: The ANL/IBM SP Scheduling Systems. In Feitelson, D., Rudolph, L., eds.: Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing. Volume 949 of LNCS. Springer-Verlag (1995) 295–303
8. The Distributed ASCI Supercomputer (DAS), `www.cs.vu.nl/das2`
9. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Benjamin/Cummings (1994)
10. Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: Proc. of the 10th IEEE Symp. on High Performance Distributed Computing (HPDC-10), IEEE Computer Society Press (2001) 7–9
11. Raman, R., Livny, M., Solomon, M.: Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In: Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12). IEEE Computer Society Press (2003) 80–89
12. Deelman et al., E.: Mapping Abstract Complex Workflows onto Grid Environments. J. of Grid Computing **1** (2003) 25–39
13. Deelman et al., E.: Pegasus: Mapping Scientific Workflows onto the Grid. In: Proc. of the 2nd European Across Grids Conference. (2004)
14. Foster, I., Vockler, J., Wilde, M., Zhao, Y.: Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In: 14th Int'l Conf. on Scientific and Statistical Database Management (SSDBM 2002). (2002)
15. Ernemann, C., Hamscher, V., Schwiegelshohn, U., Yahyapour, R., Streit, A.: On Advantages of Grid Computing for Parallel Job Scheduling. In: Proc. of the 2nd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2002). (2002) 39–46
16. Ernemann, C., Hamscher, V., Streit, A., Yahyapour, R.: Enhanced Algorithms for Multi-Site Scheduling. In: 3rd Int'l Workshop on Grid Computing. (2002) 219–231
17. Bucur, A., Epema, D.: The Maximal Utilization of Processor Co-Allocation in Multicluster Systems. In: Proc. of the Int'l Parallel and Distributed Processing Symp. (IPDPS), IEEE Computer Society Press (2003) 60–69

Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids

Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, and Aliandro Lima

Universidade Federal de Campina Grande
http://www.ourgrid.org
{elizeu,walfredo,fubica,aliandro}@dsc.ufcg.edu.br

Abstract. Data-intensive applications executing over a computational grid demand large data transfers. These are costly operations. Therefore, taking them into account is mandatory to achieve efficient scheduling of data-intensive applications on grids. Further, within a heterogeneous and ever changing environment such as a grid, better schedules are typically attained by heuristics that use dynamic information about the grid and the applications. However, this information is often difficult to be accurately obtained. On the other hand, although there are schedulers that attain good performance without requiring dynamic information, they were not designed to take data transfer into account. This paper presents *Storage Affinity*, a novel scheduling heuristic for *bag-of-tasks* data-intensive applications running on grid environments. Storage Affinity exploits a data reuse pattern, common on many data-intensive applications, that allows it to take data transfer delays into account and reduce the makespan of the application. Further, it uses a replication strategy that yields efficient schedules without relying upon dynamic information that is difficult to obtain. Our results show that Storage Affinity may attain better performance than the state-of-the-art knowledge-dependent schedulers. This is achieved at the expense of consuming more CPU cycles and network bandwidth.

1 Introduction

Each year more data are generated and need to be processed [1]. Currently, there are many scientific and enterprise applications that deal with a huge amount of data [2][3][4]. These applications are called *data-intensive*. In order to process large datasets, these applications typically need a high performance computing infrastructure. Fortunately, since the data splitting procedure is easy and each data element can often be processed independently, a solution based on data parallelism can often be employed.

Task independence is the main characteristic of parallel *Bag-of-Tasks* (BoT) applications [5][6]. A BoT application is composed of tasks that do not need to communicate to proceed with their computation. In this work, we are interested

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 210–232, 2005.

in the class of applications which has both BoT and *data-intensive* characteristics. We have named it *processors of huge data* (pHd). Shortly, pHd = *BoT* + *data-intensive*. There are innumerous important applications that fall in this category. This is the case, for instance, of data mining, image processing, and genomics.

Due to the independence of their tasks, BoT applications are normally suitable to be executed on grids [7][5]. However, since resources in the grid are connected by wide area network links (wan), the bandwidth limitation is an issue that must be considered when running pHd applications on such environments. This is particularly relevant for those pHd applications that present a data reutilization pattern. For these applications, the data reuse pattern can be exploited to achieve better performance. Data reutilization can be either among tasks of a particular application or among a succession of applications executions. For instance, in the visualization process of quantum optics simulations results [4] it is common to perform a sequence of executions of the same parallel visualization application, simply sweeping some arguments (e.g. zoom, view angle) and preserving a huge portion of the data input from the previous executions.

There exists some algorithms that are able to take data transfer into account when scheduling pHd applications on grid environments [8][9][10][11]. However, they require knowledge that is not trivial to be accurately obtained in practice, especially on a widely dispersed environment such as a computational grid [7]. For example, XSufferage [9] uses information about the CPU and network loads, as well as the execution time of each task on each machine, all of which must be known a priori, to perform the scheduling.

On the other hand, for *CPU-intensive* BoT applications, there are schedulers that do not use dynamic information, yet achieve good performance (e.g. Workqueue with Replication - WQR [12][13]). They use replication to tolerate inefficient scheduling decisions taken due to the lack of accurate information about both the environment and the application. However, these schedulers were conceived to target CPU-intensive applications and thus data transfers are not taken into account by them.

In this paper we introduce *Storage Affinity*, a new heuristic for scheduling pHd applications on grids. Storage Affinity takes into account the fact that input data is frequently reused either by multiple tasks of a pHd application or by successive executions of the application. It tracks the location of data to produce schedules that avoid, as much as possible, large data transfers. Further, it reduces the effect of inefficient task-processor assignments via the judicious use of task replication.

The rest of the paper is organized in the following way. In the next section we present the system model that is considered in this work. In Section 3, we present the Storage Affinity heuristic as well as other heuristics used for comparative purposes. In Section 4, we evaluate the performance of the discussed schedulers. Section 5 concludes the paper with our final remarks and a brief discussion on future perspectives.

2 System Model

This section formally describes the problem investigated and also provides the terminology used in the rest of the paper.

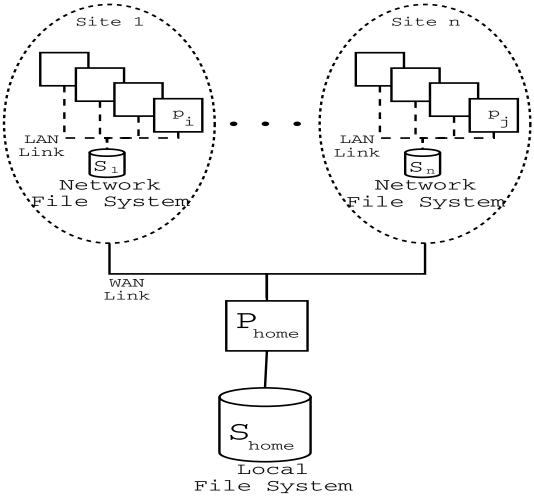


Fig. 1. The system environment model

2.1 System Environment

We consider the scheduling of a sequence of jobs[1] over a grid infrastructure. The grid G is formed by a collection of sites. Each site is comprised of a number of processors, which are able to run tasks, and a single data server which is able to store input data required in the execution of a task, and output data generated by the execution of a task. More formally:

$$G = \{site_1, \dots, site_g\}, g > 0, \text{ and } site_i = P_i \cup \{S_i\},$$

where P_i is the non-empty set of processors at $site_i$ and S_i is the data server at $site_i$. We assume that the resources owned by the various sites are disjoint, *i.e.* $\forall i, j, i \neq j, site_i \cap site_j = \varnothing$.

Processors belonging to the same site are connected to each other through a high bandwidth local area network, whose latency is small and throughput is large when compared to those experienced by the wide area networks that interconnect processors belonging to different sites. Because of this assumption we only consider one data server per site, *i.e.* the collection of data servers that may be present in a site is collapsed into a single data server.

[1] We use the terms job and application interchangeably.

We define two sets to encompass all processors (P_G) and all data servers (S_G) present in a grid G. That is to say:

$$P_G = \bigcup_{1 \leq i \leq |G|} P_i, \quad \text{and} \quad S_G = \bigcup_{1 \leq i \leq |G|} \{S_i\}.$$

We assume that the user spawns the execution of applications from a *home* machine that does not belong to the grid ($p_{home} \notin G$). Further, we assume that before the first execution of an application, all its input data are stored at the local file system of the home machine (S_{home}). The bandwidth is shared equally among the transfers initiated by the user in the *home* machine. Figure 1 illustrates the assumed environment.

2.2 Application

Job J_j, $j > 0$, is the j^{th} execution of the application J. A job is composed by a non-empty collection of tasks, in which each task is defined by two datasets: the input and the output datasets. Formally:

$$J_j = \{t_1^j, \ldots, t_n^j\}, n > 0, \text{ and } t_t^j = (I, O), I \cup O \neq \varnothing,$$

where $t_t^j.I$ and $t_t^j.O$ are the input and the output datasets of task t_t^j, respectively.

For each data server $S_i, S_i \in S_G$, let $ds_j(S_i)$ be the set of data elements that are stored at S_i before the execution of the j^{th} job was started, and let $ds(S_{home})$ be the set of data elements that are stored at the home machine. We define D_j as the set of all data elements that are available to be taken as input by the j^{th} job to be executed. D_j is given by:

$$D_j = ds(S_{home}) \cup \left\{ \bigcup_{S_i \in S_G} ds_j(S_i) \right\}.$$

We have that $t_t^j.I \subseteq D_j$ and after the execution of the j^{th} job, the set of available data elements D_{j+1} is given by the union of D_j with all data elements that have been output by J_j:

$$D_{j+1} = D_j \cup \left\{ \bigcup_{1 \leq t \leq |J_j|} t_t^j.O \right\}.$$

We also define the input dataset of the entire application as the union of the input dataset of each task in the job. It is expressed by:

$$J_j.I = \bigcup_{k=1}^{|J_j|} t_k^j.I$$

2.3 Job Scheduling and Performance Metrics

A schedule Σ_j of the job J_j comprises the schedule of each one of the tasks that form J_j. The schedule of a particular task t_t^j of J_j specifies the processor that is assigned to execute t_t^j. Note that it is possible for the same processor to be assigned to more than one task. Formally,

$$\Sigma_j = \{p_1^j, \ldots, p_n^j\}, n = |J_j|, p_t^j \in P_G, 0 \leqslant t \leqslant n$$

We assume that a task can only access the data server at the same site of the processor on which the task is running. Consequently, if all data elements in the dataset $t_t^j.I$ are not already stored at S_i, the absent data elements must be first transferred to S_i before the execution of t_t^j can be started at p_t^j. Thus, after t_t^j is executed at $p_t^j \in site_i$, S_i will have stored all data elements in the dataset $t_t^j.O$.

We measure the application execution time to evaluate the efficiency of a scheduling. Thus, the heuristic we propose in this paper and also the others that we discuss, all have a common goal, which is to minimize this metric. The application execution time, normally referred as its *makespan* [14], is the time elapsed between the moment the first task is started until the earliest moment in which all tasks have finished their execution.

3 Scheduling Heuristics

Despite the fact that pHD applications are suitable to run on computational grids, the efficient scheduling of these applications on grid environments is not trivial. The difficulty in scheduling pHD application is twofold. The first problem relates to the very nature of pHD applications, which must deal with a huge amount of data. The issue here is that the application overall performance is greatly affected by the large data transfers that occur before the execution of tasks. The second problem is related to obtaining accurate information about the performance resources will deliver to the application. Despite the fact that this information is typically not available a priori, they are input for most available schedulers. In fact, there has been a great deal of research on predicting future CPU and network performance as well as application execution time [15][16][17][18][19]. As the results of these efforts show, this is by no means an easy task. To complicate matters further, the lack of central grid control poses an obstacle for deploying resource monitoring middleware.

We can observe that the difficulty in obtaining dynamic information and the impact of large data transfers have been individually attacked. Therefore, we comment two scheduling heuristics that deal with these problems separately, Workqueue with Replication (WQR) [12] and XSufferage [9]. We also introduce our approach to address the two pHD scheduling problems together.

3.1 Workqueue with Replication

The WQR scheduling heuristic [12] has been conceived to solve the problem of obtaining precise information about the future performance tasks will experience on grid resources. Initially, WQR is similar to the traditional Workqueue scheduling heuristic. Tasks are sent at random to idle processors and when a processor finishes a task, it receives a new task to execute. WQR differs from Workqueue when a processor becomes available and there is no waiting task to start. At this point, Workqueue would just wait for all tasks to finish. WQR, however, starts replicating the tasks yet running. The result from a task comes from the first replica to finish. After the first replica completes, all other replicas are killed.

The idea behind the task replication is to improve the application performance by increasing the chances of running a task on a fast/unloaded processors. WQR achieves good performance for CPU-intensive application [12] without using any kind of dynamic information about processors, network links or tasks. The drawback is that some CPU cycles are wasted with the replicas that do not complete. Moreover, WQR does not take data transfers into account, what results in poor performance for pHd applications, as we shall see in Section 4.

3.2 XSufferage

XSufferage [9] is a knowledge-based scheduling heuristic that deals with the impact of large data transfers on pHd applications running on grid environments. XSufferage is an extension of the *Sufferage* scheduling heuristic [20]. *Sufferage* prioritizes the task that would "suffer" the most if not assigned to the processor that fastest runs it. How much a task would suffer is gauged by its *sufferage value*, which is defined as the difference between the best and the second best completion time for the task.

The main difference between XSufferage and Sufferage algorithms is the sufferage value determination method. In XSufferage, the sufferage value is calculated using the *site-level* task completion times. The *site-level* completion time of a given task is the minimum completion time achieved among all processors within the site. The *site-level sufferage* is the difference between the best and second best *site-level* completion times of the task. The other difference is that XSufferage considers input data transfers in the calculation of the completion time of the task, thus, differently from Sufferage, it requires information about network available bandwidth as input.

The algorithm input is a job J_j and a grid G. The algorithm traverses the set J_j until it finds the task t_t^j with the highest sufferage value. This task is assigned to the processor that has presented the earliest completion time. This action is repeated until all tasks in J_j are scheduled.

The rationale behind XSufferage is to consider the data location when performing the task-to-host assignments. The expected effect is the minimization of the impact of unnecessary data transfers on the application makespan. The evaluation of XSufferage shows that avoiding unnecessary data transfers indeed improves the application's performance [9]. However, XSufferage calculates sufferage values based on the knowledge about CPU loads, network bandwidth utilization and task execution times. In general, this information is not easy to obtain.

```
input   : $G$, $J_j$
output  : $\Sigma_j \cup \Sigma_r$
while ($J_j \neq \varnothing$) do
    - Get ($t_t^j$) which $SA(t_t^j)$ is the largest.
    - Schedule ($t_t^j$) to a processor at $site_i$.
    - $J_j \leftarrow J_j - t_t^j$
    if ($\forall\ p \in P_G \mid p$ is busy) then
        $waitForATaskCompletionEvent()$
    end
end
$J_r \leftarrow$ getAllRunningTasks()
while ($J_r \neq \varnothing$) do
    - Remove from ($J_r$) which:
        * $SA(t_t^j, site_i) = 0$
        * getReplicationDegree($t_t^r$) $> Degree_{min}$
    - Get the ($t_t^r$) which ($SA(t_t^r, site_i)$) is the largest.
    - Schedule replica ($(t_t^r)^d$) to a processor at $site_i$.
    if ($\forall\ p \in P_G \mid p$ is busy) then
        waitForATaskCompletionEvent()
        killAllReplicasOfTheCompletedTask()
    end
    - $J_r \leftarrow$ * getAllRunningTasks() *
end
```

Algorithm 1: Storage Affinity scheduling heuristic

3.3 Storage Affinity

Storage Affinity was conceived to exploit data reutilization to improve the performance of the application. Data reutilization appears in two basic flavors: *inter-job* and *inter-task*. The former arises when a job uses the data already used by (or produced by) a job that executed previously, while the latter appears in applications whose tasks share the same input data. More formally, the *inter-job* data reutilization pattern occurs if the following relation holds:

$$(j < k) \wedge ((J_j.I \cup J_j.O) \cap J_k.I \neq \varnothing)$$

On the other hand, the *inter-task* data reutilization pattern occurs if this other relation holds:

$$\bigcap_{t=1}^{|J_j|} t_t^j.I \neq \varnothing$$

In order to take advantage of the data reutilization pattern and improve the performance of PHD applications, we introduce the *storage affinity* metric. This metric determines *how close* to a site a given task is. By *how close* we mean *how many bytes* of the task input dataset are already stored at a specific site. Thus, *storage affinity* of a task to a site *is the number of bytes within the task input dataset that are already stored in the site*. Formally, the *storage affinity* value between t_t^j and $site_i$ is given by:

$$SA(t_t^j, site_i) = \sum_{d \in (t_t^j.I \cap ds_j(S_i))} |d|$$

in which, $|d|$ represents the number of bytes of the data element d.

We claim that information about data size and data location can be obtained a priori without difficulty and loss of accuracy, unlike, for example, CPU and network loads or the completion time of tasks. For instance, this information can be obtained if a data server is able to answer the requests about *which* data elements it stores and *how large* is each data element. Alternatively, an implementation of a Storage Affinity scheduler can easily store a history of previous data transfer operations containing the required information.

Naturally, since *Storage Affinity* does not use dynamic information about the grid and the application which is difficult to obtain, inefficient *task-to-processor* assignments may occur. In order to circumvent this problem, Storage Affinity applies *task replication*. Replicas have a chance to be submitted to faster processors than those processors assigned to the original task, thus increasing the chance of the task completion time be decreased.

Algorithm 1 presents *Storage Affinity*. Note that this heuristic is divided in two phases. In the first phase Storage Affinity assigns each task $t_t^j \in J_j$ to a processor $p \in G$. During this phase, the algorithm calculates the highest storage affinity value for each task. After this calculation, the task with the largest storage affinity value is chosen and scheduled. This continues until all tasks have been scheduled. The second phase consists of task replication. It starts when there are no more waiting tasks and there is, at least, one available processor. A replica could be created for any running task. Considering that the replication degree of a particular task is the number of replicas that have been created for the task, whenever a processor is available, the following criteria are considered to choose the task to be replicated: i) the task must have a positive storage affinity with the site that has an available processor; ii) the current replication degree of the task must be the smallest among all running tasks; and iii) the task must have the largest storage affinity value among all remaining candidates. When a task completes its execution, the scheduler kills all remaining replicas of the task. The algorithm finishes when all the running tasks complete. Until this occurs the algorithm proceeds with replications.

4 Performance Evaluation

In this section we analyze the performance of Storage Affinity, comparing it against WQR and XSufferage. We have decided to compare our approach to these heuristics because WQR represents the state-of-the-art solution to circumvent the dynamic information dependence, whereas XSufferage is the state-of-the-art for dealing with the impact of large data transfers. We have used simulations to evaluate the performance of the scheduling algorithms. These simulations were validated by performing a set of real-life experiments (see Section 4.5).

Since the performance attained by a scheduler is strongly influenced by the workload [21] [22][23], we have designed experiments that cover a wide variety of scenarios. The scenarios vary in the heterogeneity of both the grid and the application, as well as the application granularity (see Section 4.2). Our hope was not only to identify which scheduler performs better, but also to understand how different factors impact their performance.

4.1 Grid Environment

Each task has a computational cost, which expresses how long the task would take to execute in a dedicated reference processor. Processors may run at different speeds. By definition, the reference processor has *speed* = 1. So, a processor with *speed* = 2 runs a 100-second task in 50 seconds (when dedicated). Since the computational grid may comprise processors acquired at different points in time, grids tend to be very heterogeneous (i.e. their processors speed may vary widely). In order to investigate the impact of grid heterogeneity on scheduling, we consider four levels of grid heterogeneity, as shown in Table 1[2]. Thus, for heterogeneity $1x$, we always have *speed* = 10, and the grid is homogeneous. On the other hand, for heterogeneity $8x$, we have maximal heterogeneity, with the fastest machines being up to 8 times faster than the slowest ones. Note that, in all cases, the average speed of the machines forming the grid is 10.

Grid Heterogeneity	Processor Speed Distributions
$1x$	$U(10, 10)$
$2x$	$U(6.7, 13.4)$
$4x$	$U(4, 16)$
$8x$	$U(2.2, 17.6)$

Table 1. Grid heterogeneity levels and the distributions of the relative speed of processors

The grid power is the sum of the speed of all processors that comprise the grid. For all experiments we fixed the grid power to $1,000$. Since the speed of processors are obtained from the Processor Speed Distributions, a grid is "constructed" by adding one processor at a time until the grid power reaches

[2] Note the $U(x, y)$ denotes the uniform distribution in the $[x, y]$ range.

$1,000$. Therefore, the average number of processors in the grid is 100. Processors are distributed over the sites that form the grid in equal proportions. Similarly to Casanova et al [9], we assume that a grid has $U(2,12)$ sites.

For simplicity, we assume that the data servers do not run out of disk space (i.e., we do not address data replacement policies in the present work). As previously indicated, we neglect data transfers within a site. Inter-site communication is modeled as a single shared $1Mbps$ link that connects the *home* machine to several sites. It important to highlight that, in the model, $1Mbps$ is the maximum bandwidth that an application can use in the wide area network. However, the connections are frequently shared among several applications, thus, the limit is often not achieved by a particular application. We used NWS [24] real traces to simulate contention for both CPU cycles and network bandwidth. For example, a processor of *speed* = 1 and *availability* = 50% runs a 100-second task in 200 seconds.

4.2 pHd Applications

In pHd applications, the application execution time is typically related to the size of the input data. The explanation for this fact is quite simple. The more data there is to process, the longer the tasks take to complete. In fact, there are pHd applications whose cost is completely determined by the size of the input data. This is the case, for example, of a scientific data visualization application, which processes the whole input data to produce the output image [4]. There are other applications that have the cost influenced, but not completely determined, by the size of the input data. This is the case of a *pattern search* application, in which the size of the input data of each task determines an upper bound for the cost of the task, not the cost of the task itself. We simulated both kinds of applications.

The total size of the input data of each simulated application was fixed in $2GBytes$. Based on experimental data available [4], we were able to convert the amount of input data processed by each task of the visualization application into the time (in seconds) required to process the data, which is its computational cost. We have used the same proportionality factor (1.602171 ms/KByte) to calculate the computational cost of the pattern search application, as a function of the amount of data actually processed by its tasks. To determine the computational cost of each task of the pattern search application, we used an uniform distribution $U(1, UpperBound)$, in which $UpperBound$ is the computational cost to process the entire input of a particular task.

We also wanted to analyze how the relation between the average number of tasks and the number of processors in the grid would impact the performance of a schedule. Note that when both application and grid sizes are fixed, this relation is inversely proportional to the average size of the input data of the tasks that comprise the application, i.e. the *application granularity*. We have considered three application groups that are defined by the following application granularity values: $3MBytes$, $15MBytes$ and $75MBytes$.

The tasks that comprise the application can vary in size. Therefore, to simulate this variation, we introduced an application heterogeneity factor. The heterogeneity factor determines how different are the sizes of the input data elements of the tasks that form the job, and consequently their costs. The size of the input data are taken from the uniform distribution $U(AverageSize \times (1 - \frac{H_a}{2}), AverageSize \times (1 + \frac{H_a}{2}))$, in which $AverageSize \in \{3MBytes, 15MBytes, 75MBytes\}$ and $H_a \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$.

4.3 Simulation Setting and Environment

A total of $3,000$ simulations were performed, with half of them for each type of application (Visualization and Pattern Search). As we shall see in Section 4.4, $3,000$ simulations make for good precision of results (in 95% confidence interval). Each simulation consists of a sequence of 6 executions of the same job. Those 6 executions are repeated for each of 3 analyzed scheduling heuristics (Workqueue with Replication, XSufferage and Storage Affinity). Therefore, we have 18000 makespan values for each scheduling heuristic analyzed.

Our simulation tool has been developed using an adapted version of the Simgrid toolkit [25]. The Simgrid toolkit provides the basic functionalities for the simulation of distributed applications on grid environments. Since the set of simulations is itself a BoT application, we have executed it over a grid composed of 107 machines distributed among five different administrative domains (LSD/UFCG, Instituto Eldorado, LCAD/UFES, UniSantos and GridLab/UCSD). We have used the MyGrid middleware [5] to execute the simulations.

4.4 Simulation Results

In this section we show the results obtained in the simulations of the scheduling heuristics and discuss their statistical validity. We also analyze the influence of the application granularity, as well as the heterogeneity of both the grid and the application on the performance of the application scheduling.

Summary of the Results Table 2 presents a summary of the simulation results. It is possible to note that, in average, Storage Affinity and XSufferage achieve comparable performances. Nevertheless, the standard deviation values indicate that the makespan presents a smaller variation when the application is scheduled by Storage Affinity when compared to the other two heuristics. It is important to explain that the *resource wasting* percentage is given by: $\frac{TimeConsumedByKilledReplicas}{TimeConsumedByFinishedTasks}$. Obviously, we do not report any wasting values in Table 2 for XSufferage because this heuristic does not apply any replication strategy, consequently it does not kill any running task.

In order to evaluate the precision and confidence of the summarized means presented in Table 2, we have determined the 95% *confidence interval* [26] for the population mean (μ) based on those values in Table 2. That is, using the sample

Makespan (seconds)	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	14377	42919	14665
Standard deviation (σ)	10653	24542	11451
CPU Wasting	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	59.243%	1.0175%	–
Standard deviation (σ)	52.715%	4.1195%	–
Bandwidth Wasting	*Storage Affinity*	WQR	*XSufferage*
Mean ($\overline{x}$)	3.1937%	130.88%	–
Standard deviation (σ)	8.5670%	135.82%	–

Table 2. Summary of simulation results

mean, standard deviation and the sample size (number of makespan values) we estimate the confidence intervals, as shown in Table 3.

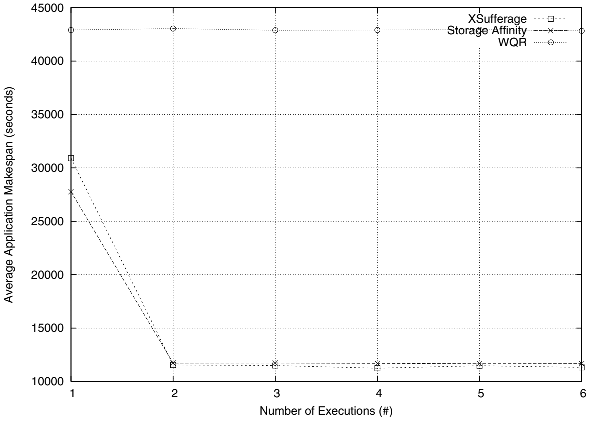


Fig. 2. Summary of the performance of the scheduling heuristics

Since the width of the confidence interval (w) is relatively small compared to the results (see Table 4), we feel that we have performed enough simulation to obtain a good precision in the results.

In Figure 2 we show the average application makespan and in Figure 3 we present the resource waste for all performed simulations with respect to all heuristics analyzed. The results show that both data-aware heuristics attain much better performance than WQR. This is because data transfer delays dominate the makespan of the application, thus not taking them into account severely hurts the performance of the application. In the case of WQR, the execution of each task is always preceded by a costly data transfer operation (as can be inferred from the large bandwidth and small CPU waste shown in Figure 3). This impairs any improvement that the replication strategy of WQR could bring. On the other hand, the replication strategy of Storage Affinity is able to cope with the lack of dynamic information and yield a performance very similar to that

Heuristic	95% *Confidence interval*
Storage Affinity	$14241 < \mu < 14513$
Workqueue with Replication	$42547 < \mu < 43291$
XSufferage	$14498 < \mu < 14832$

Table 3. 95% confidence intervals for the mean of the makespan for each heuristic.

Heuristic	w	% with respect the makespan
Storage Affinity	330	2.2%
Workqueue with Replication	330	2.2%
XSufferage	850	2%

Table 4. Width of the confidence intervals and proportion with respect the mean

of XSufferage. The main inconvenience of XSufferage is the need for knowledge about dynamic information, whereas the drawback of Storage Affinity is the consumption of extra resources due to its replication strategy (an average of 59% of extra CPU cycles and a negligible amount of extra bandwidth). From this result we can state that the Storage Affinity task replication strategy is a feasible technique to obviate the need for dynamic information when scheduling PHD applications, although at the expenses of consuming more CPU.

Application Granularity Next, we investigate the impact of application granularity on the application scheduling performance. In Figure 4 we can see the influence of the three different granularities on the data-aware schedulers. From the results presented we conclude that no matter the heuristic used, smaller granularities yield better performance. This is because smaller tasks allow greater parallelism. We can further observe that XSufferage achieves better performance than Storage Affinity only when the granularity of the application is $75 Mbytes$. This is because the larger a particular task is, the bigger its influence in the makespan of the application. Thus, the impact of a possible inefficient task-host assignment for a larger task is greater than that for a smaller one. In other words, the replication strategy of Storage Affinity is more efficient when circumventing the effects of inefficient task-host assignments when the application granularity is small. Nevertheless, for PHD applications, it is normally possible - and quite easy - to reduce the application granularity by converting a task with a large input into several tasks with smaller input datasets. The conversion is performed by simply slicing large input datasets into several smaller ones. It is important to note that there is a trade off here, because the scheduling overhead when running on real environments.

Given the above discussion, we show in Figure 5 the values for the makespan of the applications, considering only the granularities $3 Mbytes$ and $15 Mbytes$. For these simulations, Storage Affinity outperforms XSufferage by 42%, in average. Further, as can be seen in Figure 6, the percentage of CPU cycles wasted is reduced from 59% to 31%, in average. We emphasize that reducing the appli-

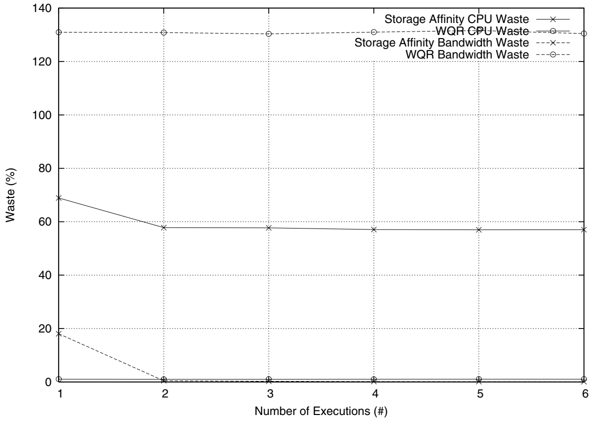


Fig. 3. Summary of resource waste

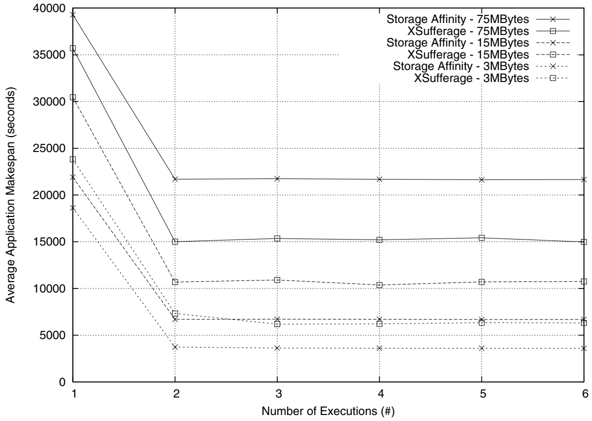


Fig. 4. Impact of application granularity

cation granularity is a good policy as smaller tasks yields more parallelism (see Figure 4).

Application Type In order to analyze the influence of the different characteristics of pHd applications on the application makespan and the resource waste, we have considered two types of applications (see Section 4.2). The results show that the behavior of the heuristics has not been affected by the different characteristics of the application. On the other hand, we found out that the waste of resources was affected by the type of application considered. Figure 9 and Figure 10 show the results attained. Recall that in the data visualization application, the computational cost of the task is completely determined by the size of its input dataset. Since Storage Affinity prioritizes the task with the largest

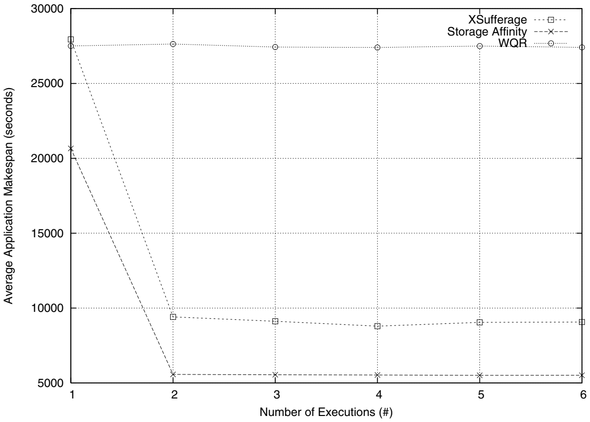


Fig. 5. Performance of the heuristics with respect to the granularities $3Mbytes$ and $15MBytes$

storage affinity value, it means that the largest tasks are scheduled first. Therefore, task replication only starts when most of the application has already been executed. In the case of the pattern search application, the computational cost of the tasks is not completely determined by the size of the input dataset of the task, thus proportionally large tasks can be scheduled at later stages in the execution of the application. Therefore, replication may start when a large portion of the application is still to be accomplished, and consequently more resources are wasted to improve the application makespan.

