Walfredo Cirne
Narayan Desai
Eitan Frachtenberg
Uwe Schwiegelshohn (Eds.)

LNCS 7698

Job Scheduling Strategies for Parallel Processing

16th International Workshop, JSSPP 2012
Shanghai, China, May 2012
Revised Selected Papers

Springer

Walfredo Cirne Narayan Desai
Eitan Frachtenberg Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

16th International Workshop, JSSPP 2012
Shanghai, China, May 25, 2012
Revised Selected Papers

Springer

Volume Editors

Walfredo Cirne
Google
1600 Amphitheater Parkway
Mountain View, CA 94043, USA
E-mail: walfredo@google.com

Narayan Desai
Argonne National Laboratory
Mathematics and Computer Science Division
Bldg 240
Argonne, IL 60439, USA
E-mail: desai@mcs.anl.gov

Eitan Frachtenberg
Facebook Inc.
1601 Willow Road
Menlo Park, CA 94025, USA
E-mail: etc@fb.com

Uwe Schwiegelshohn
TU Dortmund
Robotics Research Institute
Otto-Hahn-Str. 8
44227 Dortmund, Germany
E-mail: uwe.schwiegelshohn@udo.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35866-1 e-ISBN 978-3-642-35867-8
DOI 10.1007/978-3-642-35867-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012954530

CR Subject Classification (1998): C.2.4, D.4, C.2, D.2, H.3-4, I.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at the 16th workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) that was held in Shanghai, China, on May 25, 2012, in conjunction with the IEEE International Parallel Processing Symposium 2012.

This year 24 papers were submitted to the workshop. All submitted papers went through a complete review process, with the full version being read and evaluated by an average of four reviewers. We would like to especially thank the Program Committee members and additional referees for their willingness to participate in this effort and their detailed, constructive reviews.

This workship saw the crystallization of a trend in the parallel scheduling landscape. In addition to papers discussing traditional JSSPP topics like parallel batch scheduling, workload analysis and modeling, and resource management system software studies, Web scheduling emerged as a new topic this year. This volume includes a paper summarizing Walfredo Cirne's keynote on Web-scale scheduling at Google. This paper provides a high-level overview of Web-scale scheduling workloads, as well as an approach that Google uses to meet service availability SLAs. The paper represents an early example of a broad class of scheduling problems impacting so-called "web scale" service providers.

In addition to this topic, scheduling issues were discussed in a broader context in more established areas, from hardware scheduling, to scheduling within budget constraints, scheduling for performance, and analysis of scheduling tasks within resource management software. Adopting a broader basis for scheduling discussions was an explicit goal this year for the workshop, and will be continued in future workshops.

A major goal of the JSSPP workshop is to explore new applications of scheduling approaches to novel scenarios. Many of papers presented this year attacked new schedule fitness metrics. In presenting their DEMB system, Lee et al. developed approaches to partition query parameter spaces in order to improve query locality in a distributed pool of servers. Wu et al. presented the Critical Path-base Priority Scheduling (CPPS) algorithm that improves end-to-end performance for DAG scheduling. Tian et al. demonstrated a series of algorithms that improve makespan of jobs on chip multiprocessors (CMPs).

In a more traditional parallel job scheduling vein, Klusáček et al. offered a detailed analysis of the fairness impact of a variety of popular scheduling algorithms. On the basis of this assessment, they have built an extension to conservative backfilling, using Tabu search, that performs explicit optimization for fairness as a part of the backfilling process. This approach represents an interesting addition of optimization into an often rigid phase of the scheduling process. Krakov et al. performed a high-resolution analysis of workload data, including the use of not only job workloads, but also the actual schedule on the system as well. This comparison shows several discrepancies between these schedules and the results of simulation with commonly used scheduling algorithms. This finding suggests that simulation results may not be quite as comparable to real system performance as previously assumed. Zhang et al. analyzed 12 months of workload data from the Kraken petascale system. Niu et al. tackled the problem of inaccurate runtime estimates through the application of checkpointing to aggressive backfilling. This approach results in greatly improved system performance with low checkpointing overhead. Zakay et al. comprehensively discuss several alternative approaches to determine user session boundaries in workload data. This problem is particularly important to supercomputing centers because user sessions signal modal shifts in user expectations for scheduling.

Two papers analyzed the performance of production-grade resources that managers often used on extreme scale systems. Georgiou et al. analyzed the performance of the SLURM resource manager using a variant of NERSC's ESP toolkit for load testing. This work highlighted the difficulties in conducting benchmarking at large scales, and presented several techniques to project performance using combinations of full-scale testing and emulation. Bresford et al. discussed improvements to the LoadLeveller resource manager needed to accomodate the Blue Waters system. In order to scale to a system of this size, resource allocation was reworked to be a hybrid distributed process where node local resources are allocated at the node level. This approach trades scalability for increased communication. This paper also provides a detailed case study of improving the scalability of a production resource manager, detailing several performance bottlenecks and respective mitigation strategies.

Over the last few years, it has become clear there is a profound shift occuring in the scheduling landscape. While supercomputer workloads remain a substantial consumer of scheduling technology, and are driving the field in terms of scheduling heterogenous resources and optimizing end-to-end behavior, it has become clear that these use cases are not the only ones in need of sophisticated scheduling approaches. Cloud schedulers that support interactive Web applications and services now manage larger quantities of resources in aggregate than traditional supercomputing centers and smaller HPC systems. As scheduling needs have expanded to include both interactive and hybrid workloads, new challenges have emerged. Therefore, we strongly believe that research in the field of this workshop will remain interesting and challenging for years to come.

The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, 2862, 3277, 3834, 4376, 4942, 5798, and 6253. Since 1995 these volumes have also been available online.

September 2012

Walfredo Cirne
Narayan Desai
Eitan Frachtenberg
Uwe Schwiegelshohn

Organization

Workshop Organizers

Walfredo Cirne	Google
Narayan Desai	Argonne National Laboratory, USA
Eitan Frachtenberg	Facebook
Uwe Schwiegelshohn	TU Dortmund University, Germany

Program Committee

Henri Casanova	University of Hawaii at Manoa, USA
Julita Corbalan	Technical University of Catalunya, Spain
Dick Epema	Delft University of Technology, The Netherlands
Dror Feitelson	The Hebrew University, Israel
Ian Foster	Argonne National Laboratory, USA
Alfredo Goldman	University of Sao Paulo, Brazil
Allan Gottlieb	New York University, USA
Morris Jette	SchedMD, USA
Rajkumar Kettimuthu	Argonne National Laboratory, USA
Derrick Kondo	INRIA, France
Zhiling Lan	Illinois Institute of Technology, USA
Virginia Lo	University of Oregon, USA
Satoshi Matsuoka	Tokyo Institute of Technology, Japan
Jose Moreira	IBM T.J. Watson Research Center, USA
Bill Nitzberg	Altair Engineering
Mark Squillante	IBM T.J. Watson Research Center, USA
Dan Tsafrir	Technion, Israel
John Wilkes	Google
Ramin Yahyapour	The University of Göttingen, Germany

Table of Contents

Web-Scale Job Scheduling

Walfredo Cirne[1] and Eitan Frachtenberg[2]

[1] Google
walfredo@google.com
[2] Facebook
etc@fb.com

Abstract. Web datacenters and clusters can be larger than the world's largest supercomputers, and run workloads that are at least as heterogeneous and complex as their high-performance computing counterparts. And yet little is known about the unique job scheduling challenges of these environments. This article aims to ameliorate this situation. It discusses the challenges of running web infrastructure and describes several techniques to address them. It also presents some of the problems that remain open in the field.

1 Introduction

Datacenters have fundamentally changed the way we compute today. Almost anything we do with a computer nowadays—networking and communicating; modeling and scientific computation; data analysis; searching; creating and consuming media; shopping; gaming—increasingly relies on remote servers for its execution. The computation and storage tasks of these applications have largely shifted from personal computers to datacenters of service providers, such as Amazon, Facebook, Google, Microsoft, and many others. These providers can thus offer higher-quality and larger-scale services, such as the ability to search virtually the entire Internet in a fraction of a second.

This transition to very large scale datacenters presents an exciting new research frontier in many areas of computer science, including parallel and distributed computing. The scales of compute, memory, and I/O resources involved can be enormous, far larger than those of typical high-performance computing (HPC) centers [12]. In particular, running these services efficiently and economically is crucial to their viability. Resource management is therefore a first-class problem that receives ample attention in the industry. Unfortunately, little is understood about these unique problems outside of the companies that operate these large-scale datacenters. That is the gap we hope to address here, by discussing in detail some of the key differentiators of job scheduling at Web scale.

Web datacenters differ from traditional HPC installations in two major areas: workload and operational constraints. Few of the HPC workloads provide interactive services to human users. This is in stark contrast with datacenters that serve Web companies, where user-facing services are their *raison d'etre*. Moreover, a sizable part of the HPC workload consists of tightly coupled parallel jobs

[15] that make progress in unison (e.g., data-parallel MPI 1.0 applications). Web workloads, on the other hand, are less tightly coupled, as different tasks of a job typically serve different requests.

This difference in the amount of coupling of the applications affects the guarantees that the resource manager must deliver in each environment. Web applications typically do not require all nodes to be available simultaneously as a data-parallel MPI job does. In this sense, they are much closer to HPC high-throughput applications [7] and can better tolerate node unavailability. Unlike high-throughput applications, however, they do not finish when the input is processed. They run forever as their input is provided on-line by users.

Of course, not all workloads in Web clusters are user-facing. There are also batch jobs to enable the user-facing applications. Google web search, for example, can be thought of as three different jobs. The first crawls the web, the second indexes it, and the third serves the query (i.e., searches for it in the index). These jobs need different levels of Quality-of-Service (QoS) guarantees to run reliably, with the user-facing query-serving job having more stringent requirements. Identifying the QoS levels that different applications demand is key in the process of managing Web datacenters, as providing different Service Level Agreements (SLAs) allows for better utilization and gives flexibility to cluster management.

Moreover, the fact that a key fraction of the Web workload is user-facing has even greater significance than the QoS level: it also changes the operational requirements for the datacenter. As HPC jobs essentially run in batch mode, it is feasible for datacenter administrators to shut it down as needed, either entirely or partially. Jobs get delayed, but since maintenance is infrequent and scheduled, this is acceptable. Web companies, in contrast, never intentionally take down their services entirely. Consequently, a datacenter-wide maintenance of either hardware or software is out of question.

This paper discusses these differences in detail in Sections 2 (workload and SLAs) and 3 (operational requirements). Section 4 provides a case study of how these issues affect cluster management. It describes in detail a technique called *backup tasks* that is used by Google to assure a higher QoS for its user-facing application, running in a cluster shared with batch load. Section 5 discusses other issues in Web cluster management that, while not the focus of this paper, are relevant and important. Section 6 concludes the paper.

2 A Spectrum of SLAs

Traditional supercomputers, grids, and HPC centers can have a relatively rigid set of SLAs. Consider, for example, job reservation systems such as queue-based space-sharing schedulers [10]. Some of these schedulers can have varying and complex degrees of job priorities (e.g., Maui [14]). But the basic SLA remains the same: the system offers a firm commitment that all resource needed by a job will be available during its execution, usually ignoring the inevitable possibility of system failures and downtime. Ignoring the possibility of failure may be reasonable for some smaller clusters of commodity servers, or for some high-availability, high-end servers in supercomputers. But usually, more than by a hardware promise, it is driven by a software requirement: Many HPC jobs rely on the MPI-1 protocol for their communication (and other similar protocols) which assume that all tasks in a job remain available through the completion of a job. A failure typically means that the entire job has to restart (preferably from a checkpoint).

Schedulers for Web Clusters cannot make this assumption. The sheer scale of datacenter clusters, coupled with commoditized or mass-produced hardware, make failures a statistical certainty. But the applications are also designed for weaker guarantees. Instead of the tightly synchronous communication of parallel jobs, most Web jobs are distributed in nature, designed for fault-tolerance, and sometimes even completely stateless, so that the replacement of one faulty node with an identical node is virtually transparent to software. This leads to a primary distinction between traditional SLAs and Web SLAs: Accounting for non-deterministic availability of resources means that the resource manager makes probabilistic guarantees, as opposed to the absolute (albeit unrealistic) guarantees of supercomputers. An example of such a probabilistic guarantee can be: "this job will have a 99.99% probability of always receiving 1,000 nodes within a minute of requesting them."

This distinction, combined with the modal characteristics of Web workloads [12,21], can also mean that Web jobs can be much more varied when it comes to QoS requirements, representing an entire spectrum of responsiveness (Fig. 1). The following breakdown enumerates some common types of SLA requirements for distributed Web jobs. Note that these are examples only, and not all workloads consist of all categories, or with the same prioritization.

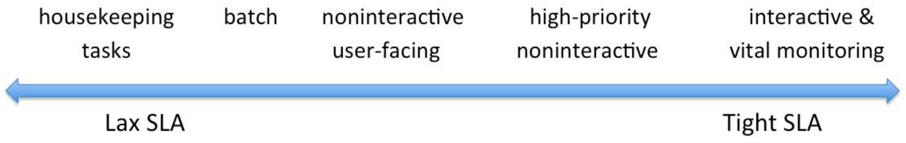

housekeeping tasks
batch
noninteractive user-facing
high-priority noninteractive
interactive & vital monitoring
Lax SLA
Tight SLA

Fig. 1. Spectrum of SLA requirements at Web clusters

- **Monitoring Jobs.** There are a small number of jobs whose sole purpose is to monitor the health of the cluster and the jobs running on it. These jobs demand very good SLAs, as their unavailability leaves systems administrators "flying in the dark".
- **User-Facing Services.** These so-called "front-end" services are what the user interacts with and represent the organization's highest priority, and therefore highest required QoS. Examples of front-ends are the search page for Google, the news feed for Facebook, and the store front for Amazon. The speed of interaction directly affects both user experience and revenue; so a high emphasis is placed on avoiding delays caused by lack of resources. Guarantees for this QoS level may include a certain minimum number of dedicated machines, or a priority to override any other services when sharing resources.

- **High Priority Noninteractive Services.** An example of such a service may be to sum up the number of clicks on an advertisement. Although not user-facing, this is a short, high-priority job, and may require guarantees such as the sum value in the database must be accurate to within the past few minutes of data. Managing these jobs, for example, may be done by servicing them on the same user-facing nodes that create these tasks, but deferring their processing past the point that the page has been presented to the user.
- **Noninteractive User Facing Services.** These are services that affect the user experience, but are either too slow to designate as interactive (in the order of many seconds to minutes) or not important enough to require immediate, overriding resources. One example of such a service builds the list of people recommendations on a social network ("People You May Know"). Whenever a user decides to add or remove a connection to their network, their list of recommended connections may require updates, which may take several seconds to compute. Although the list must be updated by the next time the user logs in, there is no need to require the user to wait for this update. Tasks for such jobs can be either scheduled on the same user-facing nodes at a lower local priority (“nice” value) [27], or on a different tier of dedicated servers, with separate queues for different SLAs.
- **Batch Jobs.** Long jobs that do not directly affect the user experience, but are important for the business. These include MapReduce and Hadoop jobs, index building, various analytics services, research and experimentation, and business intelligence. Like batch jobs on traditional supercomputers, these jobs can vary in SLA from strong guarantees to no guarantees whatsoever. A special class of jobs may have recurring computation with hard completion deadline (but otherwise no set priority). An analogue for such jobs in traditional HPC is the daily weather forecasts at ECMWF [13].
- **Housekeeping Tasks.** These can be various low-impact logging services, file-system cleanup, diagnostic metric collection, cron jobs, etc. These may be run with different variations on all active nodes in the system, at the lowest possible priority or at scheduled local downtimes. However, we must ensure they do not starve.

While the enumeration above does imply different levels of priority or SLA, the different categories overlap. For example, if disk is filling up very quickly, file-system cleanup will rise in priority and potentially take precedence over batch jobs or even user facing activities. In general, a given job is evaluated by its importance to the business and the minimal guarantees it needs, and an SLA-level is chosen.

Compute Node Sharing

Resource utilization is a very important economical and environmental concern, especially for large Web companies, due to the unprecedented scale of their clusters [12]. Maximizing utilization, in job scheduling, however, can be at odds with responsiveness and fairness [9,11,25]. In particular, Web schedulers have the choice of collocating jobs from different SLA tiers on the same compute nodes, or of allocating dedicated nodes for each desired tier.

Collocating different services on a single machine, even within virtual machines, can create hard-to-predict performance interactions that degrade QoS [19]. It can also complicate diagnosing performance and reliability issues, since there are more interacting processes and nondeterministic workload mixes. It cannot completely solve the performance problems of internal fragmentation either [12]. However, it allows for reclaiming unused resources that are allocated to guarantee we can deal with peak load, but that go unused most of the time.

Note that sharing compute nodes derives most of its appeal from the fact that different SLA tiers have varying requiremetns on resource availability. If a machine only runs high-SLA user-facing services, one cannot reclaim unused resources because they will only be needed at peak load. The existance of load that can be preempted (when user-facing services go through peak load) provides a way to use resources that would otherwise be wasted (when user-facing services are not at peak load).

Accounting

Another economic consideration for Web scheduling is accurate accounting and charging for resources while providing the agreed-upon SLAs [29]. This may not be an issue when all the services are internal to the same company with a single economic interest (such as Facebook and Google), but it is a critical issue for shared server farms, such as Amazon's EC2 cloud [1]. Part of the issue is the need to guarantee SLAs in the cloud in the presence of resource sharing and different VMs, as discussed in the previous paragraph. Trust is another aspect of the problem: How can the consumer verify if the resources delivered indeed have the correct SLA, if it is running within the provider datacenter (and thus in principle vulnerable to the provider's manipulation).

3 Operational Requirements

Cluster operation and job scheduling are two inseparable concepts for Web services. Whereas supercomputers can have scheduled full- or partial-downtime for maintenance (or even unscheduled interruptions), and then merely resume execution of the job queue from the last checkpoint, Web services strive to remain fully online at all times. Site-wide maintenance is not an option. And yet failing hardware must be repaired and software upgraded. To minimize service disruption, this process must be carefully planned as part of overall job scheduling.

One approach for scheduled upgrades is to handle the upgrade task the same way as a computational task. The downtime is estimated and treated as a job requirement, to be scheduled concurrently with existing jobs while preserving their SLAs. To protect against software failures, many maintenance tasks require a method of gradually changing the deployed service while maintaining the SLAs of currently running jobs. For example, Facebook pushes a new version of its main front-end server—a 1.5GB binary—to many thousands of servers every day [23]. The push is carried out in stages, starting from a few servers and, if no problems are detected, exponentially grows to all servers. In fact, the first stage deployment to internal servers is accessible only to employees. This is used for a final round of testing by the engineers who have contributed code to this push. The second stage of the push is to a few thousand machines, which serve a small fraction of the real users. If the new code does not cause any problems either in resource usage or in functionality, it is pushed to the last stage, which is full deployment on all the servers. Note that Facebook's service as a whole is not taken down while the code is updated, but each server in turn switches to the new version. The push is staggered, using a small amount of excess capacity to allow for a small number of machines to be down for the duration of the binary replacement. The files are propagated to all the servers using BitTorrent, configured to minimize global traffic and network interruptions by exploiting cluster and rack affinity. The time needed to propagate to all the servers is about 20 minutes.

Some applications accumulate a lot of state (such as large in-memory key-value stores) and require further coordination with the resource manager for graceful upgrades. For example, the application could be notified of an impending software upgrade, save all of its state and meta-data to shared memory (shmem), shut down, and reconnect with the data after the upgrade, assuming the version is backwards compatible.

Another operational requirement that affects scheduling is the need to add and remove server nodes dynamically without interrupting running services [12]. The implication here is that both applications and the scheduler that controls them can dynamically adjust to a varying number of servers, possibly of different hardware specifications. This heterogeneity poses an extra challenge to resource planning and management, compared to the typical homogeneous HPC cluster.

Yet another scheduling consideration that can affect both SLAs and resource utilization, especially in the cloud, is security isolation [24]. Often, jobs that are security sensitive, e.g., data with Personally Identifiable Information (PII) or financial information (credit card data, etc.) must be segregated from other jobs. Like SLAs, this is another dimension that is easier to manage with static resource allocation, as opposed to dynamically allocating entire machines to different jobs (due to issues of vulnerability explotation by previous jobs).

As mentioned in Sec. 2, one of the main philosophical differences between Web clusters and HPC clusters and supercomputers, is that Web schedulers always assume and prepare for the possibility of node failures [12]. In practice, this means that schedulers must keep track of job progress against available nodes, and automatically compensate for resources dropping off. One famous example of a fault-tolerant resource management system for Web services is Map-Reduce [8] and its open-source counterpart, Hadoop [28]. With these systems, tasks on failed machines are automatically respawned, and users have a strong expectation that their job will complete even in the face of failures, without their intervention.

An overriding principle in the operation of Web clusters and job scheduling is that it is almost always better not to start a new service than to bring down an existing one. In particular, if one wants to update a service and the new version is going to take a greater resource footprint than the current one, one must be able to answer in advance the question: is the update going to succeed? If not, admission control should reject the update.

The case study presented in the next section illustrates these points by describing one way in which resources can be scheduled dynamically for new services without interrupting existing SLAs.

4 Case Study: Guarantees for User-Facing Jobs

In the mix of jobs that run in the cloud, user-facing applications require the highest level of service. They require guarantees both at the machine level as well as the cluster level. Machine-level guarantees ensure how fast an application has access to resources in a machine it is already running, whereas cluster-level guarantees establish how fast an application has access to a new machine in case of need (e.g. raising load, or an existing machine failed).

Machine-level guarantees are needed because low latency adds business value. These applications must respond to user requests very quickly, typically within a fraction of a second. Therefore, they must have access to the CPU in very short order when they become ready to run. Likewise, we need to ensure appropriate performance isolation from other applications [19]. The easiest way to provide these guarantees is dedicating machines to user-facing jobs. Alas, this comes at the expense of lower utilization. There is therefore an incentive for finding ways to share machines between user-facing and less-stringent load while keeping strong QoS for user-facing jobs.

But machine-level guarantees are not enough per se. We must also ensure that these applications have enough resources available in spite of hardware failures and maintenance disruptions. As with machine-level guarantees, the easiest way to accomplish this is to dedicate a set of machines to the application at hand, over-provisioning to accommodate failures and disruptions. Keep in mind, however, that failures can be correlated (switch and power element nodes bring down a set of machines) and even with the dedicated assignment of machines, this is not a trivial problem.

Cluster-Level Guarantees at Google

Google cluster management does share machines between user-facing and background jobs to promote utilization. A jobs is composes by a set of tasks. The cluster-level guarantee we provide is that we will (within a probabilistic threshold) restart a task if its machine (or other hardware element) fails. In order to do so, we need to ensure we have enough spare resources to withstand hardware failures. This is accomplished by doing admission control based on *backup tasks*. The basic idea is that we only admit a user-facing job that we are confident we have enough spare resources to keep it running in face of failures and disruptions. It is better not to start a new service, as it by definition has no users to affect, than to disrupt an existing service.

To provide the guarantee that we have enough resources we need to allocate spare (backup) resources that cannot be allocated to another service job. Explicit allocation of the spare resources is needed to prevent incoming jobs (which may schedule fine) compromising the backup resources that are providing guarantees to existing jobs. Background jobs can use the backup resources, but they will be preempted if the user-facing jobs need them.

Therefore, the issue becomes how to allocate (the minimum amount of) resources to ensure we can withstand a given failure/disruption rate. We do so by scheduling backup tasks. A backup task is just a placeholder. It does not execute any code. It just reserves resources to ensure we have a place to restart real tasks in case of failure. A key feature of a backup task is that it reserves enough resources for any real task it is protecting. For example, if a backup task **B** is protecting real tasks **X** (requiring RAM = 1GB, CPU = 1.0) and **Y** (requiring RAM = 100MB, CPU = 2.0), **B** must at least reserve RAM = 1GB, CPU = 2.0. Likewise, any constraint required by a protected real task (e.g., the machine needs a public IP) must also apply to its backup tasks.

We define a *task bag* as the basic abstraction to reason about which real tasks are being protected by which backup tasks. A task bag is a set of tasks with the following properties:

1. All backup tasks in the bag are identical.
2. All real tasks in the bag can be protected by any backup task in the bag (i.e., their requirements are no bigger than the backup tasks').

Note that task bag is an internal concept of the cluster management system, being completely invisible to the user. The user aggregates tasks into jobs to make it convenient to reason about and manage the tasks. The cluster management system aggregates tasks into bags to reason about failure/disruption protection. Fig. 2 illustrates the concept with an example.

As the backup tasks in a bag have enough resources to replace any real task in the bag, as long as we do not lose more tasks than the number of backups, we will always have resources for the real tasks in the bag. That's basically the observation we rely on when doing admission control. A job is admitted if we can place its tasks in existing bags (or create new ones) and the probability of any bag losing more than its number of backups is below a threshold T. That is, a bag containing r real tasks and b backup tasks is said to be T*-safe* if the probability os losing more than b tasks (real or backup) is below T.

When Is a Task Bag Safe?

We determine if a bag is T-safe by computing the discrete probability distribution function of task failure in the bag. Such a computation takes into account correlated failures, as long as the failure domains nest into a tree. If failures

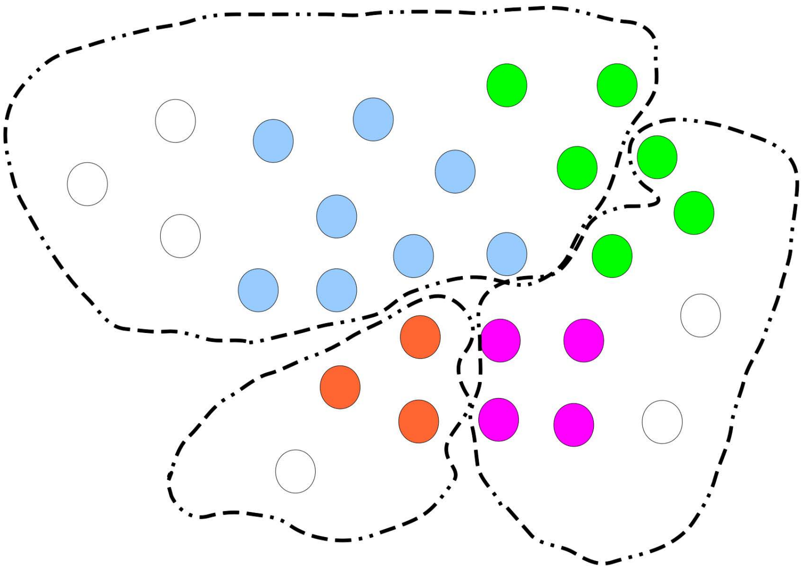


Fig. 2. Four jobs (in different colors) protected in three bags (backup tasks are in white). Note that bags are formed for the convenience of the cluster management system, which can divide tasks in any way it deems useful.

domains overlap, we approximate the actual failure graph by a tree. To the best of our knowledge, computing the pdf of task failure with overlapping failure domains is an open problem.

For the sake of explanation, we are going to assume that failures happen to machines and racks (which group a set of machines under a network switch). The generalization to more failure domains is straightforward, as we shall see. The input for our computation is the failure probabilities of each component that makes up the cluster, i.e., machine and racks in this explanation:

- $P(\mathbf{r})$ = the probability that rack $\mathbf{r}$ fails
- $P(\mathbf{m}|\neg\mathbf{r})$ = the probability the machine $\mathbf{m}$ fails but the rack $\mathbf{r}$ that holds $\mathbf{m}$ has not

Let's start with the case that all tasks in the bag are in the same rack $\mathbf{r}$ and that there is at most one task per machine. Say there are R tasks in the bag (i.e., the bag uses R distinct machines in rack $\mathbf{r}$). We seek the probability distribution function $P_{\mathbf{r}}(f = x)$, where f is the number of tasks to fail, given $P(\mathbf{r})$, $P(\mathbf{m}|\neg\mathbf{r})$, and R. Fortunately, this computation is straightforward:

- $P_{\mathbf{r}}(f > R) = 0$, as we cannot lose more tasks than we have.
- $P_{\mathbf{r}}(f = R) = P(\mathbf{r}) + P(\neg\mathbf{r}) \times PI_{\mathbf{r}}(f = R)$, i.e. we either lose the entire rack or individually lose all machines we are using.

- $P_{\mathbf{r}}(f = x < R) = P(\neg\mathbf{r}) \times PI_{\mathbf{r}}(f = x)$, as if we lose less than R machines we could not have lost the rack.
- $PI_{\mathbf{r}}(f = x) = Binomial(x, R, P(\mathbf{m}|\neg\mathbf{r}))$, where $PI_{\mathbf{r}}(f = x)$ is the probability we independently lose x machines.

Note that the computation described above gives us a discrete pdf. Therefore, to cope with multiple racks, we compute $P_{\mathbf{r}}(f = x)$ for each rack $\mathbf{r}$ the bag uses and use convolution to add them together to obtain $P(f = x)$ over the entire cluster. This is possible because we assume racks fail independently. Likewise, as long as greater failure domains nest nicely (e.g., racks are grouped in power nodes), we can separately compute their failure pdf and add them together.