Grid and Application Heterogeneity Finally, we have analyzed the impact of the heterogeneity of the grid and the application in both Scientific Visualization and Pattern Search applications.

In Figure 7 and Figure 8 we can see how the heterogeneity of the grid influences the makespan of both types of applications, considering the three heuristics discussed. It is possible to see that the two data-aware heuristics are not greatly affected by the variation of the grid heterogeneity. It is not surprising that XSufferage presents this behavior, given that it uses information about the environment. However, Storage Affinity shows that its replication strategy circumvents the effects of the variations of the speed of processors in the grid, even without using information about the environment. WQR is influenced a lot by the grid heterogeneity variation, we can see that increasing the grid heterogeneity the application makespan get worse.

Storage Affinity and XSufferage present a similar behavior with respect to application heterogeneity. Both heuristics show good tolerance to the variation of the application heterogeneity. In Figure 11 and Figure 12 we observe that the application makespan presents a tiny fluctuation for both types of application (Visualization and Search).

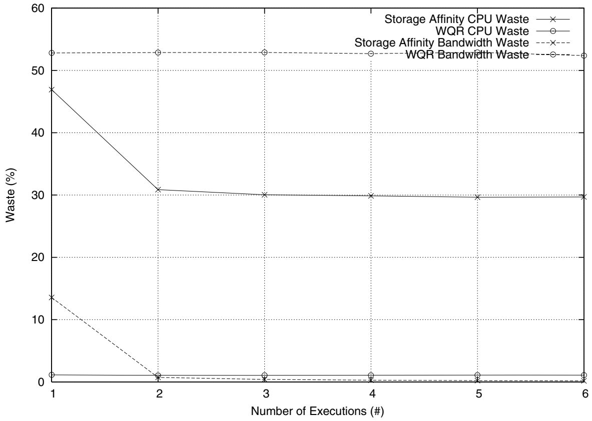


Fig. 6. Resource waste with respect to the granularities $3Mbytes$ and $15MBytes$

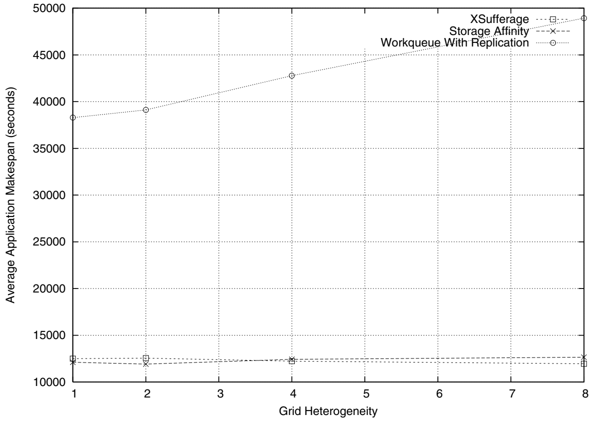


Fig. 7. Grid Heterogeneity impact for Scientific Visualization application

4.5 Validation

In order to validate our simulations, we have conducted some experiments using a prototype version of *Storage Affinity*. The *Storage Affinity* prototype has been developed as a new scheduling heuristic for MyGrid [5,27].

The grid environment used in the experiments was comprised by 18 processors located at 2 sites (Carcara Cluster/LNCC - Teresópolis, Brazil and GridLab/UCSD - San Diego, USA). The home machine (p_{home}) was located at the Laboratório de Sistemas Distribuídos/UFCG - Campina Grande, Brazil. It is important to highlight that during the experiments the resources were shared with other applications.

With respect to the application, we have used BLAST [2]. BLAST is an application that searches a given sequence of characters into a database. These

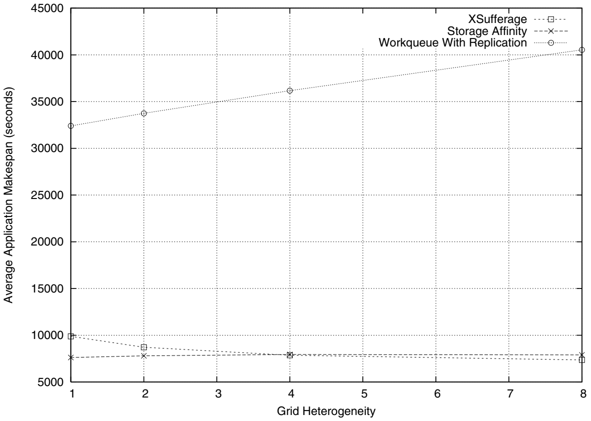


Fig. 8. Grid Heterogeneity impact for Pattern Search application

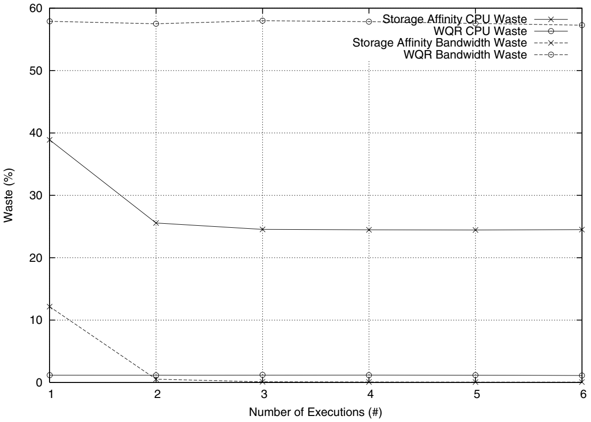


Fig. 9. Resource waste considering the Scientific Visualization Application

characters represent a protein sequence and the database contains several identified sequences of proteins. The application receives two parameters: a database and a sequence of characters to be searched. The database size is of the order of many GBytes, but it can be sliced into many slices of few MBytes. On the other hand, the size of the sequence of the characters to be searched does not surpass 4KBytes.

The application was composed of 20 tasks. Each task of the application receives a slice of 3MBytes of a large database downloaded from the BLAST site [28] and a sequence of characters smaller than 4KBytes. Since the simulations have been focused on applications that present *inter-job* data reuse pattern (Section 3.3), we have set the application to present the same data reuse pattern. Most of the input of each task was reused (the database), while a minor part of the input (the search target of few KBytes) has changed between executions.

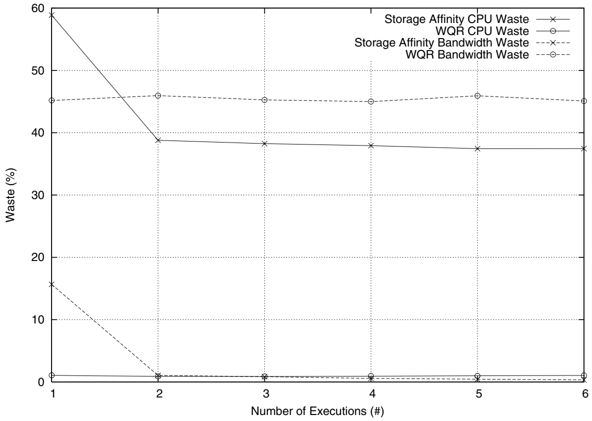


Fig. 10. Resources wasted considering the Pattern Search Application

Methodology Two scheduling heuristics have been analyzed in the experiments: *Storage Affinity* and *Workqueue with Replication*. We did not use *XSufferage* due to the very lack of deployed monitoring infrastructure that could provide resource load information. On the other hand, MyGrid already has a version of *Workqueue with Replication* heuristic available.

In order to minimize the effect of the grid dynamism on the results, the experiment consisted of *back-to-back* executions of the two scheduling heuristics (i.e. we did intermixed the experiments of both scheduling heuristics). Following this approach, 11 experiments have been executed. Each experiment consisted of 4 successive executions of the same application for each scheduling heuristic, thus adding to a total of 88 application executions.

Results In Figure 13 we present the average of the application makespan for each scheduling heuristic. Figure 14 contains the simulation of the scenario used in the experiment. The results show that both *Storage Affinity* and *Workqueue with Replication* present the same overall behavior noticed in the simulations. However, the experiment does differ from the simulation in some aspects. One is the greater fluctuation the makespan values in the experiment. This is due to the high level of heterogeneity of the grid environment, and to the fact that we were ran much fewer cases than we simulated.

We can also observe a difference between the makespan in the experiment results and the simulated scenario. We believe there are two reasons for this discrepancy. First, we could not collect CPU and network loads experienced during the real life experiments. The standard NWS logs we used instead. Therefore, the grid scenario is not quite the same for the simulation and the experiments. Second, the *Storage Affinity* prototype always queries the sites to obtain information on the existence and size of files. This costly remote operation was not modeled in the simulator. However, since the scheduler itself is the responsible

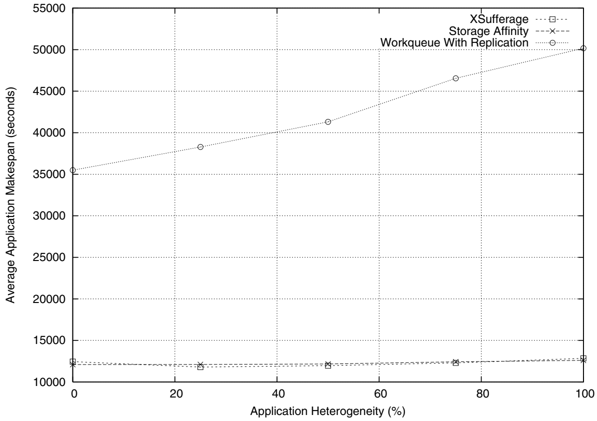


Fig. 11. Application heterogeneity impact for Scientific Visualization application

for transfering files to the sites, this information can be cached locally, thus greatly reducing the need for remote invocations during the execution of *Storage Affinity*. We are currently implementing such caching strategy and expect such a modification to greatly reduce the discrepancy between simulation and experimentation.

5 Conclusions and Future Work

In this paper we have presented Storage Affinity, a novel heuristic for scheduling PHD on grid environments. We have also compared its performance against that of two well-established heuristics, namely: XSufferage [9] and WQR [12]. The former is a knowledge-centric heuristic that takes data transfer delays into account, while the latter is a knowledge-free approach, that uses replication to cope with inefficient task-processor assignments, but does not consider data transfer delays. Storage Affinity also uses replication and avoids unnecessary data transfers by exploiting a data reutilization pattern that is commonly present in PHD applications. In contrast with the information needed by XSufferage, the data location information required by Storage Affinity is trivially obtained, even in grid environments.

Our results show that taking data transfer into account is mandatory to achieve efficient scheduling of PHD applications. Further, we have shown that grid and application heterogeneity have little impact in the performance of the studied schedulers. On the other hand, the granularity of the application has an important impact on the performance of the two data-aware schedulers analyzed. Storage Affinity is outperformed by XSufferage only when application granularity is large. However, the granularity of PHD applications can be easily reduced to levels that make Storage Affinity outperform XSufferage. In fact, independently of the heuristic used, the smaller the application granularity the

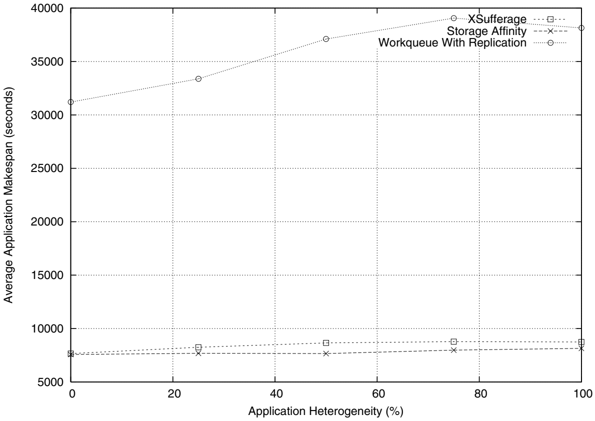


Fig. 12. Application heterogeneity impact for Pattern Search application

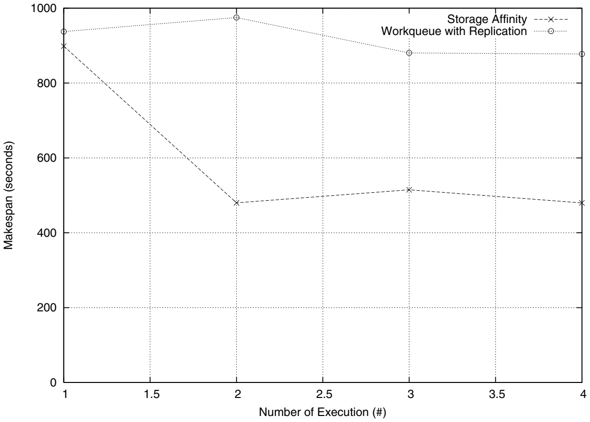


Fig. 13. Real application execution using prototype version of schedulers

better the performance of the scheduler (at least the granularity size which corresponds to an overhead starts to dominate the execution time). In the favorable scenarios, Storage Affinity achieves a makespan that is in average 42% smaller than XSufferage. The drawback of Storage Affinity is the waste of grid resources due to its replication strategy. Our results show that the wasted bandwidth is negligible and the wasted CPU can be reduced to 31%.

As future work, we intend to investigate the following issues: i) the impact of the inter-task data reutilization pattern on application scheduling; ii) disk space management on data servers; iii) the emergent behavior of a community of Storage Affinity schedulers competing for shared resources; and iv) the use of introspection techniques for data staging [29] to provide the scheduler with information about data location and disk space utilization. Finally, we are about to release a stable version of Storage Affinity within the MyGrid middleware [5,27].

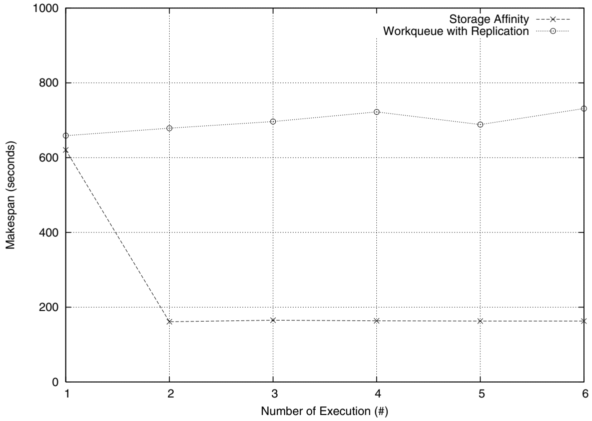


Fig. 14. Simulation of the scenario used in the experiments

We hope that practical experience with the scheduler will help us to identify aspects of our model that need to be refined.

Acknowledgments

We would like to thank HP Brazil, CNPq and CAPES for the financial support, Nazareno Andrade, Daniel Paranhos, Glaucimar Aguiar, Roberta Coelho, Lauro Costa, Marcelo Meira, Zane Cirne and Randolph Rodrigues for helping us when it was needed. We would like also to thank people at UFES, UniSantos, Eldorado and UCSD for letting us execute our simulations on their machines.

References

1. P. Lyman, Hal R. Varian, J. Dunn, A. Strygin and K. Swearingen. *How much information ?*, http://www.sims.berkeley.edu/research/projects/how-much-info-2003, October, 2003.
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman. *Basic local alignment search tool*. Journal of Molecular Biology, vol. 1, 215, pp. 403–410, 1990.
3. GriPhyN Group. http://www.GriPhyN.org, 2002, http://www.GriPhyN.org.
4. E. L. Santos-Neto, L. E. F. Tenório, E. J. S. Fonseca, S. B. Cavalcanti and J. M. Hickmann. *Parallel Visualization of the optical pulse through a doped optical fiber*. in Proceedings of Annual Meeting of the Division of Computational Physics (abstract), June, 2001.
5. W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. B. da Silva, C. O. Barros and C. Silveira. *Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach*, in Proceedings of the ICCP'2003 - International Conference on Parallel Processing, October, 2003.
6. J. Smith and S. K. Shrivastava. *A System for Fault-Tolerant Execution of Data and Compute Intensive Programs over a Network of Workstations*, Lecture Notes in Computer Science, vol. 1123, pp. 487–495, 1996.

7. I. Foster and C. Kesselman (eds.). *The Grid: Blueprint for a Future Computing Infrastructure*, 1999.
8. O. Beaumont, L. Carter, J. Ferrante and Y. Robert. *Bandwidth-centric Allocation of Independent Task on Heterogeneous Plataforms*, in Proceedings of the Internetional Parallel and Distributed Processing Symposium, April, 2002.
9. H. Casanova, A. Legrand, D. Zagorodnov and F. Berman. *Heuristics for Scheduling Parameter Sweep Applications in Grid environments*, in Proceedings of the 9th Heterogeneous Computing Workshop, May, 2000.
10. M. Faerman, A. Su, R. Wolski and Francine Berman. *Adaptive Performance Prediction for Distributed Data-Intensive Applications*, in Proceedings of the ACM/IEEE SC99 Conference on High Performance Networking and Computing, 1999.
11. K. Marzullo, M. Ogg, A. Ricciardi, A. Amoroso, A. Calkins and E. Rothfus. *NILE: Wide-Area Computing for High Energy Physics*, in Proceedings 7th ACM European Operating Systems Principles Conference. System Support for Worldwide Applications", September , 1996.
12. D. Paranhos and W. Cirne and F. Brasileiro. *Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids*, in Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing, August, 2003.
13. Z. M. Kedem and K. V. Palem and P. G. Spirakis. *Efficient Robust Parallel Computations (Extended Abstract)*, in Proceedings of ACM Symposium on Theory of Computing, 1990.
14. M. Pinedo. *Scheduling: Theory, Algorithms and Systems*, 2nd edition, August, 2001.
15. A. Downey. *Predicting queue times on space-sharing parallel computers*, in Proceedings of 11th International Parallel Processing Symposium (IPPS'97), April, 1997.
16. R. Gibbons. *A Historical Application Profiler for Use by Parallel Schedulers*, Lecture Notes in Computer Science, vol. 1291, pp. 58-77, 1997.
17. W. Smith, I. Foster and V. Taylor. *Predicting Application Run Times Using Historical Information*, Lecture Notes in Computer Science, vol. 1459, pp. 122-142, 1998.
18. R. Wolski, N. Spring and J. Hayes. *Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid*, in Proceedings of 8th International Symposium on High Performance Distributed Computing (HPDC'99), August, 1999.
19. P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz and Y. Jim. *An Architecture for a Global Internet Host Distance Estimation Service*, in Proceedings of IEEE INFOCOM, 1999.
20. O. H. Ibarra and C. E. Kim. *Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors*, Journal of the ACM (JACM), vol. 24, n. 2, pp. 280–289, 1977.
21. D. Feitelson and L. Rudolph. *Metrics and Benchmarking for Parallel Job Scheduling*, in Proceedings of Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1459, pp. 1-24, 1998.
22. D. G. Feitelson. *Metric and workload effects on computer systems evaluation*, Computer, vol. 36(9), pp. 18-25, September, 2003.
23. V. Lo, J. Mache and K. Windisch. *A Comparative Study of Real Workload Traces and Synthetic Workload Models for Parallel Job Scheduling*, in Proceedings of Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1459, pp. 25-46, 1998.

24. R. Wolski, N. T. Spring and J. Hayes. *The network weather service: a distributed resource performance forecasting service for metacomputing*, Future Generation Computer Systems, vol. 15, numbers 5-6, pp. 757-768, 1999.
25. H. Casanova. *Simgrid: A Toolkit for the Simulation of Application Scheduling*, in Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, May, 2001.
26. J. L. Devore. *Probability and Statistics for Engineering and The Sciences*, vol. 1, 2000.
27. MyGrid Site. http://www.ourgrid.org/mygrid, 2004.
28. BLAST Webpage. http://www.ncbi.nlm.nih.giv/BLAST.
29. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao. *OceanStore: An Architecture for Global-Scale Persistent Storage*, in Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, November, 2000.

Performance Implications of Failures in Large-Scale Cluster Scheduling

Yanyong Zhang[1], Mark S. Squillante[2], Anand Sivasubramaniam[3], and Ramendra K. Sahoo[4]

[1] Department of Electrical and Computer Engineering, Rutgers University,
Piscataway, NJ 08854, USA
yyzhang@ece.rutgers.edu

[2] Mathematical Sciences Department, IBM T.J. Watson Research Center,
1101 Kitchawan Road, Yorktown Heights, NY 10598-0218 USA
mss@watson.ibm.com

[3] Department of Computer Science and Engineering, Pennsylvania State University,
316 Pond Laboratory, University Park, PA 16802-6106 USA
anand@cse.psu.edu

[4] Exploratory Server Systems Department, IBM T.J. Watson Research Center,
1101 Kitchawan Road, Yorktown Heights, NY 10598-0218 USA
rsahoo@us.ibm.com

Abstract. As we continue to evolve into large-scale parallel systems, many of them employing hundreds of computing engines to take on mission-critical roles, it is crucial to design those systems anticipating and accommodating the occurrence of failures. Failures become a commonplace feature of such large-scale systems, and one cannot continue to treat them as an exception. Despite the current and increasing importance of failures in these systems, our understanding of the performance impact of these critical issues on parallel computing environments is extremely limited. In this paper we develop a general failure modeling framework based on recent results from large-scale clusters and then we exploit this framework to conduct a detailed performance analysis of the impact of failures on system performance for a wide range of scheduling policies. Our results demonstrate that such failures can have a significant impact on the mean job response time and mean job slowdown under existing scheduling policies that ignore failures. We therefore investigate different scheduling mechanisms and policies to address these performance issues. Our results show that periodic checkpointing of jobs seems to do little to ease this problem. On the other hand, we demonstrate that information about the spatial and temporal correlation of failure occurrences can be very useful in designing a scheduling (job allocation) strategy to enhance system performance, with the former providing the greatest benefits.

1 Introduction

Our growing reliance on computing and information processing services mandates not only deploying systems that can meet the performance demands imposed on such systems, but also those that are available when needed. Several technological factors are accentuating the problem of system failures, which are highly undesirable since these

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 233–252, 2005.

systems could be servicing the needs of hundreds of users. At the same time, solutions for this problem need to keep the high costs of system maintenance personnel in mind, which is growing to be a much more important factor in Total Cost of Ownership (TCO). A deep understanding of the occurrence of failures in real environments can be useful in several ways towards enhancing overall system availability. It can provide realistic data when evaluating proposed solutions, together with developing strategies for proactive prediction and remedies of faults ahead of their occurrence. Application demand for high performance is continuing to fuel research and development of large scale parallel systems. The need for processing larger datasets in existing applications, and the stringent demands of emerging applications necessitate parallelism in computational and storage devices for their deployment. The cost-effectiveness in using off-the-shelf hardware to put together clusters has contributed to a large extent in the widespread availability of parallelism, and its successful usage. At the same time, several important and challenging applications are driving the development of large scale parallel machines, such as IBM's BlueGene/L which is anticipated to have 65536 nodes.

As we continue to develop such large scale parallel systems, there are several important technological factors to keep in mind:

- Denser integration of semiconductor circuits, though preferable for performance, makes them more susceptible to strikes by alpha particles and cosmic rays [41]. At the same time, there is an increasing tendency to lower operating voltages in order to reduce power consumption. Such reduction in voltage levels can increase the likelihood of bit-flips when circuits are bombarded by cosmic rays and other particles, leading to transient errors. While memory structures are typically the target for protection against errors using informational redundancy, more recent studies [31] have pointed out that the error rates in combinational circuits are likely to surpass those of memory cells in the next decade.
- At the macro granularity, we have dense blade-systems being packed in a rack as a cluster. With a high load imposed on these dense systems – both on the CPUs and on the disks – heat dissipation becomes a very important concern, potentially leading to thermal instability that can cause system/node breakdowns [25,9].
- We find system software and applications becoming more complex. Such complexity makes them more prone to bugs and other software failures [35,23,38] (e.g. memory leaks, state corruption, etc.). These bugs/failures can cause system crashes, and it has even been suggested that one should perform pro-active shutdown/rejuvenation [39,38] to avoid catastrophic consequences.

All these factors point to the increasing occurrence of system failures in the future. Failures become more commonplace when we consider parallel systems with thousands of nodes. Rather than treat them as an exception, system design needs to recognize fault occurrence, and manage the resources across the parallel system effectively so as to hide their impact from the end users. One would ideally like to achieve the performance of a system without any failures. Even if this is difficult to attain, there should be at most a "graceful degradation" in performance under the presence of failures. Towards this goal, the present paper specifically targets the *management of CPU resources on a large scale parallel system using a general failure modeling framework that accurately* *represents the node failure characteristics reported in recent studies of extensive error logs collected from cluster systems over long periods.*

When nodes[5] fail, there are two important consequences on system performance:

- First, the process/task of the application running on this node dies, consequently loosing all its work since it began. Further, in a parallel application, tasks frequently communicate and consequently other tasks would also not be able to progress. In effect, this can cause restarting the entire application (either on the same nodes or on different nodes).
- Second, the unavailability of the failed node can cause longer queueing delays for waiting jobs.

In this paper, we focus mainly on the first issue. With transient hardware errors and software errors expected to be more prevalent than permanent failures, node reboots/restarts can fix many of these problems. The duration of unavailability would then be relatively low, given the long execution times of many of the parallel applications that we are targeting – those in the scientific domain at national laboratories and supercomputing centers. Note that the impact of node recovery time can become quite important for permanent failures, and we postpone such an investigation for future work.

There are several options for managing the nodes in a faulty environment. One could use an optimistic approach, and simply ignore the problem, assuming there would be no failures. When a node does fail, then the application (all its tasks) could be restarted as was just explained. However, as our results will show, such an approach can suffer significant performance loss compared to a system where there is no failure. At the other end of the spectrum, we could have a more pessimistic strategy, where application processes are periodically checkpointed so that when a fault occurs, the amount of work to be re-done is limited. In our results we will show that while this can be better than ignoring the problem, the overheads of checkpointing can limit its benefits.

In this paper, we investigate an alternative strategy whose main philosophy is that if we have a better idea of when and where failures occur, then one could use such information for better management of the CPUs:

- If we could predict the time for the failure, then we could checkpoint immediately before this point in time, so that we significantly limit the work lost while reducing the checkpoint overheads. However, it may be very difficult to predict the exact time for failures. If, on the other hand, *temporal prediction* of failures is possible with a coarser granularity (a window) [29], then checkpointing could be initiated only within those windows.
- If we could predict the nodes (*spatial prediction*) that fail, then we could either avoid scheduling jobs on those nodes as far as possible, or only checkpoint those nodes. The latter option may not be very fruitful since parallel applications typically require all tasks to make progress at around the same rate.

One could also use a combination of spatial and temporal prediction to specifically focus on the time and nodes where pro-active action needs to be taken to limit the work loss upon failure while limiting the overheads of checkpointing.

[5] Since we are mainly concerned with CPU management, we use the terms, *node* and *CPU*, interchangeably in the rest of the paper.

Investigation of these alternatives requires an understanding of the failure characteristics of real parallel systems executing parallel applications. Unfortunately, the research literature provides a wide variety of often conflicting results for different computing environments (hardware and software) and there seems to be a lack of consistent conclusions in previous computer failure studies. Moreover, only a few recent studies have even considered large-scale clusters and they have tended to focus on sequential commercial applications. The only exception that we are aware of is a recent study [28] of extensive error logs collected from a large-scale distributed system consisting of close to 400 machines over a period of close to 500 days, which includes some parallel applications. We therefore develop a general modeling framework that makes it possible to vary the properties of the failure patterns to span the wide range of failure characteristics found in the research literature. This framework is exploited to understand the impact of different failure characteristics on overall system performance and to propose scheduling strategies that can alleviate the performance impact of different failure attributes.

A detailed simulation study using this failure modeling framework and characterized parallel job workloads from a supercomputing center reveals that the failures do account for a significant drop in performance compared to a system without failures. As can be expected, an exact temporal prediction of node failures almost completely bridges this gap of performance loss due to failures. Our results also show that a significant portion of this gap can be bridged even if temporal prediction can be done at only a granularity of 2–4 hours. While the results from our statistical analysis demonstrate clear patterns that could be exploited to provide such coarse grain temporal prediction, the results of our simulation study further show that even greater performance benefits are possible by using the spatial (node) behavior of failures. Hence, our solution opts to exploit the statistical spatial properties of failures and does so by developing a scheduling strategy wherein nodes that have recently failed are given lower priority at being assigned a job compared to others. We demonstrate that this simple strategy suffices to extract most of the performance gap between a system with failures and one without, and does significantly better than blindly checkpointing at periodic intervals.

The rest of this paper is organized as follows. The next section provides a brief summary of work related to this study. Section 3 presents our evaluation methodology, including our system model, our failure modeling framework, and the performance metrics of interest. Simulation results of the impact of failures on system performance are provided in Section 4, followed by consideration of different failure-aware scheduling strategies in Section 5.

2 Related Work

Job scheduling plays a critical role in the performance of large scale parallel systems (e.g. refer to [8,43,44,10,12,16,18,32,33,34] and the references therein). At the same time, scheduling can be used to improve the fault-tolerance [1,27] of a system in three broad ways. First, a task can be replicated on multiple nodes so that even if a subset of these nodes fail, the execution of a task is not impacted. Studies that employ this technique ([30,17]) assume a probability for node failure to determine the number of nodes on which to replicate the task. Second, the system can checkpoint all the jobs periodically so that work loss is limited when a failure occurs, and there are several studies on tuning checkpoint parameters [21,4,22]. Third, the scheduler allocates spare nodes to a job so that it can quickly recover from potential failures [26]. With this approach there is a trade-off between using the extra node(s) to improve the response time versus time for recovery. To our knowledge, there has not been prior work in analyzing and possibly managing system resources based on node failures.

3 Evaluation Methodology

3.1 System Model

We simulate a 320-node cluster that runs parallel workloads. A parallel job consists of multiple tasks, and each task needs to run on a different node. After certain nodes are allocated to a job, they are dedicated to the job until it completes (i.e., no other jobs can run on the same nodes). Multiple parallel jobs can run side by side on different nodes at the same time.

After a job arrives, it will start execution if it is the first waiting job and the system has enough available nodes to accommodate it. Otherwise, it will be kept in the waiting queue. In this paper, all the waiting jobs are managed in the First-Come-First-Serve (FCFS) order. We also use backfilling in this exercise, which is a most commonly used scheduling technique [44] for parallel workloads. Backfilling allows a job that arrived later to start execution ahead of jobs that arrived earlier as long as its execution will not delay the start of those jobs. Estimated job execution times are required to implement backfilling.

Our experiments use a workload that is drawn from a characterization of a real supercomputing environment at Lawrence Livermore National Labs. Job arrival, execution time and size information of this environment have been traced and characterized to fit a mathematical model (Hyper-Erlang distribution of common order). The reader is referred to [11] for details on this work and the use of the model in different evaluation exercises [44]. The workload model provides (1) arrival time, (2) execution time, and (3) size (number of nodes that it needs) for each incoming job.

3.2 Failure Injection

A large number of studies have considered the characteristics of failures and their impact on performance across a wide variety of computer systems. Tang et al. [37,36] and Buckley et al. [5,6] have investigated error/failure logs collected from various VAX-cluster systems of different sizes. Lee et al. [19] and Lin et al. [20] analyzed the error trends for Tandem systems and DCE environments. Xu et al. [40] performed a study of error logs collected from a heterogeneous distributed system consisting of 503 PC servers. Heath et al. [13] considered failure data from three different clustered servers, ranging from 18 workstations to 89 workstations. Castillo et al. [7], Iyer et al. [15] and Meyer et al. [24] have explored the effects of workload on different types of computer system failures. Vaidyanathan et al. [38] demonstrated that software-related error conditions will accumulate over time which will eventually lead to crashes/failures. Sahoo et al [28] have investigated the error logs from a networked environment of close to 400 heterogeneous servers over a period of close to 500 days.