Dealing with more than one task per machine uses the same approach. We devide each rack into in the set of machines with i = 1, 2, 3, ... tasks. We then compute $PI_{\mathbf{r}i}(f = x)$ for each “sub-rack” $\mathbf{r}i$ to obtain the machine failure pdf. As each machine runs i tasks, we multiply it by i to obtain the task failure pdf. Adding these pdfs together produces the task failure pdf for the rack. Fig. 3 exemplifies the process by showing a rack in which 2 machines run 1 task of the bag ($i = 1$), 5 machines run 2 tasks each ($i = 2$), and 1 machine runs 3 tasks ($i = 3$). Note that the number of task failures is multiplied by i as there are i per machine (e.g. $PI_{\mathbf{r3}}(f = 0)$ and $PI_{\mathbf{r3}}(f = 3)$ for $i = 3$).

As an illustration of the failure pdf, Fig. 4 shows the task failure pdf for a real bag containing 1986 tasks. One may be curious about the saw-shaped pattern in the pdf. This was caused by 664 machines that had an even number of tasks (2

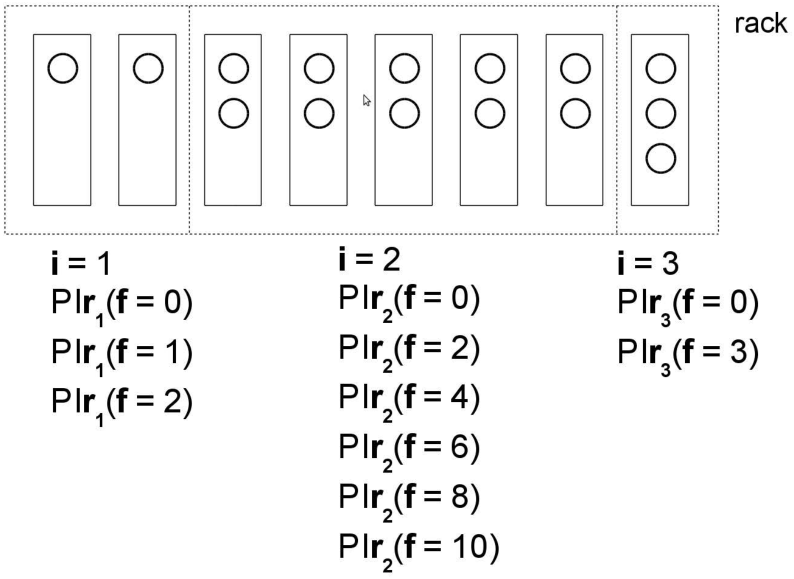


Fig. 3. A rack in which 2 machines run 1 task of the bag, 5 machines run 2 tasks each, and 1 machine runs 3 tasks

or 4 tasks per machine) whereas only 80 machines had an odd number of tasks (1 or 3 tasks per machine). This highlights another feature of the failure pdf: it is strongly influenced by where the tasks are placed.

The Bagging Process

Determining if a bag is T-safe runs in polynomial time. However, deciding how to group tasks into bags in an optimal way is NP-hard (optimality is defined as the grouping that minimizes the resources reserved by backup tasks.) In fact, this problem is akin the set cover problem [16]. We use an eager heuristic during the "bagging" process, used in admission control:

```
submission(job J):
   break J into a set of identical sub-jobs, each with identical tasks
   for each sub-job Ji:
      bag(Ji)
   if all bags are T-safe:
      commit the bagging of J
      accept J
   else:
      undo the bagging of J
      deny J
bag(Ji):
   for each existing bag B:
```

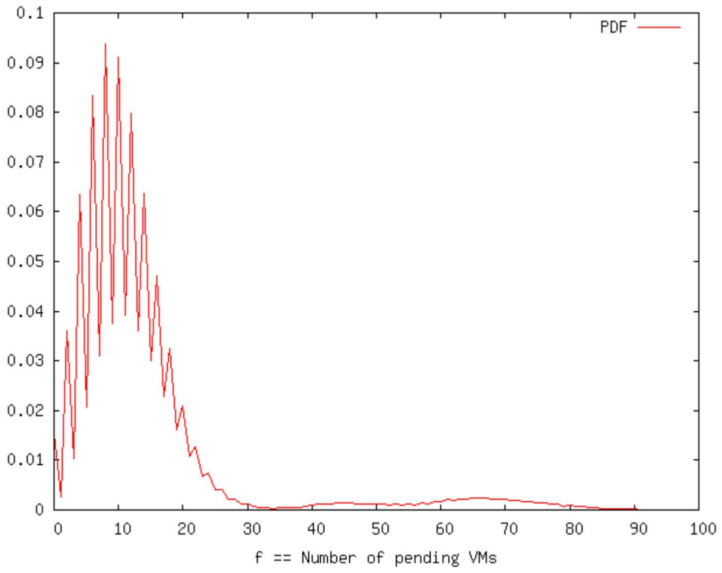


Fig. 4. The task failure pdf of a real 1986-task bag

```
      if it is possible to find enough backups for B + Ji:
         original_cost = BackupCost(B)
         add Ji to B (possibly creating new backup tasks and/or
            increasing the resources reserved by the backup tasks)
         new_cost = BackupCost(B)
         cost[B] = new_cost - original_cost
         restore B to its original form
   if it is possible to find enough backups for Ji to be in a new bag:
      cost[ALONE] = Kpenalty * BackupCost(Ji)
   if at least one solution was found:
      select the choice that gives the smallest cost
```

BackupCost() is a multidimensional quantity over all resources we allocate (CPU, memory, disk capacity, etc). As different dimensions have different scales, we compare two costs is done by multiplying together all dimensions. That is, $c_1 < c_2$ iff c_1[cpu] $\times$ c_1[memory] $\times$ ... $<$ c_2[cpu] $\times$ c_2[memory] $\times$ ...

$K_{penalty}$ is a penalty factor for us to create a new bag. It was introduced because having fewer, larger bags in principle reduces the resource consumption of backup tasks. Larger bags tend to achieve T-safety with a smaller fraction of backups because we cannot allocate a fractional backup task. For a simplistic example, assume that racks do not fail, machines fail with 1% probability, and there is always one task per machine. In this scenario, both a 1-task and an 100-task bag require 1 backup.

The bagging algorithm considers the topology of the failure domains when placing a backup task. In order to get the best failure pdf with a new backup task, we prefer to select a machine in a rack that is not yet used by the bag. If there is no such rack, we prefer a machine that is not yet used by the bag. Finally, when there is no such machine, we collocate the new backup task with other tasks already in the bag.

5 Other Issues

This exposition is by no means exhaustive of the challenges of scheduling at Web scale. Besides SLA diversity and strict operational requirements, there are many other dimensions in which Web-scale Job Scheduling poses unique challenges.

One of the most important open problems in resource management in general, but especially at Web scale, is how to increase power efficiency through job scheduling [12]. The load at large Web services varies considerably with diurnal and other patterns [4], but capacity is typically provisioned for peak load. Hosts that remain idle during non-peak hours can spend disproportionate amounts of energy [5]. What is the best use for these idle or nearly-idle machines? Should they be put to sleep? This may not be possible if they are also storage servers. Should they run low-priority batch jobs? But then what about the performance interference on the high-priority load? Should they not be acquired in the first place? But then how do we deal with peak load? Each possible solution has its own business tradeoffs. There is significant research work in progress in industry and academia to address this problem, but many challenges still remain [3,18,20,26].

Another interesting topic is the mixing of computational and storage resources in Web clusters. Supercomputers typically have their own storage infrastructure, often in the form of dedicated storage nodes with specialized hardware. The picture is less clear-cut for Web clusters that tend to use commodity hardware and often consume extremely large amounts of storage, in the order of many petabytes. For some services, such as Facebook's front-end Web servers, compute nodes are basically stateless, and all the persistent storage they require is served by specialized database and caching nodes. In other cases, such as Hadoop, the storage is collocated with the computation in the same nodes. The implications for scheduling are, how to assign jobs to nodes that would minimize storage-related bottlenecks [6,17]. Scheduling matters even for servers without persistent storage, if they use distributed in-memory storage or caching [2].

In addition to performance considerations, storage considerations also complicate other scheduling issues, such as operations (Sec. 3). Collocating storage with computation reduces the latitude for scheduling and other cluster management operations (e.g., they cannot just be powered down if other servers depend on their storage).

Finally, it is interesting to note that there is also cross pollination of ideas back from Web scheduling to HPC scheduling. One such idea is the provision of probabilistic SLAs, as suggested in the work on probabilistic backfilling [22].

6 Conclusion

It is hard to escape the realization that computer science evolves in cycles. Centralized mainframes were the common computing platform for the 1960s and 1970s. We later saw great decentralization of computation in the 1980s and 1990s with powerful personal computers. Now, owing to the sheer size of the problems we tackle and the increased scaling in management, we are again centralizing much of our computing infrastructure, this time in very large datacenters.

As in previous shifts in computer architecture organization, this trend opens a score of new problems and research opportunities. It appears, however, that much of this activity lies behind closed doors, in the R&D facilities of large Web companies. This paper hopes to shed some light on this activity, by exposing some problems and challenges, as well as describing some working novel solutions that have not been openly discussed yet.

The key aspects discussed were (1) the need and opportunities created by multiple SLA tiers, and (2) the stringent operational requirements of modern Web-scale datacenters. The backup task technology described in section 4 complements the overall discussion by illustrating how the technologies designed to tackle these challenges differ from those of the better studied HPC datacenters.

References

1. Ben-Yehuda, O.A., Ben-Yehuda, M., Schuster, A., Tsafrir, D.: Deconstructing amazon ec2 spot instance pricing. In: CloudCom 2011: 3rd IEEE International Conference on Cloud Computing Technology and Science (2011)
2. Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., Stoica, I.: PACMan: Coordinated memory caching for parallel jobs. In: Ninth USENIX Symposium on Networked Systems Design and Implementation, San Jose, CA (April 2012)
3. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: FAWN: a fast array of wimpy nodes. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 1–14. ACM, New York (2009), portal.acm.org/citation.cfm?id=1629577
4. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012, pp. 53–64. ACM, New York (2012)
5. Barroso, L.A., Hölzle, U.: The case for energy-proportional computing. IEEE Computer 40(12), 33–37 (2007), citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.5419
6. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., Aiyer, A.: Apache hadoop goes realtime at facebook. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 1071–1080. ACM, New York (2011)
7. Bricker, A., Litzkow, M., Livny, M.: Condor technical summary, version 4.1b. Technical Report CS-TR-92-1069 (January 1992), http://citeseer.ist.psu.edu/briker91condor.html
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
9. Feitelson, D.G.: Metrics for Parallel Job Scheduling and Their Convergence. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 188–1205. Springer, Heidelberg (2001)
10. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel Job Scheduling — A Status Report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
11. Frachtenberg, E., Feitelson, D.G.: Pitfalls in Parallel Job Scheduling Evaluation. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 257–282. Springer, Heidelberg (2005)
12. Frachtenberg, E., Schwiegelshohn, U.: New Challenges of Parallel Job Scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 1–23. Springer, Heidelberg (2008)
13. Holt, G.: Time-Critical Scheduling on a Well Utilised HPC System at ECMWF Using Loadleveler with Resource Reservation. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 102–124. Springer, Heidelberg (2005)
14. Jackson, D., Snell, Q., Clement, M.: Core Algorithms of the Maui Scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)

15. Jones, T., Tuel, W., Brenner, L., Fier, J., Caffrey, P., Dawson, S., Neely, R., Blackmore, R., Maskell, B., Tomlinson, P., Roberts, M.: Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In: 15th IEEE/ACM Supercomputing. ACM Press and IEEE Computer Society Press, Phoenix, AZ (2003), www.sc-conference.org/sc2003/paperpdfs/pap136.pdf
16. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103 (1972)
17. Kaushik, R.T., Bhandarkar, M.: Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In: Proceedings of the 2010 International Conference on Power Aware Computing and Systems, HotPower 2010, pp. 1–9. USENIX Association, Berkeley (2010)
18. Leverich, J., Kozyrakis, C.: On the energy (in)efficiency of hadoop clusters. SIGOPS Oper. Syst. Rev. 44(1), 61–65 (2010)
19. Mars, J., Tang, L., Hundt, R., Skadron, K., Souffa, M.L.: Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY, USA (2011)
20. Meisner, D., Gold, B.T., Wenisch, T.F.: Powernap: eliminating server idle power. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, pp. 205–216. ACM, New York (2009)
21. Mishra, A.K., Hellerstein, J.L., Cirne, W., Das, C.R.: Towards characterizing cloud backend workloads: insights from google compute clusters. SIGMETRICS Performance Evaluation Review 37(4), 34–41 (2010)
22. Nissimov, A., Feitelson, D.G.: Probabilistic Backfilling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 102–115. Springer, Heidelberg (2008)
23. Paul, R.: A behind-the-scenes look at Facebook release engineering (April 2012), http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/
24. Raj, H., Nathuji, R., Singh, A., England, P.: Resource management for isolation enhanced cloud services. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW 2009, pp. 77–84. ACM, New York (2009)
25. Sabin, G., Sadayappan, P.: Unfairness Metrics for Space-Sharing Parallel Job Schedulers. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 238–256. Springer, Heidelberg (2005)
26. Schroeder, B., Harchol-Balter, M.: Web servers under overload: How scheduling can help. ACM Trans. Internet Technol. 6(1), 20–52 (2006)
27. Sodan, A.C.: Adaptive Scheduling for QoS Virtual Machines under Different Resource Allocation – Performance Effects and Predictability. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 259–279. Springer, Heidelberg (2009)
28. White, T.: Hadoop: The Definitive Guide. Yahoo! Press, USA (2010)
29. Xiong, K., Suh, S.: Resource Provisioning in SLA-Based Cluster Computing. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 1–15. Springer, Heidelberg (2010)

DEMB: Cache-Aware Scheduling for Distributed Query Processing*

Junyong Lee[1], Youngmoon Eom[1], Alan Sussman[2], and Beomseok Nam[1,**]

[1] School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology,
Ulsan, 689-798, Republic of Korea
[2] Dept. of Computer Science
University of Maryland
College Park, MD 20742, USA
bsnam@unist.ac.kr

Abstract. Leveraging data in distributed caches for large scale query processing applications is becoming more important, given current trends toward building large scalable distributed systems by connecting multiple heterogeneous less powerful machines rather than purchasing expensive homogeneous and very powerful machines. As more servers are added to such clusters, more memory is available for caching data objects across the distributed machines. However the cached objects are dispersed and traditional query scheduling policies that take into account only load balancing do not effectively utilize the increased cache space. We propose a new multi-dimensional range query scheduling policy for distributed query processing frameworks, called *DEMB*, that employs a probability distribution estimation derived from recent queries. DEMB accounts for both load balancing and the availability of distributed cached objects to both improve the cache hit rate for queries and thereby decrease query turnaround time and throughput. We experimentally demonstrate that DEMB produces better query plans and lower query response times than other query scheduling policies.

Keywords: Multiple query optimization, Distributed query scheduling, Spatial clustering, Data intensive computing.

1 Introduction

As requirements for computing power increases and high-speed networks become more widespread, cluster computing is rapidly and widely accepted in various disciplines. For high performance data intensive computing applications, a large number of distributed and parallel query processing middleware systems have been developed and employed to solve large, complex scientific problems [14,8,15].

In distributed and parallel query processing systems, load balancing plays an important role to maximize overall system throughput by spreading the workload evenly

* This research was supported by National Research Foundation (2.110147.01) and MKE/KEIT (Ministry of Knowledge Economy) (2.120223.01) of Korea.
** Corresponding author.

across multiple servers. Besides load balancing, cache hit rate is another critical performance factor that must be accounted for to improve system throughput. However, prior distributed query scheduling policies do not take into consideration load balancing and cache hit rate at the same time. As more servers are added to the distributed query processing system, the amount of cache space available in the distributed servers increases linearly. But traditional scheduling policies such as round-robin or load-based scheduling policies do not consider cached objects that may be available in the distributed cache infrastructure, so get little benefit from the larger cache space. Hence we need more intelligent query scheduling policies in order to leverage the availability of a large number of cached objects in a distributed environment.

It is not an easy task to obtain both good load balancing and a high cache hit rate in modern heterogeneous cluster systems, especially if query patterns are also heterogeneous. Our earlier work [12] has shown that either a high cache hit rate with poor load balancing or good load balancing with low cache hit rate fails to maximize system throughput. Another difficult challenge in designing a distributed query scheduling policy is how to make a query scheduler know or predict the availability of cached objects in remote servers. Note that the query scheduling decisions are usually made in a remote front-end server, and that cached objects are evicted dynamically from remote servers, potentially at a high rate (as fast as new data objects are produced). Obviously it is not easy to keep track of cached objects in remote caches. Even if a query scheduler can keep track of them, the amount of information needed in the scheduler will become substantial as the number of servers increases, potentially making the scheduler a performance bottleneck. In order to make the scheduler scalable, the query scheduling policy must be designed to be lightweight.

In our previous work [12] we have proposed a statistical prediction-based query scheduling policy, called DEMA (Distributed Exponential Moving Average) that clusters queries on the fly to increase cache hit rate as well as to balance the query load across servers. The DEMA scheduling policy partitions a set of user queries and evenly distributes them across parallel servers to achieve load balance. DEMA has been shown to produce high cache hit rates since it preserves query locality by grouping similar queries and assigning them to the same server. If the distribution of queries through the query space is static, or slowly changes over time, the DEMA scheduling policy achieves good load balance, however if the query distribution changes rapidly or is extremely skewed, DEMA may suffer from load imbalance as we will discuss further in Section 3.1.

In this paper, we propose an alternative scheduling policy - DEMB (Distributed Exponential Moving Boundary) that overcomes the drawbacks of the DEMA scheduling policy by equally partitioning recent queries using a probability density estimation. Our experimental results show that the new scheduling policy outperforms prior query scheduling policies by a large amount and that it improves upon the DEMA scheduling policy, decreasing response time by up to 50%.

The rest of the paper is organized as follows. In Section 2 we discuss other research related to distributed query scheduling policies. In Section 3 we review the DEMA scheduling algorithm and its drawbacks. In Section 4 we introduce our new query scheduling algorithm, and discuss experimental results from simulations in Section 5. In Section 6 we conclude and discuss future work.

2 Related Work

Load-balancing problems have been extensively investigated in many different fields. Godfrey et al. [5] proposed an algorithm for load balancing in heterogeneous and dynamic peer-to-peer systems. Catalyurek et al. [3] investigated how to dynamically restore balance in parallel scientific computing applications where the computational structure of the applications change over time. Vydyanathan et al. [17] proposed scheduling algorithms that determine what tasks should be run concurrently and how many processors should be allocated to each task. Zhang et al. [20] and Wolf et al. [18] proposed scheduling policies that dynamically distribute incoming requests for clustered web servers. WRR (Weighted Round Robin) [7] is a commonly used, simple but enhanced load balancing scheduling policy which assigns a weight to each queue (server) according to the current status of its load, and serves each queue in proportion to the weights. However, none of these scheduling policies were designed to take into account a distributed cache infrastructure, but only consider the heterogeneity of user requests and the dynamic system load.

LARD (Locality-Aware Request Distribution) [1,13] is a locality-aware scheduling policy designed to serve web server clusters, and considers the cache contents of back-end servers. The LARD scheduling policy causes identical user requests to be handled by the same server unless that server is heavily loaded. If a server is too heavily loaded, subsequent user requests will be serviced by another idle server in order to improve load balance. The underlying idea is to improve overall system throughput by processing queries directly rather than waiting in a busy server for long time even if that server has a cached response. LARD shares the goal of improving both load balance and cache hit ratio with our scheduling policies, DEMA and DEMB, but LARD transfers workload only when a specific server is too heavily loaded while our scheduling policies actively predict future workload balance and take actions beforehand to achieve better load balancing.

In relational database systems and high performance scientific data processing middleware systems, exploiting similarity of concurrent queries has been studied extensively. That work has shown that heuristic approaches can help to reuse previously computed results from cache and generate good scheduling plans, resulting in improved system throughput as well as reducing query response times [8,12]. Zhang et al. [19] evaluated the benefits of reusing cached results in a distributed cache framework in a Grid computing environment. In that simulation study, it was shown that high cache hit rates do not always yield high system throughput due to load imbalance problems. We solve the problem with scheduling policies that consider both cache hit rates and load balancing.

In order to support data-intensive scientific applications, a large number of distributed query processing middleware systems have been developed including MOCHA [14], DataCutter [8], Polar* [15], ADR [8], and Active Proxy-G [8]. Active Proxy-G is a component-based distributed query processing grid middleware that employs user-defined operators that application developers can implement. Active Proxy-G employs meta-data directory services that monitor performance of back-end application servers and a distributed cache indexes. Using the collected performance metrics and the distributed cache indexes, the front-end scheduler determines where to assign incoming queries considering how to maximize reuse of cached objects [12]. The index-based scheduling policies cause higher communication overhead on the front-end scheduler, and the cache index may not predict contents of the cache accurately if there are a large number of queries waiting to execute in the back-end application servers.

3 Distributed Query Processing Scheduling Algorithms

Figure 1 shows the architecture of Active Proxy-G, a distributed and parallel query processing middleware for scientific data analysis applications. The front-end server runs a query scheduler that determines which back-end application server will process a given query. The homogeneous back-end servers retrieve raw datasets from networked storage systems and process incoming queries on cluster nodes. If the back-end servers are heterogeneous, the scheduling algorithms can be modified with appropriate weight factors.

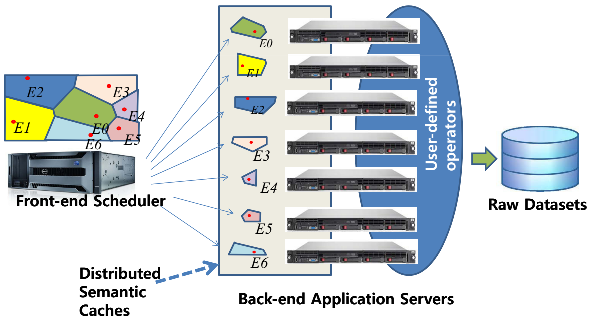


Fig. 1. Distributed Query Processing Framework with Distributed Semantic Caches

3.1 DEMA Scheduling Policy

Exponential moving average (EMA) is a well-known statistical method to obtain long-term trends and smooth out short-term fluctuations, which is commonly used to predict stock prices and trading volumes [4]. In general, EMA computes a weighted average of all observed data by assigning exponentially more weight to recent data. The formula that calculates the EMA at time t is given by

$$EMA_t = \alpha \cdot data_t + (1 - \alpha) \cdot EMA_{t-1} \tag{1}$$

where α is the weight factor, which determines the degree of weight decrease over time.

The *Distributed Exponential Moving Average (DEMA)* scheduling policy [12] estimates the cache contents of each back-end server using an exponential moving average (EMA) of past queries executed at that server. In the context of a multidimensional range query scheduling policy, we use the multidimensional center point of the query as $data_t$ in Equation 1.

Given that application servers replace old cache entries and the DEMA scheduling policy gives less weight to older query entries, the *smoothing factor* $\alpha \in (0,1)$ must be chosen so that it reflects the degree of staleness used to expunge old data. This implies that α should be adjusted based on the size of the cache space. For example, α should be 1 if a cache can contain only a single query result and α should be close to 0 if the size of the cache is large enough to store all the past query results. However, the number of cached objects in a server can be hard to predict when the sizes of cached objects can vary widely.

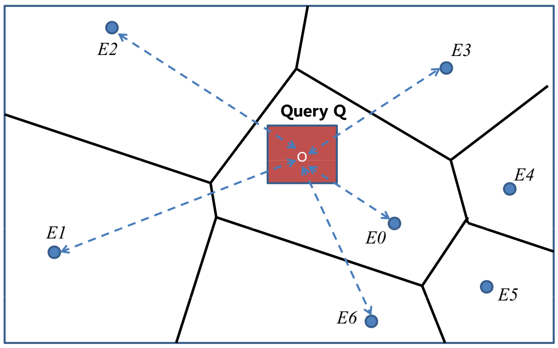


Fig. 2. The DEMA scheduler calculates the Euclidean distance between EMA points and an incoming query, and assigns the query to the server (0) whose EMA point is closest

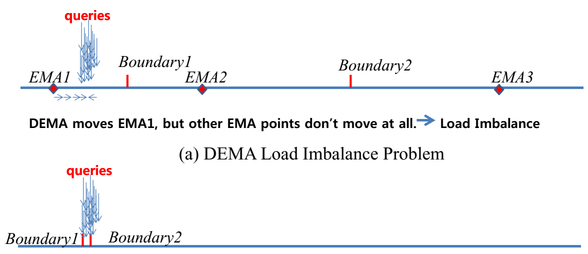
If we can keep track of both the number of current cache entries in each back-end server (k) and the last $k \cdot N$ query center points in the front-end scheduling server (where N is the number of back-end application servers), we can alternatively employ a simple moving average (SMA) instead of EMA, which takes the average of the past k query center points. SMA eliminates the weight-sum error and correctly represent the cache contents of remote back-end application servers. However, SMA does not quickly reflect the moving trend of arriving queries. Furthermore, it causes some overhead in the front-end server to keep track of the last $k \cdot N$ query center points.

In the DEMA query scheduling policy the front-end query scheduler server has one multi-dimensional EMA point for each back-end application server. For the 2 dimensional image, each query specifies ranges in the x and y dimensions, as shown in Figure 2. For an incoming query, the front-end server calculates the Euclidean distance between the center of the multidimensional range query and the EMA points of the N application servers, and chooses an application server whose current EMA point is the closest to that of the incoming query. Clustering similar range queries increases the probability of overlap between multi-dimensional range queries, and increases the cache hit rate at each back-end server. In Figure 2, the given 2 dimensional range query will be forwarded to server 0 since the EMA point of server 0 is closer to the query than any other EMA point. This strategy is called the *Voronoi assignment model*, where every multidimensional point is assigned to the nearest cell point. The query assignment regions induced from the DEMA query assignment form a *Voronoi diagram* [2].

After the query Q is assigned to the selected application server, the EMA point for that application server is updated as $EMA_{s^*} = \alpha Q + (1-\alpha)EMA_{s^*}$ (EMA_s represents the EMA point of the sth server). For every incoming query, one EMA point moves in the direction of that query, but we do not need to calculate the changing bisectors of the EMA points since DEMA scheduling algorithm compares the Euclidean distance between EMA points and a given query. The complexity of the DEMA scheduling algorithm is $O(N)$ where N is the number of back-end application servers. So the DEMA scheduling policy is very light-weight.

3.2 Load Balancing

The DEMA scheduling algorithm clusters similar queries together so that it can take advantage of cache hits. In addition to a high cache hit rate, the DEMA scheduling algorithm balances query workload across multiple back-end application servers by moving EMA points to hot spots.


queries
Boundary1
Boundary2
EMA1
EMA2
EMA3
DEMA moves EMA1, but other EMA points don't move at all. Load Imbalance
queries
Boundary1
Boundary2
(a) DEMA Load Imbalance Problem

(b) DEMB moves all the boundaries according to new query distribution

Fig. 3. DEMA Load Imbalance Problem

Ideally, we want to assign the same number of queries to back-end application servers with a uniform distribution. In DEMA, the probability of assigning a new query to a specific server depends on the query probability distribution and the size of the Voronoi hyper-rectangular cell (e.g., a range in a 1-dimensional line or area for a 2D space).

DEMA balances the server loads by trying to keep the region size inversely proportional to the probability that queries fall inside their region. For the 2D uniform distribution case, as shown in Figure 2, queries that fall inside the Voronoi region of server B's EMA ($Vor(B)$) are assigned to server B. We denote the Voronoi cell of server A's EMA as $Vor(A)$. The probability that a query arrives in a specific Voronoi cell $Vor(A)$ is proportional to the size of the $Vor(A)$. Thus, more queries are likely to land in larger cells than smaller cells for a uniform query distribution.

One important property of the DEMA scheduling policy is that an EMA point tends to move to the center of its Voronoi cell if queries arrive with a uniform distribution. In Figure 2, EMA point $E0$ is located in the lower right corner of $Vor(A)$. Since more queries will arrive in the larger part of the cell (i.e. the upper left part of $Vor(A)$), Equation 1 is likely to move the EMA point $E0$ to the upper left part of $Vor(A)$ with higher probability than to the lower right part, which will tend to move the $E0$ toward the center of the cell. That will result in moving the bisectors of $E2$ and $E0$ and the bisector of $E0$ and $E5$ to the left as well. Eventually, this property makes the size of $Vor(E2)$ decrease and the size of $Vor(E5)$ increase. As the size of large Voronoi cells becomes smaller and the size of small Voronoi cells becomes larger, the sizes of the Voronoi cells are likely to converge and effectively the number of queries processed by each back-end application servers will be similar. A similar argument can be made for higher dimensional query spaces.

A normal distribution is another commonly occurring family of statistical probability distributions. It is known that the sum of normally distributed random variables is also normally distributed [6]. If we assume that each historical query point $data_t$ is an individual random variable, the weighted sum EMA_t also should have a normal distribution. Hence the DEMA scheduling policy approximately balances the number of processed queries across multiple back-end application servers even when the queries arrive with a normal distribution.

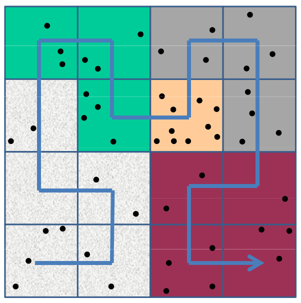
However the DEMA scheduling policy may suffer from temporary load imbalance if the query distribution changes quickly, with query hot spots moving through the query space randomly. Let us look at an extreme case. Suppose queries have arrived with a uniform distribution, and suddenly all the subsequent queries land in a very small region that is covered by a single Voronoi cell, as illustrated in Figure 3. In such an extreme case, only one corresponding EMA point will move around the new hot spot and the single server will have to process all the queries without any help from other servers. Although this may not commonly occur, we have observed that the DEMA scheduling policy suffers from failing to adjust its EMA points promptly. If a new query distribution spans multiple Voronoi cells, the EMA points slowly adjust their boundaries based on the new query distribution. However the time to adjust depends on the standard deviation of the query distribution. In the next section, we propose an alternative query scheduling policy that solves this load imbalance problem.

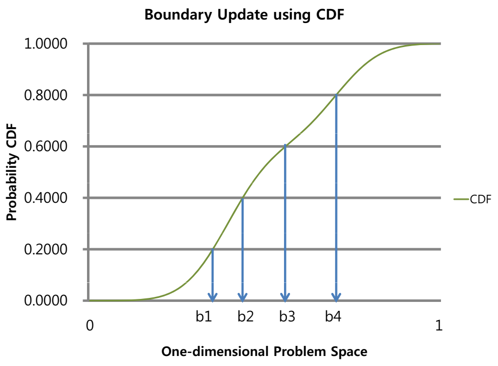
4 DEMB: Distributed Exponential Moving Boundary

To overcome the described weakness of the DEMA scheduling policy, we have devised a new scheduling policy called Distributed Exponential Moving Boundary (DEMB) that estimates the query probability density function using histograms from recent queries. Using the query probability density function, the DEMB scheduling policy chooses the boundaries for each server so that each has equal cumulative distribution value. The DEMB scheduling policy adjusts the boundaries of all the servers together, unlike DEMA, so that it can provide good load balancing even for dynamically changing and skewed query distributions.

In the DEMB scheduling policy, the front-end scheduler manages a queue that stores a predefined number (window size, WS) of recent queries, periodically enumerates those queries, and finds the boundaries for each server by assigning an equal number of queries to each back-end application server.

One challenge in the DEMB scheduling policy is to enumerate the recent multi-dimensional queries and partition them into sub-spaces that have equal probability. In order to map the multi-dimensional problem space onto a one dimensional line, we


(a) Mapping cache contents on Hilbert curve



(b) Calculating Boundaries using CDF

Fig. 4. Calculating DEMB boundaries on Hilbert curve

employ a Hilbert space filling curve [11]. A Hilbert curve is a continuous fractal space-filling curve, which is known to maintain good mapping locality (clustering), i.e. it converts nearby multi-dimensional points to close one-dimensional values.

Using the transformed one dimensional queries, we estimate the cumulative probability density function in the one-dimensional space and partition the space into N sub-ranges so that each one has the same probability as other sub-ranges. The front-end scheduler uses the sub-ranges for scheduling subsequent queries. When a query is submitted, the front-end scheduler converts the center of the query to its corresponding one-dimensional point on the Hilbert curve, determines which sub-range includes the point, and assigns the query to the back-end server that owns the sub-range.

Assigning nearby queries in one-dimensional sub-ranges takes advantage of the Hilbert curve properties. As shown in Figure 4(a), the one dimensional boundaries on the Hilbert curve cluster two dimensional queries so that they have good spatial locality. The DEMB scheduling policy achieves good load balance as well as a high cache hit rate since the scheduler assigns similar numbers of nearby queries on the Hilbert curve to each back-end server.

The DEMB scheduling policy is presented in Algorithm 1. When a new query is submitted to the scheduler, it is inserted into the queue and the oldest query in the queue is evicted, which can change the query probability density function. However if we update the boundaries of each server for every incoming query, that may cause too much overhead in the front-end scheduler. Instead, we employ a sliding window approach, where the scheduler waits for a predefined number of queries (**update interval**, UI) to arrive before updating the boundaries. Note that the update interval does not have to be equal to the window size. As the update interval is increased, the window size also has to be increased. Otherwise some queries may not be counted when calculating the query probability distribution.

In the DEMB scheduling policy, the window size WS is an important performance factor to estimate the current query probability density. As the window size becomes larger, the recent probability density function can be better estimated. However, if the query distribution changes dynamically, a large window size causes the scheduler to use a large number of old queries to estimate the query probability density function. As a result, the scheduler will not adapt to rapid changes in the query distribution in a timely manner. On the other hand, a small window size can cause a large error in the current query probability distribution estimate due to an insufficient number of sample queries. Moreover, if the window size is smaller than the size of the distributed caches, the query probability density estimation may not correctly reflect all the cached objects in the back-end servers, in addition to the moving trend of query distribution. Therefore choosing an optimal window size under various conditions is one of the most important factors when applying the DEMB scheduling policy to the distributed query processing framework.

In most systems the size of the distributed caches will be much larger than the front-end server's queue size WS. Instead of increasing the window size in order to reduce the error in the probability distribution estimate, we can calculate the moving average of the past query probability distributions, as for the DEMA scheduling policy.

After updating the query probability distribution, we choose the sub-range of each server so that each has equal probability. In Figure 4(b), the problem space is divided into 5 sub-ranges where 20% of the queries are expected for each one. However it is not practical to estimate the real probability distribution function because that requires a large amount of memory to store the histograms. Instead, we assume the query probability density function is a continuous smooth curve. Then we can make the probability density estimation process simpler. The scheduler will determine the boundaries of each server using the WS most recent queries, and apply the following equation to calculate the moving average for each boundary.

$$BOUNDARY[i]_t = \alpha \cdot CUR\_BOUND[i]_t + (1 - \alpha) \cdot BOUNDARY[i]_{t-1}$$

The weight factor α is another important performance parameter in the DEMB scheduling policy, as for DEMA. As described in Section 3.1, $alpha$ determines how earlier boundaries for each server will be considered for the current query probability distribution. However unlike the DEMA scheduling policy, the DEMB scheduling policy has two parameters to control how fast the old queries decay. One is α, and the other is the window size (WS). A large window size (WS) can be used to give more weight to the old queries instead of a low α value.

Now we show an example to see how the DEMB scheduling policy works. Suppose there are 10 back-end servers ($N = 10$), the window size is 500 queries ($WS = 500$), and the boundary update interval is 100 queries ($UI = 100$). The front-end scheduler will replace the oldest query in the queue with the newest incoming query every time a new query arrives. The incoming queries will be forwarded to one of the back-end servers using the boundaries of each server ($BOUNDARY[i]$). When the 100th query arrives, the scheduler recalculates the boundary for each server using the past 500 queries. Since there are 10 back-end servers, each server should process 50 queries to achieve perfect load balance. Hence, the new boundary between the 1st server and the 2nd server should be the middle point of the 50th query and the 51st query in the Hilbert curve ordering. Likewise, the new boundary ($CUR\_BOUND[i]$) between the

Algorithm 1. DEMB Algorithm

```
procedure
$ScheduleDEMB(Queryq)$
 1. INPUT: a client query $Q$
 2. $MinDistance \leftarrow MaxNum$
 3. $distance \leftarrow HilbertDistance(query)$
 4. for $s = 0 \to N - 1$ do
 5.    if BOUNDARY[s] is not initialized then
 6.       forward query $Q$ to server $s$.
 7.       $BOUNDARY[s] \leftarrow HilbertDistance(Q)$
 8.       return
 9.    else
10.       if $s = 0 \bigwedge distance \leq BOUNDARY[0]$ then
11.          $selectedServer \leftarrow 0$
12.       else if $BOUNDARY[s-1] < distance \bigwedge distance \leq BOUNDARY[s]$ then
13.          $selectedServer \leftarrow s$
14.       end if
15.    end if
16. end for
17. forward query $Q$ to $SelectedServer$.
18. if $QueryQueue.size() < WindowSize$ then
19.    $QueryQueue.enqueue(Q)$
20.    if $QueryQueue.size()\%N = 0$ then
21.       $UPDATE(QueryQueue)$
22.    end if
23. else
24.    $QueryQueue.dequeue()$
25.    $QueryQueue.enqueue(Q)$
26.    if $intervalCount = UpdateInerval$ then
27.       $UPDATE(QueryQueue)$
28.       $intervalCount \leftarrow 0$
29.    end if
30. end if
31. $intervalCount \leftarrow intervalCount + 1$
end procedure

procedure
$UPDATE(Queue\, QueryQueue)$
 1. INPUT: a queue that stores recent queries $QueryQueue$
 2. $LOAD \leftarrow QueryQueue.getSize()/N$
 3. for $i = 1 \to N - 1$ do
 4.    $CUR\_BOUND[i] \leftarrow (HilbertDistance(LOAD \times i) + HilbertDistance(LOAD \times i + 1))/2$
 5. end for
 6. for $i = 1 \to N - 1$ do
 7.    $BOUNDARY[i] \leftarrow alpha * CUR\_BOUND[i] + (1 - alpha) * BOUNDARY[i]$
 8. end for
end procedure
```

ith server and the $i + 1$th server should be the middle point of the $50 * i$th query and the $50 \cdot i + 1$th query. After calculating the new boundaries ($CUR\_BOUND[i]$) for the back-end servers, we compute the moving average for each boundary. If the query distribution has changed, $BOUNDARY[i]$ would move to $CUR\_BOUND[i]$. In this way, the DEMB scheduling policy makes the boundaries move according to the new query distribution.

The cost of the DEMB scheduling policy is determined by the number of servers and the level of recursion for the Hilbert curve, i.e. $O(N \cdot HilbertLevel)$. The complexity of the Hilbert curve is determined by the level of recursion in the Hilbert curve, which is usually a very small number. With a higher level of recursion, a Hilbert space filling curve can map a larger number of points onto it. For example, for a level 15 Hilbert curve about 1 billion (4^{15}) points can be mapped to distinct one-dimensional values. In our experiments, we set the level of the Hilbert curve to 15, and employed at most 50 servers. The cost of DEMB scheduling is very low, but is somewhat higher than that of DEMA, which is $O(N)$.

5 Experiments

5.1 Experimental Setup

The primary objective of the simulation study is to measure the query response time and system load balance of the DEMB scheduling policy with various query distributions. To measure the performance of query scheduling policies, we generated synthetic query workloads using a *spatial data generator*, which is available at [16]. It can generate spatial datasets with normal, uniform, or Zipf distributions. We have generated a set of queries with various distributions with different average points, and combined them so that the distribution and hot spots of queries move to different location unpredictably. We will refer to the randomly mixed distribution as the *dynamic distribution*. We also employed a Customer Behavior Model Graph (CBMG) to generate realistic query workloads [10]. CBMG query workloads have a set of hot spots. CBMG chooses one of the hot spots as its first query, and subsequent queries are modeled using spatial movements, resolution changes, and jumps to other hot spots. The query's transitions are characterized by a probability matrix. In the following experiments, we used the synthetic dynamic query distribution and a CBMG generated query distribution, with each of them containing 40,000 queries, and a cache miss penalty of 400 ms. This penalty is the time to compute a query result from scratch on a back-end server. We are currently implementing a terabyte scale bio-medical image viewer application on top of our distributed query scheduling framework. In this framework, a large image is partitioned into small equal-sized chunks and they are stored across distributed servers. Obviously the cache miss penalty is dependent on the size of the image chunks. When a cache miss occurs, the back-end server must read raw image file data from disk storage and generate an intermediate compressed image file at the requested resolution. The 400 ms cache miss penalty is set based on this scenario. We also evaluated the scheduling policies with smaller cache miss penalties, but the results were similar to the results described below.

5.2 Experimental Results - DEMB

Using various experimental parameters, we measured (1) the query response time, the elapsed time from the moment a query is submitted to the system until it completes, (2) the cache hit rate, which shows how well the assigned queries are clustered, and (3) the standard deviation of the number of processed queries across the back-end servers, to measure load balancing, i.e. lower standard deviation indicates better load balancing. In the following set of experimental studies, we focused on the performance impact of 3 parameters - window size (WS), EMA weight factor (α), and update interval (UI).

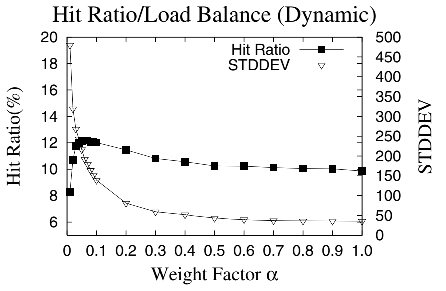
Weight Factor. The weight factor α is one of the three parameters that we can control to determine how fast the query scheduler loses information about past queries. The other two parameters are the *update interval* UI and *window size* WS. WS is the number of recent queries that front-end scheduler keeps track of. The update interval UI determines how frequently new boundaries should be calculated in the scheduler with weight factor α. As the weight factor α becomes larger, the scheduler will determine the boundaries using more recent queries. As UI becomes shorter, α will be applied to boundary calculations more frequently. Hence older queries will have less impact on the calculation of boundaries. Also, if WS is small, only a few of the most recent queries will be used to calculate the boundaries.

For the experiments shown in Figure 5 and 6, we employed 50 servers, and fixed the window size to a small number (200), i.e. only 4 recent queries are used to calculate the boundaries of each back-end server. Since the window size is small, we updated the boundaries of each server whenever 10 new queries arrive. These numbers are chosen to minimize the effects of the weight factor α on query response time.

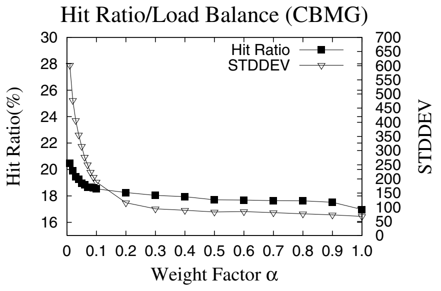
With smaller α, the boundaries move more slowly. If α is 0, the boundaries will not move at all. For both the dynamic query distribution shown in Figure 5(a) and the CBMG distribution shown in Figure 5(b), load balancing (STDDEV) becomes worse as we decrease α because the boundaries of the back-end application servers fail to adjust to the changes in the query distribution.

However the cache hit rate slightly increases from 16.9% to 20.4% as we decrease α for the CBMG query distribution because the CBMG query pattern is rather stationary and a small α value, close to 0, makes the boundaries fixed, which increases the probability of reusing cached objects. However for the dynamic query distribution stationary boundaries decrease the probability of cache reuse, hence the cache hit rate decreases from 12.2% to 8.1% as we decrease α. Figure 6 shows the average query response time determined by the cache hit rate and load balancing. As we decrease α, the query response time increases exponentially since it hurts overall system load balance greatly although it improves the cache hit rate slightly.

Both leveraging cached results and achieving good load balance are equally important in maximizing overall system throughput. However we note that a very small improvement in cache hit rate might be more effective in improving average query response time than a large standard deviation improvement in certain cases. In Figure 5(a), the load balancing standard deviation is approximately 3 times higher when the weight factor is 0.1 than it is 0.3. But the average query response times shown in Figure 6(a) are similar because the cache hit rate is slightly higher when the weight factor is 0.1.

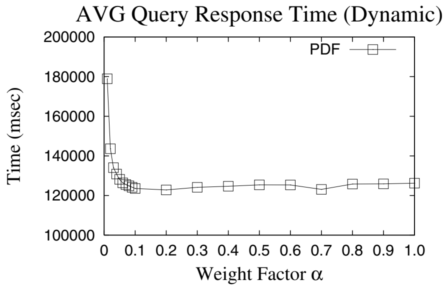


(a) Dynamic

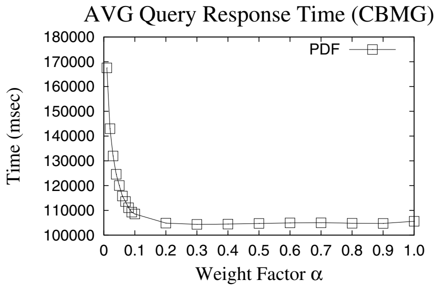


(b) CBMG

Fig. 5. Cache Hit Rate and Load Balancing with Varying Weight Factor



(a) Dynamic



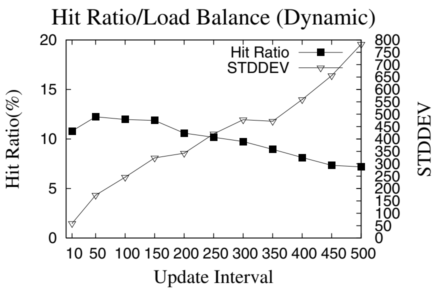
(b) CBMG

Fig. 6. Query Response Time with Varying Weight Factor

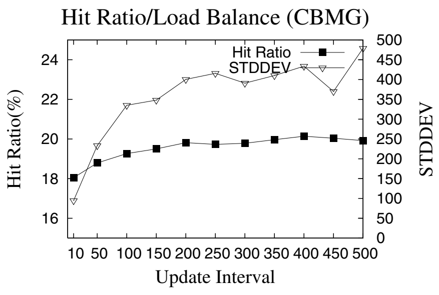
Update Interval. How frequently the scheduler updates the new boundaries is another critical performance factor in the DEMB scheduling policy, since frequent updates will make the boundaries of servers more quickly respond to recent changes in the query distribution. However frequent updates may cause large overheads in the front-end scheduler, and may not be necessary if the query distribution is stationary. Hence the update interval should be chosen considering the trade-off between reducing scheduler overhead and making the scheduling policy responsive to changes in the query distribution.

In the experiments shown in Figures 7 and 8, we measured the performance of the DEMB scheduling policy varying the update interval. In this set of experiments, the α and WS were fixed to 0.3 and 200, respectively. As we decrease the update interval, the boundaries are updated more frequently and they reflect the recent query distribution well. As a result, the DEMB scheduling policy takes good advantage of clustering and load balancing for the dynamic query distribution, as shown in Figure 7(a). As the update interval increases, the boundaries move more slowly and DEMB suffers from poor load balancing and cache misses.

With the stationary CBMG queries shown in Figure 7(b), the cache hit rate does not seem to be affected by the update interval, but the standard deviation increases slightly, although not as much as for the dynamic query distribution. This results indicate that we should update the boundaries frequently as long as that does not cause significant overhead in the scheduler.

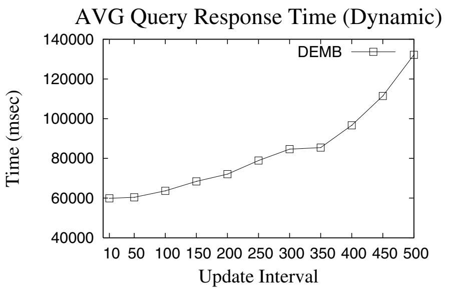


(a) Dynamic

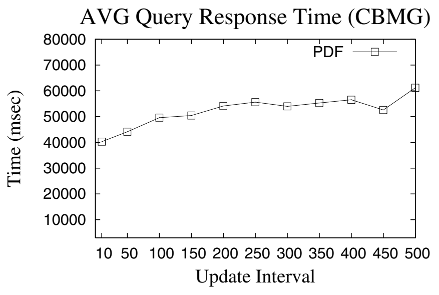


(b) CBMG

Fig. 7. Cache Hit Rate and Load Balancing with Varying Update Interval



(a) Dynamic



(b) CBMG

Fig. 8. Query Response Time with Varying Update Interval

Window Size. The front-end scheduler needs to store the recent queries in a queue so that they can be used to construct query distribution and determine the boundaries of back-end servers. With a larger window size (more queries), the front-end scheduler can estimate query distribution more accurately. Also, a larger WS allows a query to stay in the queue longer, i.e. the same queries will be used more often to construct query histograms and boundaries. On the other hand, a small WS makes the front-end scheduler uses a smaller number of recent queries to determine the boundaries, and the query distribution estimate is more likely to have large errors. In the experiments shown in Figures 9 and 10, we measured performance with various window sizes. The weight factor α and the update interval UI were both fixed to 1, in order to analyze the effects of only WS.

For the dynamic query distribution, the cache hit rate increases from 5.4% to 20% and the standard deviation decreases slowly as the window size increases up to 1000. That is because the scheduler estimates query distribution more accurately with a larger number of queries. However, if the window size becomes larger than 1000, both the cache hit rate and load balancing suffer from inflexible boundaries. Note that WS also determines how quickly the boundaries change as the query distribution changes. A large window size makes the scheduler consider a large number of old queries so will make the boundaries between servers change slowly. If the query distribution changes

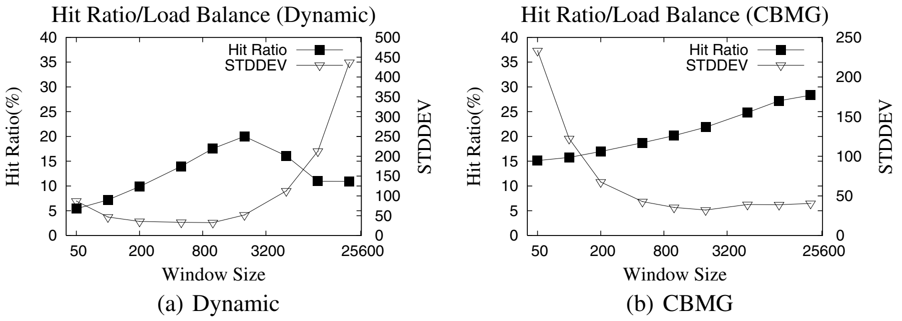


Fig. 9. Cache Hit Rate and Load Balancing with Varying Window Size

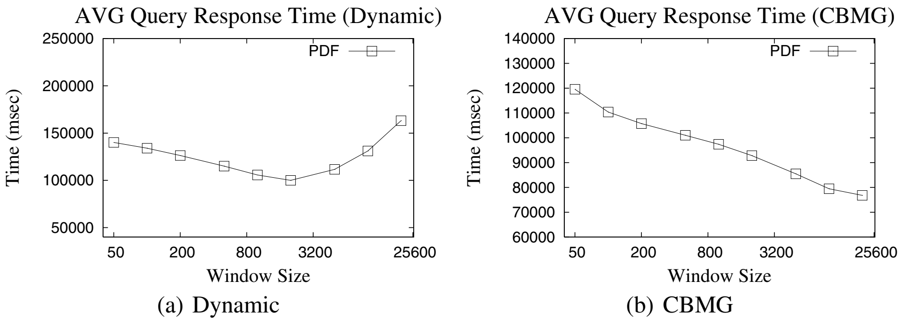


Fig. 10. Query Response Time with Varying Window Size

rapidly, a smaller window size allows the scheduler to quickly adjust the boundaries. For the CBMG query distribution, both the cache hit rate and load balancing improves as the window size increases. This is because the CBMG queries are relatively stationary over time. Hence a longer record of past queries helps the scheduler to determine better boundaries for future incoming queries.

These window size experimental results show that we need to set window size as large as possible unless it hurts the flexibility of the scheduler. Note that the window size should be orders of magnitude larger than the number of back-end servers. Otherwise, the boundaries set from a small number of queries would have very large estimation errors. For example, if the window size is equal to the number of back-end servers, the boundaries will be simply the middle point of the sorted queries, which would make the boundaries jump around the problem space. However a large window size has a large memory footprint in the scheduler, so can cause higher computational overhead in the scheduler, and the same is true for the update interval. In order to reduce the overhead from large window sizes, but to prevent estimation errors from making the boundaries move around too much, the scheduler can decrease the weight factor α instead, which will give higher weight to older past boundaries and smooth out short term estimation errors.

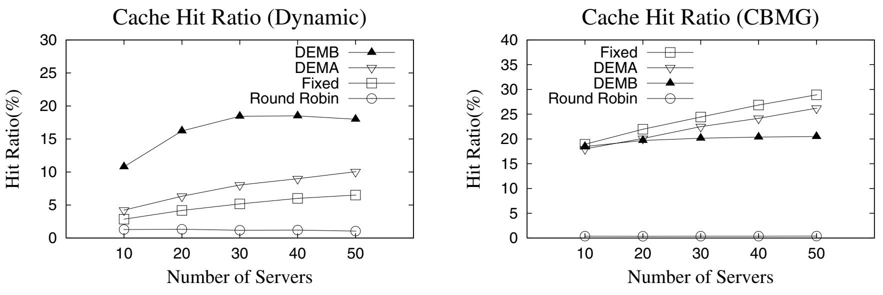
5.3 Comparative Study

In order to show that the DEMB scheduling policy performs well compared to other scheduling policies, we compared it with three other scheduling policies - round-robin, Fixed, and DEMA. For this set of experiments, we employed 50 back-end application servers and a single front-end server. In order to measure how the other scheduling policies behave under different conditions, we used both the dynamic and CBMG query distributions. The parameters for the DEMB scheduling were set to good values from the previous experiments - the weight factor α is 0.2, the update interval is 500, and the window size is 1000. These numbers are not the best parameter values we obtained from the previous experiments, but we will show that the DEMB scheduling policy shows good performance even without the optimal parameter values. Figures 11, 12, and 13 show the cache hit rate, load balance, and query response time for the different scheduling policies.

The Fixed scheduling policy partitions the problem space into several sub-spaces using the query probability distribution of the initial N queries, similar to the DEMB scheduling policy (where N is the number of servers), and each server processes subsequent queries that lie in its sub-space. But the sub-spaces are not adjusted as queries are processed, unlike DEMB. When the query distribution is stable, Fixed scheduling has a higher cache hit rate than round-robin since it takes advantage of spatial locality in the queries while round-robin does not. Since the Fixed scheduling policy does not change the boundaries of sub-spaces once they are initialized, it obtains a higher cache hit rate than the DEMA or DEMB scheduling policies for certain experimental parameter settings, as shown in Figure 11(b). For the CBMG query distribution, there are 200 fixed hot spots and the servers that own those hot spots and have large cache spaces obtain very high cache hit rates. However, when the query distribution changes dynamically the cache hit rate for the Fixed scheduling policy drops significantly, and is much lower than that of the DEMA and DEMB scheduling policy.

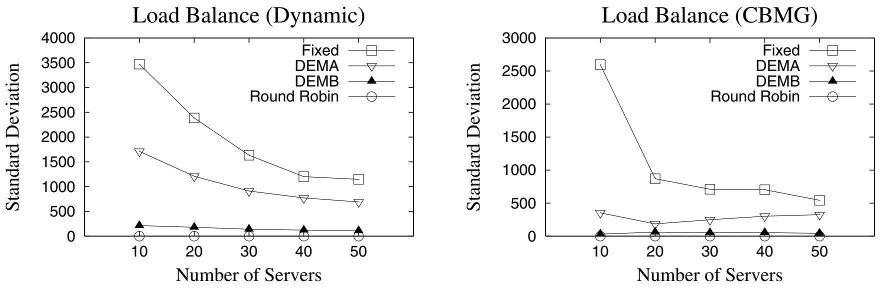
Although the Fixed scheduling policy outperforms DEMA and DEMB as measured by cache hit rate when the query distribution is stable, the Fixed policy suffers from serious load imbalance. Some servers process many fewer queries than others, if their sub-spaces do not contain hot spots. Therefore the standard deviation of the Fixed scheduling is the worst of all the scheduling policies, as shown in Figure 12, and the load imbalance problem becomes more especially bad for small numbers of servers. Due to its poor load balancing, the average query response time for the Fixed scheduling is higher than for the other scheduling policies, as shown in Figure 13. Note that the query response time is shown on a log scale.

For the dynamic query distribution, the DEMB scheduling policy is superior to the other scheduling policies for all the number of servers in terms of the query response time. DEMB's cache hit rate is about twice that of DEMA - the second best scheduling policy. The DEMA scheduling policy has a lower cache hit rate than DEMB because DEMA slowly responds to rapid changes in the incoming query distribution, as we described in Section 3.1. Note that the DEMA scheduling policy adjusts only a single EMA point per query, while DEMB adjusts all the servers' boundaries at every update interval.