Many of these studies have identified statistical properties and proposed stochastic models to represent the failure characteristics of various computer systems. This includes the fitting of failure data to Weibull, lognormal and other specific distributions, each with different parameter settings, under the assumption of independent and identically distributed failures [20,19,13]. Other studies have demonstrated that the sequence of failures on some computer systems are correlated in various ways and that the failures tend to occur in bursts [37,36,40,28]. Semi-Markov processes also have been proposed to model the time-series of failure from certain systems [14,37,36].

Unfortunately, only a few of these previous studies have even considered clustered server environments and those that have tend to focus on commercial servers like web servers, file servers and database servers. We are not aware of any studies that investigate failures within the context of large-scale clusters executing parallel applications, and no failure logs collected from such parallel computing environments are available to us. Moreover, given the wide variety of often conflicting results and the lack of consistent conclusions in previous computer failure studies, we expect that parallel computing environments with different parallel application workloads, system software and system hardware will similarly exhibit a broad range of failure behaviors. It is therefore important to have a general modeling framework that makes it possible to vary the properties of the failure patterns used to investigate parallel scheduling issues. Hence, we develop such a failure modeling framework in this section which is then exploited in Sections 4 and 5 to understand the impact of different failure characteristics on overall system performance and to propose scheduling strategies that can alleviate the performance impact of different failure attributes.

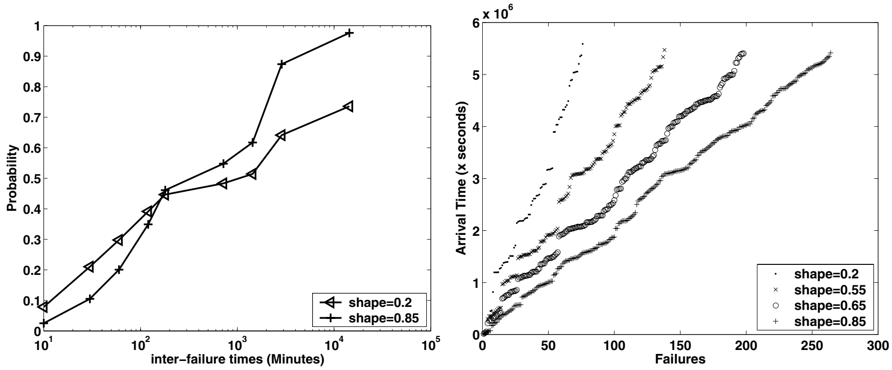
Our framework consists of models for each of the three primary dimensions of failure characteristics together with controls over each of these dimensions and their interactions. The first dimension concerns the times at which failures occur. This includes the marginal distribution for the time between failures as well as any correlation structure among the individual failures. The second dimension concerns the assignment of failures among the nodes comprising the system. This allows our framework to span the range from uniformly distributed node failures assumed in some previous failure studies to strong correlations between failures and nodes in order to yield the types of concentrations of failures on a subset of nodes as demonstrated in several recent failure studies of large-scale clusters. The third dimension concerns the down time of each failure. An overall control model is also used to directly capture any correlations or interactions among these three dimensions. Thus, there is no loss of generality in separating out the individual dimensions, while providing the ability to explicitly control and vary each aspect of the individual dimensions.

We now define the specific aspects of each dimension of our general failure modeling framework that are used in this study to generate synthetic failure workload traces each consisting of a number of failures. We use the job workload duration to determine the total number of failures (F) by making sure that the failures are spread throughout the entire span of parallel job executions.

Time of failures. Let t_i denote the time at which failure i occurs, $i = 1, \ldots, F$. Heath [13] has shown that the marginal distribution for the times between arrivals of failures in a cluster follow a Weibull distribution with shape parameters less than 1, the PDF of which can be described as $f(T) = \frac{\beta}{\eta}(\frac{T}{\eta})^{\beta-1}e^{-(\frac{T}{\eta})^{\beta}}$ where β denotes the shape parameter and η denotes the scale parameter. (Note that a Weibull distribution with shape parameter 1 corresponds to an exponential distribution.) In this paper, we use the family of Weibull distributions to generate the inter-arrival times for failures. Specifically, the parameters that are used are summarized in Table 3.2. The resulting failure arrival time distributions with different shape values are shown in Figures 3.2(a) and (b).

scale	shape	number of failures	failures/day
18000	0.2	78	1.2
	0.55	138	2.2
	0.65	198	3.2
	0.85	266	4.3

Table 1. The parameters that are used to generate the Weibull distributions



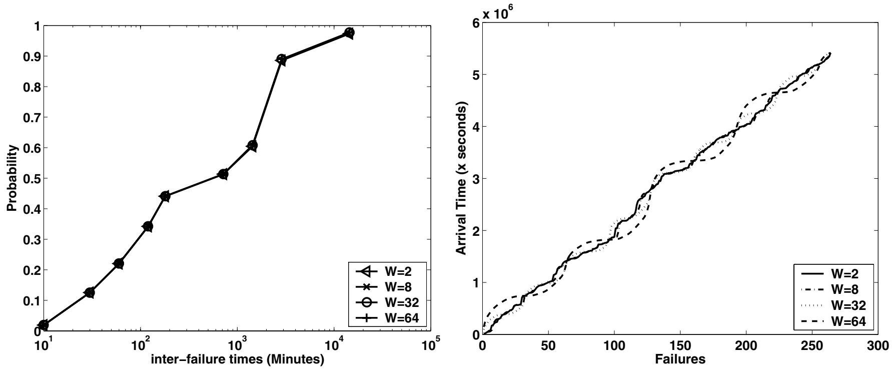
(a) Cumulated distribution function

(b) Time series

Fig. 1. The failure arrival time distribution with different shape parameters.

As noted above, the marginal distribution characterizes the statistical properties of the times between failures without any consideration of the correlation structure in the inter-failure process. Since it has been shown in [28] that there are strong temporal correlations between failure arrivals, we seek to include in our framework a general methodology for capturing different forms of temporal correlations within the inter-failure process while maintaining a perfectly consistent marginal distribution. This makes it possible for us to properly compare the impact of the inter-failure correlation structure on our results under a given marginal distribution. The following methodology is used to model the temporal correlations between failure arrival times:

- We generate a sequence of failure inter-arrival times which follow a specific Weibull distribution. Note that direct use of this time-series corresponding to assuming that the failures are independent and identically distributed.
- We break this sequence into segments, each of which contains W elements. Within each segment, we order the first $\frac{W}{2}$ elements in a descending manner, and order the remaining $\frac{W}{2}$ elements in an ascending manner. Note that the degree of correlation among the inter-failure times increases with increasing values of W.



(a) Cumulated distribution function

(b) Time series

Fig. 2. The failure arrival time distribution with different correlation parameters. $\beta = 0.85$

Once again, using this method, we can model temporal correlation between failures while maintaining a consistent marginal Weibull distribution. Figure 3.2 shows how the failure arrival time series vary with different W values. Note that $W = 2$ corresponds to the original time series and thus represents the case where there is no correlation. In this study, we shall vary the degree of correlation according to $W \in \{2, 8, 32, 64\}$.

Location of failures. Let n_i denote the location of failure i, $i = 1, \ldots, F$. Several previous failure analysis studies have shown that the spatial distribution of failures among the nodes is not uniform [37,13,28]. In fact, it has been shown in [28] that there are strong spatial correlations between failures and nodes where a small fraction of the nodes incur most of the failures. Possible reasons include: (1) some components (both hardware and software) are more vulnerable than others [37]; and (2) a component that just failed is more likely to fail again in the near future [13]. In order to capture this non-uniform behavior, we adopt the Zipf distribution to model failure locations in this study. We use α to denote the skewness parameter in the Zipf distribution. Specifically, we vary the skewness parameter of the distribution using the values 0.01, 0.5 and 0.99, where 0.01 corresponds to an environment where failures are close to being uniformly distributed among the nodes and 0.99 corresponds to a highly skewed distribution in which the majority of failures are concentrated on a relatively small number of nodes.

Down time of failures. Let r_i denote the down time of failure i, $i = 1, \ldots, F$. Failure down times can vary significantly due to the different ways of repairing the failures. If a simple reboot can re-start the system, then the down time can be relatively small (at most around minutes). However, if components need to be replaced, it could take hours or even days to recover. In this study, we use a constant value to model the down time. We vary this constant using down times of 2 minutes, 1 hour, and 4 hours.

3.3 Performance Metrics

In our simulations, we obtain the following statistics for each job: start time, work loss (the total loss of work due to failures), and completion time. These statistics are then used to calculate the following performance metrics:

- *Utilization:* The percentage of time that the system actually spends doing useful work.
- *Response Time:* The time difference between when a job completes and when it arrives to the system, averaged over all jobs.
- *Slowdown:* The ratio of the *response time* of a job to the time it requires on a dedicated system, averaged over all jobs. This metric provides an indication of the average slowdown that jobs experience when they execute in the presence of other jobs compared to their running in isolation.
- *Work Loss Ratio:* The ratio of the work loss as a result of failures to the execution time of a job, averaged over all jobs.

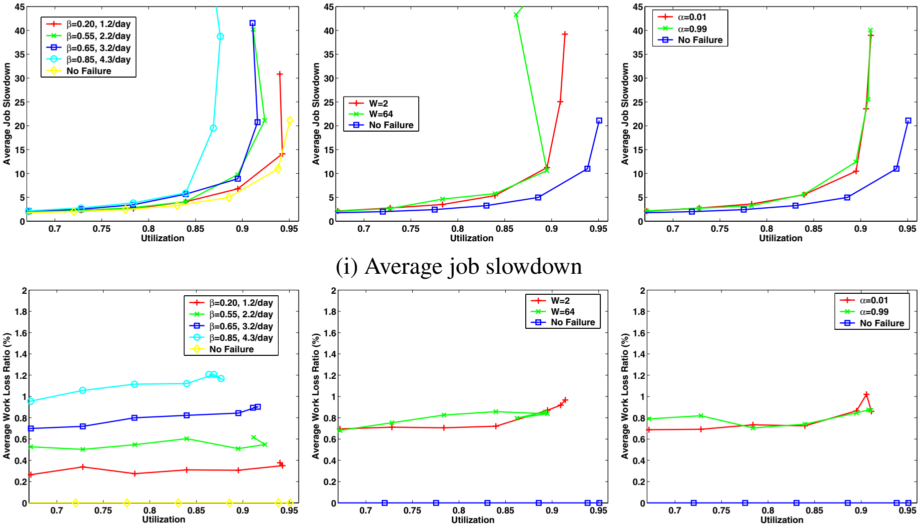
4 Impact of Failures on System Performance

We now move on to present results from detailed simulations of the system model running the parallel job workloads described in the Section 3.1 that are subjected to failures (Section 3.2).

4.1 Impact of Failure Arrival Statistics

As described early in this paper, the tasks of a parallel application often communicate with each other in order to make forward progress. Consequently, if any one task has to be restarted because of a failure, our model requires restarting all the tasks. Figures 3 illustrate the impact of the failure arrival characteristics on system performance. The graphs show the average job slowdown and average work loss ratio as a function of average job execution time. From Figure 3, we have the following observations:

- The impact of shape parameter (β). If we fix the scale parameter (η) of the Weibull distribution, varying the value of β ($\beta < 1$) will lead to different number of failures, further different inter-failure times. It thus has the most significant impact on the system performance among all the failure parameters:



(i) Average job slowdown

(ii) Average work loss ratio

(a) Impact of inter-failure time. β=*0.2, 0.55, 0.65, 0.85*; W=2; α=0.01; r =2 minutes.

(b) Impact of failure temporal correlation. β=0.85; W=*2, 64*; α=0.5; r=2 minutes.

(c) Impact of failure spatial distribution. β=0.85; W=2; α=*0.01, 0.99*; r=2 minutes.

Fig. 3. The impact of failures arrival characteristics.

- Failures can have a significant impact on the system performance (refer to Figures 3(a) (i) and (ii)). Even an average of 1.2 failures per day can increase the average job slowdown by up to 40%. An average of 4.3 failures per day will increase the job slowdown by up to 300%.

  If we look at the average work loss for different β values shown in Figure 3(a)(i), we observe an almost linear increase with β. Even a 0.2% work loss ratio suffices to cause a considerable performance degradation since these are relatively long running jobs.
- Failures have a higher impact on medium to high workloads. Let us look at the average work loss for $\beta = 0.85$. Under high workloads, the work loss is 40% higher than that under low workloads. This higher work loss ratio, together with the already high system utilization, lead to a degraded performance.

– The impact of temporal correlation parameter W (refer to Figures 3(b) (i) and (ii)). Compared to the impact of β , the impact of W is much less pronounced. We do observe that a longer-range correlation can slightly increase the average work loss and further job slowdown. A larger W can cause a more bursty failure arrivals, which can increase the chances of a job being hit by the failures.

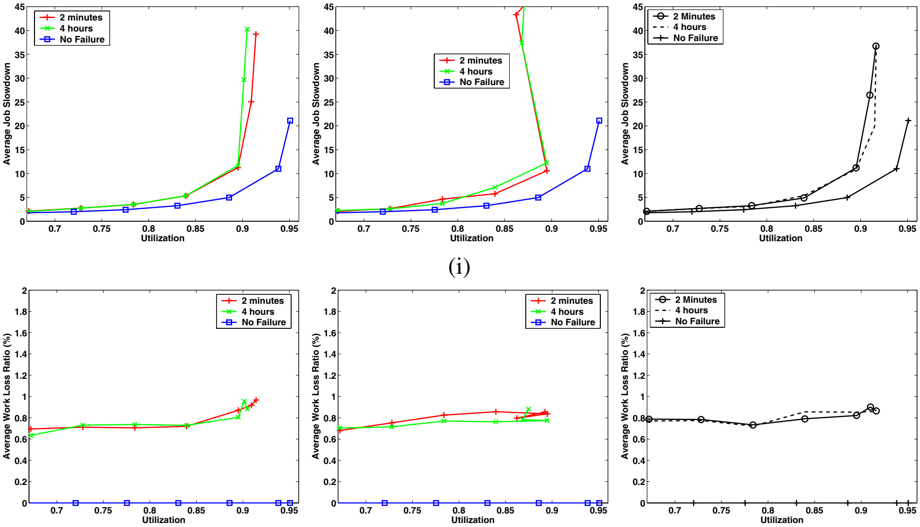
Although temporal correlation degree does not impact the average job slow down greatly, we feel that it may affect the performance of individual jobs because the same job may be hit multiple times at a higher temporal correlation degree. We are currently working on these results.

- The impact of spatial correlation parameter α (refer to Figures 3(c) (i) and (ii)). The impact of α is also less obvious compared to that of β. We observe a significantly higher work loss ratio under low loads for $\alpha = 0.99$, but this difference diminishes as the load increases. This observation may appear counter-intuitive. However, we would like to point out that this is just a simulation artifact. In our simulation, node 0 is always ranked the first, and will experience more failures than others with $\alpha = 0.99$. At the same time, when we try to schedule jobs onto the nodes, we always start from node 0 as well. Under low loads, the node utilization is low and node 0 will be available most of the time. As a result, many jobs will be affected by the failures on node 0, leading to a much higher work loss ratio.

  Further, we would like to point out that α impacts job slowdown most at medium loads. Under low loads, despite the work loss ratio difference, slowdown will not be affected due to the low load. Under high loads, different α values result in the same work loss ratio, thus leading to the same slowdown. On the other hand, the medium loads combine both the work loss ratio and reasonable loads, resulting in a more pronounced difference.

The results presented in this section are in agreement with our studies with a realistic failure trace [28].

4.2 The Impact of Failure Down Times



(i)

(ii)

(a) β=0.65; W=2; α=0.01; r=2 *minutes, 4 hours*.

(b) β=0.65; W=64; α=0.01; r=2 *minutes, 4 hours*.

(c) β=0.65; W=2; α=0.99; r=2 *minutes, 4 hours*.

Fig. 4. The impact of failure down times.

Earlier studies [] have shown that the failure down times have a great impact on the system performance for commercial servers such as file server, email server, web server, etc. However, we find that, for large-scale supercomputing clusters, an individual node's down time does *not* impact the performance significantly. As shown in Figures 4(a)-(c) (i)-(ii), the performance gap with different failure down times (varying from 2 minutes to 4 hours) is negligible. This is mainly due to the nature of the parallel workloads. These jobs cannot start execution if the system does not have enough available nodes. Therefore, in most of the times, the system will have a few free nodes while jobs are waiting to execute, even under high loads (due to system fragmentation).

In summary, failures have a great impact since the job that got hit will lose its work, but how long the failed node will remain down is not as important.

5 Failure-Aware Scheduling Strategies

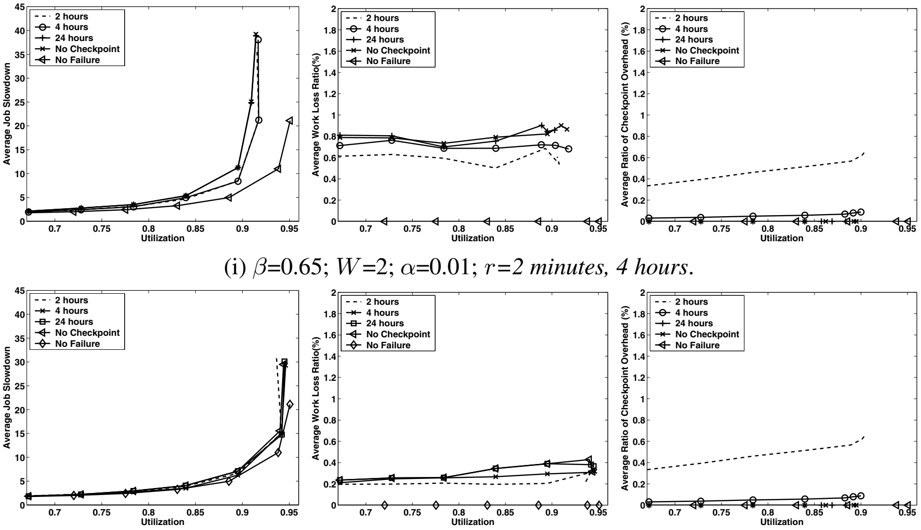
In this section, we examine different possibilities to alleviate the impact of failures, ranging from those that are oblivious to failure information (referred to as *failure-oblivious checkpointing* in section 5.1), to those that have significant knowledge about when and where failures occur (in section 5.2). Finally, we present a strategy that is based on a simple observation about the failure properties, and show that it can do a very good job of bridging this gap without requiring extensive failure prediction capabilities.

5.1 Failure-Oblivious Checkpointing

A straightforward approach to limit the impact of work loss upon failures is by checkpointing the application tasks periodically. Such an approach is oblivious to the occurrence of failures itself, and thus does not require any prediction about their occurrence. In this section, we evaluate the effectiveness of this simple approach using different intervals (2, 4, and 24 hours) for checkpointing. The scientific applications being targeted in this study are long running, and manipulate large datasets. It is not only the memory state of these applications that needs to be checkpointed but the network state of any messages that may be in transit as well. Consequently, checkpointing costs can be quite substantial, and can run into a few minutes especially with several processes swapping to a few I/O nodes [42]. We use a checkpoint cost of 5 minutes in this exercise, and the checkpoint intervals have been chosen in order to keep these overheads reasonable.

Figures 5 and 6 show the average slowdown, work loss ratio and checkpoint overhead of this approach with different failure distributions. From this set of results, we have the following observations:

- If the failures are i.i.d., oblivious checkpointing can only help the performance marginally compared to not taking any proactive actions (refer to Figures 5(a)(i-ii)). The relative performance gain due to checkpointing further decreases as the number of failures decreases (by comparing Figure 5(a)(ii) which has 1.2 failures per day to Figure 5(a)(i) which has 3.2 failures per day). With an average of 3.2 failures per day, a short checkpointing interval of 2/4 hours is better than a longer interval.



(i) β=0.65; W=2; α=0.01; r=*2 minutes, 4 hours.*

(ii) β=0.20; W=2; α=0.01; r=*2 minutes, 4 hours.*

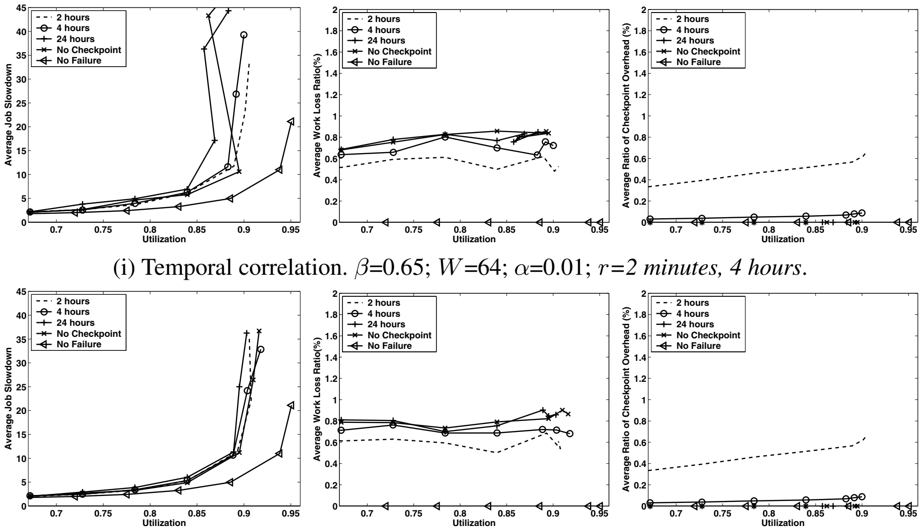
Fig. 5. failure-oblivious checkpointing for failures that are iid.

With 1.2 failures per day, we do not observe a noticeable difference between different checkpoint intervals. Although a small checkpoint interval can limit the work loss due to failures, this gain can be offset by the added checkpoint overheads. For example, if we checkpoint every 2 hours, the average work loss due to failures is less than 0.2%, but the resulted checkpoint overhead is above 0.4%, which de-emphasizes the benefits of checkpoints. At the same time, a larger checkpoint interval cannot effectively limit the work loss due to failures (Figure 5 (ii)).

– For failure traces that have temporal correlation, oblivious checkpointing does not help either (refer to Figures 6(a)(i)).
– For failure traces that have spatial correlation, e.g., following a Zipf distribution with α=0.99, the impact of oblivious checkpointing is again not obvious (refer to Figures 6(a)(ii)). Readers can look at the corresponding work loss ratios (refer to Figures 6(b)(ii)) and checkpointing overheads (refer to Figures 6(c)(ii)) to obtain further performance details.

5.2 Checkpointing Using Failure Prediction

Perfect Temporal/Spatial Prediction The previous results suggest that a checkpointing strategy oblivious to failure occurrence is not very rewarding. On the other hand, if one can predict when and where a failure occurs, then the *specific job* that would be affected can alone be checkpointed *exactly before the point of failure*. The benefits of such a perfect prediction strategy are quantified in Figure 7. The results show that the availability of exact failure information before-hand can help us schedule the checkpoint to almost completely eliminate the performance loss due to failures.



(i) Temporal correlation. β=0.65; W=64; α=0.01; r=*2 minutes, 4 hours.*

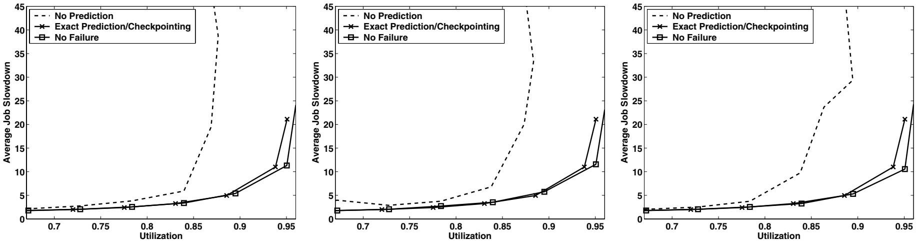
(ii) Spatial correlation. β=0.65; W=2; α=0.99; r=*2 minutes, 4 hours.*

Fig. 6. failure-oblivious checkpointing for failures that have temporal or spatial correlations.

Strategies Using Temporal Correlation While perfect prediction shows tremendous potential, it is almost impossible to be able to accurately predict when and where failures would occur. We next relax the predictability of when (temporal) and where (spatial) in the following way.

Even if one cannot predict exact times when failures occur, there could be underlying properties that make prediction at a coarser time granularity more feasible. For instance, studies have pointed out that the likelihood of failures increases with the load on the nodes [2]. At the same time, there have also been other studies [3] showing that load on clusters exhibit some amount of periodicity, e.g. higher in the day/evenings, and lower at nights. A recent study [28] has further showed that failures are correlated to the time of the day. Such insights suggest that perhaps a time-of-day based coarse-granularity prediction model may have some merit. Further, examination of our failure logs earlier in this paper shows certain patterns that could be exploited to provide such coarse grain temporal prediction. It is to be noted that our point here is not to say that such a model is feasible. Rather, we are merely trying to examine whether such a model (if developed) would be useful in alleviating the performance loss due to failures.

In our coarse-granularity temporal prediction model, we partition a day (24 hours) into n buckets (each bucket represents $\frac{24}{n}$ hours). We assume that we know exactly which bucket each failure belongs to, though we cannot predict the exact time within this bucket nor the specific node where the failure would occur. With this prediction model, at the beginning of each bucket, we know whether or not a failure will occur within this bucket. If we know a failure is about to occur, we can turn on checkpoints just for the duration of this bucket. At this time, how often we should checkpoint and



(a) failures that are i.i.d: β=0.85; W=2; α=0.01; r=2 minutes.

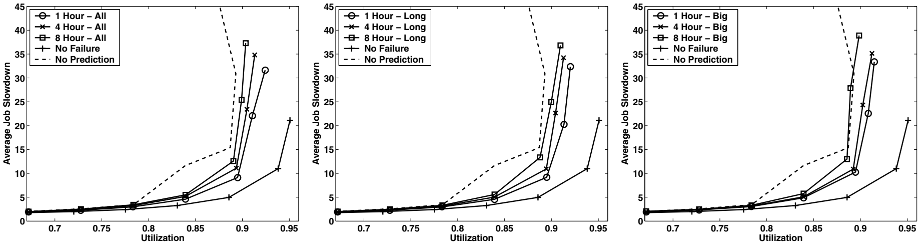
(b) failures with temporal correlation: β=0.85; W=64; α=0.5; r=2 minutes.

(c) failures with spatial correlation: β=0.85; W=2; α=0.99; r=2 minutes.

Fig. 7. Exact prediction and exact checkpointing.

which jobs to checkpoint become very important questions. In order to determine which jobs to checkpoint (*victims* of the failures), we examine the following three heuristics:

- *Checkpoint All.* We checkpoint all the jobs that are running within this bucket. Though this heuristic will not miss checkpointing the victim, it can incur higher checkpoint overheads by checkpointing more jobs than necessary.
- *Checkpoint Long.* In order to avoid excessive checkpoint overheads, this heuristic proposes to only checkpoint those jobs that have run for a certain duration (5 minutes in our experiments). The rationale is that even if we miss checkpointing the victim, it has not run long enough to incur significant work loss.
- *Checkpoint Big.* This heuristic assumes that big (in terms of the number of nodes that they use) jobs are more likely to be hit by failures because they occupy more nodes. As a result, in this heuristic we only checkpoint the k biggest jobs running within the bucket. Even though we have conducted experiments with different values of k, we only present results for $k = 1$ since the results are not very different.



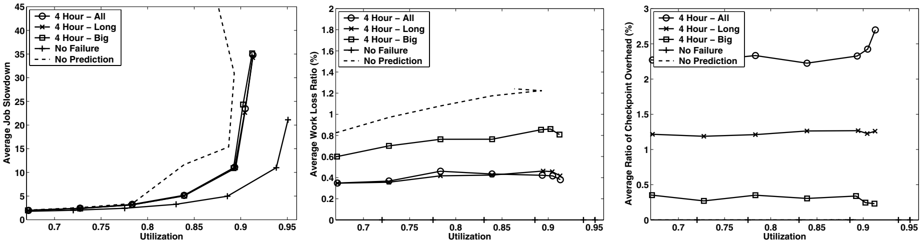
(a) Checkpoint All

(b) Checkpoint Long

(c) Checkpoint Big

Fig. 8. Checkpointing based on relaxed prediction model for i.i.d failures (β=0.85; W=2; α=0.01; r=2 minutes).

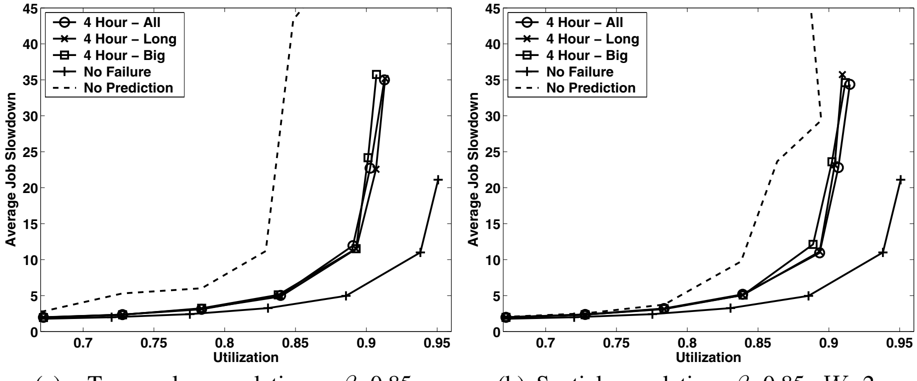
Figures 8 (a)-(c) present the performance of these heuristics for different bucket sizes (1, 4 and 8 hours). With smaller buckets, while the results are closer to perfect prediction, note that predictability at those finer granularities can become more difficult. Figure 8 shows that these heuristics can improve the performance noticeably even with bucket sizes of 8 hours.



(a) Average job slowdown (b) Average work loss ratio (c) Average checkpoint overhead

Fig. 9. Comparing three heuristics using four-hour buckets for idd failures (β=0.85; W=2; α=0.01; r=2 minutes).

Figure 9 compares these heuristics using four-hour buckets. It shows that these heuristics have comparable performances which improve job slowdown by up to 70%.



(a) Temporal correlation: β=0.85; W=64; α=0.01; r=2 minutes.

(b) Spatial correlation: β=0.85; W=2; α=0.99; r=2 minutes.

Fig. 10. Comparing three heuristics using four-hour buckets for failures that have either temporal correlation or spatial correlation.

Similar trends are observed in the other failure traces (Figures 10) while the failures that present temporal correlations benefits more from this approach since such failure traces have more bursty failure arrivals.

Strategies Using Spatial Correlation: Least Failure First (LFF) Despite the less stringent requirements from the prediction model examined in the previous section, it is quite possible as revealed in our analysis of the failure logs, that temporal prediction of failures may be very difficult to attain. At the same time, we note an important property of the failure logs - *nodes that have failed in the past are more likely to fail again* - and investigate the possibility of using this observation that can mitigate the performance loss due to failures.

We propose a scheduling strategy (rather a node assignment strategy for jobs), called *Least Failure First (LFF)*, to take advantage of this observation. The basic idea of this strategy is to give lower priority to nodes that have exhibited the most failures until that point when assigning them to jobs. Specifically, this objective is achieved by the following two optimizations:

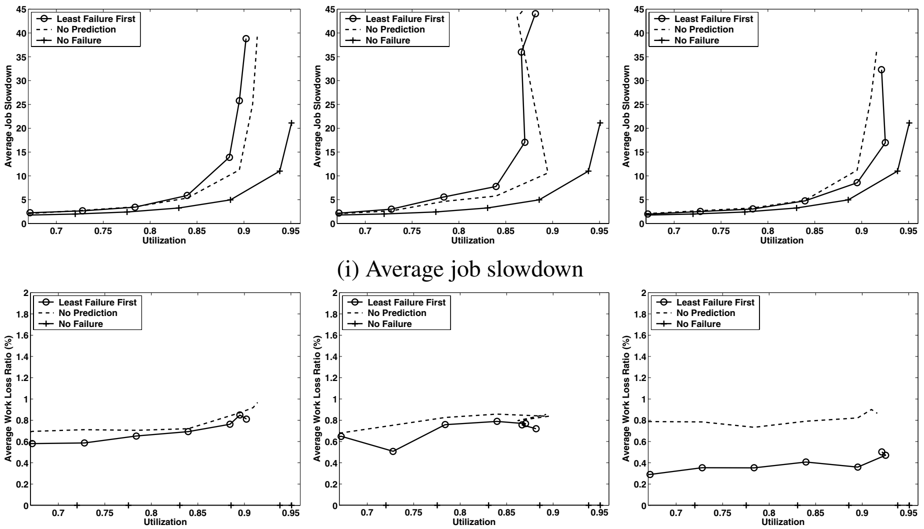
- Initial Assignment. We associate each node in the system with a failure count, which indicates the number of failures this node has experienced so far. A node that has a lower failure count is considered "safer" than another node with a higher failure count. We then sort all the nodes in ascending order of their failure counts. Amongst all the available nodes, we always allocate jobs to the safest ones (i.e., the ones with lowest failure counts).
- Migration. It is still possible that at some point a node that is not assigned to any job is more safe than another assigned to a job. To address this issue, when a job finishes, we need to migrate jobs running on less safe nodes (that started after this one) to more safe ones. As in [44], migration can be achieved by checkpointing on the original nodes and restarting on the destination nodes. We assume the checkpointing and restarting overheads to be 5 minutes. In order to avoid unnecessary overheads (thrashing), we migrate a job from node A to node B only when the difference between these two failure counts is above a certain threshold.