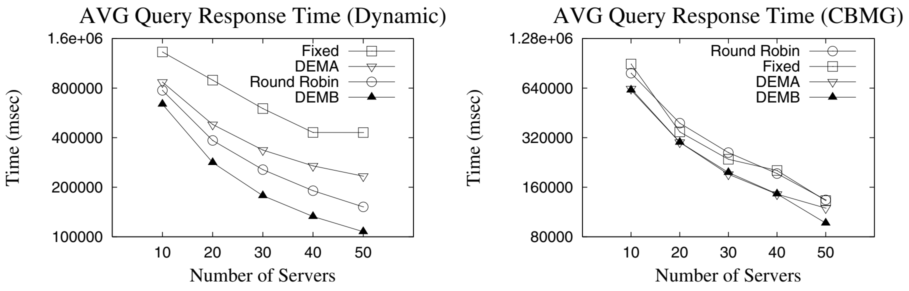
(a) Dynamic (b) CBMG

Fig. 11. Cache Hit Rate



(a) Dynamic (b) CBMG

Fig. 12. Load Balance



(a) Dynamic (b) CBMG

Fig. 13. Query Response Time

However, for the CBMG query distribution, the quick response to the changed query distribution seems to lower the cache hit rate of DEMB somewhat. As we mentioned earlier, the DEMB scheduling policy is designed to overcome the drawback of the DEMA scheduling policy, which is seen most when the query distribution changes dynamically. In CBMG distribution the query distribution pattern is stable, and both the

DEMA and the DEMB scheduling policies perform equally well. In Figure 11(b), the cache hit rate for DEMB is lower than for the DEMA scheduling policy because the boundaries determined by DEMB are likely to move rapidly, since some spatial locality for the queries is lost. When the boundaries are adjusted rapidly based on short term changes from a small number of recent queries, that may improve load balance but it will decrease the cache hit rate when the long term query distribution is stable. Due to the DEMB scheduling policy's fast response time, its load balancing performance is similar to that of the round-robin scheduling policy. The DEMA scheduling policy also balances server load reasonably well for the CBMG query distribution, but for the dynamic query distribution DEMA suffers from load imbalance due to its slow adjustment of EMA points.

Figure 13 shows that the average query response time improves as we add more servers to the system. The DEMB scheduling policy outperforms all the other scheduling policies for the dynamic query distribution. For the CBMG query distribution, the DEMB query response time is the lowest in most cases. For 50 servers, DEMB shows the best performance although its cache hit rate is not the highest. This result shows that load balancing plays an important role in large scale systems. However load balancing itself is not the only factor in overall system performance because round-robin does not show good performance.

5.4 Automated Parameter (WS) Adjustment

In the DEMB scheduling policy, the three parameters (weight factor, update interval, and window size) determine how fast the boundaries of each server adjust to the new query distribution. As seen in Figure 8, the update interval (UI) should be set close to 1 to be more responsive, i.e. the boundaries of servers must be updated for every incoming query. The weight factor (α) also determines how much weight is given to recent queries so that the boundaries adjust to the new distribution. However, when the window size (WS) is large enough, the current query distribution already captures recent changes in query distribution and the weight factor (α) does not affect query response time significantly. If the window size is not much greater than the number of backend servers (within a factor of 2 or 3), the query response time is not as resilient to changes in the weight factor when α is larger than 0.1, as shown in Figure 6. This is because the large window size (WS) and the small update interval (UI) makes a single query counted for multiple (WS) times (such as in a sliding window) to determine the boundaries for each server. Hence, the most effective performance parameter that system administrators can tune for the DEMB scheduling policy is the window size (WS) .

As shown in Figure 10, a larger window size yields lower query response times when the query distribution is stable. But if the query distribution changes rapidly, the window size must be chosen carefully to reduce the query response time. In order to automate selecting a good window size based on changes in the query distribution, we make the scheduler compare the current query distribution histogram with a moving average of the past query distribution histogram using Kullback-Leibler divergence, which is a measure of the difference between two probability distributions [9]. If the two distributions differ significantly the scheduler decreases the window size so that updates to boundaries happen more quickly. When the two distributions are similar, the scheduler gradually makes the window size bigger (but no bigger than a given maximum size), which makes the boundaries more stable over time to achieve a higher cache hit rate. In experiments not shown due to page limitation, we have observed that this automated approach improves query response time significantly.

6 Conclusion and Future Work

In this paper we have described and compared distributed query scheduling policies that take into consideration the dynamic contents of a distributed caching infrastructure with very little overhead ($O(N \times HilbertLevel)$.

In distributed query processing systems where the caching infrastructure scales with the number of servers, both leveraging cached results and achieving good load balance are equally important in maximizing overall system throughput. In order to achieve load balancing as well as to exploit cached query results, such a system must employ more intelligent query scheduling policies than the traditional round-robin or load-monitoring scheduling policies.

In this context, we proposed the DEMB scheduling policy that clusters similar queries to increase the cache hit rate and assigns approximately equal numbers of queries to all servers to achieve good load balancing. We experimentally demonstrate that DEMB produces better query plans that provide much lower query response times than traditional query scheduling policies.

References

1. Aron, M., Sanders, D., Druschel, P., Zwaenepoel, W.: Scalable content-aware request distribution in cluster-basednetwork servers. In: Proceedings of Usenix Annual Technical Conference (2000)
2. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry, Algorithms and Applications. Springer (1998)
3. Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdag, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing 69(8), 711–724 (2009)
4. Chou, Y.: Statistical Analysis. Holt International (1975)
5. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in dynamic structured p2p systems. In: Proceedings of INFOCOM 2004 (2004)
6. Grinstead, C.A., Snell, J.L.: Introduction to Probability. American Mathematical Society (1997)
7. Katevenis, M., Sidiropoulos, S., Courcoubetis, C.: Weighted round-robin cell multiplexing in a general-purpose atm switch chip. IEEE Journal on Selected Areas in Communications 9(8), 1265–1279 (1991)
8. Kim, J.S., Andrade, H., Sussman, A.: Principles for designing data-/compute-intensive distributed applications and middleware systems for heterogeneous environments. Journal of Parallel and Distributed Computing 67(7), 755–771 (2007)
9. Kullback, S., Leibler, R.A.: On information and sufficiency. Annals of Mathematical Statistics 22(1), 79–86 (1951)

10. Menasce, D.A., Almeida, V.A.F.: Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning. Prentice Hall PTR (2000)
11. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the clustering properties of the hilbert space-filling curve. IEEE Transactions on Knowledge and Data Engineering 13(1), 124–141 (2001)
12. Nam, B., Shin, M., Andrade, H., Sussman, A.: Multiple query scheduling for distributed semantic caches. Journal of Parallel and Distributed Computing 70(5), 598–611 (2010)
13. Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-aware request distribution in cluster-based network servers. In: Proceedings of ACM ASPLOS (1998)
14. Rodríguez-Martínez, M., Roussopoulos, N.: MOCHA: A self-extensible database middleware system for distributed data sources. In: Proceedings of ACM SIGMOD (2000)
15. Smith, J., Sampaio, S., Watson, P., Paton, N.: The polar parallel object database server. Distributed and Parallel Databases 16(3), 275–319 (2004)
16. Theodoridis, Y.: R-tree Portal, http://www.rtreeportal.org
17. Vydyanathan, N., Krishnamoorthy, S., Sabin, G., Catalyurek, U., Kurc, T., Sadayappan, P., Saltz, J.: An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications. IEEE Transactions on Parallel and Distributed Systems 15, 3319–3332 (2009)
18. Wolf, J.L., Yu, P.S.: Load balancing for clustered web farms. ACM SIGMETRICS Performance Evaluation Review 28(4), 11–13 (2001)
19. Zhang, K., Andrade, H., Raschid, L., Sussman, A.: Query planning for the Grid: Adapting to dynamic resource availability. In: Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), Cardiff, UK (May 2005)
20. Zhang, Q., Riska, A., Sun, W., Smirni, E., Ciardo, G.: Workload-aware load balancing for clustered web servers. IEEE Transactions on Parallel and Distributed Systems 16(3), 219–233 (2005)

Employing Checkpoint to Improve Job Scheduling in Large-Scale Systems

Shuangcheng Niu[1], Jidong Zhai[1], Xiaosong Ma[2],
Mingliang Liu[1], Yan Zhai[1], Wenguang Chen[1], and Weimin Zheng[1]

[1] Tsinghua University, Beijing, China
[2] North Carolina State University and Oak Ridge National Laboratory, USA
{niu.shuangcheng,zhaijidong,liuml07,zhaiyan920}@gmail.com,
ma@cs.ncsu.edu, {cwg,zwm}@tsinghua.edu.cn

Abstract. The FCFS-based backfill algorithm is widely used in scheduling high-performance computer systems. The algorithm relies on runtime estimate of jobs which is provided by users. However, statistics show the accuracy of user-provided estimate is poor. Users are very likely to provide a much longer runtime estimate than its real execution time.

In this paper, we propose an aggressive backfilling approach with checkpoint based preemption to address the inaccuracy in user-provided runtime estimate. The approach is evaluated with real workload traces. The results show that compared with the FCFS-based backfill algorithm, our scheme improves the job scheduling performance in waiting time, slowdown and mean queue length by up to 40%. Meanwhile, only 4% of the jobs need to perform checkpoints.

Keywords: job scheduling, backfill algorithm, runtime estimate, checkpoint/restart.

1 Introduction

Supercomputers and clusters today are usually managed by batch job scheduling systems, which partition the available set of compute nodes according to resource requests submitted through job scripts, and allocate such node partitions to parallel jobs. A parallel job will occupy the allocated node partition till *1)* the job completes or crashes, or *2)* the job runs out of the maximum wall execution time specified in its job script. Jobs that cannot immediately get their requested resources will have to wait in a job queue. The goal for parallel job schedulers is to reduce the average queue waiting time, and maximize the system throughput/utilization, while maintaining fairness to all jobs [1].

Parallel job scheduling technology has matured over the past decades. Though many strategies and algorithms have been proposed, such as dynamic partitioning [2] and gang scheduling [3], they are not widely used due to practical limitations. Instead, FCFS (First Come First Served) based algorithms with backfilling are currently adopted by most major batch schedulers running on supercomputers and clusters alike. It was first developed for the IBM SP1 parallel supercomputer installed at Argonne National Laboratory, as part of EASY (the Extensible Argonne Scheduling sYstem) [4]. Several variants of FCFS-based backfilling (referred to as *backfilling* in the rest of this paper) serve as the default setting in today's prominent job schedulers, such as LSF [5], Moab [6], Maui [7], PBS/Torque [8], and LoadLeveler [9].

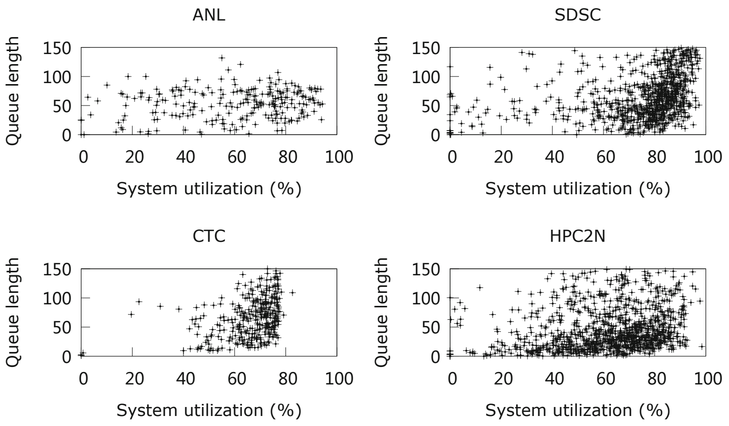


Fig. 1. Queue length vs. system utilization level based on job traces on four parallel systems

The backfilling algorithm is able to make use of idle "holes" left caused by advance reservation for jobs that arrived earlier but unable to run due to insufficient resources, by filling in small jobs that are guaranteed to finish before the reservation starts (more details of the algorithm will be given later). However, systems using such mechanism still suffer from resource under-utilization. Figure 1 illustrates such problem based on the workload traces from four production systems (available online at the Parallel Workloads Archive [10]): Intrepid at Argonne National Laboratory (ANL), Blue Horizon at the San Diego Supercomputing Center (SDSC), IBM SP2 at the Cornell Theory Center (CTC) and Linux cluster at High Performance Computing Center North, Sweden (HPC2N). The parallel job schedulers used at these systems Cobalt, Catalina, EASY, and Maui, respectively, most of which use FCFS+backfilling algorithms.[1] For each system, Figure 1 plots the average system utilization level (x axis) and the average job queue length (y axis), calculated for each day from the traces. The results show that on all four systems, it is common for a significant amount of system resource (20% to 80% of nodes) to be idle while there are (many) jobs waiting in the queue. In Peta- and the upcoming Exa-FLOP era, such system under-utilization translates into more severe waste of both compute resources and energy.

[1] Cobalt uses a WFP-based backfill algorithm.

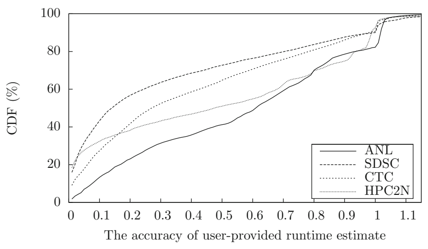


Fig. 2. Distribution of the ratio between the actual and the user-estimated job execution time. Note that supercomputers scheduling policies often allow a "grace period", such as 30 minutes, before terminating jobs that exceed their specified maximum wall time. This, plus extra storage and cleanup operations involved in job termination, explains the existence of ratios larger than 1.

One key factor leading to this defect is that the backfilling effectiveness depends on the job execution wall time estimate submitted by the user, which has been found in multiple studies to be highly inaccurate [11, 12]. By comparing the user-estimated and the actual job execution times from the aforementioned traces, we calculate the probability density function of the wall time estimate accuracy, as t_{run}/t_{req}, where t_{run} is the actual runtime and t_{req} is the user-provided estimate.

Figure 2 suggests that a large portion (∼40%) of jobs actually terminated within 20% of the estimated time. The distribution of the actual/estimated execution time ratio spread quite uniformly over the rest of the spectrum, except a noticeable burst around 1. Overall, only 17% of jobs completed within the 0.9-1.1 range of the estimated time, across the four systems. As to be discussed in more detail in the next section, such dramatic job wall time estimate error leads to significant problems in backfilling effectiveness.

To improve scheduling performance, many previous studies have targeted improving the accuracy of job execution time estimate [13–15], with only limited success. In fact, there are several factors leading to inaccurate estimates, mostly overestimates. Firstly, users may not have enough knowledge on the expected execution time of their jobs (especially with short testing jobs and jobs with new input/parameters/algorithms), and choose to err on the safe side. Secondly, in many cases the large estimated-to-actual execution time ratio is caused by unexpected (early) crash of the job. Thirdly, users tend to use a round value (such as 30 min or 1 hour) as the estimated wall time, causing large "rounding error" for short jobs. All the above factors help making job wall time overestimation a perpetual phenomenon in batch systems. Actually, a study by Cynthia et al. revealed that such behavior is quite stubborn, with little estimate accuracy improvement even when the threat of over-time jobs being killed is removed [16].

In this paper, we propose a new scheduling algorithm that couples advance reservation, backfilling, and job preemption. Our approach is motivated by the observation that on today's ever-larger systems, checkpointing has become a standard fault tolerance practice, in answer to the shortening MTBF (mean time between failures). With portable, optimized tools such as BLCR [17], parallel file systems designed with checkpointing as a major workload or even specifically for the checkpointing purpose (such as PLFS [18]), emerging high-performance hardware such as aggregated SSD storage, and a large amount of recent/ongoing research on checkpointing [19–21], the efficiency and scalability of job checkpointing has been improving significantly. Such growing checkpoint capability on large-scale systems enables us to relax the backfill conditions, by allowing jobs to be aggressively backfilled, and suspended for later execution if resources are due for advance reservation.

This approach manages to make use of idle nodes without wasting resources by aborting opportunistically backfilled jobs. By limiting the per-job checkpointing occurrences and adjusting the backfill aggressiveness, our algorithm is able to achieve a balance between improving resource utilization, controlling the extra checkpointing and restarting overhead, and maintaining scheduling fairness.

We consider our major contributions as follows:

1. *A checkpoint-based scheduling algorithm.* To our knowledge, this is the first paper that discusses how to leverage checkpointing techniques to optimize the backfilling scheduling algorithm. Although some previous works proposed preemption based backfill, no works discussed how to use checkpoint technique to improving backfilling scheduling. We analyzed the limitations of an existing FCFS-based backfilling algorithm and improved it by enabling aggressive backfilling with estimated wall time adjustment and checkpoint-based job preemption.
2. *Comprehensive evaluation with trace-driven simulation.* We conducted experiments using job traces collected from four production systems, including Intrepid, currently ranked 15th on the Top500 list. Our results indicate that compared with the classical FCFS-based backfill algorithm, the checkpoint-based approach is capable of producing significant improvements (by up to 40%) to key scheduling performance metrics, such as job average wait time, slowdown, and the mean queue length.

   In our evaluation, we also estimated the overhead incurred by checkpoint/restart operations, based on system information from the Intrepid system. Different I/O bandwidths are considered, simulation results suggest that extra checkpoint/restart operations hardly cause any impact on the key scheduling performance metrics.

The remainder of this paper is organized as follows. Section 2 reviews the traditional FCFS-based backfilling algorithm. Section 3 presents our checkpoint-based scheduling approach and gives detailed algorithms. Section 4 reports our simulation results on job scheduling performance, while Section 5 analyzes the introduced checkpoint overhead. Section 6 discusses related work, and finally, Section 7 suggests potential future work and concludes the paper.

2 Classical Backfilling Algorithm Overview

In this section, we review the classical backfilling algorithm to set a stage for presenting our checkpoint-based approach.

Most modern parallel job schedulers give static allocations to jobs with spatial partitioning, i.e., a job is allocated the number of nodes it requested in its job script and uses this partition in a dedicated manner throughout its execution. The widely used FCFS-based backfill algorithm does the following:

- maintain jobs in the order of their arrival in the job queue and schedule them in order if possible,
- upon a job's completion or arrival, dispatch jobs from the queue front and reserve resources for the first job in the queue that cannot be run due to insufficient resource available,
- based on the user estimated wall times of the running jobs, calculate the backfill time window, and
- traverse the job queue and schedule jobs that can fit into the backfill window, whose execution will not interfere with the advance reservation. Such jobs should either complete before the reserved "queue head job" start time or occupy only nodes that the advance reservation does not need to use.

A simplified version of such strategy, used in the popular Maui scheduler [7], is shown as Algorithm 1, which is also used as the classical backfilling algorithm in our trace-driven simulation experiments. Other popular schedulers such as EASY and LoadLeveler also use similar frameworks.

As can be seen from the algorithm, user-provided estimated runtime is crucial to the overall scheduling performance. Overestimation will not only postpone the reservation time, but also affect the chance for a job to be backfilled. We will see later that increased estimate inaccuracy can significantly impact the classical algorithm's scheduling performance, particularly in the "job slowdown" category.

3 Checkpoint-Based Backfilling

In this section, we explain our scheduling approach and the proposed algorithm in details. Our algorithm overcomes the limitation in the classical backfilling algorithm by weakening the impact of the user job run time estimation accuracy on the system performance. With our approach, both the wait time and the chance of backfill can be significantly improved.

3.1 Assumptions

First, we list several major assumptions made in this work:

1. *Rigid jobs:* Our design assumes that the jobs are rigid jobs, running on a fixed number of processors (nodes) during its end-to-end execution. This resource requirement is specified in the job script.

Algorithm 1. FCFS-based backfilling algorithm

```
Require: Job queue $Q$ is not empty, Resource manager $RM$
 1: /* Schedule runnable jobs */
 2: for each $job \in Q$ do
 3:    if $job.size > RM.free\_size$ then
 4:       $break$
 5:    end if
 6:    $job.predictEndtime \leftarrow job.estimateRuntime + now()$
 7:    $run(job)$
 8:    $RM.free\_size -= job.size$
 9: end for
10: if $Q.isEmpty()$ then
11:    $exit$
12: end if
13: /* Make Reservation */
14: $reserve\_job \leftarrow Q.top()$
15: $future\_available\_size \leftarrow RM.free\_size$
16: $running\_jobs\_list \leftarrow$ Sort running jobs in order of their predicted termination time
17: for each $job \in running\_jobs\_list$ do
18:    $future\_available\_size$ += $job$.size
19:    if $future\_available\_size \geq reserve\_job$.size then
20:       $reserve\_time \leftarrow job.predictEndtime$
21:       $break$
22:    end if
23: end for
24: $backfill\_window \leftarrow reserve\_time - now()$
25: $extra\_size \leftarrow future\_available\_size$ - $reserve\_job$.size
26: /* Backfilling */
27: for each $job \in Q$ do
28:    if $job.size > RM.free\_size$ then
29:       $continue$
30:    end if
31:    $predictRuntime \leftarrow$job.estimateRuntime
32:    if $predictRuntime \leq backfill\_window$ or $job.size \leq extra\_size$ then
33:       $job.predictEndtime \leftarrow predictRuntime + now()$
34:       $backfill(job)$
35:       $RM.free\_size -= job.size$
36:       if $predictRuntime > backfill\_window$ then
37:          $extra\_size -= job.size$
38:       end if
39:    end if
40: end for
```

2. *Jobs not bound to specific nodes:* Each job can execute on any set of nodes that meets the processor number requirement.
3. *Checkpoint/Restart abilities:* The applications or the supercomputing center has checkpoint/restart (C/R) abilities, which can be initiated any time by the scheduler.

Note that the first two assumptions apply to most of classical backfilling algorithms too, allowing each job to be scheduled once and assigned an arbitrary subset of all the available nodes. In the context of our checkpoint-based backfilling scheduling, however, they allow an interrupted and checkpointed job to resume its execution with the same number of nodes without any placement constraints.

3.2 Methodology Overview

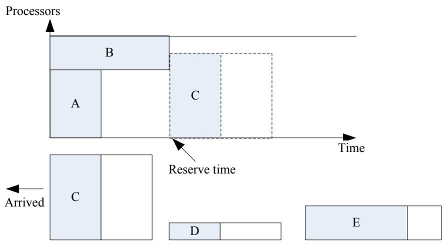
The main idea behind our checkpoint-based scheduling is to allow the queued jobs to be backfilled more aggressively, but reserve the right to suspend such jobs when they are considered to delay the execution of a higher-priority job. The flow of our algorithm works similarly as the classical one. The major difference lies in that at the time a high priority waiting job with advance reservation is expected to run, if any backfilled jobs are still running, we checkpoint rather than kill them. This allows us to cope with the highly overestimated run time and perform aggressive backfilling. In selecting backfill candidates, rather than directly using the user provided run time estimate, our scheduler predicts a job's actual execution time by intentionally *scale down* the estimated run time. According to Figure 2, most of the overestimated jobs can safely complete before the reserved deadline. For those jobs that such prediction actually underestimates their execution time, checkpointing avoids wasting the amount of work already performed, yet maintains the high priority of the job holding advance reservation.

Figure 3 illustrates the working of the checkpoint-based scheduling scheme with a set of sample jobs. Each job is portrayed as a rectangle, with the horizontal and vertical dimensions representing the 2D resource requirement: estimated wall time and processor/node number requested. The gray area within each waiting job's rectangle indicates its actual execution time. At a certain time point, jobs A and B are running, and jobs C, D and E are waiting in the queue, as shown in Figure 3(a). Upon A's completion, there are not sufficient resources for the queue head job C. The scheduler performs advance reservation for C, based on the estimated wall time of B. The moment that B terminates is the *reserve time* for C. The time from A's to B's termination forms the *backfill window*.

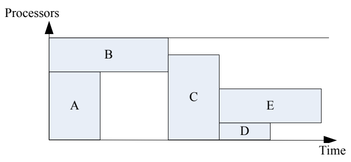
With the classical backfilling algorithm (Figure 3(b)), because D's and E's runtime estimates exceed the backfill window, D and E will not be backfilled, although D's actual runtime can fit in the window. D and E will run after C terminates, wasting resources within the backfill window.

With the checkpoint-based backfilling algorithm (Figure 3(c)), instead, both D and E will be backfilled. D will terminate before the reserved time. E, on the other hand, will be preempted at the reserved time and broken into two segments: E1 and E2. When C terminates, E will be restarted to resume its execution.

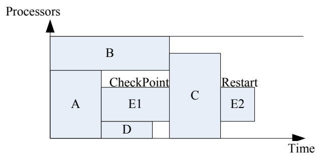
This example illustrates several advantages of the checkpoint-based scheduling approach. First, overestimated jobs benefit from eager backfilling, producing both shorter wait time and higher system utilization. Second, the checkpointed



(a) In the system, 2 jobs are running, 3 jobs are waiting.



(b) Scheduling with FCFS-based backfill algorithm



(c) Scheduling with checkpoint-based backfill algorithm

Fig. 3. With checkpoint-based backfill algorithm, job D which is overestimated benefits from the backfilling, job E also reduces the response time. Meanwhile, the top queue job is not affected.

Algorithm 2. Job.predictRuntime()

```
Require: job runtime estimate threshold $t_{thr}$, the split factor $p$
1: if $job.estimateRuntime < t_{thr}$ then
2:     return $job.estimateRuntime$
3: else
4:     if $job.isCheckPointJob()$ then
5:         return $job.estimateRuntime - job.hasRunTime$
6:     else
7:         return $job.estimateRuntime * p$
8:     end if
9: end if
```

jobs also observe a reduced response time. Finally, the priority of the job holding advance reservation is preserved.

3.3 The Checkpoint-Based Scheduling Algorithm

Next, we give more detailed description of our proposed checkpoint-based scheduling algorithm and discuss important parameters.

The main checkpoint-based scheduling framework is same with classical backfilling (Algorithm 1). It is executed repeatedly whenever a new job arrives or when a running job terminates. The major different is in Line 31 (highlighted in

Algorithm 3. Preempt algorithm

```
Require: the reserve time arrives
 1: if $RM.free\_size < reserve\_job.size$ then
 2:     $preempting\_jobs\_list \leftarrow RM.getPreemptingJobsList(reserve\_job)$
 3:     for each $job \in preempting\_jobs\_list$ do
 4:         checkpoint ($job$)
 5:         kill ($job$)
 6:         $RM.free\_size+ = job.size$
 7:         $Q.push(job)$
 8:     end for
 9: end if
10: run ($reserve\_job$)
11: $RM.free\_size- = reserve\_job.size$
```

the algorithm with underlined subroutine calls). When checking backfill eligibility and calculating the predicted end time, it according to the scheduler-predicted run time instead of the user-provided runtime estimate.

The calculating predicate runtime algorithm is the core algorithm, which is shown in Algorithm 2. It describes how the checkpoint-based scheduler performs job execution time prediction. As mentioned earlier, it scales down the user-estimated job wall time t_{req} by a factor of p, $0 < p <= 1$. However, such adjustment is only applied to jobs that are over a certain threshold t_{thr}. This is due to several considerations. First, short jobs are fairly easy to be backfilled even without such scale-down. Second, checkpointing and restart will turn out as more expensive to short jobs. Third, short jobs are often meant for testing and debugging, where a split execution might cause more degradation in user experience. Both p and t_{thr} are tunable parameters, controlling the aggressiveness of backfilling. In our experiments, we find that such simple schemes seem to work well and system administrators can select a proper parameter values based on empirical results collected from their actual workloads.

Finally, Algorithm 3 specifies the preempt scheme. It is executed whenever the reserved time arrives but there are no sufficient resources to allocate for the top priority job. In this algorithm, the backfilled jobs are checkpointed and terminated, until the top queue job gets its requested resources. The checkpointed jobs are put on the head of the queue. Also, in selecting checkpoint candidates, we start from jobs that are using more nodes, to reduce the number of preemption and checkpointing. Note that each backfilled jobs use fewer nodes than requested by the high-priority job holding advance reservation. Otherwise the latter could be scheduled at an earlier time point. Therefore, our checkpoint candidate identification is essentially performing a "best-fit" selection.

4 Evaluation Results

In this section we present our evaluation results obtained from trace-driven simulation experiments. We first analyze the performance of our proposed checkpoint-based backfill approach using real job traces, and compare it with that of the classical FCFS-based backfilling algorithm. Second, we evaluate the impact of the accuracy of user-provided estimate runtime on the performance of scheduling algorithms.

Simulator. We designed and implemented a trace-driven simulator to evaluate the performance of various of scheduling algorithms, which simulate events and states related to batch job scheduling, such as node allocation, job scheduling, job queue status, etc. The input of the simulator includes a job submission trace and a scheduling algorithm. The output of the simulator includes key performance metrics for scheduling systems, which are to be discussed in more details below.

Workload Traces. In our evaluation, we analyze four real workload traces collected from production systems from the Parallel Workload Archive [10]. Variants of backfilling is used as the default scheduling strategy in these systems. Each trace entry contain job description information items such as user-provided run time estimate, job submission time, actual execution time, and job size (number of processors requested). Below we briefly describe these traces:

– *ANL*: This trace contains entries for 68,936 jobs that were executed on a 40,960-node IBM Blue Gene/P system at Argonne National Laboratory called Intrepid. It was collected during the first 8 months of 2009 from the 40-rack production Intrepid.
– *SDSC*: This trace contains entries for 250,440 jobs that were executed on the 144-node IBM SP2 called Blue Horizon at the San Diego Supercomputer Center from April 2000 to January 2003.
– *CTC*: This trace contains entries for 79,302 jobs that were executed on a 512-node IBM SP2 at the Cornell Theory Center from July 1996 through May 1997.
– *HPC2N*: This trace contains entries for 527,371 jobs that were executed on a 120-node Linux cluster from the High-Performance Computing Center North (HPC2N) in Sweden from July 2002 to January 2006.

Metrics. We evaluate our proposed approach with four commonly used scheduling performance metrics, as defined below.

– *Wait Time (wait)*: the average per-job wait time in the job queue.
– *Bounded Slowdown (slowdown)*: the ratio of a job's response time to its actual execution time. In this work, we use bounded slowdown to reduce the impact of very short jobs on the average value, calculated as shown in the formula below. The bound value of 10 seconds is used in all our experiments.

$$\frac{WaitTime + Max(RunTime, BoundTime)}{Max(RunTime, BoundTime)}$$

– *Queue Length (qLength)*: the number of average waiting jobs in the job queue at a given time.
– *Backfill Ratio (bRatio)*: the ratio of the number of backfilled jobs to the total number of jobs.

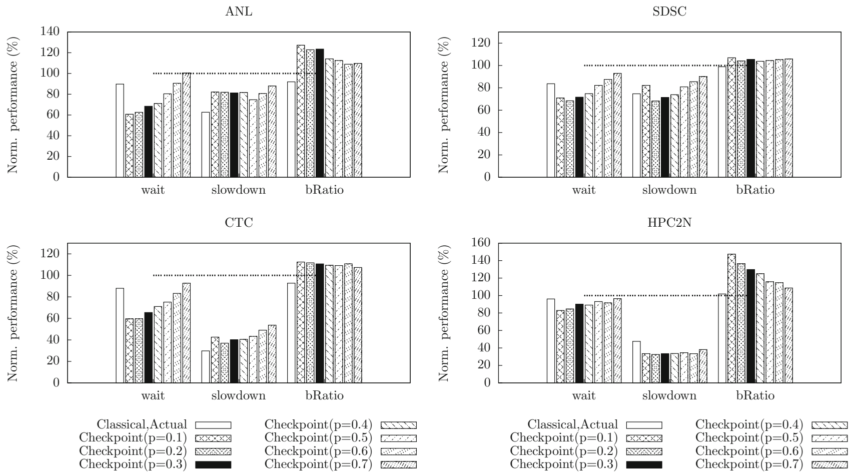


Fig. 4. Performance of checkpoint-based algorithm with different p values, normalized against the performance of the classical backfilling algorithm, which is marked by the dotted "100%" reference line. The "Classical, Actual" bar shows the performance with the classical algorithm, but supplied with the actual wall time (i.e., without any estimation error).

As will be shown later, with the checkpoint-based backfilling algorithm, most of the jobs still wait only once in the queue. Thus its wait time is the difference between its execution and submission. However, for jobs that do go through checkpointing, the wait time includes all segments of waiting the job spends in the queue.

Note that our trace-driven simulation is asynchronous, in the sense that we replay each job's submission according to the submission time specified in the trace. Therefore, even if an algorithm can improve system utilization, it will not be able to shorten the makespan for all jobs. In this context, it can be proved that the average wait time and the average queue length are equivalent. Therefore, in our results discussion, we only report the job wait time.

4.1 Performance of Checkpoint-Based Backfill Algorithm

Recall that in the checkpoint-based backfill algorithm, there are two key parameters:

- t_{thr}: The estimated wall time threshold, to mask jobs with wall time estimate smaller than this given threshold from being scaled down with p when calculating the predicted execution time. If t_{thr} is set too low, many short jobs will be checkpointed, causing increased overhead. In this paper, we fixed this parameter at 1800 seconds.
- p: The scale-down factor used to predict the actual job run time. The process of tuning p needs to consider the tradeoff between system utilization and

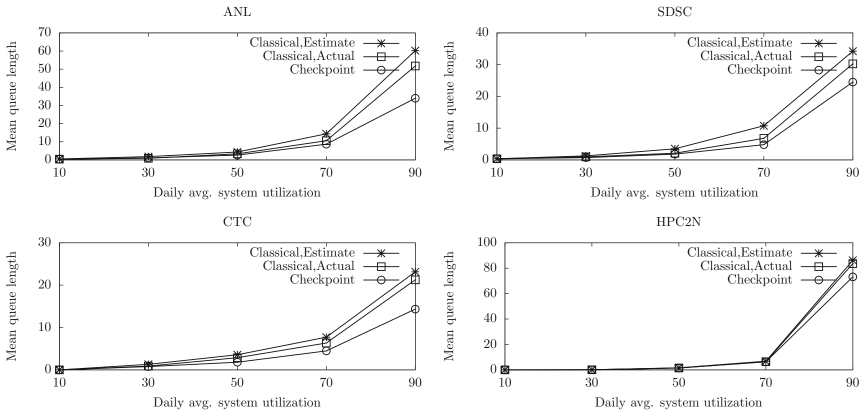


Fig. 5. Scheduling performance of different backfilling algorithms, averaged over subset of days where the system utilization level falls into certain intervals

cost. If p is set too low, many of the backfilled jobs might not finish by the reserved time for higher-priority jobs, and have to be checkpointed. In contrast, a very high value of p can reduce the backfill ratio and work quite similar to the classical algorithm. So we suggest that system administrators utilize their systems' historical job statistics in selecting a proper p.

Figure 4 shows the simulation results and compares the checkpoint-based algorithm with two variants of the classical one (with the user supplied run time estimate and with the actual job run time as a "ideal" estimate).

From the result, our algorithm performs better than both the base and ideal cases of classical algorithm in wait time (equivalent to queue length) and the backfill ratio. This is expected because of our relaxed condition. At the point of $p = 0.1$ or $p = 0.2$, our algorithm performs better than other selection of p.

One thing to note is the slowdown factor. Except for trace HPC2N, the slowdown in our algorithm can be larger than the ideal case of classical algorithm (but still far better than that of the base classical algorithm). The dominant reason for this is that the accurate estimation of small jobs can significantly improve the slowdown factor. However, it's impossible for the users to know exactly how long their jobs can run. In fact, our algorithm still performs better than classical algorithm if we directly use the runtime estimate in the collected trace. In trace set CTC and HPC2N, the slowdown decreases more than 50%.

So from these experiments, we have verified that our algorithm is more advanced than the classical algorithm in improving scheduling performance. Across the traces we obtained, it appears that a p value of 0.1 or 0.2 delivers the best overall performance. It matches the observation from Figure 2: more than 40% jobs have at least a 5-time overestimation in specifying their expected wall time.

Figure 5 further compares the three algorithms by partitioning days recorded in the trace into multiple buckets according to the daily average system utilization

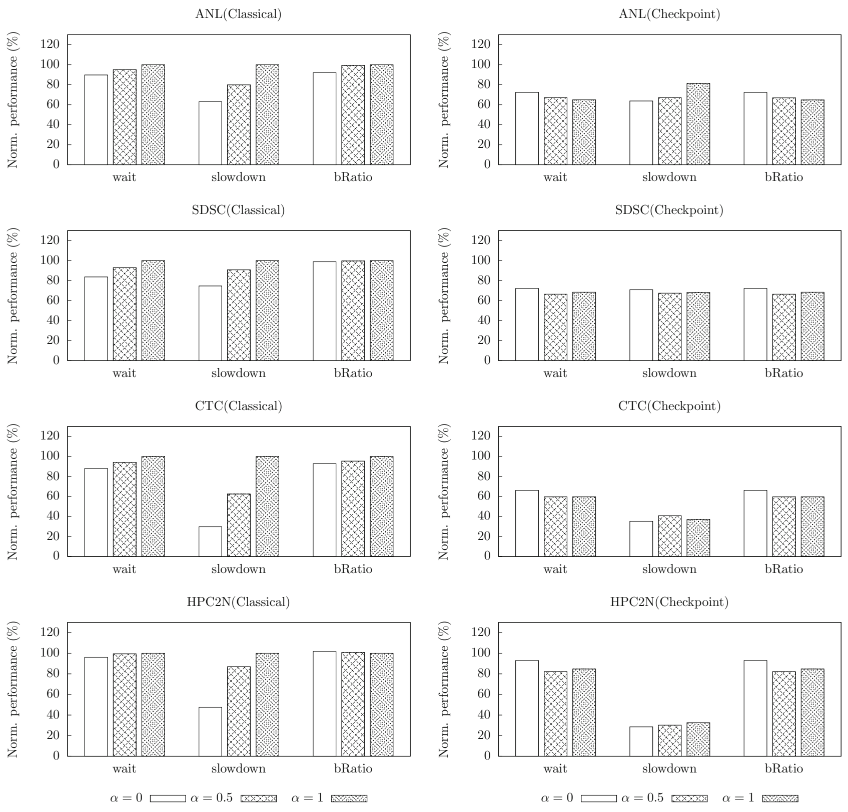


Fig. 6. Evaluation of the impact of estimate accuracy on scheduling algorithms. The left side uses classical algorithm, and right side uses the checkpoint-based algorithm with $p = 0.2$. The higher α is, the more inaccurate runtime estimate will be. When $\alpha = 0$, it is scheduled with actual runtime. When $\alpha = 1$, it is scheduled with user-provided runtime estimate.

level, and depicting average queue length over jobs in each bucket. It demonstrates that our proposed checkpoint-based algorithm significantly reduces the average queue length on most of the four platforms, especially when the system is busy.

4.2 The Impact of Estimate Accuracy on Scheduling Algorithms

To understand the impact of overestimation on algorithms' behavior, we evaluate how the two algorithms perform under different degrees of overestimation. As in the Figure 6, the result indicates checkpoint-based algorithm is more stable regarding the degree of inaccuracy changing, especially slowdown metric.

The degree of overestimation is defined as a job's actual run time dividing the runtime estimate. To do quantitative analysis, we introduce a parameter α, where $t_{sch} = t_{run} + \alpha \times (t_{req} - t_{run})$. Here t_{req} is the job's user requested run time in real world trace. We use the t_{sch} as the job's user estimated runtime to submit to the simulator. In this manner, we can obtain workload traces with different overestimation degrees by tuning the α. When $\alpha = 0$, $t_{sch} = t_{run}$ and when $\alpha = 1$, $t_{sch} = t_{req}$. The larger α is, the more inaccurate estimate time will be.

In Figure 6, we simulate the cases when α is 0, 0.5 and 1. The left side uses classical algorithm, and right side uses the checkpoint-based algorithm. It's quite clear left results vary in much larger extent when the runtime estimate tends to be more inaccurate. For example, the slowdown factor, which grows even more than twice when α changes from 0 to 1 in trace sets CTC and HPC2N.

These results indicate that classical algorithm might work well when the user estimation is accurate enough, and it's sensitive to the inaccurate factor. However, in Figure 2 we have shown it is usually unrealistic to have accurate estimate in real world system: a large portion of users tend to highly overestimate the actual run time by more than 5 times. Moreover, even in optimal case ($\alpha = 0$), classical algorithm can archive only comparable performance as checkpoint-based algorithm, which is also consistent with our results in previous section.

5 Analyzing Checkpoint/Restart Overhead

In the previous section, we don't consider the overhead of the checkpoint/restart operations. Actually the software state and temporary data are saved to the storage in checkpoint process, and are restored in restart process. In this section, we evaluate the overhead introduced by checkpoint/restart via simulation.

5.1 Checkpoint Overhead Analysis

We select the trace of Argonne Intrepid to evaluate. Intrepid is a 557 TF, 40-rack Blue Gene/P system deployed at Argonne National Laboratory. This system was ranked No. 15 in the list released in June 2011 [22]. It has 40,960 compute nodes and 640 I/O nodes, which is configured with a single I/O node managing 64 compute nodes. These I/O nodes connect to the file servers with 10 Gb Ethernet network. The peak bandwidth between the I/O node and the network is limited to 6.8 Gb/s. All the 640 I/O nodes can theoretically deliver up to 4.25 Tbps [23].

The entire Intrepid system consists of 128 dual-core file servers. The file servers connect to DataDirect 9900 SAN storage arrays through the Infiniband DDR ports, each with a theoretical unidirectional bandwidth of 16 Gbps. All the 128 file servers can theoretically deliver up to 2 Tbps [23].

Generally, the bandwidth bottleneck for the maximum data throughput lies more towards the file servers than the I/O nodes when the system is running in its full capacity. However, only partial running jobs are involved in checkpoint in the checkpoint based backfilling scheduler. The bottleneck depends on the I/O nodes bandwidth, which is limited to 0.85 GB/s per I/O node [23].

At the reserve time, it's not allowed to run the reserved job until all preempted job have done checkpoint. The delay of the reserved job depends on the maximal

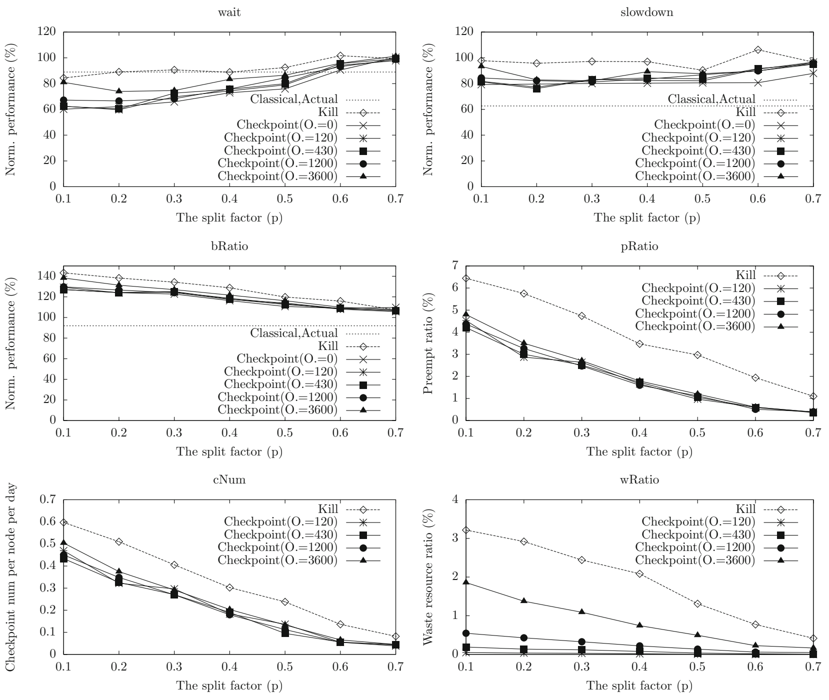


Fig. 7. Evaluation the impact of checkpoint overhead using ANL trace. Different checkpoint overhead are simulated. The simulate results that using classical backfill algorithm with runtime estimate are the baseline.

checkpoint duration of all preempted jobs. The restart process also consumes the resources, which extends the actual runtime. There is no memory requirement information in the workload trace of ANL. So we prepare for the worst, and simply assume the required memory is the total memory of the compute nodes. We assume that 70% of the bandwidth is available. Therefore, the delay time and extended runtime can be calculated as following.

$$T_{delay} = T_{chkpnt} = 2G \times 64/(0.85GB/s \times 70\%) = 215s.$$

$$T_{extend} = T_{chkpnt} + T_{restart} \leq 2 \times T_{chkpnt} = 430s.$$

Algorithm 2 should be modified to reflect the runtime extending. In the algorithm, the predicted runtime of checkpointed jobs need to add the T_{extend}.

5.2 Evaluation Results with Overhead

Our algorithm with checkpoint overhead is evaluated in Figure 7. It's compared against the same algorithm without adding overhead. To reflect the impact of the checkpoint overhead, we simulated with different T_{extend} values such as 120, 430, 1200 and 3600 seconds.

In addition to the original metrics, we also import three metrics to depict the overhead:

- *Preempt Ratio (pRatio)*: The ratio of the preempted jobs to all jobs.
- *Checkpoint number per node in a day (cNum)*: The average checkpoint number for a compute node in a day.
- *Waste Resource Ratio (wRatio)*: The ratio of the waste computing resources by preempt to all computing resources.

From the result, even adding the overhead, our algorithm still outperforms the baseline a lot when p is 0.2. The waiting time and queue length are improved by up to 40%. The slowdown is improved about 20%. In fact, Figure 7 indicates that the chances we have to do checkpoint are rare. When $p = 0.2$, the average number of checkpoint on each node is about 0.4 times per day and only 4% of the jobs need to do checkpoint. The waste resource is not more than 1.5%.

One thing to note is the intercepted point in slowdown curve in Figure 7. In fact, the runtime extending enlarges the backfill window and lets more shorter jobs to be backfilled. Thus, the backfill ratio increases, and the slowdown becomes better in some cases.

In summary, the overhead in checkpoint does not hurt the algorithm a lot, since the number of checkpoint we need to do is not quite large in our settings. Thus we stand on solid ground to conclude that the overhead is tolerable compared with the benefits gained.

6 Related Work

6.1 Job Scheduling Algorithms

Job scheduling systems take an important part in improving the efficiency of high performance computing (HPC) centers. Although a lot of scheduling algorithms have been proposed by industry and academia [2,3], FCFS-based backfilling algorithm is regarded as the most efficient algorithm for modern HPC centers. Several variants of FCFS-based backfilling algorithms serve as the default setting in a lot of famous job scheduling systems, such as LSF [5], Moab [6], Maui [7], PBS/Torque [8] and LoadLeveler [9].

User estimated runtime is a key factor affecting performance of backfilling algorithms. The first study to identify the inaccuracy of user runtime estimates was Feitelson and Mu'alem Weil [24]. There are a lot of studies trying to improve the accuracy of user estimated runtime. Chiang et al. suggested a test run before running to acquire more accurate estimated runtime [13]. Zhai et al. developed a performance prediction tool to assist an accurate estimated time [14]. Tang et al. got usable information from historical information [15]. In order to improve the estimate accuracy, Cynthia Bailey Lee et al. gave a detailed survey. However, performance prediction is a very difficult problem for HPC users. Most of user estimated runtime provided to scheduling systems is not accurate enough [16]. This results in very poor efficiency for scheduling systems.

Lots works suggested relaxing the backfilling conditions. These methods allowed jobs to be backfilled even that the estimated runtime are longer than the backfilling window. If the backfilled jobs can't finish in the specified period, uncompleted backfilled jobs are allowed to continue [15,25,26]. This strategy will postpone top-queue jobs.

Several works suggested preemption based backfill. Snell et al. studied aggressive backfill with kill-based preemption and discussed strategies that were used to select candidate preempted jobs [27]. Maui support PREEMPT backfill policy. It allows the scheduler to start backfill jobs even if required walltime is not available. If the job runs too long and interferes with another job which was guaranteed a particular timeslot, the backfill job is preempted and the priority job is allowed to run [28]. Perkovic et al. proposed "speculative backfilling" after regular backfilling [29]. If a speculatively run jobs runs longer than its speculated time, it will be killed. Most of these works focused on kill-based preemption. However, checkpoint-based preemption has different features than kill-based preemption. The latter hopes that preempted jobs have a short running time to reduce waste resources. It is more suitable for improving the short-run failure jobs. Our work uses checkpoint-based preemption, and it hopes that preempted jobs have a long running time to improve effect-cost ratio. It is more suitable for solving inaccurate user-provided runtime estimate.

Morris Jette et al. developed a preemption-based gang scheduler at Lawrence Livermore National Laboratory (LLNL) for the Cray T3D. The Gang Scheduler combines a checkpoint-based preemptive processor scheduler with the ability to relocate jobs within the pool of available processors. That approach can sustain machine utilization and provide interactive workload with a lower response time [30,31]. Different, our work is focused on traditional backfilling algorithm.

6.2 Checkpoint/Restart Techniques

As the size of HPC systems increases, reliability is becoming more and more important for HPC users. Checkpoint/Restart technique has become a standard configuration for real systems.

A lot of system-level and application-level checkpoint techniques have been proposed. Berkeley Lab's Checkpoint/Restart (BLCR) is a system-level checkpoint technique, which is the Linux kernel-based coordinated checkpoint technique [17]. BLCR is integrated with LAM/MPI and OpenMPI to provide checkpoint and restart for parallel applications. IBM proposed a special application-level checkpoint library [32] for BlueGene/P applications. This library provides support for user initiated checkpoint. Users can insert checkpoint invocation manually in the application arbitrarily. Xue et al. propose a user-level file system which can guarantee the consistency between the application and its files during checkpoint and restart [33].

A lot of work tried to reduce the overhead of checkpoint. Liu et al. used exponential distributions to model the system failure, and proposed a reliability-aware checkpoint/restart method [34]. Shastry et al. found that the optimal checkpoint interval was approximately directly proportional to the checkpoint cost while inversely proportional to shape parameter [21].

7 Conclusion

In this paper, we propose a checkpoint-based backfill algorithm, and evaluate it using real workload traces. Our analysis indicates that the checkpoint-based backfill algorithm can effectively improve the job scheduling in waiting time, slowdown and mean queue length by up to 40%. The checkpoint/restart overhead is also analyzed based on the real trace from Argonne Intrepid system. The results show that only 4% of the jobs need to be checkpointed due to preemption. This demonstrates that our checkpoint-based algorithm is able to improve overall system utilization considerably without spending significant amount of system resources on checkpoint/restart operations.

We plan to extend our work in several directions. We will develop an aging algorithm to adaptively tune the parameter p so as to achieve approximating optimal performance. Moreover, we will study other split policies, such as splitting the running process into three segments. We will also apply the checkpoint-ability to other job scheduling algorithms.

Acknowledgments. We want to express our thanks to anonymous reviewers who gave valuable suggestion that has helped to improve the quality of the manuscript. This work gets support partly from Chinese "863" project 2008AA01A204, 2010AA012403, NSFC project 61103021, NSF grants 0546301 (CAREER), 0915861, 0937908, and 0958311, in addition to a joint faculty appointment between Oak Ridge National Laboratory and NC State University, as well as a senior visiting scholarship at Tsinghua University.

References

1. Zhang, Y., Franke, H., Moreira, J., Sivasubramaniam, A.: An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. IEEE Transactions on Parallel and Distributed Systems 14(3), 236–247 (2003)
2. McCann, C., Vaswani, R., Zahorjan, J.: A dynamic processor allocation policy for iviukiprogrammed shared-memory multiprocessors. ACM Transactions on Computer Systems 11(2), 146–178 (1993)
3. Majumdar, S., Eager, D.L., Bunt, R.B.: Scheduling in multiprogrammed parallel systems, vol. 16. ACM (1988)
4. Lifka, D.: The ANL/IBM SP Scheduling System. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
5. Platform Computing Inc. Platform LSF (2012), `http://www.platform.com/products/LSFfamily/`
6. Adaptive Computing Enterprises Inc. MOAB workload manager (2012), `http://www.supercluster.org/moab/`

7. Jackson, D., Snell, Q., Clement, M.: Core Algorithms of the Maui Scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
8. Adaptive Computing Enterprises Inc. PBS/Torque user manual (2012), http://www.clusterresources.com/torquedocs21/usersmanual.shtml
9. Skovira, J., Chan, W., Zhou, H., Lifka, D.: The EASY – LoadLeveler API Project. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 41–47. Springer, Heidelberg (1996)
10. Parallel Workloads Archive (2012), http://www.cs.huji.ac.il/labs/parallel/workload/
11. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Proceedings of the International Conference on Parallel Processing Workshops, pp. 514–519. IEEE (2002)
12. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: 2001 IEEE International Workshop on Workload Characterization, WWC-4, pp. 140–148. IEEE (2001)
13. Chiang, S.-H., Arpaci-Dusseau, A., Vernon, M.K.: The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 103–127. Springer, Heidelberg (2002)
14. Zhai, J., Chen, W., Zheng, W.: PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. ACM SIGPLAN Notices 45, 305–314 (2010)
15. Tang, W., Desai, N., Buettner, D., Lan, Z.: Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–11. IEEE (2010)
16. Bailey Lee, C., Schwartzman, Y., Hardy, J., Snavely, A.: Are User Runtime Estimates Inherently Inaccurate? In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 253–263. Springer, Heidelberg (2005)
17. Berkeley Lab Checkpoint/Restart, BLCR (2012), https://ftg.lbl.gov/projects/CheckpointRestart/
18. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: Plfs: A checkpoint filesystem for parallel applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 21. ACM (2009)
19. Liu, Y., Nassar, R., Leangsuksun, C., Naksinehaboon, N., Paun, M., Scott, S.L.: An optimal checkpoint/restart model for a large scale high performance computing system. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–9. IEEE (2008)
20. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Automated application-level checkpointing of MPI programs. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 84–94. ACM (2003)
21. Mallikarjuna Shastry, P.M., Venkatesh, K.: Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications. Analysis 2(08), 2690–2697 (2010)
22. TOP500 Supercomputing web site (2012), http://www.top500.org
23. Naik, H., Gupta, R., Beckman, P.: Analyzing checkpointing trends for applications on the IBM Blue Gene/P system. In: International Conference on Parallel Processing Workshops, ICPPW 2009, pp. 81–88. IEEE (2009)

24. Feitelson, D.G., Weil, A.M.: Utilization and predictability in scheduling the ibm sp2 with backfilling. In: Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing, Parallel Processing Symposium, IPPS/SPDP 1998, pp. 542–546. IEEE (1998)
25. Ward Jr., W.A., Mahood, C.L., West, J.E.: Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 88–102. Springer, Heidelberg (2002)
26. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Transactions on Parallel and Distributed Systems 18(6), 789–803 (2007)
27. Snell, Q.O., Clement, M.J., Jackson, D.B.: Preemption Based Backfill. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 24–37. Springer, Heidelberg (2002)
28. Adaptive Computing Enterprises Inc. Preemption Policies (2012), http://www.adaptivecomputing.com/resources/docs/maui/8.4preemption.php
29. Perkovic, D., Keleher, P.J.: Randomization, speculation, and adaptation in batch schedulers. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM), p. 7. IEEE Computer Society (2000)
30. Jette, M.A.: Performance characteristics of gang scheduling in multiprogrammed environments. In: ACM/IEEE 1997 Conference on Supercomputing, pp. 54–54. IEEE (1997)
31. Jette, M., Storch, D., Yim, E.: Gang scheduler-timesharing the cray t3d, pp. 247–252. Cray User Group (1996)
32. Sosa, C., Knudson, B.: IBM System Blue Gene/P Solution: Blue Gene/P Application Development (2007), http://www.redbooks.ibm.com/abstracts/sg247287.html
33. Xue, R., Chen, W., Zheng, W.: CprFS: a user-level file system to support consistent file states for checkpoint and restart. In: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 114–123. ACM (2008)
34. Liu, Y., Nassar, R., Leangsuksun, C., Naksinehaboon, N., Paun, M., Scott, S.L.: An optimal checkpoint/restart model for a large scale high performance computing system. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–9. IEEE (2008)

Multi-objective Processor-Set Selection for Computational Cluster-Systems

N. Peter Drakenberg

Deutsches Klimarechenzentrum GmbH
Bundesstraße 45a
20146 Hamburg, Germany

Abstract. A formalization of the processor-set selection problem for parallel job-schedulers is presented and proven to be NP-hard in the strong sense. Nonetheless, a simple and straightforward algorithm for the problem is presented, and is seen to perform well in practice when used in combination with more realistic, less uniform, cost-structures.

Keywords: parallel job scheduling, interconnection networks, topology awareness, resource allocation, processor-set selection.

1 Introduction

The problem of determining *when* and *where* to run a parallel program on a parallel computer is usually referred to as job scheduling, and is but one incarnation of scheduling and resource allocation problems which recur at a variety of levels and scales in the context of parallel processing.

Parallel programs running on parallel computers are susceptible to run-time delays due to communications congestion, and it therefore benefits each program (and other simultaneously running programs) if its communicating subtasks are mapped to processors that are located close together in the communication topology of the computer system being used. For reasons of efficiency/performance, a parallel program's subtasks should thus be mapped to processors such that intensively inter-communicating pairs of subtasks are as closely located as possible and vice versa, and a variety of methods to do so have been developed over the last 25 years [3–8, 36].

Unfortunately, current parallel job-schedulers tend to assign programs to run on more-or-less randomly assembled collections of processors [26, 34], and in such cases a carefully determined mapping of program subtasks to logical processors is essentially worthless. Making parallel job-schedulers select processor collections that are closely located (i.e., 'compact') in the communication topologies of parallel computer systems is not only an efficiency/performance issue, however. For sites where single jobs never (or very very rarely) use an entire system, efficient and reliable selection of compact processor collections by the job-scheduler would enable hardware configurations with lower bisection bandwidths to remain competitive for the workloads in question, and could thereby strongly influence system procurement costs.

In this paper we present a flexible formalization of the processor-set selection problem for parallel job-schedulers. We show that the resulting optimization problem is NP-hard, and present results suggesting that the use of more realistic system models, with less homogenous cost-structures, may improve the quality of processor-set selections obtained for practically occurring problem instances.

2 Related Work

Parallel job scheduling and topology aware task mapping are well known problems that have been extensively studied since the early 1990s and early 1980s, respectively. Each field has an extensive collection of litterature associated with it, and the coverage in subsections below is necessarily very brief.