Figure 11 shows the performance results for LFF. As can be seen, for failure traces that have non-uniform spatial distribution (Figure 11(c)), LFF cuts down nearly 50% of the work loss incurred with failures by simply avoiding scheduling on failure-prone nodes as far as possible .

It is to be noted that LFF does not really require any prediction about failures. It is only exploiting a simple property of failures - a few nodes are likely to fail more often - which is not only a behavior in our failure logs but is also borne out by similar observations in other studies [37]. At the same time, it is easy to implement and can be easily integrated into existing parallel job scheduling strategies.

References

1. S. Albers and G. Schmidt. Scheduling with unexpected machine breakdowns. *Discrete Applied Mathematics*, 110(2-3):85–99, 2001.
2. S. M. andD. Andrews. On the reliability of the ibm mvs/xa operating system. In *IEEE Trans. Software Engineering*, volume October, 1987.
3. M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup E-Commerce Site. Technical Report Technical Report HPL-1999-62, HP, May 1999.
4. J. L. Bruno and E. G. Coffman. Optimal Fault-Tolerant Computing on Multiprocess Systems. *Acta Informatica*, 34:881–904, 1997.



(i) Average job slowdown

(ii) Average work loss ratio

(a) failures that are i.i.d.: β=0.65; W=2; α=0.01; r=2 minutes.

(b) failures with temporal correlation: β=0.65; W=64; α=0.01; r=2 minutes.

(c) failures with spatial correlation: β=0.65; W=2; α=0.99; r=2 minutes.

Fig. 11. Least failure first

5. M. F. Buckley and D. P. Siewiorek. Vax/vms event monitoring and analysis. In *FTCS-25, Computing Digest of Papers*, pages 414–423, June 1995.
6. M. F. Buckley and D. P. Siewiorek. Comparative analysis of event tupling schemes. In *FTCS-26, Computing Digest of Papers*, pages 294–303, June 1996.
7. X. Castillo and D. P. Siewiorek. A workload dependent software reliability prediction model. In *Proc. 12th. Intl. Symp. Fault-Tolerant Computing*, pages 279–286, June 1982.
8. D. Feitelson. A survey of scheduling in multiprogrammed parallel systems. *IBM Research Technical Report*, RC 19790, 1994.
9. K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 148–157, 2002.
10. H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. An evaluation of parallel job scheduling for asci blue-pacific. In *Proc. of SC'99. Portland OR, IBM Research Report RC 21559 , IBM TJ Watson Research Center*, November 1999.
11. H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette. Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proceedings of Supercomputing*, November 1999.
12. B. Gorda and R. Wolski. Time sharing massively parallel machines. In *Proc. of ICPP'95. Portland OR*, pages 214–217, August 1995.
13. T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *Proceedings of the ACM SIGMETRICS 2002 Conference on Measurement and Modeling of Computer Systems*, pages 217–227, 2002.
14. M. C. Hsueh, R. K. Iyer, and K. S. Trivedi. A measurement-based performability model for a multiprocessor system. In *Computer Performance and Reliability*, pages 337–352, 1987.

15. R. K. Iyer and D. J. Rossetti. Effect of system workload on operating system reliability: A study on ibm 3081. In *IEEE Trans. Software Engineering*, volume SE-11, pages 1438–1448, 1985.
16. B. Kalyanasundaram and K. R. Pruhs. Fault-tolerant scheduling. In *26th Annual ACM Symposium on Theory of Computing*, pages 115–124, 1994.
17. S. Kartik and C. S. R. Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. In *IEEE Transactions on Computer Systems*, volume 46, pages 719–724, 1997.
18. E. Krevat, J. G. Castanos, and J. E. Moreira. Job scheduling for the bluegene/l system. In *JSPP*, 2003.
19. I. Lee and R. K. Iyer. Analysis of software halts in tandem system. In *Proceedings 3rd Intl. Software Reliability Engineering*, pages 227–236, October 1992.
20. T. Y. Lin and D. P. Siewiorek. Error log analysis: Statistical modelling and heuristic trend analysis. *IEEE Trans. on Reliability*, 39(4):419–432, October 1990.
21. Y. Ling, J. Mi, and X. Lin. A Variational Calculus Approach to Optimal Checkpoint Placement. *IEEE Transactions on Computer Systems*, 50(7):699–708, July 2001.
22. G. M. Lohman and J. A. Muckstadt. Optimal Policy for Batch Operations: Backup, Checkpointing, Reorganization, and Updating. *ACM Transactions on Database Systems*, 2(3):209–222, 1977.
23. M. Lyu and V. Mendiratta. Software Fault Tolerance in a Clustered Architecture: Techniques and Reliability Modeling. In *Proceedings 1999 IEEE Aerospace Conference*, pages 141 –150, 1999.
24. J. Meyer and L. Wei. Analysis of workload influence on dependability. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 84–89, 1988.
25. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerabilityi Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.
26. J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, November 2001.
27. X. Qin, H. Jiang, and D. R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. *citeseer.nj.nec.com/qin02efficient.html*.
28. R. Sahoo, A. Sivasubramaniam, M. Squillante, and Y. Zhang. Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 389–398, 2004, to appear.
29. R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD*, pages 426–435, August 2003.
30. S. M. Shaltz, J. P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. In *IEEE Transactions on Computer Systems*, volume 41, pages 1156–1168, 1992.
31. P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, 2002.
32. M. S. Squillante. *Matrix-Analytic Methods in Stochastic Parallel-Server Scheduling Models.* Advances in Matrix-Analytic Methods for Stochastic Models, Notable Publications, 1998.
33. M. S. Squillante, F. Wang, and M. Papaefthymiou. *Stochastic Analysis of Gang Scheduling in Parallel and Distributed Systems.* Technical Report, IBM Research Division, 1996.

34. M. S. Squillante, Y. Zhang, A. Sivasubramanian, N. Gautam, J. E. Moreira, and H. Franke. Modeling and analysis of dynamic coscheduling in parallel and distributed environments. *Performance Evaluation Review*, 30(1):43–54, June 2002.
35. M. Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems. In *Proceedings of The 21st International Symposium on Fault Tolerant Computer Systems (FTCS)*, pages 2–9, 1991.
36. D. Tang and R. K. Iyer. Impact of correlated failures on dependability in a vaxcluster system. In *IFIP Working Conference on Dependable Computing for Critical Applications*, 1991.
37. D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modelling of a vaxcluster system. In *Proceedings 20th. Intl. Symposium on Fault-tolerant Computing*, pages 244–251, 1990.
38. K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. In *Proceedings of the ACM SIGMETRICS 2001 Conference on Measurement and Modeling of Computer Systems*, pages 62–71, June 2001.
39. K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. In *SIGMETRICS 2001*, pages 62–71, 2001.
40. J. Xu, Z. Kallbarczyk, and R. K. Iyer. Networked windows nt system field failure data analysis. *Technical Report CRHC 9808 University of Illinois at Urbana- Champaign*, 1999.
41. J. Zeigler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.
42. Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. The Impact of Migration on Parallel Job Scheduling for Distributed Systems . In *Proceedings of 6th International Euro-Par Conference Lecture Notes in Computer Science 1900*, pages 245–251, Aug/Sep 2000.
43. Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 133–142, May 2000.
44. Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang- scheduling backfilling and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.

Are User Runtime Estimates Inherently Inaccurate?

Cynthia Bailey Lee[1,2], Yael Schwartzman[1], Jennifer Hardy[1], and Allan Snavely[1,2]

[1] San Diego Supercomputer Center
University of California, San Diego
9500 Gilman Drive
La Jolla CA 92093-0505, USA
{cl,yaels,jhardy,allans}@sdsc.edu

[2] Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla CA 92093-0114, USA

Abstract. Computer system batch schedulers typically require information from the user upon job submission, including a runtime estimate. Inaccuracy of these runtime estimates, relative to the actual runtime of the job, has been well documented and is a perennial problem mentioned in the job scheduling literature. Typically users provide these estimates under circumstances where their job will be killed after the provided amount of time elapses. Also, users may be unaware of the potential benefits of providing accurate estimates, such as increased likelihood of backfilling. This study examines user behavior when the threat of job killing is removed, and when a tangible reward for accuracy is provided. We show that under these conditions, about half of users provide an improved estimate, but there is not a substantial improvement in the overall average accuracy.

1 Introduction

It is a well-documented fact that user-provided runtime estimates are inaccurate. Characterizations of this error in various real workload traces can be found in several classic and recent papers. Cirne and Berman [1] showed that in four different traces, 50 to 60% of jobs use less than 20% of their requested time. Ward, Mahood and West [7] report that jobs on a Cray T3E used on average only 29% of their requested time. Chiang, Arpaci-Dusseau and Vernon [4] studied the workload of a system where there is a 1-hour grace period before jobs are killed, but found that users still grossly overestimate their jobs' runtime, with 35% of jobs using less than 10% of their requested time (includes only jobs requesting more than one minute). Similar patterns are seen in other workload analyses [2,3,5].

Many factors contribute to the inaccuracy of user estimates. All workloads show a significant portion of jobs that crash immediately upon loading. This

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 253–263, 2005.

is likely more indicative of users' difficulties with configuring their job to run correctly, than difficulties with providing accurate runtime estimate [2]. However, a job's runtime may also vary from run to run due to load conditions on the system. In an extreme example, Jones and Nitzberg [9] found that on an Origin system where different jobs on the same node share memory resources, job runtime varied 30% on a lightly loaded system, to 300% on a heavily loaded system.

Mu'alem and Feitelson [2] note that because many systems kill jobs after the estimated time has elapsed, users may be influenced to "pad" their estimates, to avoid any possibility of having their job killed. Therefore, we believe that it is important to be precise about what users are typically asked to provide, which is a time after which they would be willing to have their jobs killed, and to distinguish this from the abstract notion of an estimate of their jobs' runtime. This leads us to prefer the term, *requested runtime* for the former, reserving the term *estimated runtime* for a best guess the user can make without any penalty (and possibly even with an incentive for accuracy).

This paper focuses on two specific causes of error in user provided runtime estimates:

1. Requested runtimes are used as a "kill time"—in other words, jobs are killed after the provided time has elapsed.
2. Users may be insufficiently motivated to provide accurate runtime estimates. Many users are likely unaware of the potential benefits of providing an accurate request, such as higher probability of receiving quicker turnaround (because of an increased likelihood of backfilling), or this motivation may not be strong enough to elicit maximum accuracy.

A significant unanswered question is, can and would users be accurate if these two barriers to accuracy were removed? This study addresses this question by asking users of the Blue Horizon system at the San Diego Supercomputer Center (SDSC) [8] for a non-kill-time estimate of their jobs' runtime, and offering rewards for accuracy.

The rest of the paper is organized as follows. In Section 2, we describe the experiment design. In Sections 3 and 4 we present the results of the accuracy of users' non-kill estimates, and their confidence in their estimates, respectively. Section 5 reviews related work on the impact of user inaccuracy on scheduler performance. Finally, Sections 6 and 7 present the conclusions and future work.

2 Survey Experiment Design

Users of the Blue Horizon system submit jobs by using the command *llsubmit*, passing as an argument the name of a file called the job *script*. The script contains vital job information such as the location and name of the *executable*, the number of *nodes* and *processors* required, and a *requested runtime*. An analysis of the requested runtimes from the period prior to the experiment shows that the error has a similar distribution to that observed in other workloads. Specifically, a majority of jobs use less than 20% of their requested time.

During the survey period, users were prompted for a non-kill-time estimate of their jobs' runtime by the llsubmit program, randomly one of every five times they run. We asked at the moment of job submission, as this will be the most timely and realistic moment to measure the user's forecasting abilities. The traditional requested runtime is not modified in the job script, we merely reflect that value back to the user and ask them to reconsider it, with the assurance that their response in no way affects this job.

Users were notified of the study, by email and newsletter, a week prior to the start of the survey period. The notification included information about prizes to reward the most accurate users (with consideration given also to frequency of participation). One MP3 player (64MB Nomad, approximate value: 80 USD) and 18 USB pen drives (64MB, approximate value: 20 USD) were awarded. The prizes were intended to provide a tangible motivation for accuracy and thus to elicit the most accurate estimates users are capable of providing.

The text of the survey is as follows. First, the user is reminded of the requested runtime (kill time) provided in their script. The user is then queried for a better estimate. Finally, the user is asked to rate their confidence in the new estimate they provided, on a scale from 0 to 5 (5 being the highest). This question was designed to test if users could self-identify as good or poor estimators. The survey does not provide default values. The text of the survey (with sample responses) is shown below:

> You have been randomly selected to participate in a two-question survey about job scheduling (as posted on www.npaci.edu/News). Your participation is greatly appreciated. If you do not wish to participate again, type NEVER at the prompt and you will be added to a do-not-disturb list.
> In the submission script for this job you requested a 01:00:00 wall-clock limit. We understand this may be an overestimate of the wall clock time you expect the job to take. To the best of your ability, please provide a guess as to how long you think your job will actually run.
> NOTE: Your response to this survey will in no way affect your job's scheduling or execution on Blue Horizon.
> Your guess (HH:MM:SS)? ***00:10:00***
> Please rate(0-5) your confidence in your guess: (0 = no confidence, 5= most confident): ***3***
> Thank you for your participation.Your Blue Horizon job will now be submitted as usual.

3 User Accuracy

Over the 9-week period of the survey there were 10,397 job submissions. However, only 2,870 of those ran until completion (many jobs are withdrawn while still waiting in the queue or cancelled while running). Since approximately one out of every five job submissions were requested to complete the survey, 2,478 of the jobs that ran until completion were not surveyed. Furthermore, we did not survey automated submissions (81) or jobs that requested less than 20 minutes of runtime (172). We had 21 timeouts, where there was no response for more than 90 seconds; and 59 jobs that were submitted by the 11 people that decided not to take part in the survey.

Of the 143 jobs that ran until completion and completed the survey, 20 had equal or slightly higher runtimes than their requested runtime. This situation could either indicate that the user was very accurate or, more likely, that the job got killed once it reached its requested runtime due to scheduling policies. We decided to discard these survey entries since it was not possible to determine whether the job was completed or killed from the information we collected. In 16 of the responses, the estimate given in response to the survey was *higher* than the requested runtime in the script. Taken at face value, this means that upon further reflection, the user thought the job would need *more* time than they had requested for it, in which case the job is certain to be killed before completing. Some of these responses appeared to be garbage (e.g. "99:99:99") from users who perhaps did not really want to participate in the study or just hoped a random response had some chance of winning a prize. In our analysis, all of these higher responses were discarded, as well as a survey response indicating an expected runtime of 0 seconds.

Fifty-six of the survey response runtime estimates were the same as the requested runtime in the script. Of the 51 responses where users provided a tighter estimate, users cut substantially (an average of 35%) from the requested time (see Figure 1). The average *inaccuracy* in this group decreased from 68% to 60%. By inaccuracy we mean the percent of requested (or estimated) time that was unused or exceeded (in the case of estimates it is possible, though unusual, in this survey, to underestimate the runtime), as given in the following formula:

$$Inaccuracy = abs(base - actualruntime)/base \tag{1}$$

Where *base* is either the requested runtime or the estimated time from the survey. So for example, an estimated time inaccuracy of 68% means either that 32% of the estimated runtime was used, or that 168% of the estimated time was used. A requested time inaccuracy of 68% means that 32% of the requested runtime was used.

Because not all users tightened their estimates, overall the inaccuracy decreased from an average of 61% to 57%. Those users who did not tighten their estimate were notably less inaccurate than those who did revise it; their initial inaccuracy was 55%. To fully understand our two metrics it is helpful to understand an example. A not atypical user requested their job to run for 120 minutes, revised (estimated) the runtime to 60 minutes in response to the survey, and the job actually ran for 50 seconds (!). In this example the user tightened their estimate by 50%. But the inaccuracy of the request is 99%, and the inaccuracy of the estimate is improved only 1% down to 98%. Intuitively, many users are

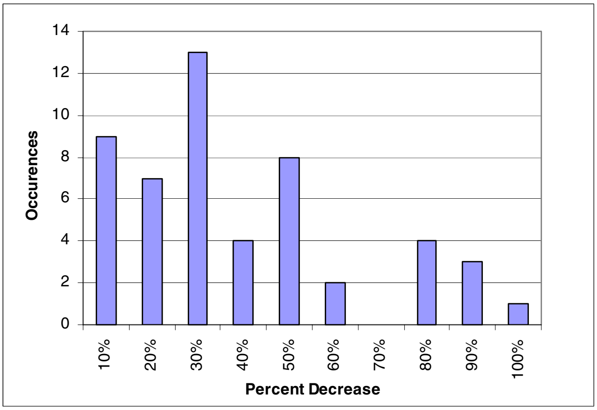


Fig. 1. Histogram of percent decrease from the requested time to the estimate provided in response to the survey (includes only responses that were different from the requested time). Categories represent a number of respondents up to the label, e.g. 20% represents 7 responses that were between 10% (exclusive) and 20% (inclusive) decreased from the requested time in the script.

substantially improving extreme overestimates, still without making the bounds very tight.

In Figure 3 we show the comparison between the requested runtime in the script, and the actual runtime for the survey entries. The results are similar to those seen in Figure 2, where we see the same information but for the entire workload during the survey period, suggesting that the survey entries collected are a representative population sample. The results are also similar to those seen in the literature, in particular see [2]. Figure 4 shows the results if the estimate provided in the survey is used, instead of the requested runtime in the script. Note that no job's actual runtime can exceed the requested runtime, but because the survey responses were unconstrained in terms of being a kill time, the actual runtime can be either more or less than this estimate. The great majority of survey responses were still overestimates of the actual runtime. We cannot be sure why this is so, but it may be a lingering tendency due to users having been conditioned to overestimate by system kill-time policies.

Some degree of improvement can be seen in the pattern of error, for example a cluster of points on the right to right-bottom area of Figure 3 is largely dissipated in Figure 4. We can see that users still tend to round their times to 12, 24 and 36 hours in the survey, but not quite as heavily.

4 User Confidence

It is likely that even the most motivated of users will not always be able to provide an accurate runtime request or estimate. But it may be useful if users can at

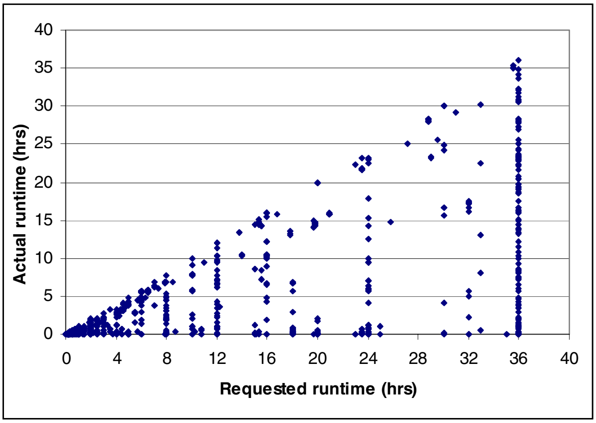


Fig. 2. Comparison of actual runtime and requested runtime for all jobs on Blue Horizon during the survey period (Figure 3 shows the same data but only for jobs in the survey.) Note that some data points are overlapping.

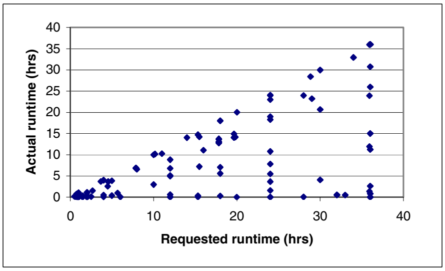


Fig. 3. Correlation between requested runtime and actual runtime. Note that some data points are overlapping.

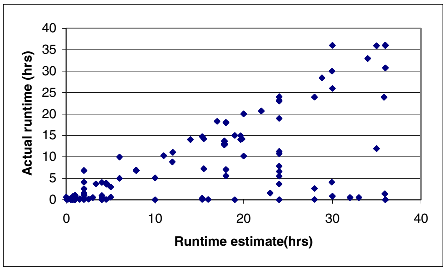


Fig. 4. Correlation between users' survey runtime estimates and actual job duration. Note that some data points are overlapping.

least self-identify when they are unsure of their forecast. In our study, we asked users to rate their confidence in the runtime estimate they provided in response to the survey on a scale from 0 (least confident) to 5 (most confident). Figure 5, below, shows the distribution of responses. In a majority (70%) of the responses, users rated themselves as most confident or very confident (5 or 4 rating) in the estimate. This is in spite of the fact that, overall, the accuracy of the requested runtimes and runtime estimates was poor (though typical, as observed in other workloads). It may be that users did not significantly adjust their forecasts of their jobs' runtime to account for possible crashes and other problems [11,12].

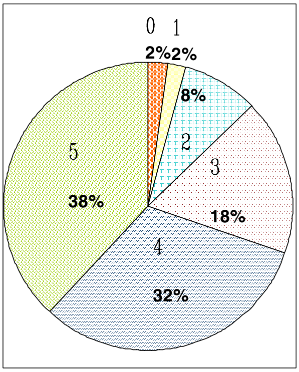


Fig. 5. Distribution of user accuracy self-assessments (i.e. confidence).

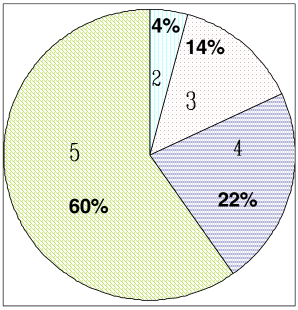


Fig. 6. Distribution of user accuracy self-assessments in users who did not change their requested runtime in response to the survey.

The responses can be divided into those users who provided a revised estimate in response to the survey, and those who reiterated the requested runtime in their script. In Figure 6, we see that in 60% of responses that were the same as the

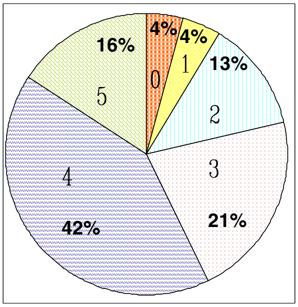


Fig. 7. Distribution of user accuracy self-assessments in users did change their requested runtime in response to the survey.

requested runtime, users rated themselves as most confident (5), with another 22% rated very confident (4). No users in this group rated themselves as low or very low confidence (1 or 0). In contrast, of those responses that were a different estimate (Figure 7), most users rated themselves somewhere in the middle (4 or 3).

Psychologists Kruger and Dunning [10] have observed that people who are most ignorant of a subject area are *more* likely to overestimate their own abilities than those who are knowledgeable. We wondered if our results were an instance of the same phenomenon. In other words, perhaps users reiterated the same requested runtime out of ignorance, and were then very self-confident, as predicted by Kruger and Dunning. However, it appears that users who did not change in response to the survey, and had high confidence, did on average have more accurate estimates (as seen in Figure 8). For the unchanged responses, there is a clear pattern of decreasing average inaccuracy as the confidence increases. For both changed and unchanged responses, the highest average inaccuracy is seen in the category of lowest confidence, and the lowest average inaccuracy is seen in the category of highest confidence.

5 Impact of User Inaccuracy on Scheduler Performance

One might ask what impact user inaccuracy has on scheduler performance—why worry if user estimates are inaccurate? Indeed, some studies have shown that if workloads are modified by setting the requested times to *R * actual runtime*, average slowdown for the EASY and conservative backfilling algorithms actually *improves* when R = 2 or R = 4, compared to R = 1 (total accuracy) [3,14]. Similar results have been shown when R is a random number with uniform distribution between 1 and 2, or between 1 and 4, etc. [2,14].

But simply taking the accurate time and multiplying it by a factor does not mimic the "full badness of real user estimates" [2]. Still, in other studies where real user-provided times were used [2,3], some scheduling algorithms did perform

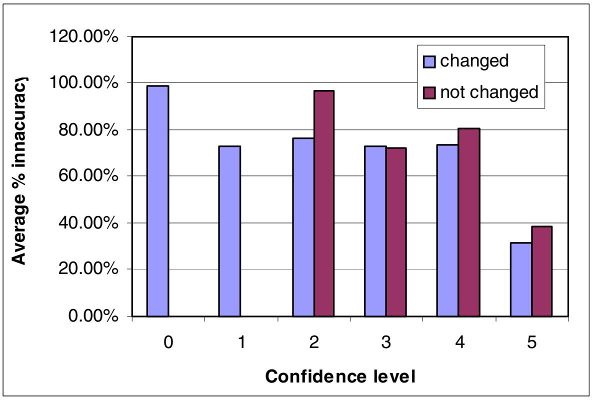


Fig. 8. Average percent inaccuracy of user survey responses, separated into those responses that were changed and not changed with respect to the requested runtime in the script.

equivalently or slightly better, compared to the same workload with completely accurate times.

However, some other algorithms experience significant performance degradation as a result of user inaccuracy [4,5]. Also, even for an algorithm such as conservative backfilling, which shows a slightly improved average slowdown with inaccurate estimates, it is at the cost of less useful wait time guarantees at the time of job submittal, and causing an increased tendency to favor small jobs over large jobs (which may or may not be desirable) [4,14].

Asking the user for a more accurate time, as we have done in this study, is not the only approach to mitigating inaccuracy. One suggestion is to weed out some inaccurate jobs through speculative runs, to detect jobs that immediately crash [5,13]. Or, the system could generate its own estimates for jobs with a regular loop structure, via extrapolation from timings of the first few iterations [4]. Another proposal [6] is to charge users for the entire time they requested, not only the time they actually used. This idea, meant to discourage users from "padding" their estimates, may seem unfair to users and thus be unattractive to implement.

6 Conclusion

Mu'alem and Feitelson [2] documented and modeled discrepancies between user provided time limits and actual execution time on several HPC systems, including Blue Horizon. We analyze a more recent trace, with similar results. We then ask the question, are users capable of providing more accurate runtime estimates?

To answer this question, we surveyed users upon job submittal, asking them to provide the best estimate they can of their job's runtime, with the assurance that their job will not be killed after that amount of time has elapsed.

We have demonstrated that some users will provide a substantially revised estimate but that, on average, the accuracy of their new estimates was only slightly better than their original requested runtime. On the other hand, many users were able to correctly identify themselves as more or less accurate in their estimating than other users.

An inherent weakness in our survey experiment design is that we can never be sure if users are motivated "enough" to provide the best estimates they can. In other words, it is not clear if a bigger or better prize offering would have elicited better estimates from users. However, that most users were very confident in their estimates indicates that perhaps many were in fact exhibiting their "best" in our study.

7 Future Work

In future work, we will measure the impact that better user estimates have on supercomputer performance. We intend to carry out additional surveys to find a scheduling system that understands user behavior and uses this knowledge as a key scheduling factor. The survey will possibly include educating feedback in order to measure users' improvement over the lifetime of the experiment. In addition, we wish to help users improve their estimates. One possible way to accomplish this is by educating them about the potential benefits of providing accurate estimates, other than the prizes offered specifically for this study. For example, our prototype web-based tool Blue View visually presents the Blue Horizon scheduler's current plans for running and queued jobs. We hope this tool will give incentive to the users to give shorter time estimates with the promise that their jobs will fit the backfill slots shown in it. Furthermore, this tool will also give the user the opportunity to mold their job according to what is readily available.

8 Acknowledgements

This work was sponsored in part by NSF cooperative agreement STI-00230925 "Data Intensive Grid Benchmarks," DOE SciDAC award titled "Performance Evaluation Research Center," DoD HPCMP award titled "HPC Performance Evaluation," and NSF cooperative agreement ACI-9619020 through computing resources provided by the National Partnership for Advanced Computational Infrastructure at the San Diego Supercomputer Center. We gratefully acknowledge all the users of Blue Horizon who participated in the survey, and the staff who supported the administration of the survey. We wish to thank the workshop referees for their thorough and very useful suggestions. Finally, Dan Tsafrir provided helpful correspondence.

References

1. W. Cirne and F. Berman. “A comprehensive model of the supercomputer workload,” Proceedings of IEEE 4th Annual Workshop on Job Scheduling Strategies for Parallel Processing. Cambridge, MA. 2001.
2. 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 and Distributed Systems, 12(6). June 2001.
3. S. Srinivasan, R. Kettimuthu, V. Subramani and P. Sadayappan. “Characterization of Backfilling Strategies for Parallel Job Scheduling,” Proceedings of 2002 International Workshops on Parallel Processing, August 2002.
4. S-H. Chiang, A. Arpaci-Dusseau and M. K. Vernon. “The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance,” Proceedings of the 4th Workshop on Workload Characterization, D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, eds. July 2002.
5. B. G. Lawson, and E. Smirni. “Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems,” Proceedings of 8th Workshop on Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, eds. July 2002.
6. I. Stoica, H. Abdel-Wahab and A. Pothen. “A Microeconomic Scheduler for Parallel Computers,” IPPS’95 Workshop: Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph, eds. April 1995.
7. W. A. Ward Jr., C. L. Mahood and J. E. West. “Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy,” Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph, eds. July 2002.
8. Blue Horizon, National Partner for Advanced Computing Infastructure (NPACI). www.npaci.edu/Horizon/
9. J. P. Jones and B. Nitzberg. “Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization,” Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph, eds. April 1999.
10. J. Kruger and D. Dunning. “Unskilled and unaware of it: How difficulties in recognizing one’s own incompetence lead to inflated self-assessments,” Journal of Personality and Social Psychology, 77(6). December 1999.
11. D. Lovallo and D. Kahneman. “Delusions of Success,” Harvard Business Review, 81(7). July 2003.
12. R. Buehler. “Planning, personality, and prediction: The role of future focus in optimistic time predictions,” Organizational Behavior and Human Decision Processes, 92(1/2). September-November 2003.
13. D. Perkovic and P. Keleher. “Randomization, Speculation, and Adaptation in Batch Schedulers,” Proceedings of Supercomputing 2000. November 2000.
14. D. Zotkin and P. Keleher. “Sloppiness as a Virtue: Job-Length Estimation and Performance in Backfilling Schedulers,” Proceedings of the 8th HPDC, Redondo Beach, CA. August 1999.

Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW*

Gaurav D. Ghare and Scott T. Leutenegger

Department of Computer Science
University of Denver
Denver, CO 80208-0189
{gghare,leut}@cs.du.edu

Abstract. Idle computation cycles of a shared network of workstations are increasingly being used to run batch parallel programs. For one common paradigm, the batch program task running on an idle workstation is preempted when the owner reclaims the workstation. This owner interference has a considerable impact on the execution time of a batch program, especially in the case of large parallel programs. Replication of batch program tasks has been used to reduce the impact of owner interference. We show analytically that replication can significantly improve parallel program speedup. Perhaps surprisingly, replication can also improve efficiency for certain workloads. We present analysis to quantify the amount of speedup and efficiency improvement. Furthermore, we provide analysis to help determine whether extra available workstations should be used for increasing job parallelism or for task replication.

1 Introduction

Networks of workstations (NOWs) have been used to run parallel programs for some time [1,2,3,4,5,6,7,8,9,10]. The NOW may be dedicated as in the case of Beowulf [11] or shared as in the case of Condor [12,13]. When the NOW is shared with workstation owner processes, it is referred to as a shared network of workstations (SNOW). The speedup of parallel programs running on a SNOW can be greatly affected by workstation owner interference. In this paper we consider a SNOW where parallel jobs are run in an opportunistic fashion as in Condor. In such an environment, workstation owner processes have preemptive priority over batched parallel programs. Workstation reclamations by owner processes stop execution of the batch job task and hence may significantly impact the parallel job response time.

One approach to prevent a workstation reclamation from impacting parallel job performance is to replicate some or all of the parallel tasks [5,6]. When replication is used, parallel job performance is not affected by reclamations as long as, at least one replica of each task completes without interference.

In [5] the author compares competition protocols and migration protocols for sequential and distributed programs on variable-speed processors. A SNOW is

* This work was supported by the NSF grant ACI-9733658.