2.1 Parallel Job Scheduling

The dominant scheme for scheduling parallel jobs on parallel computers is known as *variable partitioning* [14], the meaning of which is that when scheduled to run, each job is assigned a set of processors (i.e., a partition) of the requested size that it keeps and uses throughout its lifetime. In early parallel job schedulers partitions were allocated strictly in a first-come first-served (FCFS) manner to submitted jobs, with the result that system utilization most typically was below 60 % [18]. A widely used refinement of the scheme just described is to also apply a technique known as *backfilling* [23], whereby jobs that are not first in line to be started are nonetheless started, when doing so is possible without affecting the expected starting-time of the job that is currently first in line to be started. Through backfilling and other improvements, such as ordering jobs by priority rather than strictly FCFS, it has become entirely realistic to achieve utilization figures above 80 % [18, 25, 30].

All currently used parallel job schedulers (e.g., PBS, LSF, LoadLeveler, SLURM, Sun/Oracle GE) use variable partitioning schemes with backfilling and order waiting jobs by priority. Additional features such as fair-share scheduling, consumable resources, and job preemption are also supported in many cases. Strategies for choosing the actual subsets of nodes to use for each job, are however essentially limited to selecting the least loaded nodes, selecting available nodes in some constant sequential order, or selecting nodes such that the number of consecutive sets of nodes is minimized. However, due to fragmentation and boundary effects, these strategies tend not to yield particularly encouraging results for recent generations of parallel computer systems.

2.2 Topology Aware Task Mapping

Substantial theoretical and practical research on interconnect topologies and topology-aware mapping of tasks to processors was performed from the early 1980s to the mid 1990s. Heuristic techniques such as pairwise exchange were initially suggested [6, 21], but subsequently found not to scale well. Instead, methods were developed based on recursive partitioning [13] and graph contraction [3]. Yet other methods were developed based on techniques such as simulated annealing [7] and genetic algorithms [8]. The mapping problems considered were not always constrained to having predefined tasks. Mapping of recurrence relations [29] and loop iterations [11] onto regularly connected grids were other aspects of mapping being studied.

Due to the deployment in the mid 1990s and onwards of virtual cut-through and wormhole routing [19, 27] and the emergence of faster interconnects, message latencies became relatively unimportant and research in topology aware mapping of computations died down. However, with the reemergence of three-dimensional torus networks in recent top-of-the-line supercomputers [1, 10, 17], and due to other reasons as well, message latencies are again gaining in significance and interest in topology aware task mapping is increasing [4, 5, 26, 36].

Processor-set selection by job-schedulers is an area that has been investigated previously [2, 33, 34], but to our knowledge only for mesh and torus topologies, whereas hierarchical topologies have received very little attention.

3 Problem Formulation and Notation

There exists a vast variety of communication network topologies for parallel computing, but because of the dominance of flat and hierarchically structured network topologies (e.g., Clos networks [9] and fat-trees [22]) in current and recent Top500 rankings,[1] we only consider such network topologies in the present paper. For a fat-tree network with 256 compute nodes, overall performance differences of 200–400% due to bad processor selections have been reported [26], and our interest in flat and hierarchical topologies is thus not misplaced.

3.1 A Concrete Sample-System

Current high-performance computer systems are typically composed of several hundred or thousand compute-nodes, connected to one another by communication networks of various kinds, as well as power-distribution networks, cooling networks (for water-cooled systems), etc.

Figure 1 shows a fairly typical 256-node/2048-core system with nodes arranged in 10 racks with 25 nodes in each rack, and with the six nodes thereby not accounted for located in the center-rack (together with login-nodes, management-nodes, and a 288-port Infiniband switch). The limit of 25 nodes per rack is to prevent excessive floor loading, and as expected nodes 1–25 (counted from the bottom and upwards) are located in rack 1, nodes 26–50 in rack 2, and so on.

The system's job-scheduler communicates with the nodes over a dedicated control/monitoring network (plain ethernet) in which a separate switch is responsible for

[1] November 2011: 41.8 % InfiniBand, 44.8 % Gigabit Ethernet,
November 2010: 45.2 % InfiniBand, 42.8 % Gigabit Ethernet.

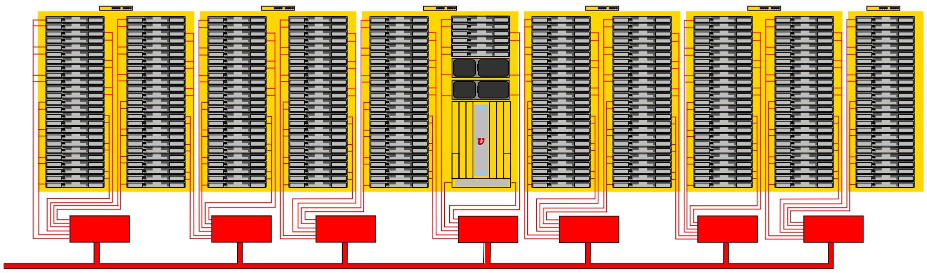


Fig. 1. A 256-node system, with its power- (red) and control- (yellow boxes) networks

the nodes in consecutive pairs of racks. A yellow rectangle has been placed behind each group of nodes managed by a common switch in Fig. 1. The power distribution network is shown in red in Fig. 1. The red boxes shown below the compute node racks represent fuse-blocks, and as can be seen in the figure, nodes 1, 4, 5, 8, 9, 12, 13 of each rack are connected to one power-line, while nodes 2, 3, 6, 7, 10, 11 of the same rack are connected to a different power-line, and in quite a few cases also to a different fuse-block.

The communication between compute-nodes is performed over InfiniBand. Internally, the 288-port InfiniBand switch used has a Clos topology [9, 32], and has 12 compute nodes connected to each line-card (local routing of messages is performed by each line card). Compute nodes are connected in-order to line-cards, so that nodes 1–12 are connected to line-card 1, nodes 13–24 to line-card 2, nodes 25–36 to line card 3 etc.

With all the different networks that are involved and their differing structures, it is clear that no single hierarchy (such as can be defined when using SLURM [35] or PBS Pro job schedulers) will suffice to simultaneously take the various networks of the system into account. To make matters concrete, it is desirable that a job makes use of as few InfiniBand line-cards as possible (for communication efficiency reasons), but at the same time it should ideally also make use of as few power-lines as possible (so that each given job is affected by fewer blown fuses), and it should be assigned to nodes below as few different control-network switches as possible (so that each given job can be harmed by as few failed switches as possible).

Finally, the system in Fig. 1 is air-cooled, with cold air being provided from below on the system's font-side. For this reason the use of physically lower positioned nodes is preferrable over the use of physically higher positioned nodes, and because of flow of hot air around the sides of the system from rear to front, it is more desirable to use centrally positioned nodes than to use nodes nearer to the sides.

With all things considered, it is clear that the limitations and constraints of the various networks present in a cluster system can interact in non-trivial ways with respect to processor set selection, and that when cooling and energy consumption issues are considered, no two compute-nodes are truly identical unless they also occupy exactly the same physical location. These properties stand in noticeable contrast to most idealized scheduling and selection problems, in which sets of indentical resources and/or objects tend to be assumed.

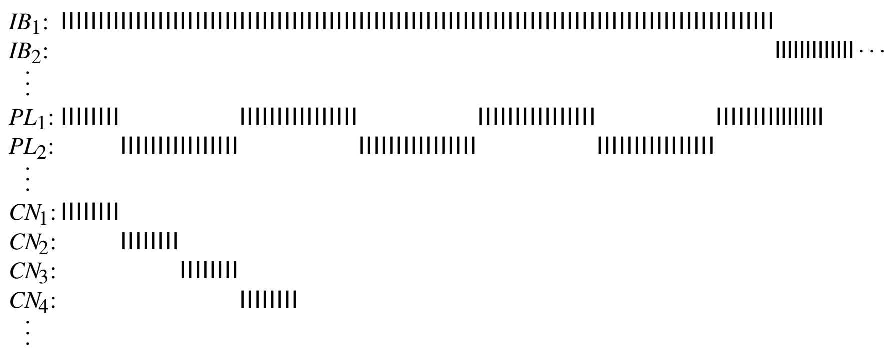


Fig. 2. Typical processor-set definitions, shown as bit-vectors

3.2 Problem Formalization

To address the processor-set selection (PSS) problem such that the different types of constraints mentioned above can be accounted for, we define a set of processor numbers corresponding to each practical constraint that involves that particular collection of processors. For example, since compute nodes 1–12 are attached to line-card 1 on the InfiniBand switch and compute nodes 13–24 are attached to line-card 2, etc., we define sets: $IB_1 = \{1, \ldots, 96\}$, $IB_2 = \{97, \ldots, 192\}$, etc. Similarly, we define sets $PL_1 = \{1, \ldots, 8, 25, \ldots, 40, 57, \ldots, 72, 89, \ldots, 104\}$, $PL_2 = \{9, \ldots, 24, 41, \ldots, 56, 73, \ldots, 88\}$, etc., for processors connected to a common power-line, and continue in the same manner with processor-sets for control-network switches, as well as for each group of processors on the same compute-node (CN_1, CN_2, etc.). For the processor-set definitions given above it is assumed that each compute-node has eight processors, and an incomplete but nonetheless illustrative rendering of such processor-sets is given in Fig. 2 (above). A complete example of processor-set definitions is also presented in the appendix.

Now, let $U = \{1, \ldots, m\}$ for some $m \in \mathbb{N}$ be the set of all processors that are under the job-scheduler's control. For each processor-set $P_i \subseteq U$, $i \in \{1, \ldots, n\}$, defined as described above (i.e., $P_1 = IB_1$, $P_2 = IB_2$, etc.), an associated ℓ-element cost-vector $\mathbf{c}_i \in \mathbb{N}^\ell$ is defined. Given the collection of defined processor sets and their associated cost-vectors, we view the total cost of a processor-set selection $S \subseteq U$ as being given by the vector-sum (denoted by $\oplus$) of those cost-vectors $\mathbf{c}_i$ for which the corresponding processor-set P_i has at least one element in common with S. That is, the cost of a processor selection S can be defined as:[2]

$$cost(S) = \bigoplus_{i=1}^{n} \mathbf{c}_i \Big[(P_i \cap S) \neq \emptyset \Big], \tag{1}$$

from which it follows that the cost of a selection is itself also a cost-vector.

[2] Using the notation introduced in [16, page 24] whereby a pair of brackets enclosing a boolean expression evaluates to 1 when the enclosed expression evaluates to true and evaluates to 0 when the enclosed expression evaluates to false.

Cost-vectors are compared lexicographically, which means that we consider the cost $\mathbf{c}_i \in \mathbb{N}^\ell$ as being lower than $\mathbf{c}_j \in \mathbb{N}^\ell$ when $\mathbf{c}_i \prec \mathbf{c}_j$, with the definition of the latter notation given by:

$$\begin{aligned}\mathbf{c}_i \prec \mathbf{c}_j \;\equiv\; & (\mathbf{c}_i)_1 < (\mathbf{c}_j)_1 \vee \big((\mathbf{c}_i)_1 = (\mathbf{c}_j)_1 \wedge (\mathbf{c}_i)_2 < (\mathbf{c}_j)_2\big) \vee \cdots \\ & \cdots \vee \big((\mathbf{c}_i)_1 = (\mathbf{c}_j)_1 \wedge \cdots \wedge (\mathbf{c}_i)_{\ell-1} = (\mathbf{c}_j)_{\ell-1} \wedge (\mathbf{c}_i)_\ell < (\mathbf{c}_j)_\ell\big)\,,\end{aligned}$$

wherein the notation $(\mathbf{c}_i)_k$ is used to indicate the kth element of $\mathbf{c}_i \in \mathbb{N}^\ell$.

Given a set $P_\mathrm{a} \subseteq U$, of currently available processors and a number of required processors $r \in \mathbb{N}$, the processor-set selection problem is to find a solution, X, to the following optimization problem:

$$\min_{\substack{\prec \\ X \subseteq P_\mathrm{a}}} \left\{ cost(X) \;\middle|\; |X| \geq r \right\}, \tag{2}$$

or determine that no such solution X exists.

3.3 Motivations

Compared to plain scalar values, lexicographically ordered cost-vectors have the advantages that different concerns (e.g., communication costs, electrical reliability, energy consumption, etc.) are easy to keep separate and differently prioritized, and that cost-constraints to decide the acceptability of selections can be defined in terms of individual cost-vector components. When the cost of each processor-set is limited to being a single scalar value, such tasks become more difficult and thus error-prone.

As described above, a processor-set is defined for a group G of processors to indicate that it is desirable (with respect to some criterion) to let the processors of G be selected for jobs of size $\leq |G|$. That is, selecting processors from two different processor-sets P_1 and P_2, with cost-vectors $\mathbf{c}_1$ and $\mathbf{c}_2$, when any one of P_1 or P_2 alone would suffice, leads to a total selection cost of $\mathbf{c}_1 \oplus \mathbf{c}_2$ instead of $\min_\prec(\mathbf{c}_1, \mathbf{c}_2)$. In many real cases, however, the cost implied by a processor-set definition only arises when the processor-set is straddled by a processor selection and not when the requested number of processors can all be selected within the processor-set, suggesting that common and essential problem-features are not captured by the model.

The idea behind the processor-set/cost-vector model for processor selection, however, is that the *surplus* costs incurred by undesirable processor selections will cause minimization procedures to find more suitable selections whenever such exist,[3] and that it therefore should be irrelevant whether the base (i.e., minimum) total cost of each particular processor selection is zero or some other value. In the case of P_1 and P_2 just above, for example, it is not important that the minimum selection cost is $\min_\prec(\mathbf{c}_1, \mathbf{c}_2)$ instead of $\mathbf{0}$, but what is important is that any attempt at selecting processors from both sets will lead to a total selection cost that lies as much above this minimum selection cost as is given by the (lexicographically) larger value of $\mathbf{c}_1$ and $\mathbf{c}_2$.

[3] In this context we consider such minimization procedures to be exact, despite the results in Section 4 and Section 6.

4 Complexity of the Processor-Set Selection Problem

The theorem below establishes that PSS is NP-hard in the strong sense, which in turn implies that an algorithm that efficiently delivers exact answers for all conceivable problem instances is unlikely to exist.

Theorem 1. *The PSS problem is NP-hard in the strong sense.*

Proof: The proof is by reduction from the set-union knapsack problem (SUKP). The SUKP is defined as follows:

> There is a universe of m elements, denoted by $1, \dots, m$, and n items, with the set of elements constituting item i denoted by P_i, and such that the union of all items is the set of all elements, $\bigcup_{i=1}^{n} P_i = \{1, \dots, m\}$. The value of item i is denoted by v_i and the weight of element j is denoted by s_j. The capacity of the knapsack is b. For any $K \subseteq \{1, \dots, n\}$, we define P_K as $P_K = \bigcup_{i \in K} P_i$. The objective of the SUKP is to find a solution (K) to the following mathematical program:
>
> $$\max \left\{ \sum_{i \in K} v_i \;\middle|\; \sum_{j \in P_K} s_j \le b,\ K \subseteq \{1, \dots, n\} \right\}, \tag{3}$$
>
> or less formally expressed; to find a collection of items of maximum total value such that the weight of their constituent elements does not exceed the knapsack capacity b.

In [15], Goldschmidt *et al.* show that SUKP is NP-hard in the strong sense, even in the case when $|P_i| = 2$, $s_j = 1$ and $v_i = 1$, for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$.

Given $b \ge m$, the problem in Eq. (3) is trivial. The solution K is simply chosen to contain all items P_i, $i = 1, \dots, n$. Given $b < m$, one or more elements (and the items containing these elements) must be removed from the solution for $b \ge m$. The elements to remove in order to satisfy Eq. (3) are those for which the items that are consequentially removed contribute the least to the overall value of the knapsack. With R denoting the set of elements to remove, and assuming that $s_j = 1$ for all $j \in \{1, \dots, m\}$, this can be expressed as:

$$\min_{R \subseteq U} \left\{ \sum_{i=1}^{n} v_i \Big[(R \cap P_i) \ne \emptyset \Big] \;\middle|\; |R| \ge m - b \right\}, \tag{4}$$

where $U = \{1, \dots, m\}$. Assuming single-element cost-vectors, and that $P_{\mathrm{a}} = U$, the minimization problem in Eq. (4) is identical to that in Eqs. (1–2), and the proof is thereby complete. ■

5 A Simple Processor-Set Selection Algorithm

As should be expected from the results arrived at in Section 4, the simple algorithm presented below is by no means guaranteed to find optimal solutions.

The main idea behind the algorithm SELECTPROCESSORSET (in Fig. 3) is to start with the set of all currently available processors P_a being the set of selected processors, and then successively remove subsets of P_a that correspond to defined processor sets (i.e., $P_i, i \in \{1, \ldots, n\}$), until removing any further such subsets would leave less than the required number, r, of processors selected.

The way in which processor-sets P_i are removed from the set of selected processors has some similarities to the concept of *reaching* as used in dynamic programming [12], in which case solutions to subproblems are computed before it is known whether these solutions will be of use in obtaining the final solution.

The algorithm begins by populating the array *remove* such that *remove*$[k]$ holds the defined processor set (i.e., one of $P_i, i \in \{1, \ldots, n\}$) of size k with the lexicographically largest associated cost-vector (among processor-sets of size k). The algorithm then proceeds to its main phase, in which *remove*$[k]$ is processed in sequence (for $k = 1, \ldots, |P_a| - r - 1$), by forming the union of *remove*$[k]$ with each element of C_p (i.e., the defined processor-sets). The size of each processor-set, s', so obtained is determined, and when the total cost of s' is lexicographically greater than that of the current value of *remove*$[|s'|]$, the value of s' will replace the current value of *remove*$[|s'|]$.

When *remove*$[|P_a| - r - 1]$ has been processed in the manner just described, the algorithm's suggestion for the best set of processors to remove from P_a such that q processors remain (where $q \geq r$) is stored in *remove*$[|P_a| - q]$, and consequently, the value given by $P_a \setminus ($*remove*$[|P_a| - r])$ is returned as the result of the SELECTPROCESSORSET algorithm.

Note that actual implementations (*vide infra*) of the algorithm in Fig. 3 compute cost-vectors for processor-sets when forming the unions $s \cup (p_k \cap P_a)$, or shortly thereafter, and do not as in Fig. 3 repeatedly redo this calculation (in MAXCOSTSET). This is done in Fig. 3 for the purpose of simplifying the presentation.

6 Algorithm Implementation and Evaluation

The algorithm in Fig. 3 and a worst-case exponential-time algorithm for the set-union knapsack problem[4] described by Goldschmidt *et al.* [15] were first implemented in Common Lisp [31], along with a simple framework to perform processor-set selection driven by the workload trace described below.

Having observed that the Lisp implementation of the algorithm in Fig. 3 behaved and performed reasonably, it was reimplemented in ANSI C [20] and interfaced to the parallel environment queue selection (PQS) API of the Sun Grid Engine (SGE) job-scheduler, and evaluated as described in Section 6.3 below.

[4] Similarly to how the algorithm in Fig. 3 operates internally, the set-union knapsack algorithm was used to determine which processors *not* to select.

```
Algorithm. SelectProcessorSet( $r$, $P_a$, $C_p$, $C_v$ )
Inputs:  $r \in \mathbb{N}\backslash\{0\}$: the number of processors requested,
         $P_a \in \mathbb{P}(\mathbb{N})$ : the set of currently available processors,
         $C_p = \langle p_1, \ldots, p_n \rangle$ : a sequence of processor-sets,
         $C_v = \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ : a sequence of corresponding cost-vectors.
Output: $S$: a set of processors ($\emptyset$ if selection impossible).
begin
  if $r \leq |P_a|$ then
    for $k \leftarrow 1$ to $n$ do
      $q \leftarrow p_k \cap P_a$;
      $remove[|q|] \leftarrow$ MaxCostSet( $q$, $remove[|q|]$, $C_p$, $C_v$ );
    od
    for $j \leftarrow 1$ to $|P_a| - r - 1$ do
      if $remove[j] \neq \emptyset$ then
        $s \leftarrow remove[j]$;
        for $k \leftarrow 1$ to $n$ do
          $s' \leftarrow s \cup (p_k \cap P_a)$;
          if $|s'| > |s|$ then
            $remove[|s'|] \leftarrow$ MaxCostSet( $s'$, $remove[|s'|]$ , $C_p$, $C_v$ );
          fi
        od
      fi
    od
    $S \leftarrow P_a \setminus remove[|P_a| - r]$;
  else
    $S \leftarrow \emptyset$;
  fi
end

proc MaxCostSet( $s_1, s_2, \langle p_1, \ldots, p_n \rangle, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ ) : $\mathbb{P}(\mathbb{N}) \equiv$
  $\mathbf{c}_1 \leftarrow \mathbf{0}$;
  $\mathbf{c}_2 \leftarrow \mathbf{0}$;
  for $k \leftarrow 1$ to $n$ do
    $\mathbf{c}_1 \leftarrow \mathbf{c}_1 \oplus \left[(p_k \cap s_1) \neq \emptyset\right] \mathbf{v}_k$;
    $\mathbf{c}_2 \leftarrow \mathbf{c}_2 \oplus \left[(p_k \cap s_2) \neq \emptyset\right] \mathbf{v}_k$;
  od
  if $\mathbf{c}_1 \prec \mathbf{c}_2$ then
    return $s_2$;
  elif $\mathbf{c}_2 \prec \mathbf{c}_1$ then
    return $s_1$;
  else
    return (if $rectrand(0.0, 1.0) < 0.5$ then $s_1$ else $s_2$ fi);
  fi
end
```

Fig. 3. The simple processor-set selection algorithm

6.1 Workload Details

The workload used for evaluation is a trace of submitted and executed jobs corresponding to one week (24×7 hours) of wall-clock time on the system described in Section 3.1 (and depicted in Fig. 1). The trace begins at a time directly following a restart of the entire system, and jobs started prior to the window of observation need therefore not be considered.

Looked upon in further detail, the workload used can be seen to have the following characteristics:

21902 jobs in total with a mean job-size of 44.6 ± 27.4 processors (5.57 ± 3.43 nodes), and with 6692 jobs using 8 or fewer processors (i.e., using at most one node). For jobs using 8 processors or less, the mean run-time is 243 ± 390 sec. The mean run-time of jobs using more than 8 processors is 848 ± 3170 sec., and the mean run-time of jobs using more than 8 processors and more than 900 sec. of run-time is 8375 ± 6445 sec. (i.e., 2.3 ± 1.8 hours).

In all cases, above mentioned run-times refer to wall-clock times and have thus not been scaled by the number of participating processors. Note that unlike the corresponding situation for more conventional scheduling algorithms, when evaluating processor-set selection algorithms it is important that the workload exhibits substantial variation in the number of unused processors. This criterion is satisfied by the described workload.

On the system in question, the (wall-clock) run-time is limited to 8 hours for all parallel jobs. For this reason it is common practice to use rather long *job-chains* (i.e., the last action of a running job is usually to submit a new instance of itself). In order to avoid obtaining misleading results, the use of job-chains must be properly accounted for when replaying the job-submission traces.

6.2 Prototype Evaluation

For the evaluation of Lisp algorithm implementations, the workload described above was simplified by considering each compute-node to have only one processor (instead of eight), and dividing the processor counts of all requests by eight.

The processor-set/cost-vector configuration comprised a total of 334 processor-set definitions and associated (5-element) cost vectors, corresponding to InfiniBand line-cards(22), control-network switches (7), power-lines (41), fuse-blocks (8) and compute-nodes (256). The Figure-3-algorithm was run with two different sets of cost-vectors for the mentioned processor-set definitions. In the first such set of cost-vectors, the cost of compute-node processor-sets has been set to reflect the relative desirability of using some compute nodes over others from an energy- and cooling-perspective, as described in Section 3.1. In the second set of such cost-vectors, all compute-node processor-sets have been assigned identical cost-vectors.

Runs with the described workload-trace were performed with the simple algorithm using both sets of cost-vectors for all processor-selection requests, whereas the exact

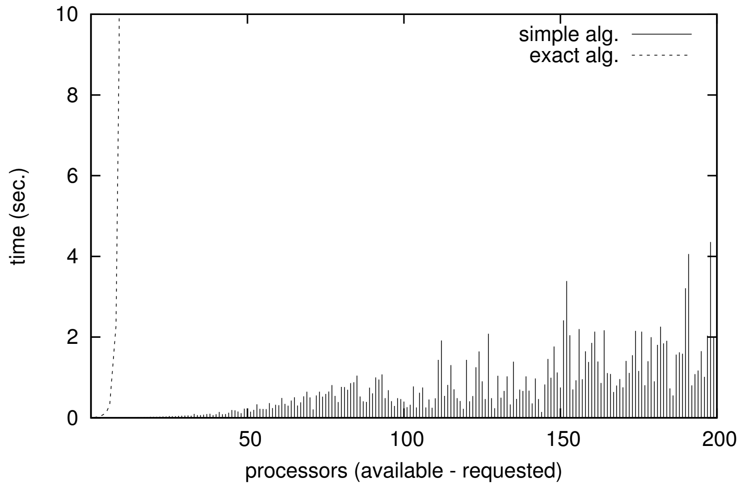


Fig. 4. Mean execution-times of the Lisp implementations of the simple and the exact processor-set selection algorithms, as a function of $|P_a| - r$, for each request. All measurements were made using compiled Lisp code with CMUCL [24] (release 19d) running on an Intel Core2 Duo T1700 1.8 GHz processor with 2Gb physical memory.

algorithm was run only when the problem size was small enough ($|P_a| - r \leq 12$) that its execution-time was still reasonable (see Fig. 4). The selection decisions made by the simple algorithm using the more realistic (i.e., non-uniform) cost-vectors were the ones actually used to change the selected-state of nodes maintained by the simulation, and thus influencing the starting conditions for subsequent processor selections.

For the limited subset of selection problems that could be handled by the exact algorithm, the simple algorithm using the more realistic cost-vectors arrived at a different selection than the exact algorithm (also using realistic cost-vectors) for only 23 out of 4272 multi-node jobs/requests. When comparing results obtained for the two different sets of cost-vectors and using the uniform cost-vectors to judge selection quality, selections of equal cost were obtained with both sets of cost-vectors for 15155 out of 15210 multi-node jobs. For the remaining 55 jobs, better processor selections were obtained using realistic cost-vectors than using uniform cost-vectors for 41 jobs ($\sim$75 %), even though solution quality was judged according to the uniform cost-vectors. Correspondingly for the 4272 requests that could be handled by the exact algorithm, the exact algorithm (now using uniform cost-vectors) arrived at better processor selections than the simple algorithm using realistic cost-vectors for 4 jobs, and the simple algorithm using realistic cost-vectors in turn arrived at better processor selections than the same algorithm using uniform cost vectors for 4 jobs and worse for only 1 job, with solution quality again judged according to the uniform cost-vectors. We view this as indications that artificially introduced non-uniformness in cost-structure definitions may contribute to improved processor-set selection quality.

Prototyping the algorithms in Lisp allowed us to focus efforts on key issues, and postpone less immediately relevant matters such as file-formats for describing processor-sets and cost-vectors, implementations of bit-vectors, and dynamic memory management, that need to be addressed when using a language such as C. However, as can be inferred from Fig. 4, showing mean execution-times (without error-bars, since these would have completely cluttered the diagram), the execution-times fluctuate noticeably, and one of the main reasons for this variability is that bit-vectors have been implemented as `bignums` [31], the sizes of which vary in correspondance with the most significant bit that is set.

6.3 SGE-Implementation Evaluation

The evaluation of the SGE-interfaced algorithm implementation used the workload described in Section 6.1 without modification (i.e., assuming 2048 processors in total and 8 processors per node). Compared to the description in Section 6.2, processor-set definitions were expanded as is implied by having 8 processors per node instead of only 1, and cost-vectors were expanded by adding a new first element, through which the selection of processors on as few different nodes as possible was made the primary objective.

A distinct SGE-master was set up on one of the management hosts of the system described in Section 3.1, and its 256 compute nodes were cloned and subsequently simulated through Xen hypervisors and virtual machines on 16 compute nodes of the same system (i.e., with 16 virtual compute nodes on each real compute node). Since the wall-clock execution-time of each job is known from the workload-trace, each job simply sleeps an amount of time corresponding to its execution-time,[5] and therefore no substantial load arises on the virtual machines. This method of replaying the workload-trace enabled us to observe the described algorithm and its implementation under very authentic conditions, with a very modest impact on the physical system.

The mean recorded execution times of the SGE-interfaced processor-set selection algorithm as a function of the number of processors *not* to select for each corresponding job (i.e., $|P_\mathrm{a}| - r$) is shown in Fig. 5 (a). We find the observed running-times of the algorithm to clearly be within acceptable limits, particularly in view of the fact that systems of the kind in question should preferably be sufficiently heavily used that the number of idle processors only rarely can be counted in the hundreds or thousands, and that in the common case that parallel jobs are always given complete nodes for themselves, problem sizes can be reduced as was done in Section 6.2. Finally, Fig. 5 (a) does indicate some execution-time irregularities, but the exact sources of these are currently not clear to us.