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 264–287, 2005.

treated as a collection of variable-speed processors where background programs have lower priority than foreground owner programs. Competition protocols use replication to reduce the impact of owner interference. Simulation results of the performance of competition for distributed programs are presented. However there is no mathematical analysis of the performance of competition protocols for distributed programs. Competition policy issues such as, how to allocate workstations for replicating tasks of a program, are not studied.

The authors in [6] demonstrate that owner interference considerably impacts the performance of a parallel program running in a SNOW environment. Futhermore, they demonstrate that using task replication can significantly improve job response time. The study only considers the loss of up to one workstation in a batch of tasks. Tasks are assumed to be of different sizes allowing for a mix of short and long service demand tasks. Performance of replication for programs with tight coscheduling requirements is not studied. They consider one no-replication and two replication strategies to alleviate the negative impact of owner interference: no-replication (NR) adds the extra workstations to the general pool, single replication (SR) replicates the largest task in a batch using one extra workstation and uses other extra workstations in the general pool, and complete replication (CR) replicates the K largest tasks using K extra workstations. They show with experimental workloads that CR performs better than SR. Our work differs in that we provide analysis and proofs instead of simulation, we consider synchronized workloads, we study various tradeoffs in how best to allocate K extra workstations, and we consider the tradeoff between using extra processors for parallelism versus replication.

Rosenberg [10] develops a model for devising a schedule that maximizes the amount of work accomplished from an embarassingly parallel workload. Owner interference is considered as an adversial process between the owner and the user running background programs. The instantaneous probability of workstation reclamation is assumed to be known. In [2] the authors consider sharing a bag of identically complex tasks in a heterogeneous network of workstations (HNOW). The problem considered is accomplishing as much work as possible on the HNOW, during a prespecified fixed period of time. Neither of [2,10] consider the effect of replication for reducing the impact of owner interference on parallel program performance.

In this paper we show that replication can result in significant speedup improvements and we analytically quantify parallel task replication benefits for two workload models: tightly-coupled barrier synchronized and loosely-coupled barrier synchronized. We allow for multiple workstation reclamations during the execution of a batch of tasks. We assume knowledge of the probability that a workstation may be reclaimed before a task completes, but do not assume knowledge of the instantaneous probability of workstation reclamation as in [2,10]. We analytically study how to distribute extra workstations not only among tasks of a program but also between two programs.

In a dedicated parallel processing machine, increasing a parallel program's workstation allocation beyond the program's maximum parallelism will reduce workstation efficiency without any speedup improvement [14]. We show that this assumption does not hold in the case of a SNOW. On the contrary, and somewhat surprisingly, additional workstation allocation in the form of task replication can result in improvements in efficiency as well as speedup.

The rest of the paper is organized as follows. In Sect. 2 we describe our machine and workload models. In Sect. 3 we present replication analysis and results for both workload models considered. In Sect. 4 we present analysis and explore the trade-off between using additional workstations for replication versus for increasing job parallelism. In Sect. 5 we prove lemmas that are used in theorems. We state our conclusions in Sect. 6.

2 Machine and Workload Model

In this section we explain our overall system model, specific parallel program models, and performance metrics.

2.1 System and Workstation Model

We assume a SNOW system of N homogeneous workstations. As in the Condor system [12,13], we assume workstations execute both owner jobs and batched jobs. We assume batch *jobs* are parallel programs, that are decomposed into *tasks*, and then the tasks are run in parallel on idle workstations. For simplicity we assume that all the tasks are of an equal length, that owner jobs are sequential processes local to the workstation, and migration and checkpointing overheads are absorbed in the workload demand. Similarly, homogeneity of workstations is assumed for simplicity.

We assume workstation owner processes have preemptive priority over batch tasks. As soon as an owner job begins execution after an idle period, the workstation is reclaimed, and the running batch task, if any, is preempted immediately. All task work completed since the last checkpoint is lost.

We assume workstation reclamations are independently and identically distributed, and that the probability of a workstation being reclaimed during the length of a task unit is p_r, $0 < p_r < 1$. When a task is preempted the task is allocated to another idle workstation and is restarted on that workstation. The partial work completed by the task is lost and the task is restarted at the beginning on the newly allocated workstation. The task only restarts after the current length of a task unit is finished. The time when one set of job tasks is deemed to be completed and the next set of tasks started, may represent the point in the program where barrier synchronization takes place or when a job checkpoints.

We assume we have enough workstations at hand so when one (or more) of the workstations is reclaimed we have another one (or more) upon which to run the task(s). Thus a job always has a fixed number of workstations allocated to it. We make this assumption to simplify analysis, but it should have no effect on the qualitative results gleaned from this study.

When we say a task is replicated d times, we mean there are a total of $d > 1$ identical tasks scheduled, and the task completes when one (or more) of them finishes. However if $d = 1$, we say the task has not been replicated.

2.2 Parallel Job Models

The batch workload consists of parallel programs, or jobs, each with N tasks, where $N \geq 1$. A job completes when all of the N tasks have completed. For simplicity, we assume all tasks to be of unit length. Different inter-task synchronization constraints impact the job performance. We have recently begun exploration of the effectiveness of replication for a Master-Worker workload [15], but for now consider the following workload models:

1. **Tightly-Coupled Barrier**
   We assume a job is composed of N equal sized parallel tasks. A series of B barriers must be completed. A barrier is reached when each of the N tasks completes. All tasks must be simultaneously scheduled for forward progress. If any of the N tasks is preempted, all N tasks must be started over. Thus, a barrier is only achieved after a task-demand sized time interval where none of the N workstations is reclaimed.
   We assume that tasks always start at fixed intervals of time. When a workstation is reclaimed, the task must be moved and hence the completed work of *all* job tasks (of the current barrier) is lost and all these tasks must be re-executed from the beginning. We also assume, for simplicity and to model task migration time, that the tasks can be restarted only at the beginning of the next interval.
   We only explicitly model one barrier, since the analysis is the same for B barriers, with the response time being B times that in the case of one barrier.
2. **Loosely-Coupled Barrier**
   The model for loosely-coupled is the same as tightly-coupled with one difference: the only synchronization constraint during the time between barriers is waiting for each of the other $N - 1$ tasks to reach the barrier. Thus preemption of one task does not affect the completion of the other $N - 1$ tasks. It is necessary for all the N workstations to have finished their current task before the job (any of its tasks) can proceed to the next barrier. Again, we do not explicitly model more than one barrier, since the analysis is the same for B barriers, with the response time being B times that of one barrier.

2.3 Performance Metrics and Notation

Our primary metrics of interest are batch job speedup and efficiency [14]. Since we want to study the impact of replication, we define S_d to be the speedup of a job when each of its tasks is replicated d times. Likewise, R_d and E_d are the response time and efficiency assuming each task is replicated d times. In the case when the tasks of a program have different degrees of replication, d represents the relevant degree of replication being discussed. We summarize our notation in Table 1.

Table 1: Notation

N	Parallelism of a program
d	Degree of replication
p_r	Probability of workstation reclamation
S_d	Speedup, where d is the degree of replication
R_d	Response Time, where d is the degree of replication
E_d	Efficiency, where d is the degree of replication
$F(t)$	CDF of the probability distribution
$\mu_{N:N}$	Mean of a maximum of N geometric variables
B	Number of barriers in a program, each having N tasks
f	Sequential fraction of a parallel program
T_i	Task i of a program

3 Analysis and Results

In this section we present replication analysis and results for the tightly coupled barrier and loosely coupled barrier synchronized workloads. For the sake of readability we state the relevant Lemmas before the Theorem statements. The proofs of the Lemmas are in section 5

3.1 Tightly Coupled Barrier

Here we present our analysis and results for the tighly-coupled barrier synchronization model. Assume a job has parallelism N and that the probability of a workstation reclamation during task execution is p_r. The probability that a task completes is $(1 - p_r)$. The program completes a set of N tasks, if all the workstations complete the task allocated to them. Thus the probability that the job completes a set of N tasks is $(1 - p_r)^N$. Assuming the program has a linear speedup on a dedicated set of workstations, the expected speedup of the program on a non-dedicated set of workstations is $S_1 = N(1 - p_r)^N$.

When a task is replicated on two workstations, the task gets completed when either one of them completes. The probability that the task is completed during the first allocation is $(1-p_r^2)$. If all the tasks in a program are replicated then the probability that a program completes a set of tasks is $(1 - p_r^2)^N$. The expected speedup of the replicated program (assuming linear speedup) is $S_2 = N(1-p_r^2)^N$.

The ratio of speedup with replication to the speedup without replication is

$$\frac{S_2}{S_1} = \frac{N(1 - p_r^2)^N}{N(1 - p_r)^N} = \left(\frac{1 - p_r^2}{1 - p_r}\right)^N$$

$0 \leq p_r \leq 1$ implies $(1 - p_r^2) \geq (1 - p_r)$ and the speedup ratio grows exponentially as the parallelism N is increased. The speedup with replication is guaranteed to be at least as much as the speedup without replication, if not better.

In Fig. 1 we present plots of the ratio of speedup with replication over the speedup with no replication. Unless varied, we assume $p_r = 0.1$, $N = 32$, and d (replication degree) $= 2$. We chose N to be 32 as a typical sized parallel program. Even though replication provides a much better performance improvement for higher values of p_r, we select $p_r = 0.1$ because running tightly coupled parallel jobs on a SNOW with high owner interference may not be practical.

In all three graphs, on the y-axis we plot the ratio of speedup with replication over the speedup with no replication. In Fig. 1(a) we vary job parallelism. We see that the ratio increases with the job parallelism. This is because as N increases, the probability of a task being preempted, and all of the tasks requiring a restart, also increases. With replication, *both* a task and its replica must be preempted before a barrier needs to be re-executed. Hence, the speedup ratio increases significantly.

In Fig. 1(b) we vary p_r. We see that as p_r increases, the utility of replicating tasks increases.

In Fig. 1(c) we vary the degree of replication, d. We see that for the parameters chosen, increasing the degree of replication up to 3 significantly improves job speedup.

Replication can also increase the efficiency [14] defined as the ratio of speedup to the number of workstations, $E(n) = S(n)/n$. Assuming all the tasks of a program are replicated once, the number of additional workstations allocated to the parallel program is equal to N, where N is the parallelism of the program and is equal to the number of workstations allocated to the program without replication. The efficiency is improved if $E_2/E_1 \geq 1$ where E_2 , E_1 represent the efficiency of the program with and without replication respectively. $E_2/E_1 = S_2/(2S_1)$, so the efficiency is improved if $S_2/S_1 \geq 2$. This happens when :

$$\frac{(1-p_r^2)^N}{(1-p_r)^N} \geq 2$$

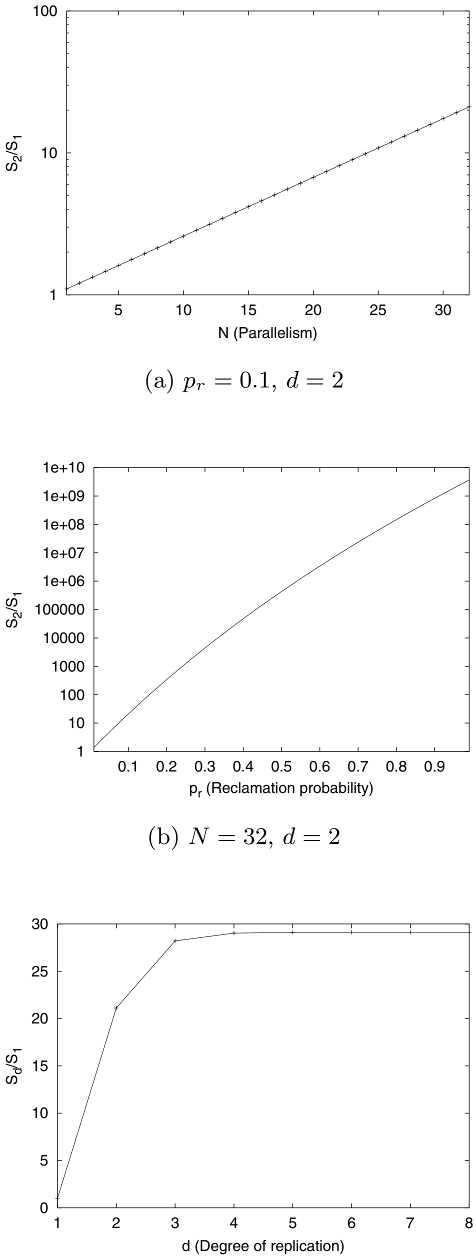
$$\frac{1-p_r^2}{1-p_r} \geq 2^{1/N}$$

$$\log_2\left(\frac{1-p_r^2}{1-p_r}\right) \geq \frac{1}{N}$$

$$N \geq \frac{1}{\log_2\left(\frac{1-p_r^2}{1-p_r}\right)}$$

Given p_r, we can calculate the parallelism for which the efficiency of the program will be improved by replicating all its tasks. We can use the knowledge of N and p_r to determine the degree of improvement (or degradation) replication causes in the efficiency.

In the general case when each task i of the program of parallelism N is replicated d_i times, where $d_i \geq 1$, the speedup of the replicated program is $N\prod_{i=1}^{N}(1-p_r^{d_i})$. Replication causes improvement in efficiency if $N\prod_{i=1}^{N}(1-p_r^{d_i}) \geq (1-p_r)^N\sum_{i=1}^{N} d_i$.



(a) $p_r = 0.1,\ d = 2$

(b) $N = 32,\ d = 2$

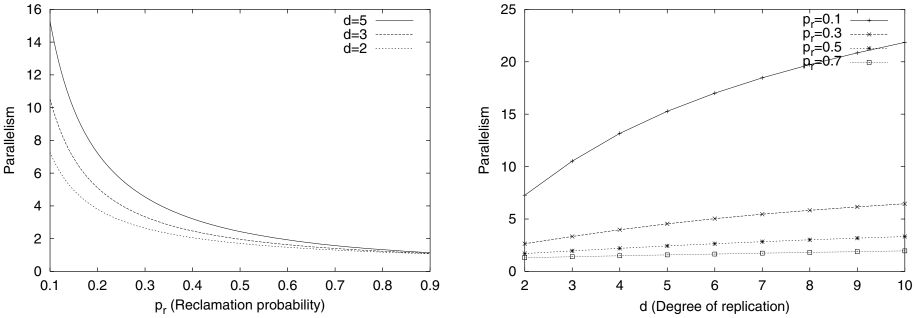
(c) $p_r = 0.1,\ N = 32$

Fig. 1: Speedup Ratio (Replication over No-Replication)

Figure 2 shows the minimum parallelism a program must have, to achieve improved efficiency by replication. Programs that are larger (have higher parallelism) than the minimum parallelism will have better efficiency with replication than without replication. All the tasks of a program are replicated to an equal degree.

In Fig. 2(a) we vary p_r. We plot the minimum parallelism needed when the degree of replication is 2, 3 and 5. For a low value of p_r (low owner interference), replication improves efficiency only for large programs. When the owner interference is high, replication improves performance significantly and is thus able to improve the efficiency of both large and small programs.

In Fig. 2(b) we vary the degree of replication d on the x-axis and view its effect on the minimum parallelism which is plotted on the y-axis. A lower degree of replication improves the efficiency for programs with smaller parallelism than does a higher degree of replication. Note, for higher degrees of replication where the efficiency ratio is less than 1, it is the case that higher speedup is achieved, but at the cost of a lower efficiency.



(a) (b)

Fig. 2: Minimum parallelism for which replication improves efficiency

Allocating Extra Workstations to the Same Program with Equal Replication. In the previous sections we considered the benefit of replicating all of the tasks relative to no replication. In this section we investigate the best way to allocate additional replicas when we do not have enough workstations available to replicate all the tasks an additional time. Suppose we have k workstations ($1 \leq k \leq N$) to allocate to a program of parallelism N, whose tasks are all replicated d times. The question we have is : Is it better to use all these k workstations to replicate just one task of the program k additional times, or to use them to replicate k tasks of the program 1 additional time?

Lemma 5.1: If $0 < p < 1$, $n > 0$ and $k > 0$, then $\forall m > n$, we have

$$\frac{1-p^{n+k}}{1-p^n} > \frac{1-p^{m+k}}{1-p^m}$$

Theorem 3.1. *Replicating k tasks one additional time gives better speedup (and response time) than replicating one of the tasks k additional times.*

Proof. Let S_{d+k}, S_{d+1} denote the estimated speedups when one task is replicated k additional times and when k tasks are replicated one additional time respectively. We have

$$S_{d+k} = N(1-p_r^d)^{N-1}(1-p_r^{d+k})$$

$$S_{d+1} = N(1-p_r^d)^{N-k}(1-p_r^{d+1})^k$$

For $k = 1$, we have $S_{d+1} = S_{d+k} = N(1-p_r^d)^{N-1}(1-p_r^{d+1})$. For $k = 2$, we have

$$S_{d+k} = N(1-p_r^d)^{N-1}(1-p_r^{d+2})$$

$$S_{d+k} = N(1-p_r^d)^{N-1}(1-p_r^{d+1})\frac{(1-p_r^{d+2})}{(1-p_r^{d+1})}$$

$$S_{d+1} = N(1-p_r^d)^{N-1}(1-p_r^{d+1})\frac{(1-p_r^{d+1})}{(1-p_r^{d})}$$

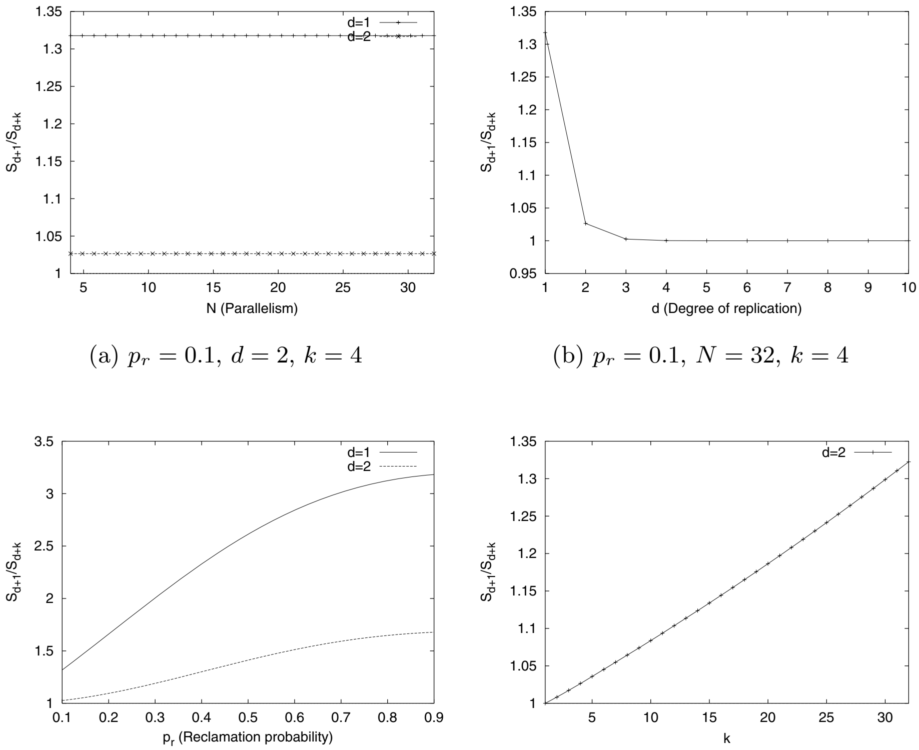
By Lemma 5.1 (Sect. 5) we know $S_{d+1} > S_{d+k}$. Similarly, for $k > 2$, we can reapply Lemma 5.1 $k-1$ times to show that $S_{d+1} > S_{d+k}$.

Thus it is better to replicate k tasks of the program one additional time, than to replicate one task k times. □

In Fig. 3 we present plots of the ratio of speedup (S_{d+1}) when one workstation is allocated to each of k tasks over the speedup (S_{d+k}) when all k workstations are allocated to one of the tasks. In all four graphs, we plot the speedup ratio S_{d+1}/S_{d+k}, as the y-axis. Unless varied, we assume $p_r = 0.1$, $d = 2$, $N = 32$, and the number of extra workstations to be allocated $k = 4$. Before allocation of the extra workstations, each of the tasks has an equal degree of replication.

In Fig. 3(a) we vary the parallelism N. We plot N only for values where $N \geq k$. Assume, without loss of generality, that the allocation of the k workstations is done among the first k tasks. For every value of N allocating the additional workstation evenly results in a speedup ratio of 1.03 for $d = 2$ and 1.32 for $d = 1$. From the expressions for S_{d+1} and S_{d+k}, we can see that the tasks $k+1$ through N contribute equally to the speedup in both cases. Consequently, they do not affect the ratio S_{d+1}/S_{d+k}. For $N \geq k$, the ratio of the speedups is independent of N.

In Fig. 3(b) we vary the initial degree of replication d (before allocating the k extra workstations) and study its effect on the ratio S_{d+1}/S_{d+k}. For a lower initial degree of replication, the allocation of k workstations to the program has



(a) $p_r = 0.1$, $d = 2$, $k = 4$

(b) $p_r = 0.1$, $N = 32$, $k = 4$

(c) $d = 2$, $N = 32$, $k = 4$

(d) $p_r = 0.1$, $d = 2$, $N = 32$

Fig. 3: Speedup ratio (One-to-each over All-to-one)

a greater effect. As d increases, the k workstations have a smaller effect on the overall speedup. Thus the ratio of speedups approaches 1 for higher values of d.

In Fig. 3(c) we vary p_r. The benefit of allocating the k workstations is greater for higher amounts of owner interference. Thus the benefit of distributing the extra workstations among the k tasks is more pronounced for higher values of p_r.

In Fig. 3(d) we vary the number of extra workstations k. When $k = 1$ the ratio of speedups equals 1. As the number of extra workstations is increased, the difference in the performance of the two allocation policies increases.

Allocating an Extra Workstation to the Same Program with Non-identical Replication. Let us assume we have a program J with N tasks (T_1, T_2, ... T_N) running in parallel. Let us assume each task T_i has d_i replicas. If we have an extra workstation to allocate to one of these tasks, we wish to determine the allocation that will maximize speedup.

Lemma 5.1: If $0 < p < 1$, $n > 0$ and $k > 0$, then $\forall m > n$, we have

$$\frac{1-p^{n+k}}{1-p^{n}} > \frac{1-p^{m+k}}{1-p^{m}}$$

Theorem 3.2. *Speedup is maximized when the extra workstation is allocated to the task that has the least replicas.*

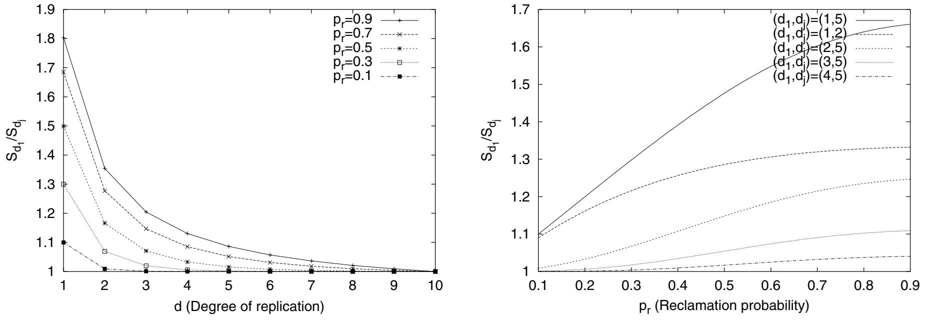
Proof. The speedup of the program before allocating the extra workstation is given by $S = N(1-p_r^{d_1})(1-p_r^{d_2})...(1-p_r^{d_N}) = N\prod_{i=1}^{N}(1-p_r^{d_i})$. Let S_j denote the speedup of the program when the extra workstation is allocated to task T_j (i.e. task T_j is replicated one additional time). $S_j = N(1-p_r^{d_j+1})\prod_{i\neq j,i=1}^{N}(1-p_r^{d_i})$. Also,

$$\frac{S_j}{S} = \frac{N(1-p_r^{d_j+1})\prod_{i\neq j,i=1}^{N}(1-p_r^{d_i})}{N\prod_{i=1}^{N}(1-p_r^{d_i})}$$

$$\frac{S_j}{S} = \frac{(1-p_r^{d_j+1})}{(1-p_r^{d_j})}$$

We can assume, without loss of generality, that T_1 has the least replicas, i.e. $d_1 \leq d_j, \forall j \neq 1$. By Lemma 5.1 we know that $S_1/S \geq S_j/S, \forall j, 1 < j \leq N$. Thus $S_1 \geq S_j$. Therefore we maximize speedup by allocating the extra workstation to the least replicated task. □

In Fig. 4 we plot the ratio of the speedup when we allocate the extra workstation to the task with degree of replication d_1 over the speedup when we allocate the extra workstation to the task with degree of replication d_j, where $d_1 \leq d_j$.



(a) (b)

Fig. 4: Speedup Ratio (Allocate-to-lower-d over Allocate-to-higher-d)

In Fig. 4(a), we vary d_1 with a fixed value of $d_j = 10$. As the difference in the initial degrees of replication between the two tasks (before the extra workstation is allocated) is low, the ratio of the speedups approaches 1. It is more important to allocate the extra workstation to the least replicated task, when the difference in their degrees of replication is high.

In Fig. 4(b), we vary p_r. At higher values of owner interference using the extra workstation to replicate the least replicated task becomes more important.

Allocating Extra Workstations to Identical Programs. We now consider the case where we have TWO parallel jobs, each with equal parallelism, and want to allocate additional replicas in the best way possible. Consider the problem of allocating k extra workstations to jobs J_1 and J_2 which each have $N \geq k$ tasks. Each task of both the programs has d replicas. In other words, the parallelism and replication of each job is equal. Our objective is to allocate the additional k workstations so as to minimize mean response time.

Lemma 5.3: If $x \geq 1$, $m \geq 0$, $n \geq 0$ and wlog $m \geq n$, then $1+x^{m+n} \geq x^m + x^n$.

Theorem 3.3. *Giving some extra workstations to each program results in a better mean response time than giving all of them to one of the programs.*

Proof. Let R_{d+k} be the mean response time when all k additional workstations are allocated to one of the programs, say J_1. Let R_{d+k_1} be the mean response time when k_1, $(0 < k_1 < k)$, workstations are allocated to J_1 and $k - k_1$ workstations are allocated to J_2. Assuming that the program is composed of one barrier, and all the tasks are of unit length, the expected response time of J_1 (or J_2) before allocation of the extra workstations is $\frac{1}{(1-p_r^d)^N}$. Therefore we have

$$R_{d+k} = \frac{1}{2}\left(\frac{1}{(1-p_r^d)^N} + \frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k}\right)$$

and

$$R_{d+k_1} = \frac{1}{2}\left(\frac{1}{(1-p_r^d)^{N-k_1}(1-p_r^{d+1})^{k_1}} + \frac{1}{(1-p_r^d)^{N-k+k_1}(1-p_r^{d+1})^{k-k_1}}\right)$$

Since $d \geq 1$, we know $\frac{1-p_r^{d+1}}{1-p_r^d} \geq 1$. As $k \geq 1$ and $0 < k_1 < k$, applying Lemma 5.3 we get

$$1 + \left(\frac{1-p_r^{d+1}}{1-p_r^d}\right)^k \geq \left(\frac{1-p_r^{d+1}}{1-p_r^d}\right)^{k-k_1} + \left(\frac{1-p_r^{d+1}}{1-p_r^d}\right)^{k_1}$$

$$\frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k}\left[1 + \frac{(1-p_r^{d+1})^k}{(1-p_r^d)^k}\right] \geq$$
$$\frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k}\left[\frac{(1-p_r^{d+1})^{k-k_1}}{(1-p_r^d)^{k-k_1}} + \frac{(1-p_r^{d+1})^{k_1}}{(1-p_r^d)^{k_1}}\right]$$

$$\frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k}\left[\frac{1}{(1-p_r^d)^k(1-p_r^{d+1})^{-k}}+1\right] \geq$$
$$\frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k}$$
$$\left[\frac{1}{(1-p_r^d)^{k-k_1}(1-p_r^{d+1})^{-k+k_1}}+\frac{1}{(1-p_r^d)^{k_1}(1-p_r^{d+1})^{-k_1}}\right]$$

$$\frac{1}{(1-p_r^d)^N}+\frac{1}{(1-p_r^d)^{N-k}(1-p_r^{d+1})^k} \geq$$
$$\frac{1}{(1-p_r^d)^{N-k_1}(1-p_r^{d+1})^{k_1}}+\frac{1}{(1-p_r^d)^{N-k+k_1}(1-p_r^{d+1})^{k-k_1}}$$

Hence $R_{d+k} \geq R_{d+k_1}$, so we get a better mean response time by splitting the extra workstations among the two programs than by giving them all to one of them. □

Theorem 3.4. *The mean response time of two programs is minimized when the extra workstations are split equally among the two programs.*

Proof. Now, for a fixed k, we find k_1 so as to minimize the mean response time. Note, we can minimize the mean response time, by minimizing

$$\left(\frac{1-p_r^{d+1}}{1-p_r^d}\right)^{k-k_1}+\left(\frac{1-p_r^{d+1}}{1-p_r^d}\right)^{k_1}$$

Let $a=\frac{1-p_r^{d+1}}{1-p_r^d}$, and let $f(x)=a^{k-x}+a^x$.

$$f'(x)=\log a\ (a^x-a^{k-x})$$

$f'(x)=0$ when $a^x=a^{k-x}$ which is true when $x=k/2$.

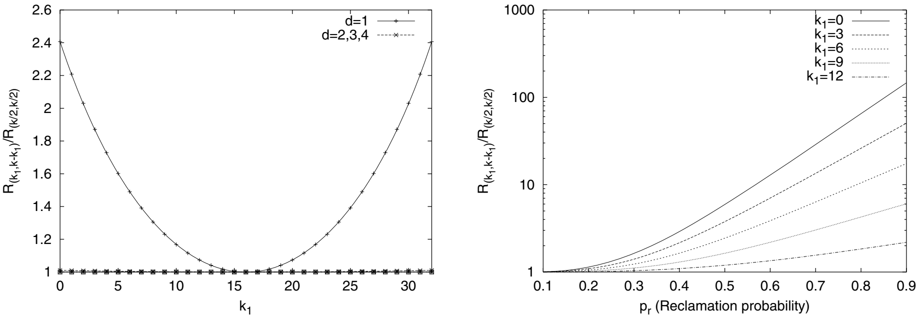
$$f''(x)=\log^2 a\ (a^x+a^{k-x})$$

$f''(k/2)>0$, therefore $x=k/2$ is the minima of $f(x)$.

Hence, we get the best mean response time when we equally share the additional workstations among the two programs J_1 and J_2. □

In Fig. 5 we plot the effect of distributing k extra workstations among two identical programs on their mean job response time. Unless varied, we assume $N=32$, $p_r=0.1$, $d=2$, $k=32$. A distribution (k_i,k_j) means, without loss of generality, k_i workstations are allocated to the first program and k_j workstations are allocated to the second program.

In both the graphs, we plot the ratio of the mean job response time for the distribution $(k_1,k-k_1)$ over the response time for the distribution $(k/2,k/2)$. We use different values of k_1 for the plots.



(a) $p_r = 0.1$, $d = 2$, $N = 32$, $k = 32$

(b) $d = 2$, $N = 32$, $k = 32$

Fig. 5: Mean Response Time Ratio

In Fig. 5(a), we vary k_1 (x-axis) in the range 0 through $k = N = 32$. We plot the curves corresponding to d values of 1, 2, 3 and 4. In each case, the plot is symmetric about the line $x = k/2$ and the minimum mean job response time is reached for $(k/2, k/2)$. The ratio is higher for lower values of d. So it is especially important to divide the extra workstations equally among the two programs for low initial degree of replication. As seen earlier, increasing replication for higher values of d does not improve performance significantly. Thus the benefit of distributing the workstations among the programs is low for d = 2, 3, 4. In the plot the difference between d = 2, 3 and 4 is not discernible.

In Fig. 5(b), we vary p_r along the x-axis and plot the response time ratio from top to bottom for k_1 values of 0, 3, 6, 9 and 12. The y-axis has a logarithmic scale. A higher owner interference causes the ratio to increase significantly. For high values of p_r, even a slight imbalance in the allocation of the workstations among the two programs has a stiff penalty in terms of mean job response time.

3.2 Loosely Coupled Barrier

Here we consider barrier synchronized programs whose tasks do not have to be co-scheduled. So a job makes a barrier so long as all the tasks in the set are completed irrespective of whether they were all running simultaneously or not. In this loosely coupled scenario, the time a program needs to reach a barrier is the maximum of the time needed for each task of the corresponding set to complete. When a task is replicated, it is sufficient for one of the replicas to finish. Thus we only need to consider the replica that has the minimum completion time. Hence for a program with replicated tasks, the time needed to complete a set of tasks (reach a barrier) is the maximum of the time required to complete each of the individual tasks, which is in turn the minimum of the completion times of the task's replicas.

The probability that a task gets completed follows a geometric distribution with parameter $(1 - p_r)$. The c.d.f of the distribution is given by $F(t) = 1 -$

$(1-(1-p_r))^t = 1-p_r^t$. $F^N(t) = (1-p_r^t)^N$. The mean of the maximum of N geometric variables with parameter $1-p_r$ [16] is $\mu_{N:N} = \sum_{t=0}^{\infty}(1-F^N(t)) = \sum_{t=0}^{\infty}(1-(1-p_r^t)^N)$.

Now consider the case where each task is replicated d times. The time needed to complete a task is the minimum of d geometric variables with the parameter $1-p_r$. The minimum of d geometric variables (with parameter $1-p_r$) follows a geometric distribution with parameter $1-(1-(1-p_r))^d = 1-p_r^d$. The c.d.f of the geometric distribution with parameter $1-p_r^d$ is given by $F_d(t) = 1-(1-(1-p_r^d))^t = 1-p_r^{dt}$. The mean of the maximum of N such geometric variables (with parameter $1-p_r^d$) is given by $\mu'_{N:N} = \sum_{t=0}^{\infty}(1-F_d^N(t)) = \sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N)$.

$\mu'_{N:N}$ gives the mean time required to complete one barrier. If a program has B such barriers, then the time needed to complete the program is $B\mu'_{N:N}$. Similarly the time needed to complete a program with no replication is $B\mu_{N:N}$.

The completion time of the sequential version of the program is BN. Hence the speedup of the program without replication is $S_1 = \frac{BN}{B\mu_{N:N}} = \frac{N}{\sum_{t=0}^{\infty}(1-(1-p_r^t)^N)}$ and the speedup of the program with d replicas is $S_d = \frac{BN}{B\mu'_{N:N}} = \frac{N}{\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N)}$. The ratio of the speedup with replication to the speedup without replication is given by

$$\frac{S_d}{S_1} = \frac{N/\left(\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N)\right)}{N/\left(\sum_{t=0}^{\infty}(1-(1-p_r^t)^N)\right)}$$

$$\frac{S_d}{S_1} = \frac{\sum_{t=0}^{\infty}(1-(1-p_r^t)^N))}{\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N))}$$

Since $0 < p_r < 1$ and $d \geq 1$, we have $p_r^d \leq p_r$. Given $t \geq 0$, $p_r^{dt} \leq p_r^t$. $(1-p_r^{dt}) \geq (1-p_r^t)$ and $(1-p_r^{dt})^N \geq (1-p_r^t)^N$, where $N \geq 1$. Hence, $(1-(1-p_r^{dt})^N) \leq (1-(1-p_r^t)^N)$, $\forall t \geq 0$. Therefore $S_d \geq S_1$. Note for $d > 1$, $S_d > S_1$.

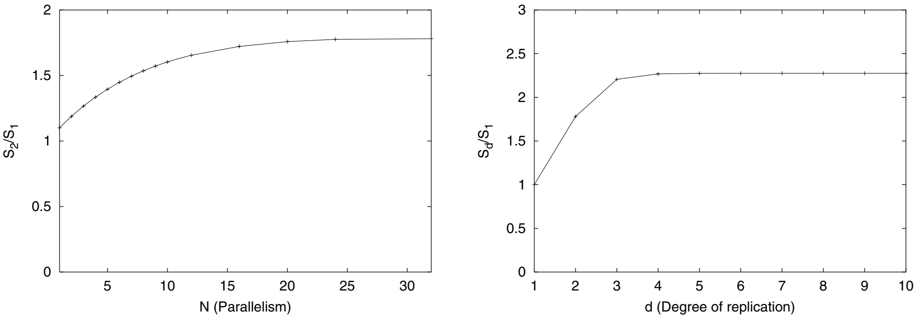
From the above we conclude that replication always results in better speedups when $0 < p_r < 1$.

In both the graphs of Fig. 6 we plot, on the y-axis, the ratio of speedup with replication over the speedup without replication. Unless varied, we assume $N = 32$, $p_r = 0.1$ and $d = 2$.

In Fig. 6(a), we plot the speedup ratio as we vary the parallelism N on the x-axis. The speedup ratio is greater than 1, as replication provides at least as much speedup as no replication. The speedup improvement is better for larger jobs (higher values of N) than for smaller jobs. The graph tends to level off for larger N, because for larger N values, the chances of one of the workstations being reclaimed are much higher, so adding just one degree of replication is not as effective (as compared to smaller jobs).

In Fig. 6(b), we plot the speedup ratio as we vary the degree of replication d. The speedup improvement increases significantly until about $d = 3$ (for $N = 32$, $p_r = 0.1$), after which the improvement levels off.

Allocating Extra Workstations to the Same Program with Equal Replication. Now consider the problem of allocating $1 \leq k \leq N$ additional workstations to a program whose tasks are all replicated d times.



(a) $p_r = 0.1$, $d = 2$

(b) $p_r = 0.1$, $N = 32$

Fig. 6: Speedup Ratio (Replication over No-Replication)

Lemma 5.2: If $0 < p < 1$, $t > 0$, $d > 0$, and $n \geq 1$, then $\forall k$, k,n integers and $1 \leq k \leq n$, we have

$$(1 - p^{dt})^{n-k}(1 - p^{(d+1)t})^k \geq (1 - p^{dt})^{n-1}(1 - p^{(d+k)t})$$

Theorem 3.5. *Replicating k tasks one additional time gives better speedup (and response time) than replicating one of the tasks k additional times.*

Proof. Let S_{d+k}, S_{d+1} denote the speedups when one task is replicated k additional times ($d + k$ times) and when k tasks are replicated one additional time ($d + 1$ times) respectively. Thus we have

$$S_{d+k} = \frac{N}{\sum_{t=0}^{\infty}(1 - (1 - p_r^{dt})^{N-1}(1 - p_r^{(d+k)t}))}$$

$$S_{d+1} = \frac{N}{\sum_{t=0}^{\infty}(1 - (1 - p_r^{dt})^{N-k}(1 - p_r^{(d+1)t})^k)}$$

$$\frac{S_{d+1}}{S_{d+k}} = \frac{\sum_{t=0}^{\infty}(1 - (1 - p_r^{dt})^{N-1}(1 - p_r^{(d+k)t}))}{\sum_{t=0}^{\infty}(1 - (1 - p_r^{dt})^{N-k}(1 - p_r^{(d+1)t})^k)}$$

From Lemma 5.2 we know that $\forall t \geq 0$, $(1 - p_r^{dt})^{N-k}(1 - p_r^{(d+1)t})^k \geq (1 - p_r^{dt})^{N-1}(1 - p_r^{(d+k)t})$. Therefore, $1 - (1 - p_r^{dt})^{N-k}(1 - p_r^{(d+1)t})^k \leq 1 - (1 - p_r^{dt})^{N-1}(1 - p_r^{(d+k)t})$ which implies

$$\frac{S_{d+1}}{S_{d+k}} \geq 1$$

Thus it is better to replicate k tasks one additional time than to use all the k workstations to replicate just one process k times more. □

Allocating Extra Workstations to the Same Program with Different Replication. Suppose we have a program that has one task, T_1 with m total replicas and another task, T_2 with n total replicas, where $m > n$. Since the change in speedup of the program, after allocating $k \geq 1$ additional workstations to one of these tasks, only depends on the change in speedups due to the additional replication of one of these tasks, we can assume, without loss of generality, that the program only has 2 tasks T_1, T_2.

Lemma 5.1: If $0 < p < 1$, $n > 0$ and $k > 0$, then $\forall m > n$, we have

$$\frac{1 - p^{n+k}}{1 - p^n} > \frac{1 - p^{m+k}}{1 - p^m}$$

Theorem 3.6. *Allocating extra workstations to the task that is replicated the least results in better speedup (and response time) than allocating extra workstations to other tasks.*

Proof. The speedup of the program when the k extra workstations are allocated to T_1 is

$$S_{m+k} = \frac{N}{\sum_{t=0}^{\infty}(1 - (1 - p_r^{(m+k)t})(1 - p_r^{nt}))}$$

and the speedup of the program when the k workstations are used to replicate T_2 is

$$S_{n+k} = \frac{N}{\sum_{t=0}^{\infty}(1 - (1 - p_r^{mt})(1 - p_r^{(n+k)t}))}$$

where, $N = 2$.

From Lemma 5.1, we know $\forall t \geq 0$, $(1-p_r^{mt})(1-p_r^{(n+k)t}) \geq (1-p_r^{(m+k)t})(1-p_r^{nt})$. Hence, $(1-(1-p_r^{mt})(1-p_r^{(n+k)t})) \leq (1-(1-p_r^{(m+k)t})(1-p_r^{nt}))$. Therefore, $S_{n+k} \geq S_{m+k}$. So allocating the k additional workstations to the task which is replicated fewer times (before the allocation) is better. □

Allocating Extra Workstations to Identical Programs. Consider two programs J_1 and J_2 which have $N \geq k \geq 1$ tasks each. The tasks of both the programs are replicated a total of d times each. Now we wish to allocate k additional workstations to the two programs so as to minimize the mean response time of the programs. We claim that allocating all k workstations to just one of the two programs is at least as good (better in most cases) as allocating some to each program.

Lemma 5.3: If $x \geq 1$, $m \geq 0$, $n \geq 0$ and wlog $m \geq n$, then $1+x^{m+n} \geq x^m+x^n$.

Theorem 3.7. *The mean response time is lower when all the extra workstations are allocated to either one of the programs, rather than split among both the programs.*

Proof. Let R_{d+k} be the mean response time of the programs when k workstations are all allocated to one of the programs and none to the other. Let R_{d+k_1} be the mean response time of the programs when k_1, $1 \leq k_1 < k$ workstations are allocated to one of the programs and $k - k_1$ workstations are allocated to the other program. Here we assume the (identical) programs both have one set of tasks (barrier).

$$R_{d+k} = \frac{1}{2}\left[\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N) + \sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k)\right]$$

$$\begin{aligned} R_{d+k_1} = \ & \frac{1}{2}[\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1}) \\ & \qquad + \sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1})] \end{aligned}$$

For $d \geq 1$, $t \geq 0$ we have $(1-p_r^{(d+1)t}) \geq (1-p_r^{dt})$. Therefore, $\left(\frac{1-p_r^{(d+1)t}}{1-p_r^{dt}}\right) \geq 1$. By Lemma 5.3, we have

$$1 + \left(\frac{1-p_r^{(d+1)t}}{1-p_r^{dt}}\right)^k \geq \left(\frac{1-p_r^{(d+1)t}}{1-p_r^{dt}}\right)^{k_1} + \left(\frac{1-p_r^{(d+1)t}}{1-p_r^{dt}}\right)^{k-k_1}$$

$$1 + \frac{(1-p_r^{(d+1)t})^k}{(1-p_r^{dt})^k} \geq \frac{(1-p_r^{(d+1)t})^{k_1}}{(1-p_r^{dt})^{k_1}} + \frac{(1-p_r^{(d+1)t})^{k-k_1}}{(1-p_r^{dt})^{k-k_1}}$$

$$(1-p_r^{dt})^k + (1-p_r^{(d+1)t})^k \geq (1-p_r^{dt})^{k-k_1}(1-p_r^{(d+1)t})^{k_1} + (1-p_r^{dt})^{k_1}(1-p_r^{(d+1)t})^{k-k_1}$$

Multiplying both sides by $(1-p_r^{dt})^{N-k}$, we get

$$\begin{aligned} & (1-p_r^{dt})^N + (1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k \geq \\ & \qquad (1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1} + (1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1} \end{aligned}$$

$$\begin{aligned} & -(1-p_r^{dt})^N - (1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k \leq \\ & \qquad -(1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1} - (1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1} \end{aligned}$$

$$\begin{aligned} & 2-(1-p_r^{dt})^N - (1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k \leq \\ & \qquad 2-(1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1} - (1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1} \end{aligned}$$

$$\begin{aligned} & (1-(1-p_r^{dt})^N) + (1-(1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k) \leq \\ & (1-(1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1}) + (1-(1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1}) \end{aligned}$$

$$\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^N)+\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k}(1-p_r^{(d+1)t})^k)\leq$$
$$\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k_1}(1-p_r^{(d+1)t})^{k_1})$$
$$+\sum_{t=0}^{\infty}(1-(1-p_r^{dt})^{N-k+k_1}(1-p_r^{(d+1)t})^{k-k_1})$$

Thus $R_{d+k} \leq R_{d+k_1}$, so we get a better mean response time if we allocate all k workstations to one of J_1 or J_2. □

4 Replication Vs Parallelization

4.1 Tightly Coupled Barrier

Suppose we have a program J_1 with a sequential fraction f. Assume there is no upper bound on the maximum parallelism. By Amdahl's Law, we know that the speedup of this program has an upper bound of $S(N) = 1/(f+(1-f)/N)$, where N is the number of tasks of J_1. We shall assume that this speedup is achieved by the program when run on a set of N dedicated workstations, even though in a real scenario, the speedup achieved is much lower due to other constraints. Let us further assume that each of the N tasks has been replicated d times. Thus we are running J_1 on dN workstations. If we have $dN+1$ workstations available to us, we need to find out if it is more profitable (in terms of speedup) to increase the parallelism of J_1 to $N+1$ or to replicate an existing task one additional time.

Let S_d denote the speedup of J_1 when run on a SNOW with increased parallelism of $N+1$, and let S_{d+1} denote the speedup of J_1 when run on a SNOW with parallelism N but with extra replication. Note, for simplicity we assume p_r remains constant when the parallelism is increased to $N+1$. We have $S_d = S(N+1)(1-p_r^d)^N(1-p_r)$ and $S_{d+1} = S(N)(1-p_r^d)^{N-1}(1-p_r^{d+1})$. $S_{d+1} > S_d$ when : $S(N)(1-p_r^{d+1}) > S(N+1)(1-p_r^d)(1-p_r)$. The gain in improvement when the parallelism is increased by 1 is given by S_{d+1}/S_d.

$$\begin{aligned}\frac{S_{d+1}}{S_d} &= \frac{S(N)(1-p_r^d)^{N-1}(1-p_r^{d+1})}{S(N+1)(1-p_r^d)^N(1-p_r)}\\ &= \frac{S(N)(1-p_r^{d+1})}{S(N+1)(1-p_r^d)(1-p_r)}\\ &= a\frac{S(N)}{S(N+1)}\end{aligned}$$

where $a = \frac{(1-p_r^{d+1})}{(1-p_r^d)(1-p_r)}$. Note $a > 1$ for $d > 0$.

$$\begin{aligned}\frac{S_{d+1}}{S_d} &= a\frac{f+(1-f)/(N+1)}{f+(1-f)/N} \\ &= a\frac{(Nf+f+1-f)N}{(Nf+1-f)(N+1)} \\ &= a\frac{N+N^2f}{(N+1)+(N^2-1)f}\end{aligned}$$

$$\frac{S_{d+1}}{S_d}-1 = b[a(N+N^2f)-(N+1)-(N^2-1)f]$$

where $b = 1/[N+1+(N^2-1)f]$. Note $b > 0$ for $N > 0$.

$$\begin{aligned}\frac{S_{d+1}}{S_d}-1 &= b[aN+aN^2f-N-N^2f-(1-f)] \\ &= b[(a-1)fN^2+(a-1)N-(1-f)] \\ &= (a-1)b\left[fN^2+N-\frac{1-f}{a-1}\right]\end{aligned}$$

Example 4.1 (N=1). If $N = 1$,

$$\frac{S_{d+1}}{S_d}-1 = (a-1)b\left[f+1-\frac{1-f}{a-1}\right] = b(af+a-2)$$

Thus it is always better to increase replication (rather than parallelism) if $f > 2/a - 1$.

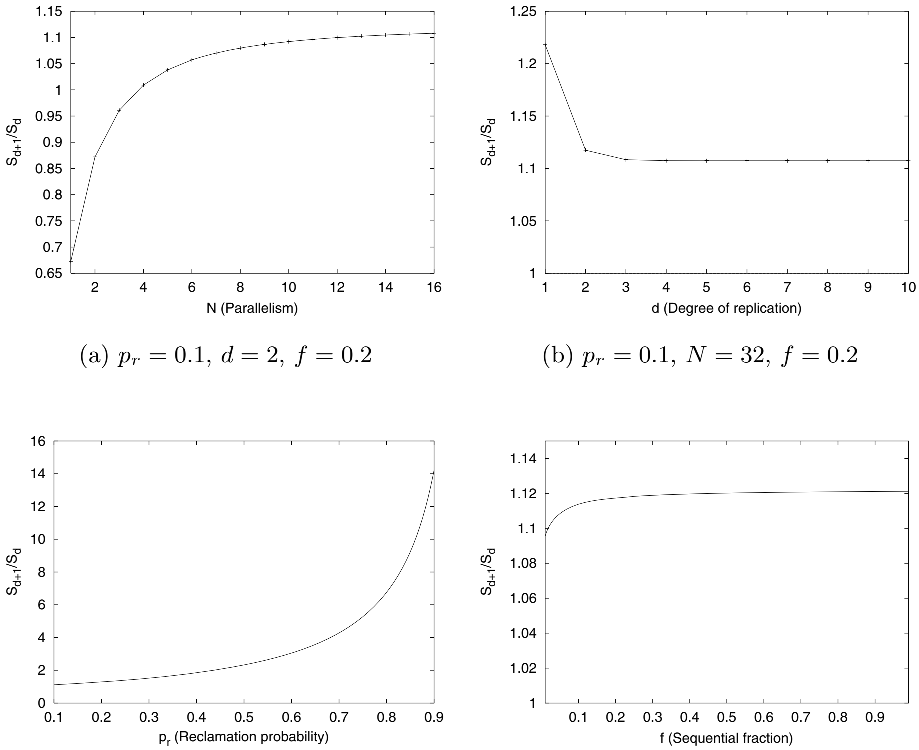
In Fig. 7 we plot the ratio of speedup with increased replication over the speedup with increased parallelism. Unless varied, we assume $d = 2$, $N = 32$, $p_r = 0.1$ and the sequential fraction of the parallel program $f = 0.2$.

In Fig. 7(a), we vary N along the x-axis and study its effect on the ratio of speedup with increased replication over speedup with increased parallelism. For this choice of parameters it is better to increase replication for all $N \geq 4$. For $N < 4$, increasing parallelism gives a better performance. When N is large, using just one extra workstation to replicate a task has a reduced effect on overall performance. Thus we see the ratio levelling off.

In Fig. 7(b), we vary the initial degree of replication d (before using the extra workstation). An increase in the initial degree of replication, means the amount of improvement possible, by replicating one of the tasks once more, is lower. Hence the improvement by increased replication is less relative to the improvement possible by increasing parallelism. So we see a drop in the speedup ratio. For higher values of d, using one extra replication of one task of the program has a low effect on overall speedup and thus we see the speedup ratio levelling off.

In Fig. 7(c), we vary p_r. Replication is especially helpful when owner interference is high. Thus when p_r increases, the speedup ratio also increases significantly.

In Fig. 7(d), we vary the sequential fraction of the program f. We notice that the sequential fraction of the program has a very low effect on the speedup ratio.



(a) $p_r = 0.1,\ d = 2,\ f = 0.2$

(b) $p_r = 0.1,\ N = 32,\ f = 0.2$

(c) $d = 2,\ N = 32,\ f = 0.2$

(d) $p_r = 0.1,\ d = 2,\ N = 32$

Fig. 7: Speedup Ratio (Replication over Parallelism)

5 Proofs

Lemma 5.1. *If* $0 < p < 1$, $n > 0$ *and* $k > 0$, *then* $\forall m > n$, *we have*

$$\frac{1 - p^{n+k}}{1 - p^n} > \frac{1 - p^{m+k}}{1 - p^m}$$

Proof. Since $k > 0$, $n > 0$ and $0 < p < 1$, $p^n > p^{n+k}$. And also since $m > n$, $p^{m-n} < 1$ which implies $(1 - p^{m-n}) > 0$. Therefore, we have

$$p^n(1 - p^{m-n}) > p^{n+k}(1 - p^{m-n})$$

$$p^n - p^m > p^{n+k} - p^{m+k}$$

$$-p^m - p^{n+k} > -p^n - p^{m+k}$$

$$1 - p^{n+k} + p^{m+n+k} - p^m > 1 - p^{m+k} + p^{m+n+k} - p^n$$

$$1 - p^{n+k} + p^m(p^{n+k} - 1) > 1 - p^{m+k} + p^n(p^{m+k} - 1)$$
$$(1 - p^{n+k})(1 - p^m) > (1 - p^{m+k})(1 - p^n)$$
$$\frac{1 - p^{n+k}}{1 - p^n} > \frac{1 - p^{m+k}}{1 - p^m}$$

□

Lemma 5.2. *If* $0 < p < 1$, $t > 0$, $d > 0$, *and* $n \geq 1$, *then* $\forall k$, *k,n integers and* $1 \leq k \leq n$, *we have*

$$(1 - p^{dt})^{n-k}(1 - p^{(d+1)t})^k \geq (1 - p^{dt})^{n-1}(1 - p^{(d+k)t})$$

Proof. Since $d > 0$ and $t > 0$, from Lemma 5.1 we have for $1 \leq i \leq k - 1$

$$\frac{1 - p^{dt+t}}{1 - p^{dt}} \geq \frac{1 - p^{(d+i)t+t}}{1 - p^{(d+i)t}}$$

Therefore,

$$\frac{(1 - p^{dt+t})^{k-1}}{(1 - p^{dt})^{k-1}} \geq \prod_{i=1}^{k-1} \frac{1 - p^{(d+i)t+t}}{1 - p^{(d+i)t}}$$

$$(1 - p^{(d+1)t})^{k-1} \geq (1 - p^{dt})^{k-1} \prod_{i=1}^{k-1} \frac{1 - p^{(d+i+1)t}}{1 - p^{(d+i)t}}$$
$$\geq (1 - p^{dt})^{k-1} \frac{1 - p^{(d+k)t}}{1 - p^{(d+1)t}}$$

$$(1 - p^{(d+1)t})^k \geq (1 - p^{dt})^{k-1}(1 - p^{(d+k)t})$$
$$(1 - p^{dt})^{n-k}(1 - p^{(d+1)t})^k \geq (1 - p^{dt})^{n-1}(1 - p^{(d+k)t})$$

□

Lemma 5.3. *If* $x \geq 1$, $m \geq 0$, $n \geq 0$ *and wlog* $m \geq n$, *then* $1+x^{m+n} \geq x^m+x^n$.

Proof. Case 1: $n = 0$

$n = 0$ implies $x^{m+n} = x^m$ and $x^n = 1$. So $1 + x^{m+n} = 1 + x^m = x^m + x^n$. Thus, $1 + x^{m+n} \geq x^m + x^n$.

Case 2: $n > 0$

Let $f(x) = 1 + x^{m+n} - x^m - x^n$. Now, $f'(x) = (m + n)x^{m+n-1} - mx^{m-1} - nx^{n-1}$.

$$f'(x) = (m + n)x^{m+n-1} \left[1 - \frac{m}{m+n}\frac{1}{x^n} - \frac{n}{m+n}\frac{1}{x^m}\right]$$
$$= (m + n)\, x^{m+n-1} \left[1 - \frac{1}{x^n}\left(\frac{m}{m+n} + \frac{n}{m+n}\frac{1}{x^{m-n}}\right)\right]$$

Since, $x \geq 1$ and $m \geq n$, we have $\frac{1}{x^{m-n}} \leq 1$, and hence

$$\left(\frac{m}{m+n} + \frac{n}{m+n}\frac{1}{x^{m-n}}\right) \leq 1$$

Because $n > 0$, $\frac{1}{x^n} \leq 1$, therefore $f'(x) \geq 0$ so $f(x)$ is increasing when $x \geq 1$. At $x = 1$, $f(1) = 0$ which implies $f(x) \geq 0$, for $x \geq 1$. Hence, $1 + x^{m+n} \geq x^m + x^n$. □

6 Conclusions

We have analyzed the performance improvements resulting from task replication of batch parallel programs running on a SNOW. Specifically, we have derived formulas to calculate the speedup and efficiency improvements due to task replication. With our analysis we have shown that replicating tasks of parallel programs can result in significant speedup improvements. Also, for some workloads, replication can also improve efficiency. Furthermore, when the probability of workstation reclamation rises, the speedup and efficiency improvements due to replication increase. Likewise, as job parallelism increases, replication becomes even more beneficial in improving speedup.

We have also analyzed the problem of using extra workstations to replicate tasks of a parallel program and shown how to distribute the extra workstations among the tasks. Specifically, for the workload models considered, if we have k extra workstations, we have shown it is better to replicate k tasks one additional time than to replicate one of the tasks k additional times. If there is only one extra workstation, we have shown it is best to allocate the extra workstation to the least replicated task. Finally, if we have extra workstations to distribute among two identical programs, distributing the workstations equally between the two programs gives least mean response time for a tightly coupled workload and giving all the extra workstations to one of the programs gives least mean response time for a loosely coupled workload.

Lastly, we have presented an analysis of the trade-off between using an extra workstation to increase parallelism and increasing replication, and have found that replication can be more beneficial than increasing parallelism for a range of tightly-coupled workloads.

We have made a strong case for considering the use of replication in the design and implementation of scheduling policies for SNOWs.

We plan on further investigating the speedup improvements of replication for the master-worker workload and for workloads with un-equal task demands. We also plan to consider the problem of distributing extra workstations among several batch parallel programs.

References

1. Acharya, A., Edjlali, G., Saltz, J.: The utility of exploiting idle workstations for parallel computation. In: Proc. 1997 ACM SIGMETRICS. (1997) 225–236

2. Adler, M., Gong, Y., Rosenberg, A.: Optimal sharing of bags of tasks in heterogeneous clusters. In: Proc. of the fifteenth annual ACM Symposium on Parallel Algorithms and Architectures. (2003) 1–10
3. Anderson, T., Culler, D., Patterson, D.: A case for networks of workstations: Now. IEEE Micro (1995)
4. Arpaci, R., Dusseau, A., Vahdat, A., Liu, L., Anderson, T., Patterson, D.: The interaction of parallel and sequential workloads on a network of workstations. In: Proc. 1995 ACM SIGMETRICS. (1995)
5. Cho, S.: Competitive Execution in a Distributed Environment. PhD thesis, University of California, Los Angeles (1996)
6. Heymann, E., Senar, M., Luque, E., Livny, M.: Evaluation of strategies to reduce the impact of machine reclaim in cycle-stealing environments. In: Proc. First IEEE/ACM International Symposium on Cluster Computing and the Grid. (2001) 320–328
7. Leutenegger, S., Sun, X.: Limitations of cycle stealing for parallel processing on a network of homogeneous workstations. Journal of Parallel and Distributed Computing **43** (1997) 169–178
8. Pruyne, J., Livny, M.: Interfacing condor and pvm to harness the cycles of workstation clusters. Journal on Future Generations of Computer Systems **12** (1996)
9. Pruyne, J., Livny, M.: Managing checkpoints for parallel programs. In: Workshop on Job Scheduling Strategies for Parallel Processing, IPPS 96. (1996)
10. Rosenberg, A.L.: Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. IEEE Transactions on Parallel and Distributed Systems **13** (2002) 179–191
11. Sterling, T., Becker, D., Savarese, D., Dorband, J., Ranawake, U., Packer, C.: Beowulf: A parallel workstation for scientific computation. In: Proc. of the International Conf. on Parallel Processing. (1995)
12. Litzkow, M., Livny, M.: Experience with the condor distributed batch system. In: Proc. of IEEE Workshop on Experimental Distributed Systems. (1990)
13. Litzkow, M.J., Livny, M., Mutka, M.W.: Condor - a hunter of idle workstations. In: Proc. of the 8th International Conference on Distributed Computing Systems (ICDCS). (1987) 104–111
14. Eager, D., Zahorjan, J., Lazowska, E.: Speedup versus efficiency in parallel systems. IEEE Transactions on Computers **38** (1989) 408–423
15. Goux, J., Kulkarni, S., Yoder, M., Linderoth, J.: An enabling framework for master-worker applications on the computational grid. In: Proc. of the 9th IEEE International Symposium on High Performance Distributed Computing. (2000) 43–50
16. David, H.: Order Statistics. John Wiley and Sons, Inc. (1970)

LOMARC — Lookahead Matchmaking for Multi-resource Coscheduling

Angela C. Sodan and Lei Lan

University of Windsor, Computer Science
401 Sunset Ave., Windsor ON N9B 3P4, Canada
acsodan@cs.uwindsor.ca, lan_lei@hotmail.com

Abstract. Job scheduling typically focuses on the CPU with little work existing to include I/O or memory. Time-shared execution provides the chance to hide I/O and long-communication latencies though potentially creating a memory conflict. We consider two different cases: standard local CPU scheduling and coscheduling on hyperthreaded CPUs. The latter supports coscheduling without any context switches and provides additional options for CPU-internal resource sharing. We present an approach that includes all possible resources into the schedule optimization and improves utilization by coscheduling two jobs if feasible. Our LOMARC approach partially reorders the queue by lookahead to increase the potential to find good matches. In simulations based on the workload model of [12], we have obtained improvements of about 50% in both response times and relative bounded response times on hyperthreaded CPUs (i.e. cut times by half) and of about 25% on standard CPUs for our LOMARC scheduling approach.

1 Introduction

The primary goal in job scheduling is to provide good response times to users. A secondary goal is to improve utilization. Both objectives may conflict with each other though often improved response times also mean improved (though potentially not optimum) utilization. The relationship between them is typically not well expressed. The best response-time behavior so far has been reported for gang scheduling [16,6]. Gang scheduling is a time-sharing approach and means that all processes of the same job are scheduled across nodes at the same time by globally synchronous time slicing [6]. Gang scheduling has shortcomings as regards latency hiding for I/O and long-latency communication. Latency hiding plays an increasingly significant role for the emerging class of data-intensive applications like datamining. Long-latency hiding is important also for potential grid applications. Loosely coordinated coscheduling [1,24,17,34,27] (avoiding the globally synchronous execution and enabling to release the CPU if waiting) and relaxed combinations of gang and local CPU scheduling [23] provide alternatives performing better in this regard. Loosely coordinated coscheduling requires modifications of the communication software and potentially the OS, typically using a spin-blocking approach to release the CPU after some time

D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.): JSSPP 2004, LNCS 3277, pp. 288–315, 2005.

of waiting and a priority boost to schedule processes that have been waiting for communication. Hyperthreading processors like the Xeon and the new Intel Pentium 4 make it possible to run two applications at the same time without any context switches and without the need to change the communication software. However, the processes compete for CPU-internal and network resources in addition to memory and I/O. Thus, interesting new options for time-shared execution are available but have to be handled carefully. The target architecture considered is a cluster with high-performance network (like Myrinet or Quadrics [35]) and user-level communication. In this paper, we only consider single-CPU nodes (hyperthreaded or standard) but our approach would be extendible to multi-way nodes. Considering the possibilities of hyperthreaded CPUs, we limit our coscheduling to a maximum of two jobs, i.e. a multiprogramming level of 2. In the following, we use the term coscheduling in the sense of running multiple jobs together.

The objectives for our own LOMARC job scheduler are:

- Inclusion of all relevant resources (CPU, network, disk, memory)
- Support of standard time-shared execution on standard CPU and coscheduling on hyperthreaded CPUs
- Increase of utilization though keeping basic primary goal of improved response times
- Usage of application characteristics via a-priori knowledge
- Usage of otherwise standard state-of-the-art scheduling approaches (priorities, backfilling etc.)

We address our objectives by the following innovative solutions:

- Optimizing the schedule by matching two applications whenever possible to share resources for high utilization
- Calculating estimates for response-time impact and utilization improvement while considering reordering to match jobs
- Including application characteristics (CPU, network, disk, memory) via an integrated cost model to estimate matchability and slowdowns
- Relaxing the scheduling order and sorting jobs more flexibly by permitting jobs to move ahead in the schedule if they pair well with other jobs though potentially to some extent pushing other jobs backward in the queue

We apply the standard per-job approach for scheduling jobs, i.e. do not attempt any global optimization. The reasons are that global optimization has a high – $O(n^3)$ – time complexity and that its benefit is even questionable, considering that the submissions are dynamic and, in standard approaches with priorities, the overall context of jobs changes permanently.