As explicitly stated and as implied, respectively, in the previous discussion, the algorithm was run with a processor-set and cost-vector configuration such that the primary objective was to select processors on as few different nodes as possible and the secondary objective was to select processors connected to as few different InfiniBand line-cards as possible. The third-, fourth-, fifth-, and sixth-level objectives were to minimize the use of control-network switches, power-lines, fuse-blocks, and compute-node

[5] In order to ensure proper treatment by the SGE job-accounting machinery, the sleep operations were performed by letting an `mpiexec`-command in each job-script invoke `sleep` commands (in parallel) that slept for the appropriate length of time.

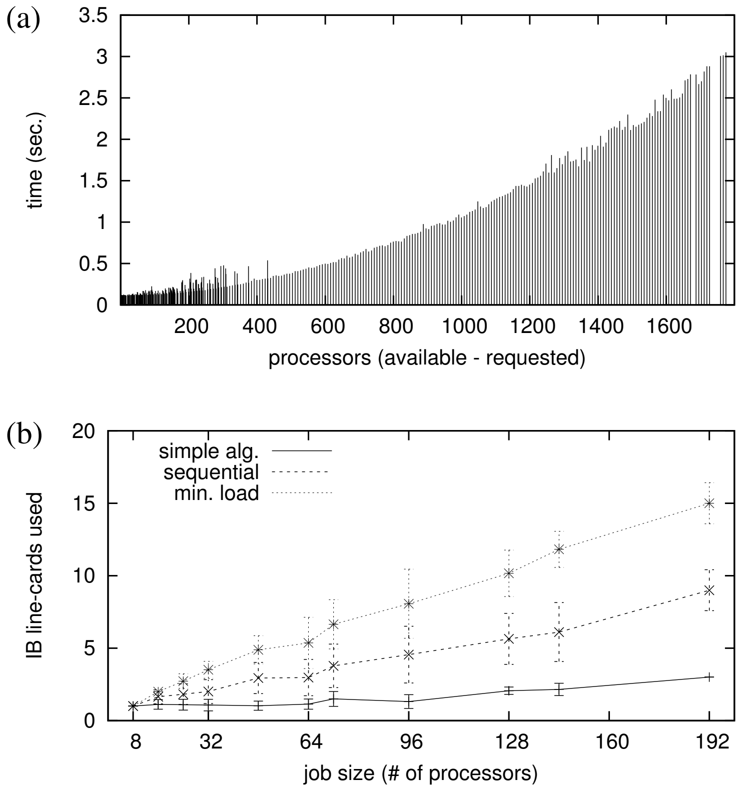


Fig. 5. (a) Mean execution-times of the SGE-interfaced processor-set selection algorithm as a function of $|P_{\mathrm{a}}| - r$, measured on a Sun Fire X4100 with 2.4 GHz AMD Athlon processors and 4 Gb physical memory. (b) Average number of different InfiniBand line-cards used by jobs of various sizes for three different processor-set selection methods (error-bars indicate standard deviations).

energy/cooling costs, respectively. The primary objective was achieved for all jobs (in part because all jobs in the workload request a multiple of 8 processors). Fig. 5 (b) presents the outcome with respect to the secondary objective, compared to the two processor selection strategies that the SGE has built-in (i.e., sequentially by increasing node numbers, and least loaded nodes). With respect to communication locality, as can be seen in Fig. 5 (b), the processor-set selection method described in this paper represents a clear improvement over both of the strategies provided in the SGE, and which are widely used in practice (also by other job-schedulers).

7 Summary and Conclusions

We have presented a model for processor selection by parallel job-schedulers that is conceptually simple, easy to understand, and flexible with respect to the kinds of constraints that can be accounted for. The model is also easily extended such that different processor-set and cost-vector definitions can be given for different ranges of job sizes.[6] In this way, jobs of different sizes can be steered towards different regions of a system, constraints only affecting jobs of specific sizes can be accounted for, and processor-set/cost-vector definitions can be kept shorter.

The resulting minimization problem for optimal processor selection was proven to be NP-hard in the strong sense. A simple (approximative) algorithm is nonetheless presented and shown to run sufficently fast to be practically useful and to yield processor selections of acceptable quality and that represent a clear improvement over processor selection strategies that are currently in widespread practical use. The quality of solutions obtained by the algorithm appears to also benefit slightly from less uniformly defined processor-set costs, suggesting that more realistic cost models can bring both direct and indirect advantages.

Concerning job-scheduler based processor-set selection in general, it has been observed on multiple occasions that imposing topology-related constraints on processor selection usually leads to longer waiting times and reduced overall system utilization (e.g., see [2, 28]), suggesting that such mechanisms bring little benefit. On the other hand, by successfully imposing topology-related constraints on processor selection, it may be possible to purchase a larger number of processors (because of a less expensive communications network), in which case a higher total throughput may be delivered despite reduced system utilization.

Acknowledgements. The author would like to thank Dipl. Ing. Frank Rambo for explaining the electrical wiring of the 256-node/2048-core cluster shown in Fig. 1, Dipl. Ing. Christian Schwede for setting up the Xen-based virtual replica of the already mentioned cluster, and Dipl. Inf. Gerald Vogt for patiently answering questions concerning energy consumption, cooling, and air-flow in the machine-rooms.

References

1. Adiga, N.R., et al.: Blue Gene/L torus interconnection network. IBM J. Res. Develop. 49(2/3), 265–276 (2005)
2. Aridor, Y., et al.: Resource allocation and utilization in the Blue Gene/L supercomputer. IBM J. Res. Develop. 49(2/3), 425–436 (2005)
3. Berman, F., Snyder, L.: On mapping parallel algorithms onto parallel architectures. J. Parall. Distr. Comput. 4(5), 439–458 (1987)
4. Bhanot, G., et al.: Optimizing task layout on the Blue Gene/L supercomputer. IBM J. Res. Develop. 49(2/3), 489–500 (2005)
5. Bhatelé, A., Bohm, E., Kalé, L.V.: Optimizing communication for Charm++ applications by reducing network contention. Concur. Pract. Exp. 23(2), 211–222 (2011)
6. Bokhari, S.H.: Assignment Problems in Parallel and Distributed Computing. Kluwer Academic Publishers, Norwell (1987)

[6] In fact, job-size differentiated processor-set and cost-vector definitions are supported by all algorithm implementations mentioned in this paper. These facilities were not used in the described evaluation runs, however.

7. Bollinger, S.W., Midkiff, S.F.: Heuristic technique for processor and link assignment in multicomputers. IEEE Trans. Comput. C-40(3), 325–333 (1991)
8. Chokalingam, T., Arunkumar, S.: Genetic algorithm based heuristics for the mapping problem. Comput. Oper. Res. 22(1), 55–64 (1995)
9. Clos, C.: A study of non-blocking switching networks. Bell Sys. Tech. J. 32(2), 406–424 (1953)
10. Cray Inc., Seattle, WA 98104, U.S.A.: Cray XT System Overview (2009), publication No. S–2423–22
11. Darte, A., Robert, Y.: Mapping uniform loop nests onto distributed memory architectures. Parallel Computing 20(5), 679–710 (1994)
12. Denardo, E.V.: Dynamic Programming: models and applications. Dover Publications, Mineola (2003)
13. Ercal, F., Ramanujam, J., Saddayappan, P.: Task allocation onto a hypercube by recursive mincut bipartitioning. J. Parallel Distrib. Comput. 10(1), 35–44 (1990)
14. Feitelson, D.G., Rudolph, L.: Parallel Job Scheduling: Issues and Approaches. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 1–18. Springer, Heidelberg (1995)
15. Goldschmidt, O., Nehme, D., Yu, G.: Note: On the set-union knapsack problem. Nav. Res. Logist. 41(6), 833–842 (1994)
16. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics, 2nd edn. Addison-Wesley, Reading (1994)
17. IBM Blue Gene Team: Overview of the IBM Blue Gene/ P project. IBM J. Res. Develop. 52(1/2), 199–220 (January/March 2008)
18. Jones, J.P., Nitzberg, B.: Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999. LNCS, vol. 1659, pp. 1–16. Springer, Heidelberg (1999)
19. Kermani, P., Kleinrock, L.: Virtual cut-through: A new computer communication switching technique. Computer Networks 3(4), 267–286 (1979)
20. Kernighan, B.W., Richie, D.M.: The C Programming Language, 2nd edn. Prentice-Hall, Englewood Cliffs (1988)
21. Lee, S.Y., Aggarwal, J.K.: A mapping strategy for parallel processing. IEEE Trans. Comput. C 36(4), 433–442 (1987)
22. Leiserson, C.E.: Fat-Trees: Universal networks for hardware-efficient supercomputing. IEEE Trans. Comput. C-34(10), 892–901 (1985)
23. Lifka, D.: The ANL/IBM SP Scheduling System. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
24. MacLachlan, R.A.: CMUCL User's Manual. Carnegie-Mellon University (November 2006), release 19d
25. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parall. Distr. Sys. 12(6), 529–543 (2001)
26. Navaridas, J., et al.: Effects of job and task placement on parallel scientific applications performance. In: Proc. 17th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing, pp. 55–61 (February 2009)
27. Ni, L.M., McKinley, P.K.: A survey of wormhole routing techniques in direct networks. Computer 26(2), 62–76 (1993)
28. Pascual, J.A., Navaridas, J., Miguel-Alonso, J.: Effects of Topology-Aware Allocation Policies on Scheduling Performance. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 138–156. Springer, Heidelberg (2009)
29. Quinton, P., van Dongen, V.: The mapping of linear recurrence relations on regular arrays. J. VLSI Signal Process. 1(2), 95–113 (1989)

30. Skovira, J., Chan, W., Zhou, H.: The EASY – LoadLeveler API Project. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 41–47. Springer, Heidelberg (1996)
31. Steele, J. G.L.: Common Lisp: the Language, 2nd edn. Digital Press, Burlington (1990)
32. Voltaire Ltd., Herzliya, Israel: Voltaire GridVision Integrated Grid Directors User Manual (May 2007), part Number: 399Z00038
33. Wan, M., Moore, R., Kremenek, G., Steube, K.: A Batch Scheduler for the Intel Paragon MPP System with a Non-Contiguous Node Allocation Algorithm. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 48–64. Springer, Heidelberg (1996)
34. Weisser, D., et al.: Optimizing job placement on the Cray XT3. In: Proceedings of Cray User Group 2006, Lugano, Switzerland (2006)
35. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003)
36. Yu, H., Chung, I.H., Moreira, J.: Topology mapping for Blue Gene/L supercomputer. In: Proc. of 2006 ACM/IEEE Conf. on Supercomputing, SC 2006. ACM, New York (2006)

Appendix: Configuration File Format

The decisions made by the presented algorithm for processor set selection are governed by a configuration file, that specifies the collection of available queues, the collections of processor sets, the corresponding cost-vectors, and the cost criteria any actual processor-set selection must satisfy in order to be seen as being acceptably good.

A concrete example of such a configuration file is provided in Fig. 6. The hypothetical target system is assumed to be deployed analogously to that shown in Fig. 1, but in order to keep the configuration file within the allotted page-limits, it comprises only 6 racks with 7 nodes in each rack, for a total of 42 nodes, with a relative positioning as follows, when looking at the system's front side:

n07	n14	n21	n28	n35	n42
n06	n13	n20	n27	n34	n41
n05	n12	n19	n26	n33	n40
n04	n11	n18	n25	n32	n39
n03	n10	n17	n24	n31	n38
n02	n09	n16	n23	n30	n37
n01	n08	n15	n22	n29	n36

As seen in Fig. 6, a configuration file consists of five separate parts, each beginning with a distinct keyword. The first part (line 1 in Fig. 6), simply names the queues in which jobs may run for which processor-sets are to be selected. The second part (lines 3–11) specifies how many processors that are available on each host and for which queues. More complicated cases than what is shown on lines 3–11 can also be handled. For example, by writing

```
[cluster@k007=1,3,5|standby@k007=2,4,6]
```

it is specified that processors 1, 3 and 5 are available through the queue `cluster` and processors 2, 4 and 6 through the queue `standby`, on the node `k007`.

Processor-Set Definitions

The processor-sets (such as described in Section 3.2 above) that are to be used are specified following the keyword `groups`. Distinct processor-sets for the processors of each indivdual compute node are defined on lines 13–21 in Fig. 6. The 42 nodes of our hypothetical target system are assumed to be connected to 7 different 6-port line-cards. Processor-sets corresponding to the processors whose nodes are directly attached to each line-card are defined on lines 21–27 of Fig. 6. For example, through

```
ib1={n01,n02,n03,n04,n05,n06},
```

a processor-set named `ib1` is defined, that comprises all processors on nodes `n01` through `n06`, and this in turn corresponds exactly to the set of processors attached to line-card 1 on the switch.

Analogously, the remaining processor-set definitions on lines 28–40 in Fig. 6, correspond to other factors such that it may be desirable that they influence the processor-set selection for jobs, and as discussed in Section 3.

Processor-Set Cost Definitions

As shown in Fig. 6, the definitions of cost-vectors associated with processor sets can depend on the size of the requesting job. This can be used for a variety of purposes, such as imposing stricter communication locality requirements for small jobs and selecting processor-sets for differently sized jobs starting from different physical regions of a machine, etc.

In Fig. 6, processor-set costs for 1–4 processor jobs are defined separately from those for jobs of all other sizes. Although this has been done mainly to make it clear that doing so is possible, it also provides an opportunity to discuss the costs choosen for the node processor sets. Even though the system is composed of logically identical nodes, they are obviously not physically in exactly the same location, and each node thereby has a slightly different physical environment. The different physical environments of the nodes can induce an ordering according to which the use of some nodes is preferable over the use of other nodes, in the absence of other constraints, and assuming the system is not currently operating under full load (i.e., that some nodes need not be used).

In the present case, with cooling by cold air provided from below at the front-side of the system, the use of lower placed nodes is preferable to the use of higher placed nodes, and because of flow hot air around the sides of the system from rear to front it is more desirable to use centrally positioned nodes than nodes nearer to the sides. The cost-vector definitions on lines 44–64 in Fig. 6 are intended to express precisely the cooling related preferences w.r.t. processor-set selection that have just been described.

Constraint Definitions

Finally, the fifth and final part of a configuration file, contains definitions of cost constraints that must be satisfied by processor-set selections. As shown on lines 103–107 in Fig. 6, each constraint is simply a boolean-valued expression, and when this expression does not evaluate to true for a proposed processor-set selection, the job scheduler is informed that the job in question should not be allowed to start.

Just as for cost-definitions, constraints can be differently defined for jobs of different sizes. In combination with job-size and queue differentiated processor-set costs this enables a rather precise control over when a potential processor-set selection is considered to be of sufficiently high quality to be used.

Minor Implementation Details

Due to the size and complexity of configuration files, they are not read as such by the processor-set selection machinery. Instead, configuration files are parsed and validated by a separate program, that for valid configuration files create corresponding (machine independent) binary configuration files, and these in turn are read by the processor-set selection machinery.

At regular time-intervals (every 5 min. currently), the processor-set selector checks the last modification time of its binary configuration file, and reloads it if it has changed. It is thereby easily arranged to make use of different processor-set selection strategies at different times of day or during weekends *vs.* workdays, etc. The separately performed configuration-file validation (and conversion into binary form), prevents simple mistakes (e.g., syntactic errors) made when preparing and/or modifying configuration files from influencing processor-set selection behaviour and decisions.

```
queues: cluster,standby

slots: [n01=1..4],[n02=1..4],[n03=1..4],[n04=1..4],[n05=1..4],
       [n06=1..4],[n07=1..4],[n08=1..4],[n09=1..4],[n10=1..4],
           ⋮          ⋮          ⋮          ⋮          ⋮
       [n36=1..4],[n37=1..4],[n38=1..4],[n39=1..4],[n40=1..4],
       [n41=1..4],[n42=1..4]

groups: h01={n01},h02={n02},h03={n03},h04={n04},h05={n05},
        h06={n06},h07={n07},h08={n08},h09={n09},h10={n10},
            ⋮         ⋮         ⋮         ⋮         ⋮
        h36={n36},h37={n37},h38={n38},h39={n39},h40={n40},
        h41={n41},h42={n42},ib1={n01,n02,n03,n04,n05,n06},
        ib2={n07,n08,n09,n10,n11,n12},
                  ⋮          ⋮
        ib7={n37,n38,n39,n40,n41,n42},
        en1={n01,n02,n03,n04,...,n11,n12,n13,n14},
        en2={n15,n16,n17,n18,...,n25,n26,n27,n28},
        en3={n29,n30,n31,n32,...,n39,n40,n41,n42},
        ps1a={n07,n06,n03,n02},ps1b={n05,n04,n01},
        ps2a={n14,n13,n10,n09},ps2b={n12,n11,n08},
        ps3a={n21,n20,n17,n16},ps3b={n19,n18,n15},
        ps4a={n28,n27,n24,n23},ps4b={n26,n25,n22},
        ps5a={n35,n34,n31,n30},ps5b={n33,n32,n29},
        ps6a={n42,n41,n38,n37},ps6b={n40,n39,n36},
        pwr1={n01,n02,n03,n04,n05,n06,n07,n09,n10,n13,n14},
```

Fig. 6. Configuration file for processor-set selection

```
        pwr2={n08,n11,n12,n15,n16,n17,n18,n19,n20,n21},
        pwr3={n22,n23,n24,n25,n26,n27,n28,n30,n31,n34,n35},
        pwr4={n29,n32,n33,n36,n37,n38,n39,n20,n41,n42}

costs:
    when PEs in [1,4]:
        cluster@h01=[0,0,0,0,129],cluster@h02=[0,0,0,0,135],
        cluster@h03=[0,0,0,0,141],cluster@h04=[0,0,0,0,147],
        cluster@h05=[0,0,0,0,153],cluster@h06=[0,0,0,0,159],
        cluster@h07=[0,0,0,0,165],cluster@h08=[0,0,0,0,115],
        cluster@h09=[0,0,0,0,121],cluster@h10=[0,0,0,0,127],
        cluster@h11=[0,0,0,0,133],cluster@h12=[0,0,0,0,139],
        cluster@h13=[0,0,0,0,145],cluster@h14=[0,0,0,0,151],
        cluster@h15=[0,0,0,0,101],cluster@h16=[0,0,0,0,107],
        cluster@h17=[0,0,0,0,113],cluster@h18=[0,0,0,0,119],
        cluster@h19=[0,0,0,0,125],cluster@h20=[0,0,0,0,131],
        cluster@h21=[0,0,0,0,137],cluster@h22=[0,0,0,0,102],
        cluster@h23=[0,0,0,0,108],cluster@h24=[0,0,0,0,114],
        cluster@h25=[0,0,0,0,120],cluster@h26=[0,0,0,0,126],
        cluster@h27=[0,0,0,0,132],cluster@h28=[0,0,0,0,138],
        cluster@h29=[0,0,0,0,116],cluster@h30=[0,0,0,0,122],
        cluster@h31=[0,0,0,0,128],cluster@h32=[0,0,0,0,134],
        cluster@h33=[0,0,0,0,140],cluster@h34=[0,0,0,0,146],
        cluster@h35=[0,0,0,0,136],cluster@h36=[0,0,0,0,130],
        cluster@h37=[0,0,0,0,136],cluster@h38=[0,0,0,0,142],
        cluster@h39=[0,0,0,0,148],cluster@h40=[0,0,0,0,154],
        cluster@h41=[0,0,0,0,160],cluster@h42=[0,0,0,0,166]
    otherwise:
        cluster@h01=[0,0,0,0,129],cluster@h02=[0,0,0,0,135],
                    ⋮                          ⋮
        cluster@h41=[0,0,0,0,160],cluster@h42=[0,0,0,0,166],
        cluster@ib1=[4,0,0,0,0],cluster@ib2=[4,0,0,0,0],
        cluster@ib3=[4,0,0,0,0],cluster@ib4=[4,0,0,0,0],
        cluster@ib5=[4,0,0,0,0],cluster@ib6=[4,0,0,0,0],
        cluster@ib7=[4,0,0,0,0},cluster@en1=[0,1,0,0,0],
        cluster@en2=[0,1,0,0,0],cluster@en3=[0,1,0,0,0],
        cluster@ps1a=[0,0,2,0,0],cluster@ps1b=[0,0,2,0,0],
        cluster@ps2a=[0,0,2,0,0],cluster@ps2b=[0,0,2,0,0],
        cluster@ps3a=[0,0,2,0,0],cluster@ps3b=[0,0,2,0,0],
        cluster@ps4a=[0,0,2,0,0],cluster@ps4b=[0,0,2,0,0],
        cluster@ps5a=[0,0,2,0,0],cluster@ps5b=[0,0,2,0,0],
        cluster@ps6a=[0,0,2,0,0],cluster@ps6b=[0,0,2,0,0],
        cluster@pwr1=[0,0,0,4,0],cluster@pwr2=[0,0,0,4,0],
        cluster@pwr3=[0,0,0,4,0],cluster@pwr4=[0,0,0,4,0]
    end-costs
```

Fig. 6. (*Continued*)

```

constraints:
   when PEs in [1,4] : cost <= [0,0,0,0,166],
   when PEs in [5,16] : cost <= [8,1,6,8,628],
   when PEs in [17,32] : cost <= MINCOST(32) + [0,0,0,0,64],
   when PEs in [33,64] : cost <= MINCOST(64) + [4,1,2,0,38],
   otherwise : cost <= ceil(1.2*MINCOST(PEs)) + [4,0,2,0,28]
```

Fig. 6. (*Continued*)

On Workflow Scheduling for End-to-End Performance Optimization in Distributed Network Environments

Qishi Wu[1], Daqing Yun[1], Xiangyu Lin[1], Yi Gu[2], Wuyin Lin[3], and Yangang Liu[3]

[1] Department of Computer Science
University of Memphis
Memphis, TN 38152
{qishiwu,dyun,xlin}@memphis.edu

[2] Department of Management, Marketing, Computer Science, and Information Systems
University of Tennessee at Martin
Martin, TN 38238
ygu6@utm.edu

[3] Atmospheric Sciences Division
Brookhaven National Laboratory
Upton, NY 11793
{wlin,lyg}@bnl.gov

Abstract. Next-generation computational sciences feature large-scale workflows of many computing modules that must be deployed and executed in distributed network environments. With limited computing resources, it is often unavoidable to map multiple workflow modules to the same computer node with possible concurrent module execution, whose scheduling may significantly affect the workflow's end-to-end performance in the network. We formulate this on-node workflow scheduling problem as an optimization problem and prove it to be NP-complete. We then conduct a deep investigation into workflow execution dynamics and propose a Critical Path-based Priority Scheduling (CPPS) algorithm to achieve Minimum End-to-end Delay (MED) under a given workflow mapping scheme. The performance superiority of the proposed CPPS algorithm is illustrated by extensive simulation results in comparison with a traditional fair-share (FS) scheduling policy and is further verified by proof-of-concept experiments based on a real-life scientific workflow for climate modeling deployed and executed in a testbed network.

Keywords: Scientific workflows, task scheduling, end-to-end delay.

1 Introduction

Next-generation computational sciences typically involve complex modeling, large-scale simulations, and sophisticated experiments in studying physical phenomena, chemical reactions, biological processes, and climatic changes [13, 25]. The processing and analysis of simulation or experimental datasets generated in these scientific applications require the construction and execution of domain-specific workflows consisting of many interdependent computing modules[1] in distributed network environments such as Grids

[1] Workflow modules are also referred to as tasks/subtasks, activities, stages, jobs, or transformations in different contexts.

or clouds for collaborative research and discovery. The network performance of such scientific workflows plays a critical role in the success of targeted science missions.

The computing workflows of scientific applications are often modeled as Directed Acyclic Graphs (DAGs) where vertices represent computing modules and edges represent execution precedence and data flow between adjacent modules, and could be managed and executed by either special- or general-purpose workflow engines such as Condor/DAGMan [1], Kepler [23], Pegasus [12], Triana [11], and Sedna [31]. In practice, workflow systems employ a static or dynamic mapping scheme to select an appropriate set of computer nodes in the network to run workflow modules to meet a certain performance requirement. No matter which type of mapping scheme is applied, it is often unavoidable to map multiple modules to the same node (i.e. node reuse) for better utilization of limited computing resources, leading to possible concurrent module execution. For example, in unitary processing applications that perform one-time data processing, workflow modules may run concurrently if they are independent of each other[2]; while in streaming applications with serial input data, even modules with dependency may run concurrently to process different instances of the data. Of course, concurrent modules on the same node may not always share resources if their execution times do not overlap completely due to their misaligned start or end times. The same is also true for concurrent data transfers.

Generally, in the case of concurrent module execution, the node's computing resources are allocated by kernel-level CPU scheduling policies such as the round-robin algorithm to ensure fine-grained fair share (FS). Similarly, a network link's bandwidth is also fairly shared by multiple independent data transfers that take place concurrently over the same link through the use of the widely deployed TCP or TCP-friendly transport methods. Such system-inherent fair-share scheduling mechanisms could reduce the development efforts of workflow systems, but may not always yield the best workflow performance, especially in distributed environments. We shall use an extremely simplified numerical example to illustrate the effect of on-node workflow scheduling.

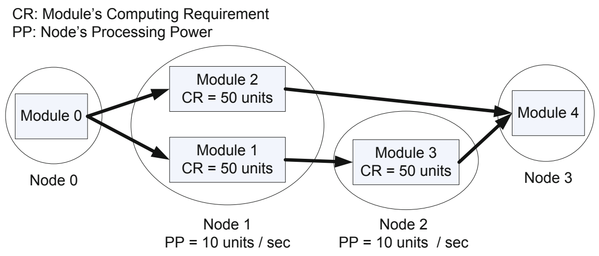


Fig. 1. A simple numerical example illustrating the effect of scheduling

As shown in Fig. 1, a workflow consisting of five modules is mapped to a network consisting of four nodes. For simplicity, we only consider the execution time of Modules 1, 2, and 3. With a fair-share scheduler, the workflow takes 15 seconds to complete along the critical path (i.e. the longest path of execution time) consisting of Modules 0, 1, 3, and 4. However, if we let Module 1 execute exclusively ahead of Module 2, the completion time of the entire workflow is cut down to 10 seconds. This example reveals

[2] Two modules in a workflow are independent of each other if there does not exist any dependency path between them.

that the workflow performance could be significantly improved if concurrent modules on the same node are carefully scheduled.

The work in this paper is focused on task scheduling for *Minimum End-to-end Delay* (MED) in unitary processing applications under a given workflow mapping scheme. We formulate this on-node workflow scheduling problem as an optimization problem and prove it to be NP-complete. Based on the exact *End-to-end Delay* (ED) calculation algorithm, *extED* [17], we conduct a deep investigation into workflow execution dynamics and propose a *Critical Path-based Priority Scheduling* (CPPS) algorithm that allocates the node's computing resources to multiple concurrent modules assigned by the given mapping scheme to the same node to minimize the ED of a distributed workflow. We also provide an analytical upper bound of the ED performance improvement when the critical path remains fixed during workflow scheduling. The proposed CPPS algorithm is implemented and tested in both simulated and experimental network environments. The performance superiority of CPPS is illustrated by extensive simulation results in comparison with a traditional fair-share scheduling policy, and is further verified by proof-of-concept experiments based on a real-life scientific workflow for climate modeling deployed and executed in a testbed network.

The rest of the paper is organized as follows. In Section 2, we conduct a survey of related work on both workflow mapping and scheduling. In Section 3, we provide a formal definition of the workflow scheduling problem under study. In Section 4, we design the CPPS algorithm. In Section 5, we implement the proposed algorithm and present both simulation and experimental results. We conclude our work in Section 6.

2 Related Work

In this section, we provide a brief survey of related work on workflow optimization. There are two aspects of optimizing distributed tasks to improve the performance of scientific workflows: i) assigning the component modules in a workflow to suitable resources, referred to as workflow mapping[3], and ii) deciding the execution order and resource sharing policy among concurrent modules on computer nodes or processors, referred to as workflow/task scheduling. Both problems have been extensively studied in various contexts due to their theoretical significance and practical importance [7,28,30].

When network resources were still scarce in early years, workflow modules were often mapped to homogeneous systems such as multiprocessors [6,22]. As distributed computing platforms such as Grids and clouds are rapidly developed and deployed, research efforts have shifted to workflow mapping in heterogeneous environments to utilize distributed resources at different geographical locations across wide-area networks [8,9,28]. However, several of these studies assume that computer networks are fully connected [30] or only consider independent tasks in the workflow [10]. These assumptions are reasonable under certain circumstances, but may not sufficiently model the complexity of real-life applications in wide-area networks. In real workflow systems, a greedy or similar type of approach is often employed for workflow mapping.

[3] The workflow mapping problem is occasionally referred to as a scheduling problem in the literature. In this paper, we designate it specifically as a mapping problem to differentiate it from the workflow/task scheduling problem under study.

For instance, Condor implements a matchmaking algorithm that utilizes classified advertisements (ClassAds) mechanism for mapping user's applications to suitable server machines [27]. Most existing mapping algorithms are centralized, but decentralized methods are desired for higher scalability. More recently, Rahman *et al.* proposed an approach to workflow mapping in a dynamic and distributed Grid environment using a Distributed Hash Table (DHT) based d-dimensional logical index space [26].