We have tested our approach via an event-based simulation and the workload described in [12], comparing it to standard space-shared job scheduling. Our tests include investigations of different heuristics, focusing either on utilization or response-time impact. We present a maximum slowdown model for the cases of resource competition and validate our estimates by practical tests with synthetic programs on a cluster with hyperthreaded CPUs.

2 Related Scheduling Work

Space/time-shared execution of parallel programs has been shown to outperform mere space sharing by providing better response times [16]. The typical practically applied approach for time/space sharing is gang scheduling which means globally synchronized execution of parallel programs [20]. We have shown that with adaptive space allocation, we can obtain even better response times with a lower multiprogramming level [25]. This has the essential benefit of reducing memory pressure. Furthermore, gang scheduling has shortcomings with respect to the overhead involved and not being able to hide I/O and other long latencies unless the application internally is doing that. Most parallel applications avoid I/O and compute in-core. However, data-intensive applications like datamining are emerging. Several different approaches have been proposed for a loosely coordinated form of coscheduling (implicit and dynamic coscheduling, periodic boost) which is more flexible and can hide latencies [1,24,17,34]. See also [27] for a survey. Most loosely-coordinated coscheduling approaches apply spin-blocking at the waiting side to avoid wasting CPU time if the partner process is not currently scheduled. Furthermore, some form of priority boost is applied at the receiving side for processes that are waiting for communication but are not currently scheduled. These mechanisms are supposed to keep jobs coscheduled if they are in synchrony and drive processes into synchrony if they are not currently coscheduled but communicating with each other. Loosely coordinated coscheduling is, however, in experimental status. One system reflecting some of these findings is Sun MPI [28], though in own experiments on an SMP server, we found that it does not satisfactorily accomplish coscheduling [25,26]. To overcome the I/O problems of gang scheduling and the problems of proper coscheduling for applications with high communication intensity, flexible coscheduling with a combination of gang and local CPU scheduling has been proposed [23,8]. The main idea is to keep frequently communicating applications gang scheduled, while relaxing the scheduling toward local CPU scheduling for coarse-grain applications that potentially have I/O or communication with long latencies. The decision can be made dynamically and per node.

One possible approach to schedule jobs with different combinations of I/O-bound and computation-bound jobs in gang scheduling is to reorder the gang-matrix rows to match jobs in the schedule and schedule them together [33]. The benefit of this approach is that it is dynamic, i.e. does not depend on pre-knowledge about characteristics and can accommodate different phases of the programs, e.g. jobs switching between I/O-bound and computation-bound phases. Then, jobs can be paired or not be paired in different phases. However, this approach needs to use the maximum I/O time of different jobs per row and requires a larger number of rows for choice, i.e. a high multiprogramming level. However, a large multiprogramming level is undesirable as regards memory pressure and the probability of actually finding pairs on large machines with potentially many different jobs per row is low. Flexible coscheduling as described above [23,8] overcomes the problem of different jobs in the row behaving differently and the dependence on the maximum per row but still depends on which jobs are randomly allocated to the same nodes as candidates for matching.

Most approaches apply a heuristic on a per-job basis to allocate jobs and determine the schedule. There is little work to perform a more global optimization. One approach optimizes the job ordering during backfilling (instead of using the common first-fit heuristic) to obtain better response times and utilization. A certain lookahead window is applied and the solution found via dynamic programming [22]. Slack-based scheduling [29] not only considers multiple factors for priority calculation but is more ambitious as regards finding optimum schedules. The approach, in principle, permits free reordering of the whole queue but sets constraints by the slack that represents maximum delays per job. In a practical setting, the approach boils down to a number of different possible heuristics. In this approach, priority-based heuristics performed best and utilization-based ones worst.

For all approaches of job scheduling, memory pressure creates constraints for scheduling which can increase fragmentation and response time significantly [21,2]. All of the above consider only one resource (I/O or memory) in addition to the computation. The approach in [10] can handle several resources, trying to balance the overall resource usage. The approach is applied during backfilling and searches the whole queue to find the best match. In [5], an optimal resource allocation in the sense of adapting the size of the job is found by, at the time of submission, simulating different possible job sizes with the current job queue and selecting the optimum.

3 Hyperthreading

Hyperthreading is a special case of simultaneous multithreading [30] with 2 threads (of the same or different applications) running simultaneously, based on the idea of letting multiple threads share the internal CPU resources in each cycle to increase their utilization. This addresses the problem that modern superscalar processors often cannot keep all their resources busy with a single program. The Xeon hyperthreaded physical CPU has only minor extensions (5% die) to support multiple architectural states – the rest of the resources including the L1 data cache and the L2/L3 unified caches are shared [14]. Hyperthreading is not limited to the Xeon processor but will become widespread with the Intel Pentium 4. However, the effectiveness of Hyperthreading depends on how well a single thread already would utilize the resources of the CPU and to what extent the threads compete for resources – such as integer and floating-point units – or complement each other. Furthermore, the impact of stalls due to insufficient instruction-level parallelism and branch misses is reduced. Another problem is the sharing of the cache which is typically a scarce resource anyway. The impact of this effect depends on the cache behavior of the program. If the working set is large but just fits nicely into the cache (which may mean that the application is cache-optimized), the competition of a second process/thread running on the CPU can severely slow down the program. However, future versions of hyperthreaded CPUs may perform better by increased cache sizes. Applications that sequentially run over a large set of data in a single pass may perform very well because having little locality (this may apply to, e.g., many datamining applications in comparison to, e.g., a matrix multiplications which use the same rows and columns multiple times). If the program has no cache locality (because of irregular accesses or poor implementation), the effects of longer-latency memory accesses can even be mitigated. Though, memory can equally well create an additional problem if the machine architecture does not provide sufficient memory bandwidth to support two processes as this is often the case [3]. Parallel applications typically use different data subsets per process/thread and thus compete for the cache. In addition, scientific applications often use floating-point operations. If they are already well optimized, they can keep the floating-point resources busy with a single thread [11]. [13] comes to the conclusion that scientific applications typically show less improvement than business applications (10%-30% vs. 60%). Symbiotic scheduling [31] and MASA [18] monitor resource conflicts among running jobs on single-CPU simultaneous multithreading processors and coschedule the jobs that have the least resource contention.

Hyperthreading provides a different option of coscheduling by running multiple applications together on the same physical CPU. This saves overhead for context switches and coordination. Especially applications that are dominated by floating-point operations can run well together with applications that are dominated by integer operations [18]. Though, the threads have to share the network, with communication not only creating network contention but also memory-access contention. In [11], the communication effects were studied and, for communication-intensive benchmarks, a degradation in performance was observed. In [18], an approach is presented to set affinity to certain physical or logical CPUs at user level. This would make it possible to extend our approach to run on dual SMP nodes. Furthermore, the involved modification of the OS-internal CPU scheduling can be used to switch hyperthreading dynamically on and off (i.e. switch from multithreading mode MT to single-threaded mode ST). This can be done by using the privileged (OS) instruction hlt (HALT).

4 The Slowdown Estimation and Empirical Evaluation

4.1 The Slowdown Estimation

For the following discussion, we first need to define our view of slowdown. Note that we always assume two jobs being coscheduled.

Definition *individual-execution-slowdown*: The factor in execution time by which an application A runs slower in joint execution with another application B ($T_{A,B}$) than it would run on its own (T_A), i.e. $sl_{A,B} = T_{A,B}/T_A$.

Note that this definition is different from the slowdown definition in loosely coordinated coscheduling such as implicit coscheduling [1] which bases on jobs normally running twice as long in joint time-shared execution. Thus, the slowdown is the relative factor beyond that, i.e. $T_{A,B}/(2T_A)$ if $T_A \leqq T_B$. For example, if two jobs with equal runtime together run 3 times as long, the slowdown is considered to be $3/2 = 1.5$. Since our concern is increasing utilization, this view is not appropriate for us.

Previous research [13,11] has investigated the performance on hyperthreaded SMP nodes and/or cluster for applications as a whole. Thus, no detailing into computation and synchronization/communication cost was done and no I/O was considered. Below we present a slightly more detailed model which estimates the maximum slowdown. We split execution time into the fraction of computation time f_{comp}, the fraction of communication time f_{comm}, and the fraction of I/O time f_{io}. For simplification, we assume that $f_{comp} + f_{comm} + f_{io} = 1$, i.e. we currently do not consider any application-internal latency hiding. For applications with many short communications, we may actually attribute most of the communication time (similar to [7]) as computation time because most of the time ($f_{comm,O,Lmcopy}$) is spent on the CPU for setting up the communication, copying to and from buffers, polling to wait on communication, and copying between host and NI (network interface) memory (because typically being buffered and handled via Programmed I/O – PIO). Long communication involves little CPU time because employing Direct Memory Access – DMA – and zero-copy communication [35]. Similarly, I/O spends a certain amount of time $f_{io,OS}$ in OS handling – especially buffer copying – on the CPU. We basically assume I/O is to the local disk – if I/O goes to an I/O server, the message-passing part ($f_{io,comm}$) has to be attributed to the network. Thus,

- $f_{CPU} = f_{comp} + f_{comm,O,Lmcopy} + f_{io,OS}$
- $f_{network} = f_{comm} - f_{comm,O,Lmcopyn} + f_{io,comm}$
- $f_{disk} = f_{io} - f_{io,OS} - f_{io,comm}$

with f_{CPU} being the time on the CPU, $f_{network}$ the time on the network, and f_{disk} the time on the disk. We assume that the time on each resource is distributed more or less equally along the execution time (e.g., not I/O clustering at the beginning and end of the job).

In the perfect case, applications would exploit different resources all the time but typically $T_{A,B} \geqq T_A$.

Disk, network, and CPU usages do not conflict with each other. In the general case, applications use all three resources though in different shares. Race conditions may apply and, in the worst case, the applications are using the same resources at the same time, and we therefore have to estimate competition on resources. Cost estimates have to consider worst case behavior per node because the probability for the worst case to happen on at least one node increases with larger number of nodes, converging to a probability of 1 (if applications need to synchronize with each other). The potential for conflicts is described below for the different resources.

Communication: Two jobs may communicate at the same time: the communication will be serialized on the NI and in the DMA. On different nodes, communications may interleave in different order, leading to delays for both applications. Since according to our measurements, non L2/L3 cache integer operations have little slowdown, we can ignore additional CPU time from added polling time. Thus, we estimate the slowdown as

$$sl_{A,B,network} = 1 + k_{network} * min\{f_{network,A}, f_{network,B}\}/f_{network,A}$$

with $k_{network}$ being a factor for potential superlinear slowdowns from network contention.

Hyperthreaded CPUs: they compete for floating-point and integer CPU-internal resources and for the cache. The former serializes instructions, the latter creates additional cache misses. The exact resource competition depends on how much instruction parallelism is available per application and which execution resources are needed at any time vs. the available resources in the CPU. In [13], the major difference made is between integer- and floating-point-dominated applications. However, in own measurements, we found a somewhat more complex relationship. As regards the cache, we found that often cache-miss latencies can be hidden within the application or among applications. Thus, coscheduling two applications with cache conflict does not necessarily reduce performance significantly more than if there are no conflicts. Furthermore, the sum of the cache-space needs does not linearly translate into cache misses because caches are not perfectly LRU (Least Recently Used) but n-way direct (the Xeon L2/L3 caches are 8-way) caches that may lead to replacements even if the working set still fits into the cache. We estimate

$$sl_{A,B,CPU} = 1 + k_{CPU} * (f_{A,B,competing} + (sl_{A,B,mem} - 1))* \\ min\{f_{CPU,A}, f_{CPU,B}\}/f_{CPU,A}$$

with $f_{A,B,competing}$ being the fraction of the code competing for CPU-internal resources; and k_{CPU} being a factor expressing potential superlinear effects from CPU contention or effects from running the CPU in hyperthreaded vs. single-threaded mode. Detailed modelling would require an advanced cache/CPU cost model and a detailed application model (access patterns, instructions mixture) which goes beyond the scope of this paper. Similar arguments apply to $sl_{A,B,mem}$ which expresses the slowdown from paging if the two applications do not fit into memory together.

I/O: the system calls for I/O will be partially serialized, may interfere with each other by going to different tracks (and therefore adding seek times), and compete for buffer space. However, the different I/O calls may also provide potential for OS-internal optimization or overlapping each other on the disk. The details depend on the OS. We make the assumption that the same serialization of cost applies as for the other cost components, i.e.

$$sl_{A,B,disk} = 1 + k_{disk} * min\{f_{disk,A}, f_{disk,B}\}/f_{disk,A}$$

with k_{disk} being a factor for potential superlinear slowdowns from disk contention.

The above leads to the following ***overall maximum slowdown***:

$$sl_{A,B} = sl_{B,A} = 1+ \\ k_{CPU} * (f_{A,B,competing} + (sl_{A,B,mem} - 1)) * min\{f_{CPU,A}, f_{CPU,B}\}+ \\ k_{network} * min\{f_{network,A}, f_{network,B}\}+ \\ k_{disk} * min\{f_{disk,A}, f_{disk,B}\}$$

Note that the slowdown for A and B is the same (since we count the shared parts) and that the maximum slowdown according to the above formula is 2 as long as no memory conflicts are involved. Then, a slowdown of 2 corresponds to time-sharing on a standard CPU and any slowdown > 2 means a decrease in utilization. The slowdown is the lower, the more different the characteristics of the two applications are, i.e. the smaller the shared parts on the different resources.

As an example, if all k factors and $sl_{A,B,mem}$ are 1 and $f_{CPU,A} = 0.4$, $f_{network,A} = 0.1$, $f_{disk,A} = 0.5$, $f_{CPU,B} = 0.5$, $f_{network,B} = 0.4$, $f_{disk,B} = 0.1$, and $f_{A,B,competing} = 0.5$, then $sl_{A,B} = 1 + 0.5 * 0.4 + 0.1 + 0.1 = 1.4$.

Information about application characteristics can be obtained by monitoring shortened sample runs or by monitoring normal application runs and keeping the information for future runs in performance databases [9]. A tool like Paradyn [15] may be used to obtain the standard characteristics f_{comp}, f_{comm}, f_{io}. Vtune [32] can obtain performance counters for CPU-internal usage and measure, for example, retired floating-point operations and cache misses to obtain estimates about CPU-internal resource usage and conflicts.

Considering the discussion above, we can now compare our coscheduling on hyperthreaded CPUs to loosely coordinated coscheduling. The latter can hide I/O latency though I/O intensive applications can significantly disturb the coordinated execution of intensively communicating jobs (and cause process switches and delays). Thus, both types of jobs should not be coscheduled. However, this negative effect does not exist on hyperthreaded processors because both applications can continue to execute at any time. For loosely coordinated coscheduling of communication and/or computation-dominated jobs, the best results obtained so far are about a factor of 2.4 slowdown, and it is not even sure whether these results generalize. Thus, the benefits are more limited. We only coschedule jobs if we can obtain a benefit, i.e. a slowdown below a $sl_{limit} \leqq 2$. As a benefit of loosely coordinated coscheduling, it is less sensitive to the cache though the spin-block also in a negative cache impact (process switches on standard CPUs invalidate the whole cache) [27].

Above, we have made the simplification not to consider application-internal latency hiding. Such consideration is, however, possible. We only have to make sure to recognize that no external latency-hiding potential is available anymore for the corresponding fractions of the code. We can simply mark these fractions as the combination of the typically two resource types. An estimation on the safe side, then, is to count the whole combined fraction for each of the corresponding resource types when estimating conflicts. Latency hiding (and improved resource usage) is still possible for such applications if matching with an application which is dominant in the third resource type.

4.2 Empirical Evaluation of Slowdowns

We have tested slowdowns with synthetic applications on a cluster with Intel Xeon processor and Myrinet interconnect, running MPICH-GM with user-level MPI communication. L2 cache size is 512 k and memory size per node 512 Mbyte.

The operating system is Linux 2.6. The compiler used was gcc with option "-O". In all measurements, we use f_{comp}, f_{comm}, and f_{io} due to our current lack of low-level monitoring tools that could reveal the CPU, network, and disk fraction. We could find hardly any difference for our test program for the CPU set to MT or ST mode (for the "streaming" application below, we found 1.5% but otherwise none at all). If runtimes were different under coscheduling, slowdowns are calculated for the shared execution time.

We first investigate hyperthreaded CPU behavior and ran applications dominating in either float (double) or integer calculations, using complex multiplication or simpler add instructions, running totally in L1 cache (using 400 bytes) or using some (40k) or much (400k) of the L2 cache. The code sequences are simple and easily fit into the cache. The summary of results can be seen in Table 1. As far as L2 usage is involved, we have modelled an access patterns that runs over the same data structure serially per iteration (thus, all data would be repeatedly replaced if not fitting totally into the cache). We have used two variants: one where adjacent array elements are accessed (creating dependencies among iteration steps) and one where only elements of the same index are used from 3 different arrays (avoiding dependencies among iteration steps). The former is similar to calculating the stencil in a mesh computation. Our results are to a large extent consistent with research as far as available – though we also found some interesting differences in the details. In [13], scientific applications benefited between 10% and 30% by running each with two threads on a hyperthreaded CPU. However, even performance on a dual SMP was not optimal. Thus, translating the hyperthreading improvement to the relative best-possible threaded performance, the slowdowns according to our definition were approximately 1.4 which is not worse than the up to 30% improvement measured for business applications. [11] shows slowdowns up to 3, including communication, for cache-intensive applications. Since the tests were done by increasing the number of processes per application, however, also the speedup behavior changed (speedup curves typically flatten with larger number of processes) and the results therefore appear to be too negative.

Table 1. Slowdown for different types of computation. + means application uses add operations, $*$ means it uses mult operations; the number indicates the size of the data in L2 cache.

	int + 0	int + 40k	int + 400k	int * 0	int * 40k	int * 400k	float any type
2x same, dependencies	1	1	1.4	1.2	1.2	1.25	between 2 and 2.2
int and float of same type	1/1.2	1.1/1.2	1.2/1.2	1.8/1.6	2/1.6	2/1.8	
2 x same, no dependencies	3.7	1.2	4.2	1.7	1.3	2.6	between 2 and 2.2

Float calculations coschedule poorly – slowdowns are between 2 and 2.2. However, they are insensitive to the cache which obviously is due to the fact that the execution time is dominating and L2 cache or memory-access latencies can be hidden. We also tested a 2D FFT calculation (using float instructions and 32 Mbyte space, i.e. exceeding the cache capacity in a repetitive execution of the algorithm). Coscheduling resulted only in a slowdown of 1.3. Our synthetic programs are extreme cases of almost exclusive float instructions and thus represent a maximum of contention – which is unlikely to occur in any real program. Integer calculations coschedule much better as long as there are access dependencies. Coscheduling integer and float works well for add operations but less well for multiplication. Coscheduling of integer calculations becomes poor, however, for the "streaming" calculations which fill the instruction pipeline well and accomplish maximum execution speed. The slowdown is up to 4.2 if there are cache conflicts. Memory latencies can no longer be hidden. However, we assume that such calculations on integers are unlikely to occur in practice but typically are used on floats. For "streaming" float calculations the slowdown is in the same range as for the dependent calculations. For detailed modelling, an integration of instruction and cache cost would be needed.

In Table 2, we show results from running applications together with a) different mixtures of communication and computation, and b) different communication granularity. In all cases, $f_{io} = 0$. The applications are run on 4 nodes and are loosely synchronous, communicating with all 3 other neighbors, sending to them and receiving from them in each communication phase. The computations are of type "int+" and run within L2 cache to focus on the effects of CPU vs. network. Note that the short communication is spending a significant amount of time on the CPU via PIO (with integer operations), whereas the long communication employs DMA and zero copy in a rendezvous protocol. Communication cost results into 13.3 μsec for a message with 200 bytes and into 120 μsec for a message with 18k bytes. In all cases, the actual slowdown is lower than the estimated maximum slowdown. As can be seen from the table, the slowdown is different for each application if running coarse- and fine-grain communication together. The application with the finer communication (smaller and more communications) suffers more. The explanation is that if the communications interleave, the finer-grain communications are stretched more, adding idling time to this application. If applications are slowed down to different extent, it would be important to make sure that enough non-competitive time is left for the application with the larger slowdown to catch up with communication (as not possible with $f_{network}$ being 0.6 for both applicationss). Thus, additional conditions for the matching may be necessary to ensure that $f_{network,A} + f_{network,B} \leqq 1$ (considered in the parameter settings of our LOMARC coscheduling algorithm). We also coscheduled a 2D FFT program on 4 nodes (employing all-to-all communication in 10% of the overall runtime) which resulted in a slowdown of 1.4 – which is exactly the projected slowdown from the known sl_{CPU}.

Finally, we show in Table 3 our results of testing different classes of applications together. The I/O is repeatedly reading a file sequentially in 1 k blocks from

Table 2. Runtimes and slowdowns for coscheduling two applications with different mixtures of computation and communication (and different communication granularities). The left number represents the row application, the right number represents the column application. Numbers in italic and parenthesis show estimated maximum slowdown (left) and estimated slowdown if there would be no CPU slowdown (right). C_{size} is the number of bytes per communication.

	$f_{comm} = 0.4$, $C_{size} = 200$	$f_{comm} = 0.4$, $C_{size} = 18k$	$f_{comm} = 0.6$, $C_{size} = 200$	$f_{comm} = 0.6$, $C_{size} = 18k$
$f_{comm} = 0.4, C_{size} = 200$	1 / 1 *(2, 1.4)*	1.1 / 1 *(2, 1.4)*	1 / 1.1 *(1.8, 1.4)*	1.25 / 1 *(1.8, 1.4)*
$f_{comm} = 0.4, C_{size} = 18k$		1.2 / 1.2 *(2, 1.4)*	1 / 1.35 *(1.8, 1.4)*	1.2 / 1.25 *(1.8, 1.4)*
$f_{comm} = 0.6, C_{size} = 200$			1.1 / 1.1 *(2, 1.6)*	1.7 / 1.1 *(2, 1.6)*
$f_{comm} = 0.6, C_{size} = 18k$				1.2 / 1.2 *(2, 1.6)*

the local disk. The communicating application is running a standard pingpong test. Note that the combinations using two communication or two I/O intensive applications are stress-tests only – LOMARC would not normally coschedule such applications.

Table 3. Slowdown if running different classes of applications together. C_{size} is the number of bytes per communication and F_{size} the file size.

	$f_{comm} = 1$, $C_{size} = 200$	$f_{comm} = 1$, $C_{size} = 18k$	$f_{comp} = 1$, $int + 0$	$f_{io} = 1$ $F_{size} = 600Mbyte$
$f_{comm} = 1, C_{size} = 200$	1.4 / 1.4	2.3 / 1.2	1 / 1	1 / 1
$f_{comm} = 1, C_{size} = 18k$		1.3 / 1.3	1 / 1	1 / 1
$f_{comp} = 1, int + 0$			1 / 1	1 / 1
$f_{io} = 1, F_{size} = 600Mbyte$				2 / 2

The results show that there is no negative impact if the job classes are different. Doing the same test with "float+400k", we found minor slowdowns (1.1 or 1.2) if coscheduling computation with communication or I/O applications. The explanation is that we are here including computational times in I/O and communication and both involve memory accesses, too.

Finally, we have studied the effect of paging. Using the same type of application as for cache measurements, we have compared the effect of running two applications with 400 Mbyte and 260 Mbyte memory usage each. The slowdown is 4.8 in the former and 2.1 in the latter case for floats. In the optimal case of fast non-dependent int calculations with 3 arrays, the slowdown is about 300 (though the absolute paging cost is similar). Thus, memory conflicts can be severe.

In summary, our measurements show that there are no unexpected superlinear slowdowns for normal cases and that the k factors can typically be set to 1. Conversely, the slowdowns actually measured are in many cases much lower than our maximum estimate if setting the k factors to 1, i.e. simply adding the cost for the overlapping shares (though the slowdowns may increase with larger numbers of nodes). Note that though we have done our best efforts to extract behavior with typical code, our tests are not exhaustive and a separate study on hyperthreaded, communication, and I/O behavior would be needed. However, the parameters used in our simulations are consistent with results reported in other research with full applications.

5 The Look-Ahead Scheduling Algorithm

5.1 The General Algorithm

We apply a standard job-scheduling algorithm with the following features

- Usage of priorities, classifying the jobs into short, medium, and long and allocating priorities according to these classes; usage of aging to prevent starvation
- First-fit during allocation of jobs onto nodes
- Flexible and dynamic allocation of nodes (no fixed and contiguous partitions required)
- Backfilling (EASY backfilling)

We basically keep short response times as the primary schedule-optimization objective and exploit utilization as far as it does not contradict good response times. However, we propose different heuristics, mainly aiming at either optimization for response times (as it would be meaningful during the day) or optimization for utilization (as it would be meaningful during the night). Memory consumption currently only plays the role of a constraint.

The key special features in our LOMARC scheduling approach are:

- Estimating the utilization gain
- Estimating the impact on the response times
- Allocating jobs to free nodes by themselves if the accumulated node requests in the queue $\leqq$ the available nodes by 20% (machine is weakly loaded)
- Finding a possible best match for the next job subject to scheduling among
  - The remaining jobs in the waiting queue
  - The running jobs

This means that LOMARC never coschedules jobs if the machine is weakly loaded, i.e. there are empty nodes to run the job. We classify jobs into CPU-bound, disk-bound, and network-bound, according to which of f_{CPU}, f_{disk}, or

$f_{network}$ dominates. Only medium and long jobs are considered for coscheduling – short ones are not worth the effort.

LOMARC can schedule either on standard or hyperthreaded CPUs with the following scheme:

- On a standard CPU, we only schedule CPU-bound and disk-bound jobs together. Only they can benefit as regards CPU utilization in this case.
- On a hyperthreaded CPU, more options exist to coschedule jobs. We consider joint execution of CPU-bound and CPU-bound jobs, CPU-bound and network-bound jobs, and network-bound and I/O-bound jobs in addition to CPU-bound and disk-bound jobs.

Thus, LOMARC does not depend on any special coscheduling software (gang or implicit coscheduling). However, LOMARC depends on the option to share the network [27]. Such sharing is, however, provided by the widespread standard native GM communication library for Myrinet and the MPICH and LAM MPI implementations that build on top of GM [35].

Fig. 1 shows pseudo code of the abstracted LOMARC algorithm. Fig. 2 and Fig. 3 graphically demonstrate the matchmaking.

Our LOMARC algorithm depends on knowing the characteristics of the applications as regards the fractions of time on CPU, network, and disk and making correct upper bound estimations for slowdowns. We assume the applications to be occasionally monitored (we have accompanying research work running on this topic). If the estimates are severely wrong in a negative sense, one application may be preempted and its execution be completed when the other one is finished [19]. Shorter overall job runtimes than estimated, however, do not hurt at all as we can try to find a new match if one job finishes.

5.2 The Utilization-Gain and Response-Time-Impact Calculation

Fig. 4 shows the search for the best match among all jobs in the waiting queue (if searching there) and the definition of matchable jobs. We first check whether job classes can be matched (e.g. whether their requirements fit). Furthermore, we estimate the slowdown according to our description above. If the slowdown is less than a certain threshold sl_{limit} ($MAX\_SLOWDOWN$), the job becomes a candidate for matching. Different heuristics can be applied as explained below. Either response-time impact and utilization gain can be estimated.

The calculation of the response-time impact does not consider any detailed packing, i.e. does not calculate any actual schedule. The reason is that the packing anyway is subject to change under dynamic submission with priorities. Furthermore, the complexity of incorporating such calculation is high − backfilling has $O(n^2)$ time complexity and, if trying all jobs in the waiting queue to find the optimum, complexity increases to $O(n^3)$. Thus, we simply assume that a perfect packing would be possible (by taking $work = runtime * size$ for each job and adding the corresponding work up for all jobs) and determine all delays on the basis of this simple heuristic. A future improvement might be to calculate exact

```
while (! waiting_queue.is_empty ()) {    //run over all jobs in queue
current_job = waiting_queue.first;       //  as long as can be scheduled
while (current_job.size<=freenodes.size){//enough space for job
  if(current_job.is_medium_or_long_job())//try find a match for job
    match = find_match (current_job);    //  among remaining jobs in
  allocate_nodes (current_job);          //  waiting queue
    if (match != null)                   //coallocate match on
     coallocate_nodes(current_job,match);//  same nodes
    if (end_of_queue) return ();
      else current_job = waiting_queue.first;
  }
if(current_job.is_medium_or_long_job){   //job won't fit on free nodes
  match = find_match_among_running (current_job);
                                         //co-schedule with running job
  if (match != null)                     //find best match among running
                                         //  jobs
    coallocate_nodes(match,current_job); //allocate job on same nodes
  if (match == null)                     //job does not match any job
  break;                                 //job cannot be scheduled now;
                                         //  continue with backfilling
}                                        //end of loop over queue
backfill();                              //try to backfill(applying
                                         //  same matching as above)
```

Fig. 1. Abstracted LOMARC scheduling algorithm as invoked upon job-termination or submission.

order for the first few jobs in the queue and apply the heuristic estimate for the rest.

As regards utilization, a detailed utilization metric would have to consider the maximum capacity of hyperthreaded CPUs, disk, and network and their utilization by each application (making detailed resource and application models necessary). Therefore, instead of absolute utilization, we consider the relative utilization improvement on the basis of the scheduled applications.

Definition ***Relative Utilization Gain***: We consider the overlap in time the two jobs run together and calculate how much faster the jobs run if coscheduled than they would run if scheduled individually. We have the following two options: to consider a timeless metric ($U_{gain,2}$) or to include the shared (overlap) runtime that is affected by the utilization change ($U_{gain,1}$). This leads to the following two formulas:

$$U_{gain,1} = (min\{S_A, S_B\} * (2/sl_{A,B} - 1) - \\ |S_A - S_B| * (1 - 1/sl_{A,B})) * (min\{T_A, T_B\}/\ max\{T_A, T_B\})\}/max\{S_A, S_B\}$$

$$U_{gain,2} = (min\{S_A, S_B\} * (2/sl_{A,B} - 1) - |S_A - S_B| * (1 - 1/sl_{A,B}))/max\{S_A, S_B\}$$

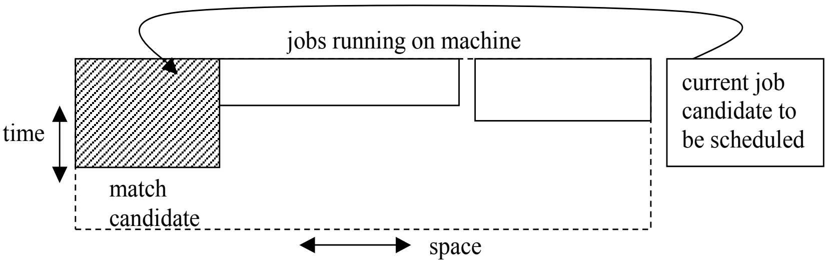


Fig. 2. Finding best match among currently running jobs.

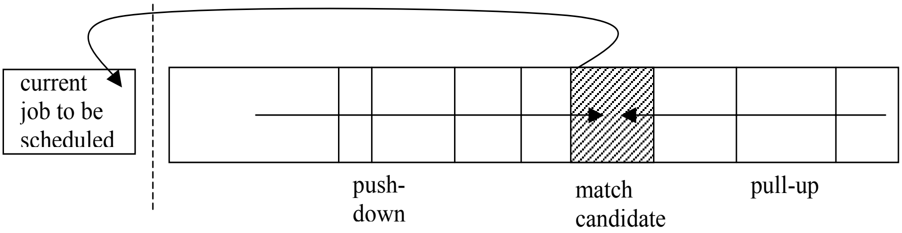


Fig. 3. Reordering the job queue if finding a match in the waiting queue.

with S being job size. These formulas consider both the win in the overlapped part and the fact that the application with larger size is – if needing synchronization – slowed down without any utilization win in the non-overlapped part.

As regards relative response times, the impact from reordering the queue can be estimated in the following way:

- Jobs in front of the job that is matched and thus moved ahead get delayed: For them, we calculate an estimate of the impact by the sum of all relative delays. We call these jobs ***push-down*** jobs.
- Jobs behind the job that is matched get scheduled earlier, assuming that the match decreases the joint runtime of the two jobs vs. running them on their own: For these jobs, we calculate an estimate of the impact by the sum of all relative improvements. We call these jobs ***pull-up jobs***.

In both cases, we include a prediction about future job submissions and the impact of these jobs on response times. We do a one-level prediction, calculating new job submissions in the time interval which we estimate for the execution of the jobs that are currently in the queue. To do so, we use parameters (average work) from the workload model. We simplify the calculation of relative response times by taking them relative from the current time on. See Fig. 5 for the details of the algorithm.

```
find_match (job) {
  maxprofit = 0; match = null;
  for each_job_in_queue (match_cand) {
    if (matchable (job, match_cand)){
      slowdown_cand = slowdown (job, match_cand);// determine slowdown
      if (slowdown_cand <= MAX_SLOWDOWN) {       // check slowdown limit
        switch (heuristic) {
          case 1:                                // utilization gain 1
            profit = utilization_gain_1 (job, match_cand);
          case 2:                                // utilization gain 2
            profit =utilization_gain_2(job,match_cand);
          case 3:                                // response times
            profit = response_time (job, match_cand, slowdown_cand);
        }
      if profit > maxprofit                      // keep best match
       {maxprofit = profit;
        match = match_cand;}
       }
   } }
  return match;
}

matchable(jobi, jobj){
  if (jobi.memory + jobj.memory <=1)
    if (jobi_is_CPU_intensive && jobj_is_CPU_intensive)
      return true;
      else (if jobi.type !=jobj.type ) return true;
  return false;
}
```

Fig. 4. Finding best match in waiting queue and definition of matchable jobs.

The complexity of our algorithm is $O(n^2)$. However, the worst case for searching through all jobs in the queue – $O(nlgn)$ – is always met if we look for the optimum match. To check whether we can reduce cost, we also incorporate a simplified version in our experiments that takes the first match.

6 Experimental Results

Our experiments are based on an event simulation with parameter settings and workload modelling as described below. The machine modelled is a cluster with 128 single-CPU nodes.

6.1 Metrics and Workloads

We use the following metrics to evaluate the performance of our LOMARC scheduling algorithm:

```
response_time (jobi, jobj, slowdown) {
   pairruntime = min (jobi.runtime, jobj.runtime) * (slowdown-1) +
                 max (jobi.runtime, jobj.runtime);
   pairsize = max (jobi.size, jobj.size);
   improvement = jobi.runtime*jobi.size / n_nodes;
   delay = (pairruntime * pairsize - jobi.runtime*jobi.size) / n_nodes;
   response_decrease = jobj.runtime*jobj.size / n_nodes - delay;
   response_increase = delay / responsetime;

   //estimate delay for push-down jobs
   for (all push_down_jobs (jobn)) {
      response_time += jobn.runtime * jobn.size / n_nodes;
      response_increase += delay / response_time; }

   // response time improvement for job being moved
   response_time += jobj.runtime* jobj.size / n_nodes;
   response_decrease = (response_time - jobj.runtime* slowdown
      * jobj.size / n_nodes) / response_time;
   // estimate improvement for pull-up jobs
   for (all_pull_up_jobs) {
      response_time += jobn.runtime * jobn.size / n_nodes;
      response_decrease += improvement /  response_time; }
   for (future_arrival_short_jobs(jobn)) {
      response_time  = jobn.runtime * jobn.size  / n_nodes;
      response_increase+= delay / response_time; }
   for (future_arrival_med_or_long jobs (jobn)) {
      response_time  = jobn.runtime * jobn.size / n_nodes;
      response_decrease+= improvement / response_time; }
   return (response_decrease - response_increase) / (number(push_down_job)
      - number(pull-up_jobs));
   }
```

Fig. 5. Pseudo code for abstracted calculation of utilization gain and response-time impact. Calculates increase/decrease relative to normal response time.

- Average response times
- Average relative bounded response times: response time in relation to run-time time, bounded by a 60 sec minimum runtime to avoid overly high impact of very small jobs
- Utilization: percentage of used-nodes time over the makespan; i.e., ratio of the accumulated used nodes and the product of makespan T and number of nodes P
- Utilization efficiency: if coscheduling, also considers positive improvements by increasing the utilization per CPU, indirectly reflected by a shortened makespan: $E = \sum_i p_i t_i / PT$ with p_i and t_i being size and runtime per job
- Makespan: the runtime of the whole job batch

We have used the model in [12] for the workload generation. This model is a complex statistical workload description, considering job sizes, job runtimes, and job interarrival times. The model includes correlations between sizes and runtimes, fractions of sequential jobs, fractions of power-of-two sizes, and differing interarrival times according to day/night-cycles. All numbers are generated in logarithmic space. A two-stage uniform distribution is used for job sizes (including probabilities for serial and power-of-two job sizes), a hyper-Gamma distribution for job runtimes, and two Gamma distributions for interarrival times (one for peak times and one for the overall daily cycle). The parameters of the model are extracted from three traces of supercomputing centers and propose a generalization from the three test cases. The nice feature of this generalized model is that it can be adapted to different machine sizes and, thus, be applied to our machine size of 128 nodes. We have modelled 8,000 jobs.

Furthermore, we have modified the original workload by shortening the job interarrival times, determined by the α parameter of the Gamma distribution. Workload 1 is the original workload, Workload 2 and Workload 3 have smaller α parameters as shown in Table 4. The table also shows the resulting load value $Load = (r - n)/(P * a)$ with r being the mean runtime, n the mean job size, and a the mean job interarrival time [12].

Table 4. Workloads modelled.

	Workload 1	Workload 2	Workload 3
α	10.23	9.83	8.83
Load	10.6	13	21

To the best of our knowledge, there do not exist any studies on the distribution of the application's resource-usage characteristics as regards CPU, network, and disk. We model the following mixtures

- M1: 40% CPU-bound, 30% network-bound, 30% disk-bound
- M2: 40% CPU-bound, 10% network-bound, 50% disk-bound
- M3: 30% CPU-bound, 50% network-bound, 20% disk-bound

We perform the majority of our tests with the mixture M1 which can be considered the mixture we expect to see on clusters with a share of scientific and datamining applications. We do some comparisons that include M3 as a representation of what might be the conventional mixture and M2 which might be the mixture for clusters specializing on datamining.

Detailed job characteristics are generated randomly, using an equal distribution per value range, according to the following scheme:

- CPU-bound jobs: f_{CPU} in [0.5,0.9), f_{disk} in [0.05,0.4) with $f_{CPU} + f_{disk}$ in [0.6,0.95)

- Disk-bound jobs: f_{disk} in [0.4,0.65), $f_{network}$ in [0.05,0.4) with $f_{disk}+f_{network}$ in [0.5,0.8)
- Network-bound jobs: $f_{network}$ in [0.4,0.65), f_{disk} in [0.05,0.4) with $f_{network} + f_{disk}$ in [0.5,0.8)

As regards the CPU-behavior, we model different probabilities that the CPU-parts of the two applications go well or poorly together, i.e. increase or decrease utilization. We set the probability for the former case to p=0.33 and for the latter to p=0.67. We assume sl_{CPU}=1.4 in the former and sl_{CPU}=2 which picks two typical cases from our measurements in Sect. 4.2 and is consistent with results from the literature. In the latter case, LOMARC does not schedule CPU-bound applications together.

Memory consumption is modelled by random generation for each job in [0.05,1] with 70% of the jobs in [0.05,0.5], 25% in (0.5,0.8), and 5% in [0.8.1]. 1 represents the maximum memory size that is available for applications. This distribution is roughly modelled as an average over existing memory studies as in [4]. LOMARC does not coschedule jobs that do not fit into memory together but, for the comparison with other scheduling approaches, we need to model the memory slowdown and set sl_{mem}=2.5. This is a lower value from our measurements in Sect. 4.2, i.e. an optimistic assumption about other approaches that do not care about memory conflicts.

6.2 Experiments with LOMARC Scheduler

To evaluate the benefits of our approach, we compare to

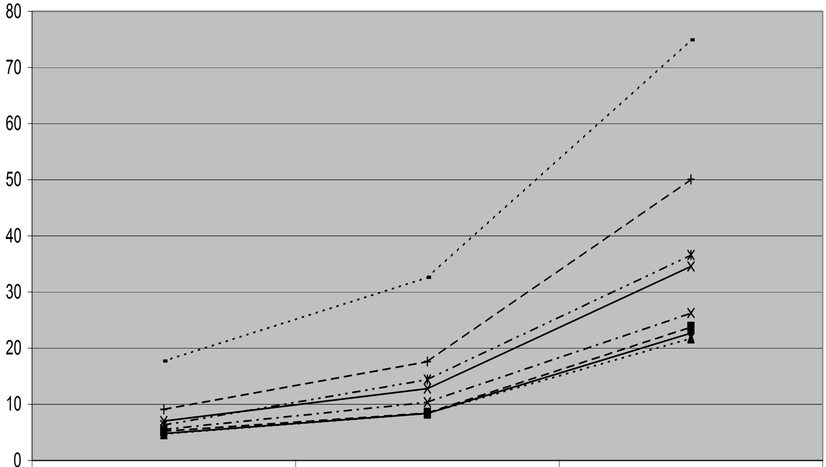
- Standard single-job scheduling (mere space sharing PSS)
- Always coscheduling two jobs if running on a hyperthreaded CPU (AC)
- Coscheduling two jobs that are adjacent in the queue if they are a match according to the LOMARC definition (AM) if running on hyperthreaded CPUs

For our LOMARC approach, we test the following variants:

- Scheduling on standard CPUs (L-N) using $U_{gain,1}$
- Optimization with different heuristics on hyperthreaded CPUs: utilization $U_{gain,1}$ (L-U1) and $U_{gain,2}$ (L-U2), response-time impact (L-R), and a variant which selects the first match found (L-FM)

We set the maximum acceptable slowdown $MAX\_SLOWDOWN$ to 1.6. For all approaches, we use priorities and EASY backfilling. We define job classes in the following way: runtimes in [1sec,1min] are classified as short, in (1min,1h] as medium, and in (1h,45h] as long with 45h being the maximum runtime modelled. Aging (to prevent starvation) is based on average waiting time T_{age}. Per each T_{age}, the priority of one job will be boosted to a higher level, so it will take a long job 2 T_{age} to have the same priority as a short job.

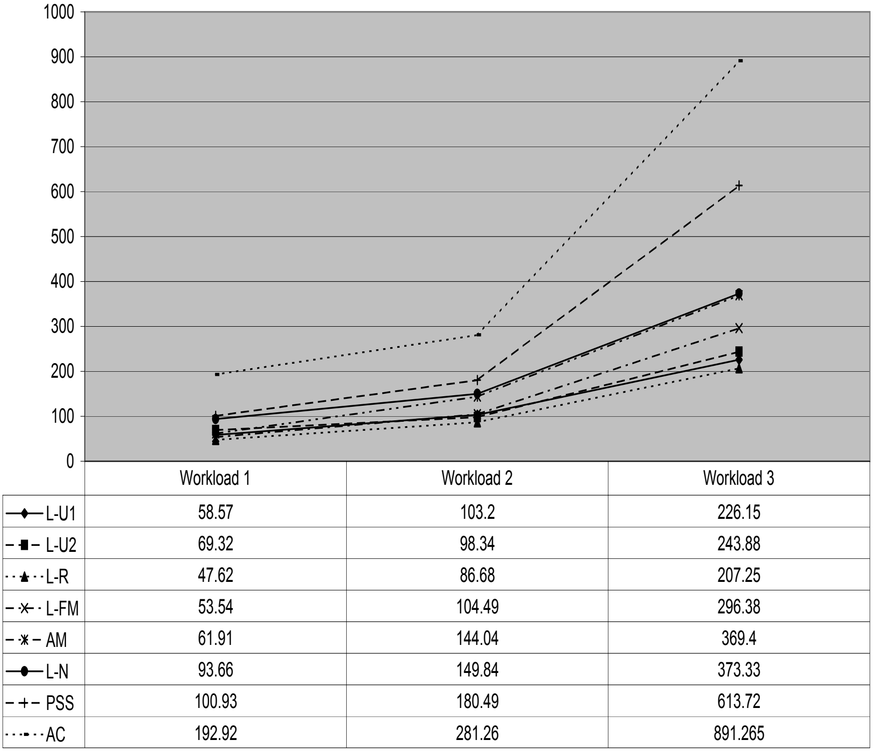
In Fig. 6 and Fig. 7, we show the results from comparing several LOMARC variants (L-U1 L-U2, L-R, L-FM, and L-N) with PSS, AC, and AM under the 3 different workloads W1, W2, and W3. In all cases, the characteristics mix M1 is used. For all workloads, all LOMARC variants perform clearly better than all other approaches. The arbitrary coscheduling AC is significantly worse than space sharing PSS and, thus, not a reasonable choice. This demonstrates that detailed match considerations are necessary to make coscheduling on a hyperthreaded CPU meaningful.



	Workload 1	Workload 2	Workload 3
—◆—L-U1	4.79	8.37	22.69
-■- L-U2	5.21	8.46	23.76
··▲··L-R	4.72	8.39	21.79
-×- · L-FM	5.51	10.39	26.26
-✳- AM	6.29	14.4	36.6
—×—L-N	7.01	12.8	34.56
-+- PSS	9.11	17.61	50.07
···▪···AC	17.69	32.6	74.9

Fig. 6. Response times for different scheduling approaches and different workloads.

We can see that with the workload becoming heavier, our approaches, L-U1, L-U2, L-R, L-FM and L-N, show more obvious improvement over other approaches in response time, relative bounded response time and effective utilization. The improvement in response time of L-R increases from 48% to 56%, and the improvement in relative bounded response time of L-R increases from 50% to 66% compared to PSS. Thus, response time and relative bounded response time are approximately reduced to half by using our approach.



	Workload 1	Workload 2	Workload 3
L-U1	58.57	103.2	226.15
L-U2	69.32	98.34	243.88
L-R	47.62	86.68	207.25
L-FM	53.54	104.49	296.38
AM	61.91	144.04	369.4
L-N	93.66	149.84	373.33
PSS	100.93	180.49	613.72
AC	192.92	281.26	891.265

Fig. 7. Relative bounded response times for different scheduling approaches and different workloads.

Comparing our different LOMARC heuristics, all are pretty close to each other as regards response times. However, L-U1 performs slightly better than L-U2. L-U1 provides almost the same results as L-R for all workloads. The differences are more pronounced for the relative bounded response times. L-U2 is again worse than L-U1. Obviously, $U_{gain,1}$ provides the more adequate estimate. L-R is better than L-U1, especially for W1 where it is better by 19% whereas only better by 16% for W2, and by 8% for W3. To perform better as regards relative response times is the expected result for a metric focusing on them. Selecting simply the first match in L-FM is not too much worse if the workload is lighter (W1) but becomes worse than the workload becomes heavier where there are more choices to select the match but ignored by this approach. Response times are by 17% worse than L-R under W3 and relative bounded response times by 24%. Relating the performance to PSS, the improvement in response times of L-FM vs. PSS is 40% for W1, 41% for W2, and 48% for W3. The improvement in relative bounded response times is 47% for W1, 42% for W2, and 52% for W3. Thus, the much simpler heuristic provides still very good results. AM that only matches adjacent jobs in the queue is still doing significantly better than PSS but still significantly worse than L-R and also worse than L-FM, especially for heavier workloads. Considering scheduling on standard CPUs (L-N), LOMARC still provides significant improvements: as regards response times 23% for W1, 27% for W2, and 31% for W3. Relative bounded response times are improved by 7% for W1, 17% for W2, and 40% for W3.

The makespans for Workload 1 are about 10 weeks and are by only 5% improved by L-R vs. PSS. This indicates that there are often not enough jobs to fully utilize the machine. The improvement for Workload 2 is 20% and for Workload 33%.

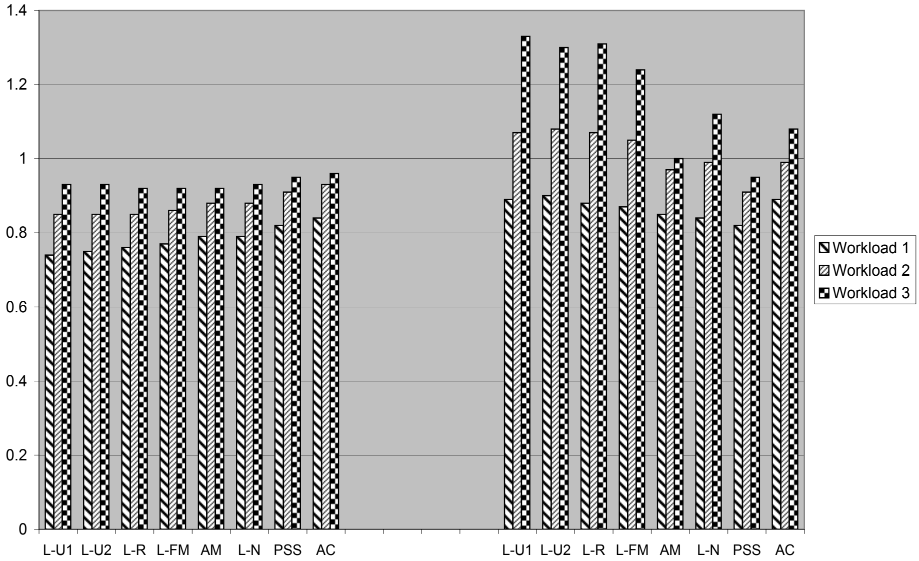


Fig. 8. Utilization (left) and utilization efficiency (right) for different scheduling approaches and different workloads.

In Fig. 8, we show utilization and utilization efficiency for all approaches. Utilization is almost the same for all approaches and for all approaches improves if the workload becomes heavier (because more options for packing exist). For utilization efficiency, LOMARC shows improvements, especially under heavier workloads, for L-U1, L-U2, and L-R: 8.5% for W1, 19% for W2, and 38% for W3. However, there are no relevant differences between L-U1, L-U2, and L-R. This means using a heuristic which focuses on utilization does not make any difference. L-FM is slightly worse – the improvement is 6% for W1, 15% for W2, and 31% for W3. L-N only accomplishes 2% improvement for L1, 9% for W2, and 18% for W3.

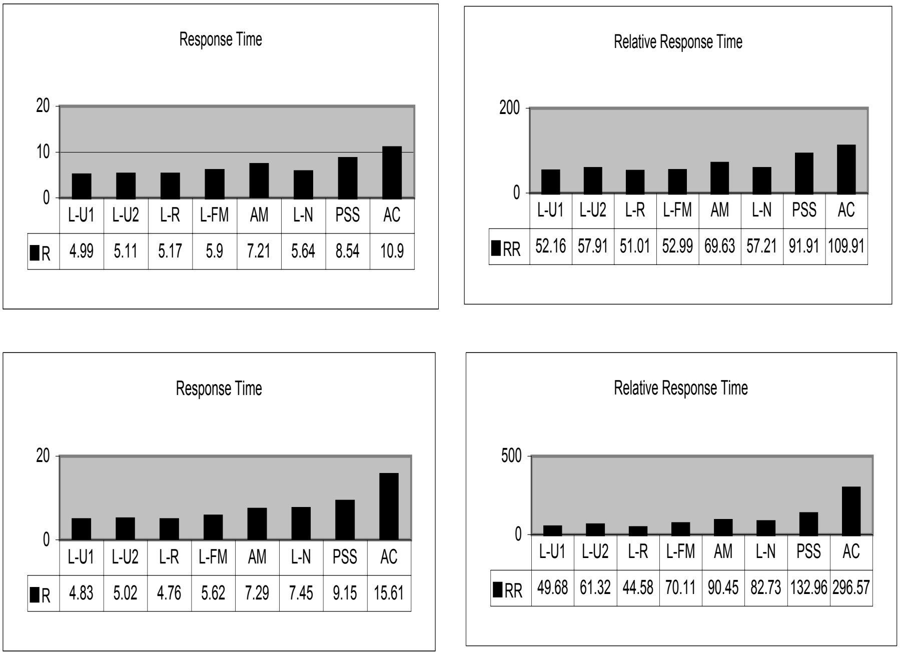


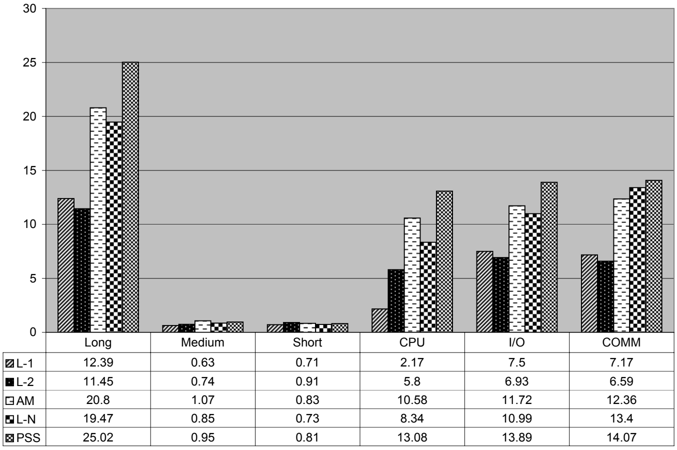
Fig. 9. Average response times and average relative bounded response times for M2 (upper row) and M3 (lower row).

To check how much difference the assumptions about the characteristics mix make, we present response times and relative bounded response times for Workload 1 and M2 and M3 in Fig. 9.

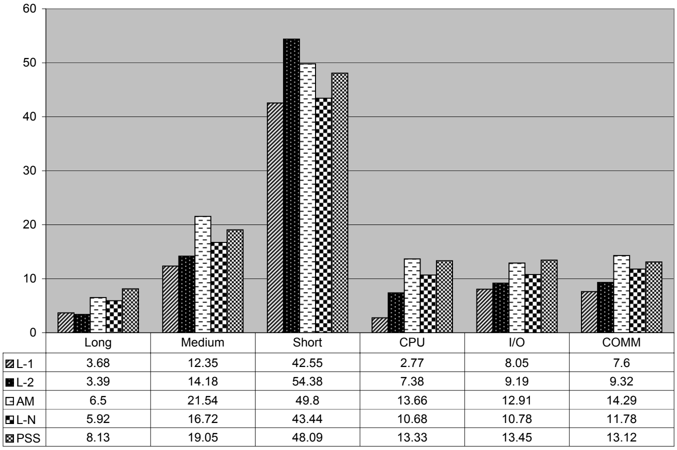
For M2 and M3, the relative improvements of L-U1, L-U2, and L-R vs. PSS and AC are similar to M1. However, for M2, L-N now improves upon PSS by 34% in response times and by 38% in relative bounded response times and for M3 it is closer to PSS than under M1. This is consistent with the expectation because in M2 there are more disk-bound jobs that can still be coscheduled with CPU-bound jobs and, in M3, there are fewer of them.

In Fig. 10, we show response times and relative bounded response times for different job classes. We see that LOMARC favors medium and long jobs due to its job-scheduling policy but creates no disadvantage for short jobs. CPU-intensive jobs benefit most because they have more opportunities for coscheduling.

Finally, we investigate the detailed behavior of L-U1, L-U2, and L-R (under M1) by looking at the average queue lengths, the number of jobs left in each comparison step for finding a match, and which job in the end is selected.



	Long	Medium	Short	CPU	I/O	COMM
▨ L-1	12.39	0.63	0.71	2.17	7.5	7.17
■ L-2	11.45	0.74	0.91	5.8	6.93	6.59
□ AM	20.8	1.07	0.83	10.58	11.72	12.36
◪ L-N	19.47	0.85	0.73	8.34	10.99	13.4
⊠ PSS	25.02	0.95	0.81	13.08	13.89	14.07



	Long	Medium	Short	CPU	I/O	COMM
▨ L-1	3.68	12.35	42.55	2.77	8.05	7.6
■ L-2	3.39	14.18	54.38	7.38	9.19	9.32
□ AM	6.5	21.54	49.8	13.66	12.91	14.29
◪ L-N	5.92	16.72	43.44	10.68	10.78	11.78
⊠ PSS	8.13	19.05	48.09	13.33	13.45	13.12

Fig. 10. Response times (top) and relative bounded response times (bottom) for different job classes.

See Table 5. As we can see, after meeting all the constraints, the number of jobs left as candidates to choose from by the different heuristics is relatively small: for W1 between 5 and 7. With this small number of choices, there is not much room for the different heuristics to create different effects. For all heuristics, on average the 3rd match candidate is selected. L-U1 and L-R select the 4th match candidate under Workload W2. For Workload 3, we see a significant difference: L-U1 selects the 6th job and L-R the 4th which is an expected effect as optimizing with a focus on response times should be more reluctant to select a job which has a position further down in the queue. However, the results for response times and relative bounded response times as discussed above do not really confirm this as the actual improvement of L-R is higher for W1.

Table 5. Average queue lengths, average numbers of jobs left under the different constraints, and average job selected for candidates.

		Average Queue Length	Medium or Long Job	$SizeB \leqq SizeA$	Memory Fit	Matchable	$Slowdown \leqq Max$	Number Selected
L-U1	W1	36	24	10	8	6	5	3
	W2	81	45	21	12	10	8	4
	W3	213	86	40	20	16	14	6
L-U2	W1	40	25	11	8	6	5	3
	W2	77	49	25	16	12	11	5
	W3	233	85	39	21	16	14	7
L-R	W1	36	24	13	11	8	7	3
	W2	80	51	31	21	15	14	4
	W3	222	87	48	25	20	17	4

7 Summary and Conclusion

We have presented an approach to find matches between two jobs on hyperthreaded and standard CPUs for better resource utilization via coscheduling. The approach partially reorders the queue and searches for the best match while estimating impacts on relative bounded response times and utilization. In simulations, we have shown that our LOMARC scheduler clearly outperforms standard space sharing as regards response times and relative bounded response times by reducing them to about half their original value on hyperthreaded CPUs and to about 3/4 on standard CPUs. The heuristic performing best is to estimate the response-time impact when selecting the best match. The improvement is accomplished by an improvement in utilization efficiency from running multiple jobs with complementary resource requirements. Each individual application is unlikely to accomplish the same internally, especially if the application does not use multithreading per CPU but simply doubles the number of processes.

Future work includes a more detailed investigation of hyperthreading behavior, a refined slowdown model, experiments with other simplified heuristics (like making the choice between the first three candidates only or selecting a candidate if it is beyond a certain match threshold), and testing the scheduler with conservative backfilling which may be more sensitive to whether utilization or response-time impact is considered. Furthermore, extension to multi-way nodes is of interest. Then, another choice is to schedule one or multiple applications on the different CPUs per node. For such nodes, applications are more likely to be prepared to use multithreading per node and may already use the network very intensively. Thus, there may be fewer options for coscheduling as regards network usage but also new options in using physical and virtual CPUs.

Acknowledgements

This research was partially supported by NSERC and by CFI (Grant No. 6191) with contributions from OIT and IBM. Thanks to Xuemin Huang for providing a basic version of the job-scheduling simulator.

References

1. Dusseau, A., Arpaci, R., and Culler, D.E.: Implicit Scheduling – Efficient Distributed Scheduling for Parallel Workloads on Networks of Workstations. Proc. SIGMETRICS Conf. Measurement and Modelling of Computer Systems, Philadelphia/PA, USA (1996)
2. Batat, A., Feitelson, D.G.: Gang Scheduling with Memory Considerations. Proc. IPDPS (2000)
3. Behr, P., Pletner, S., and Sodan, A.C.: The PowerMANNA Architecture. Proc. IEEE Conf. on High Performance Computer Architecture (HPCA), Toulouse, France (2000) 277–286.
4. Chiang, S.-H., Vernon, M.K.: Characteristics of a Large Shared Memory Production Workload. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (2001)
5. Cirne, W., Berman, F.: When the Herd is Smart: Aggregate Behavior in the Selection of Job Request. IEEE Trans. on Parallel and Distributed Systems **14(2)** (Feb. 2003)
6. Feitelson, D.G.: Job Scheduling in Multiprogrammed Parallel Systems, Extended Version. Technical Report, IBM, August 1997, RC 19790 (87657)
7. Figueira, S.M., Berman, F.. A Slowdown Model for Applications Executing on Time-Shared Clusters of Workstations. IEEE Transactions on Parallel and Distributed Systems **12(6)** (June 2001)
8. Frachtenberg, E., Feitelson, D., Petrini, F., and Fernandez, J. : Flexible CoScheduling – Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. Proc. Int. Parallel and Distributed Processing Symposium (IPDPS), Nice, France (2003)
9. Gibbons, R.A.: Historical Application Profiler for Use by Parallel Schedulers. Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Lecture Notes in Computer Science 1291, Springer Verlag (1997)
10. Leinberger, W., Karypis, G., and Kumar, V.: Job Scheduling in the Presence of Multiple Resource Requirements. Proc. IEEE/ACM Supercomputing Conf.(SC), Seattle/WA, USA (1999)
11. Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., and Rooholamini, R.: An Empirical Study of Hyper-Threading in High Performance Computing Clusters. Linux HPC Revolution (2002)
12. Lublin, U., Feitelson, D.G.: The Workload on Parallel Supercomputers – Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing **63(11)** (Nov. 2003) 1105–1122
13. Magro, W., Peterson, P., and Shah, S.: Hyper-Threading Technology: Impact on Compute-Intensive Workloads. Intel Technology Journal Q1 **6(1)** (2002)
14. Marr, D., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., and Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q1 **6(1)** (2002)

15. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., and Newhall, T.: The Paradyn Parallel Performance Measurement Tools. IEEE Computer, Special issue on performance evaluation tools for parallel and distributed computer systems **28(11)** (Nov. 1995) 37–46
16. Moreira, J.E., Chan, W., Fong, L.L., Franke, H., and Jette, M.A.: An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. Supercomputing'98, Nov. 1998.
17. Nagar, S., Banerjee, A., Sivasubramaniam, A., and Das, C.R.: A Closer Look at Coscheduling Approaches for a Network of Workstations. Proc. ACM SPAA. Saint Malo, France (1999)
18. Nakajima, J., Pallipadi, V.: Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling. Proc. USENIX 2nd Workshop on Industrial Experiences with Systems Software, Boston/MA, USA (Dec. 2002)
19. Nikolopoulos, D.S., Polychronopoulos, C.D.: Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs. Proc. International Parallel and Distributed Processing Symposium (IPDPS), Fort Lauderdale/CA, USA (April 2002)
20. Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems. Proc. 3rd Intl. Conf. Distributed Comp. Systems (1982) 22–30
21. Setia, S., Squillante, M., and Naik, V.K.: The Impact of Job Memory Requirements on Gang-Scheduling Performance. Performance Evaluation Review (March 1999)
22. Shmueli, E., Feitelson, D.G.: Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (2003)
23. da Silva, F.A.B., Scherson, I.D.: Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler. Proc. 11th Symp. On Computer Architecture and High Performance Computing, Natal, Brazil (1999)
24. Sobalvarro, P.G., Pakin, S., Weihl, W.E., and Chien, A.A.: Dynamic Coscheduling on Workstation Clusters. Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (1998)
25. Sodan, A.C., Huang, X.: Adaptive Time/Space Sharing with SCOJO. Conf. on High Performance Computing Systems (HPCS), Winnipeg/Manitoba (2004)
26. Sodan, A.C., Riyadh, M.: Coscheduling of MPI and Adaptive Thread Applications in a Solaris Environment. Proc. IASTED PDCS, Cambridge/MA, USA (2002)
27. Sodan, A.C.: Loosely Coordinated Coscheduling in the Context of Other Dynamic Job Scheduling Approaches – A Survey. Concurrency & Computation: Practice & Experience. To appear.
28. SUN HPC ClusterTools 4 Performance Guide. SUN Microsystems, retrieved from http://www.sun.com/products-n-solutions/hardware/docs/Software/ (August 2001)
29. Talby, D., Feitelson, D.G.: Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling. Proc. IPPS (1999)
30. Tullsen, D., Eggers, S., and Levy, H.: Simultaneous Multithreading – Maximizing On-chip Parallelism. Proc. Ann. Int. Symp. on Computer Architecture (ISCA) (1995)
31. Tullsen, D.M., Snavely, A.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2000)
32. Vtune Performance Analyzer. Intel Corporation, retrieved from http://www.intel.com (April 2004)
33. Wiseman, Y., Feitelson, D.G.: Paired Gang Scheduling. IEEE Trans. Parallel & Distributed Systems (2003)

34. Zhang, Y., Sivasubramaniam, A., Moreira, J., and Franke, H.: A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. Proc. Int. Conf. on Supercomputing (ICS), Santa Fe / NM, USA (2000)
35. Zhou, Y., Sodan, A.C.: Survey of Zero-Copy Optimization in User-level Communication and Adaptive Knowledge-Based Solution. Conf. on High Performance Computing Systems (HPCS) (2004)

Author Index