Considering the limit in the scope and amount of networked resources, a reasonable workflow mapping algorithm inevitably results in a mapping scheme where multiple modules are mapped to the same node, which necessitates the scheduling of concurrent modules. On-node processor scheduling has been a traditional research subject for several decades, and many algorithms have been proposed ranging from proportional, priority-based, to fair share [18, 19], to solve scheduling problems under different constraints with different objectives. Most traditional processor scheduling algorithms consider multiple independent processes running on a single CPU with a goal to optimize the waiting time, response time, and turnaround time of individual processes, or the throughput of the system. The multi-processor and multi-core scheduling on modern machines has attracted an increasing amount of attention in recent years [21, 24].

Our work is focused on a particular scheduling problem to achieve MED of a DAG-structured workflow in distributed environments under a given mapping scheme. A similar DAG-structured project planning problem dated back to late 1950's and was tackled using Program/Project Evaluation Review Technique and Critical Path Method (PERT/CPM) [14, 20]. PERM is a method to analyze the involved tasks in completing a given project and identify the minimum time needed to complete the total project. CPM calculates the longest path of planned activities to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. The workflow scheduling problem differs from the project planning problem in that the computing resources are shared among concurrent modules and the execution time of each individual module is unknown until a particular task scheduling scheme is determined. This type of scheduling problems are generally NP-complete. Garey *et al.* compiled a great collection of similar or related multi-processor scheduling problems in [15], which were tackled by various heuristics such as list scheduling and simple level algorithm [16].

In practical system implementations, concurrent tasks are always assigned the same running priority since the fair share of CPU cycles is well supported by modern operating systems that employ a round-robin type of scheduling algorithms. However, in this paper, we go beyond the system-inherent fair-share scheduling to further improve the end-to-end performance of scientific workflows through the use of an on-node scheduling strategy taking into account the global workflow structure and resource sharing dynamics in distributed environments.

3 Workflow Scheduling Problem

In this section, we construct analytical cost models and formulate the workflow scheduling problem, which is proved to be NP-complete.

3.1 Cost Models

We model a workflow as a Directed Acyclic Graph (DAG) $G_w = (V_w, E_w)$, $|V_w| = m$, where vertices represent computing modules starting from the start module w_0 to the end module w_{m-1}. A directed edge $e_{i,j} \in E_w$ represents the dependency between a pair of adjacent modules w_i and w_j. Module w receives a data input of size z from each of its preceding modules and performs a predefined computing routine, whose computing requirement (CR) or workload is modeled as a function of the aggregate input data sizes. Module w sends a data output to each of its succeeding modules after it completes its execution. In this workflow model, we consider a module as the minimal execution unit, and ignore the inter-module communication cost on the same node. For a workflow with multiple start/end modules, we can always convert it to this model by adding a virtual start/end module with $CR = 0$ and connected to all the original start/end modules with data transfer size $z = 0$.

We model an overlay computer network as an arbitrary weighted graph $G_c = (V_c, E_c)$, consisting of $|V_c| = n$ nodes interconnected by $|E_c|$ overlay links. A normalized variable PP_i is used to represent the overall processing power of node v_i without specifying its detailed system resources. Link $l_{i,j}$ from v_i to v_j is associated with a certain bandwidth $b_{i,j}$. It is assumed that the start module w_0 serves as a data source on the source node v_s without any computation to supply all initial data needed by the application and the end module w_{m-1} performs a terminal task on the destination node v_d without any further data transfer.

Based on the above models, we can use an existing mapping algorithm to compute a mapping scheme under the following constraints on workflow mapping and execution/transfer precedence [17]:

- Each module/edge is required to be mapped to only one node/link.
- A computing module cannot start execution until all its required input data arrive.
- A dependency edge cannot start data transfer until its preceding module finished execution.

A mapping scheme $M : G_w \rightarrow G_c$ is mathematically defined as follows:

$$M : G_w \rightarrow G_c = \begin{cases} w \rightarrow c, \forall w \in V_w, \exists c \in V_c; \\ l(c_{i'}, c_{j'}) \in E_c, \begin{cases} \text{if } w_i \rightarrow c_{i'}, w_j \rightarrow c_{j'}, e(w_i, w_j) \in E_w, \\ 0 \leq i, j \leq |V_w| - 1, 0 \leq i', j' \leq |V_c| - 1. \end{cases} \end{cases} \quad (1)$$

Once a mapping scheme is obtained, we can further convert the above workflow and network models to virtual graphs as follows: each dependency edge in the workflow is replaced with a virtual module whose computational workload is equal to the corresponding data size, and each mapped network link is replaced by a virtual node whose processing power is equal to the corresponding bandwidth. This conversion facilitates workflow scheduling as we only need to focus on the module execution time. We use t_w^s and t_w^f to denote the execution start and finish time of module w. For convenience, we tabulate the notations used in the cost models in Table 1, some of which will be used in the algorithm design.

Table 1. Parameters in the cost models and algorithm design

Parameters	Definitions
$G_w = (V_w, E_w)$	a computing workflow
m	the number of modules in the workflow
w_i	the i-th computing module
w_0	the start computing module
w_{m-1}	the end computing module
$e_{i,j}$	the dependency edge from module w_i to w_j
z	the data size of dependency edge e
CR_i	the computing requirement of module w_i
$G_c = (V_c, E_c)$	a computer network
n	the number of nodes in the network
v_i	the i-th network of computer node
v_s	the source node
v_d	the destination node
PP_i	the processing power of node v_i
$l_{i,j}$	the network link from node v_i to v_j
$b_{i,j}$	the bandwidth of link $l_{i,j}$

3.2 Problem Definition

Since the start time of a module in the workflow depends on the end time of its preceding module(s), even independent modules assigned to the same node may not always run in parallel. Therefore, task scheduling is only applicable to concurrent modules that run simultaneously at least for a certain period of time. We formally define the On-Node Workflow Scheduling (ONWS) problem as follows:

Definition 1. (***ONWS***) *Given a DAG-structured computing workflow $G_w = (V_w, E_w)$, a heterogeneous overlay computer network $G_c = (V_c, E_c)$ and a mapping scheme $M : G_w \rightarrow G_c$, we wish to find a job scheduling policy on every computer node with concurrent modules such that the mapped workflow achieves:*

$$\text{MED} = \min_{\text{all possible job scheduling policies}} (T_{\text{ED}}) . \tag{2}$$

3.3 NP-Completeness Proof

We first transform the original ONWS problem to its decision version, referred to as ONWS-Decision, with a set of notations commonly adopted in the definitions of traditional multi-processor scheduling problems in the literature.

Definition 2. (***ONWS-Decision***) *Given a set T of tasks t, each having a length(t) and executed by a specific processor p(t), a number $m \in Z^+$ of processors, partial order $\prec$ on T, and an overall deadline $D \in Z^+$, is there an m-processor preemptive schedule for T that obeys the precedence constraints and meets the overall deadline?*

Note that in Definition 2, the workflow in the original ONWS problem is expressed as a set of tasks with a partial-order relation and the given mapping scheme is expressed as a set of node assignments or requirements for all tasks. Also, we consider preemptive scheduling in our problem. The difficulty of ONWS-Decision is given by Theorem 1, which is proved by showing that the *m*-processor bound unit execution time (MBUET) system scheduling problem in [16], an existing NP-complete problem as defined in Definition 3, is a special case of ONWS-Decision.

Definition 3. (***MBUET***) *Given a set T of tasks, each having length(t) = 1 (unit execution time) and executed by a specific processor p(t), an arbitrary number* $m \in Z^+$ *of processors, partial order* $\prec$ *of a forest on T, and an overall deadline* $D \in Z^+$*, is there an m-processor schedule for T that obeys the precedence constraints and meets the overall deadline?*

Theorem 1. *The ONWS-Decision problem is NP-complete.*

Proof. Obviously, the MBUET scheduling problem is a special type of ONWS-Decision problem with $length(t)$ restricted to be 1 for all tasks $t \in T$ and partial order $\prec$ restricted to be a forest. For preemptive scheduling as considered in ONWS-Decision, a task is not required to finish completely without any interruption once it starts execution. However, if we restrict the length of all tasks to be 1 (the smallest unit of time) in the input of ONWS-Decision, each task would finish in its entirety once it is assigned to the designated processor for execution. Moreover, the partial order $\prec$ of a forest is a special DAG structure of workflow topology. Therefore, the MBUET problem polynomially transforms to the preemptive ONWS-Decision scheduling problem. Since MBUET is NP-complete [16], the general ONWS-Decision problem is also NP-complete. The validity of the NP-completeness proof by restriction is established in [15], where "restriction" constrains the given, not the question of a problem.

4 Algorithm Design

4.1 Analysis of Resource Sharing Dynamics in Workflows

Since the end-to-end delay (ED) of a workflow is determined by its critical path, i.e. the path of the longest execution time, a general strategy for workflow scheduling is to reduce the execution time of critical modules (i.e. modules on the critical path) by allocating to them more resources than those non-critical modules that are running concurrently. Ideally, the MED would be achieved if all possible execution paths from the start module to the end module have the same length of execution time.

We first present a theorem on the resource sharing dynamics among the concurrent modules assigned to the same node, which will be used in the design of our scheduling algorithm.

Theorem 2. *The finish time of the last completed module among k concurrent modules executed on the same node with a single fully-utilized processor of processing power PP is a constant,* $\frac{\sum_{i=1}^{k} CR_i}{PP}$.

Proof. Since the total workload of all k modules is fixed, i.e. $\sum_{i=1}^{k} CR_i$, and the processor is fully operating, no matter how the modules are scheduled, the total execution time remains unchanged and is bounded by the finish time of the last completed module.

To understand the significance of workflow scheduling, we investigate a particular scheduling scenario shown in Fig. 2 where the best performance improvement could be achieved over a fair-share scheduling policy with the following conditions:

– There are k modules running concurrently on the same node v_1 during their entire execution time period;
– Among k modules, one is a critical module and $k-1$ are non-critical modules, and all of them are of the same computing requirement (CR);
– Each non-critical module is the only module on its execution path (except for the start and end modules).

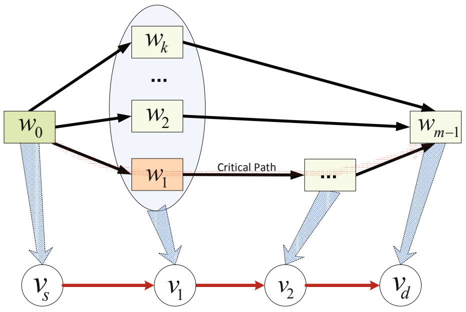


Fig. 2. A scheduling scenario that achieves the upper bound performance

Based on the scheduling scenario depicted in Fig. 2, we have the following theorem:

Theorem 3. *The workflow's MED performance improvement of any on-node task scheduling policy with a fixed critical path over fair share is upper bounded by 50%.*

Proof. We first justify that the conditions in Fig. 2 are necessary for achieving the upper bound of the workflow's MED improvement.

Let x be the original execution time of the critical module on v_1 using the fair-share scheduling policy. If there were any non-critical modules on v_1 that finished before x, even if we let the critical module w_1 run exclusively first, it would take longer than $\frac{x}{k}$ to complete. Thus, we would not be able to reduce the execution time of the critical module to the minimum time possible, i.e. $\frac{x}{k}$. On the other hand, if there were any non-critical modules on v_1 that finished after x, from Theorem 2, we know that the total execution time on this node would be greater than x, and hence we would not be able to reduce the length of the critical path to x.

The above discussion concludes that all k modules on node v_1 must share resources during their entire execution time and finish at the same time x to achieve the maximum possible reduction on the length of the critical path. Since we consider fair share as the comparison base in Theorem 2, the CRs of all k modules must also be identical. In this scheduling scenario, if we let the critical module w_1 execute exclusively first, its finish time $t^f_{w_1}$ would be reduced from x to $\frac{x}{k}$, which is the best improvement possible with k concurrent modules on the same node.

We use y to denote the sum of the execution time for the rest of the modules on the critical path (excluding w_1). From Theorem 2, we have $max(t^f_{w_i}) = x$, $i = 2,3,\ldots,k$, which means that the latest finish time of the last completed non-critical module would be still x. Since the new length of the critical path (i.e. MED) becomes $\frac{x}{k}+y$, the upper bound MED improvement is achieved if this new length is equal to the latest finish time of any non-critical module, i.e. $\frac{x}{k}+y=x$. It follows that $y=\frac{k-1}{k}x$. Therefore, the MED improvement over fair share is $\Delta = \frac{(x+y)-x}{x+y} = \frac{k-1}{2k-1}$, which is 1/2 or 50% as $k \to \infty$.

4.2 Critical Path-Based Priority Scheduling (CPPS) Algorithm

We propose a *Critical Path-based Priority Scheduling* (CPPS) algorithm to solve the ONWS problem. The CPPS algorithm produces a set of processor scheduling schemes for all mapping nodes that collectively cut down the execution length of the critical path in the entire workflow to achieve MED.

The pseudocode of the proposed CPPS algorithm is provided in Alg. 1, which first calculates the critical path based on the fair-share (FS) scheduling, and then uses Algs. 2 and 3 to compute the new task schedule of concurrent modules on each mapping node. In most cases, the new schedule outperforms the FS schedule. However, if the new schedule does not improve the MED performance, the CPPS algorithm simply rolls back to the FS schedule.

In Alg. 2, we calculate the independent set of each module in line 8 and find the set set_0 of modules with the earliest start time in line 12. If there is only one module in set_0, this module is executed immediately; otherwise, a scheduling strategy needs to be decided. In line 18, we recalculate the percent of processing power that is allocated to those modules in set_0. In lines 18-32, a new scheduling policy is generated and applied to all concurrent modules.

Alg. 3 performs the actual on-node scheduling, where only two cases need to be considered: 1) when $per(w_{cp})$ is 100% and other $per(w_{non\_cp})$ is 0%, 2) when all $per(w)$

Algorithm 1. CPPS($G_w{}'$, $G_c{}'$, f)
Input: A converted workflow graph $G_w{}' = (V_w{}', E_w{}')$, a converted network graph $G_c{}' = (V_c{}', E_c{}')$, a given mapping scheme $f : G_w{}' \to G_c{}'$.
Output: the scheduling policies on all the mapping nodes.

```
1: Calculate the execution time $T_{FS}$ for all the modules in $G_w{}'$ using a fair-share scheme;
2: Calculate the critical path $CP_{FS}$ and its length $T(CP_{FS})$ based on $T_{FS}$;
3: Calculate the execution time $T_{CPPS}$ for all the modules in $G_w{}'$ using Alg. 2, which in turn
   uses Alg. 3 to decide the priorities for all the modules;
4: Calculate the new $CP_{CPPS}$ and its length $T(CP_{CPPS})$ based on $T_{CPPS}$;
5: if $T(CP_{CPPS}) \geq T(CP_{FS})$ then
6:     return the fair-share schedule and $T(CP_{FS})$.
7: else
8:     return the new CPPS schedule and the ED $T(CP_{CPPS})$ of the mapped workflow.
```

Algorithm 2. extEDOnNode($G_w{}'$, $G_c{}'$, f, T_{FS}, CP_{FS})

Input: A converted workflow graph $G_w{}' = (V_w{}', E_w{}')$, a converted network graph $G_c{}' = (V_c{}', E_c{}')$, a mapping scheme $f : G_w{}' \to G_c{}'$, the execution time T_{FS} of each module under the fair-share scheme, and the critical path CP_{FS} based on T_{FS}.

Output: the new execution time T_{CPPS} of all modules and the ED $T(CP_{CPPS})$ of the mapped workflow.

```
1: $t^s(set)$: the set of start times of all the modules in a $set$;
2: $t^f(set)$: the set of finish times of all the modules in a $set$;
3: $ids(w)$: the independent set of module $w$ on the same node (excluding $w$);
4: $est(set) = \{w | w \in set$ and $t^s_w = \min(t^s(set))\}$, i.e. the set of modules with the earliest start
   time in a $set$;
5: $ready(set) = \{w | w \in set$ and $w$ is "$ready$" to execute$\}$;
6: $TBF(w)$: the partial workload of module $w$ to be finished;
7: **for all** module $w_i \in V_w{}'$ **do**
8:    Find $ids(w_i)$;
9:    Set $w_i$ as "$unfinished$";
10: Set $w_0$ as "$ready$";
11: **while** exist "$unfinished$" modules $\in V_w{}'$ **do**
12:    Find $set_0 = est(ready(V_w{}'))$;
13:    **for all** module $w_i \in set_0$ **do**
14:       **if** $|ids(w_i)| == 0$ **then**
15:          Execute $w_i$, calculate $T_{CPPS}(w_i)$ and set $w_i$ as "$finished$";
16:       **else**
17:          Find $set_1 = \{w | w$ is "$ready$" and $w \in est(ids(w_i) \cup w_i)\}$;
18:          OnNodeSchedule($G_w{}'$, $G_c{}'$, $f$, $T_{FS}$, $T_{CPPS}$, $w_i \cup ids(w_i)$, $CP_{FS}$);
19:          Estimate $t^f(set_1)$;
20:          Find $set_2 = \{w \mid w \in ids(w_i)$ & $w \notin set_1$, $t^s(w) < \min(t^f(set_1))$, and $w$ is "$ready$"
             and "$unfinished$"$\}$;
21:          **if** $|set_2| > 0$ **then**
22:             **for all** module $w_j \in set_1$ **do**
23:                From $t^s(w|w \in set_1)$ to $\min(t^s(set_2))$: 1) finish part of $TBF(w_j)$ with the new
                   scheduling policy determined in line 18; 2) calculate the partial amount of
                   $T_{CPPS}$;
24:                Update the new $t^s_{w_j} = \min(t^s(set_2))$;
25:          **else**
26:             **for all** module $w_j \in set_1$ **do**
27:                **if** $t^f_{w_j} == \min(t^f(set_1))$ **then**
28:                   Execute $w_j$, calculate $T_{CPPS}(w_j)$, and set $w_j$ as "$finished$";
29:                **else**
30:                   From $t^s(w|w \in set_1)$ to $\min(t^f(set_1))$: 1) finish part of $TBF(w_j)$ with the
                      new scheduling policy determined in line 18; 2) calculate the partial amount
                      of $T_{CPPS}$;
31:                   Update the new $t^s_{w_j} = \min(t^f(set_1))$;
32:    Mark all ready modules as "$ready$";
33: Compute the CP based on the time components $T_{CPPS}(w_i)$ for all $w_i \in V_w{}'$;
34: **return** the ED $T(CP_{CPPS})$ of the mapped workflow.
```

Algorithm 3. **OnNodeSchedule**($G_w{'}$, $G_c{'}$, f, T_{FS}, $w_i \cup ids(w_i)$, CP_{FS})

Input: A converted workflow graph $G_w{'} = (V_w{'}, E_w{'})$, a converted network graph $G_c{'} = (V_c{'}, E_c{'})$, a given mapping scheme $f : G_w{'} \to G_c{'}$, and a module set $w_i \cup ids(w_i)$ that combines w_i and its independent set $ids(w_i)$.

Output: the percentage $per(w)$ of resource allocation for all modules in $w_i \cup ids(w_i)$.

```
1: $w_{cp}$: module of the critical path $cp$ (i.e. critical module);
2: $w_{non\_cp}$: module that is not on the critical path (i.e. non-critical module);
3: $per(w)$: percentage of processing power allocated to module $w$;
4: $t_{OnNode}$: the estimated execution time of a module under the new scheduling strategy;
5: $CP(w)$: the critical path (CP) consisting of module $w$;
6: $CP_L(w)$: the left CP segment from the start module to module $w$ (i.e. the CP of the left-side
   partial workflow ending at $w$);
7: $CP_R(w)$: the right CP segement from module $w$ to the end module (i.e. the CP of the right-side
   partial workflow starting at $w$);
8: $LFT(n_i)$: the latest possible finish time of concurrent modules assigned to node $n_i$;
9: $T_{exclusive}(w_i)$: execution time of $w_i$ when running exclusively on its mapping node $map(w_i)$;
10: $TBF(w)$: the partial workload of module $w$ to be finished;
11: $map(w)$: the node to which module $w$ is mapped;
12: for all modules $w_i$ in $w_i \cup ids(w_i)$ do
13:     Calculate $CP(w_i) = CP_L(w_i) + CP_R(w_i)$ based on $T_{FS}$ and $T_{CPPS}$;
14: Find $w_{cp} = \{w | CP(w) = max\{CP(w) | w \in ids(w_i) \cup w_i\}\}$;
15: Estimate execution time $T_{exclusive}(w_{cp})$;
16: Calculate the latest possible finish time $LFT(map(w_i))$ of modules mapped to $map(w_i)$;
17: bool $flag$ = false;
18: for all modules $w_i \in w_i \cup ids(w_i)$ do
19:     Estimate the amount of time $T_{tbf}$ needed to finish $TBF(w_i)$;
20:     if $w_i$ is $w_{non\_cp}$ then
21:         $\Delta t_1 = min\{LFT(map(w_i)) - T_{tbf}, T_{exclusive}(w_{cp})\}$;
22:         $\beta = |\{w_k \in CP(w_i)$ and $w_k \in \cup\{CP(w_j) | w_j \in ids(w_i)\}\}|$;
23:         if $T(CP(w_i)) + \beta \Delta t_1 \geq T(CP_{FS})$ then
24:             $flag = $ true;
25:             $\Delta t_2 = T(CP_{FS}) - T(CP(w_i))$;
26:             $t_{OnNode} = T_{tbf} + \Delta t_2$;
27:             Calculate $per(w_i)$ according to $t_{OnNode}$;
28:             mark $w_i$;
29: if $flag$ == true then
30:     for all modules $w_i \in w_i \cup ids(w_i)$ do
31:         if $w_i$ is not marked in line 28 then
32:             $per(w_i) = \left(1 - \sum_{w_i \in w_i \cup ids(w_i)}^{w_i\ marked} per(w_i)\right) \Big/ \left(\left|\{w_i \cup ids(w_i)\}\right| - \left|\{w_i | w_i\ marked\}\right|\right)$;
33: else
34:     for all modules $w_i \in w_i \cup ids(w_i)$ do
35:         if $w_i$ is $w_{cp}$ then
36:             $per(w_i) = 1.0$;
37:         else
38:             $per(w_i) = 0.0$;
39:     return the ID of $w_i$.
40: return $per(w_i)$ for all modules $w_i \in w_i \cup ids(w_i)$.
```

are between 0 and 100%. Whenever possible, we wish to run the critical module exclusively first to cut down the length of the globe CP as much as possible. Accordingly, Alg. 2 considers the following two possible scenarios for all modules $w_i \in w_i \cup ids(w_i)$ depending on the output of Alg. 3:

1) When $per(w_{cp}) = 100\%$ and $per(w_{non\_cp}) = 0$: since Alg. 3 returns the ID of w_{cp}, we execute the critical module w_{cp} from the time point $t^s(w|w \in set_1)$ to $\min(t^s(set_2))$ exclusively; while for other modules, we suspend them to wait for w_{cp} to finish. During this period of time, w_{cp} may be entirely or partially completed. We then execute all unfinished modules (their unfinished part $TBF(w_i)$) that are mapped to node $map(w_{cp})$ in a fair-share manner until the number of concurrent modules assigned to this node changes again.
2) When all $per(w)$ are between 0 and 100%: we execute each module with its own percent $per(w_i)$ of resource allocation until the number of concurrent modules changes.

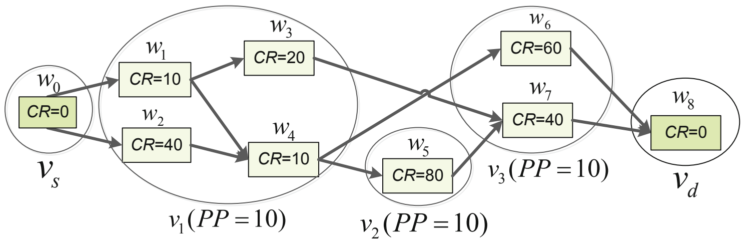


Fig. 3. A simple numerical example used for illustrating the scheduling algorithms

We shall use a simple numerical example to illustrate the CPPS scheduling process. As shown in Fig. 3, a computing workflow consisting of nine modules $w_0, w_1, \ldots,$ and w_8 is mapped to a computer network consisting of 5 nodes v_s, v_1, v_2, v_3 and v_d. The modules' computing requirements (CR) and the nodes' processing powers (PP) are marked accordingly except for the start/end modules mapped to the source/destination nodes, whose execution time is not considered (i.e. $CR = 0$).

As shown in Fig. 4, we first compute the execution start and end time $T_{FS} : t^s_w | t^f_w$ of each module under the fair-share (FS) scheduling policy. In this scheduling scenario, the workflow takes 20 seconds to complete along the critical path (CP) consisting of modules $w_0, w_2, w_4, w_5, w_7,$ and w_8.

Now we describe the CPPS scheduling process. Fig. 5 illustrates the scheduling status when modules w_3 and w_4 are being scheduled in the CPPS algorithm. At this point of time, the left shadowed part has been scheduled by CPPS while the right shadowed part is still estimated based on FS. Note that the module start time t^s_w is always counted from the point when the module is ready to execute, not from the point when it actually starts execution. Right after w_1 finishes execution, both w_3 and w_4 are ready to execute with possible resource sharing. To decide the scheduling between them, we need to compute the longest (critical) path that traverses each of them by concatenating the left and right segments of its critical path. For w_3, the length of its left CP segment

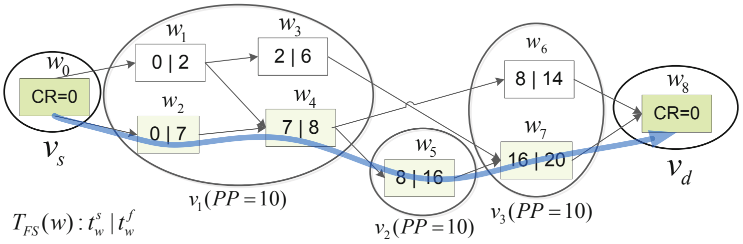


Fig. 4. The execution start time t_w^s and finish time t_w^f of each module calculated under the FS scheduling policy, listed in the form of $T_{FS}(w) : t_w^s|t_w^f$

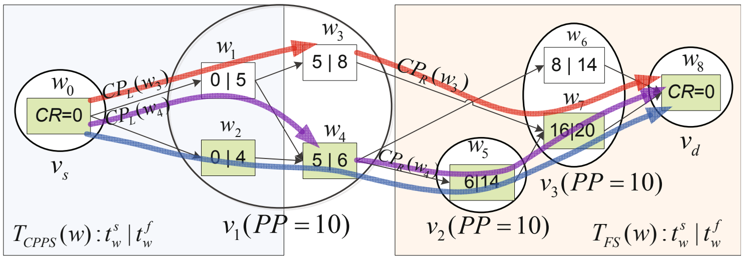


Fig. 5. The scheduling status when modules w_3 and w_4 are being scheduled in the CPPS algorithm. The left shadowed part has been scheduled by CPPS while the right shadowed part is estimated based on FS.

(w_0, w_1, and w_3), denoted as $CP_L(w_3)$, is 5 under CPPS, while the length of its right CP segment (w_3, w_7, and w_8), denoted as $CP_R(w_3)$, is 8 (including its own execution time), which is estimated using the FS-based measurements in Fig. 4. Similarly, for w_4, the lengths of its left and right CP segments are 5 under CPPS and 13 based on FS as shown in Fig. 4, respectively. Since $CP(w_4) = CP_L(w_4) + CP_R(w_4) = 18$ is longer than $CP(w_3) = CP_L(w_3) + CP_R(w_3) = 13$, w_4 is to be executed exclusively first. It is estimated that the length of $CP(w_3)$ would not exceed the length of the original global CP_{FS} based on FS, which is 20, even if we let w_4 run to its completion with exclusive CPU utilization, so in this case, we assign the entire CPU to w_4 until it finishes; otherwise, w_4 would execute exclusively until a certain point and then share with w_3 in a fair manner such that the length of $CP(w_3)$ does not exceed the length of the original global CP_{FS} based on FS. The new execution time of w_3 and w_4 is updated in Fig. 5. This scheduling process is repeated on every mapping node until all the modules are properly scheduled.

5 Performance Evaluation

We evaluate the performance of the proposed CPPS algorithm in both simulated and experimental network environments in comparison with the fair-share scheduling policy, which is commonly adopted in real systems.

5.1 Simulation-Based Performance Comparison

Simulation Settings. In the simulation, we implement the *CPPS* algorithm in C++ and run it on a Windows 7 desktop PC equipped with Intel(R) Core(TM)2 Duo CPU E7500 of 2.93GHz and 4.00GB memory.

We develop a separate program to generate test cases by randomly varying the problem size denoted as a three-tuple $(m, |E_w|, n)$, i.e. m modules and $|E_w|$ edges in the workflow, and n mapping nodes in the network. For a given problem size, we randomly vary the module complexity and data size within a suitably selected range of values, and create the DAG topology of a workflow as follows: 1) Lay out all the modules sequentially; 2) For each module, create an input edge from a randomly chosen preceding module and create an output edge to a randomly chosen succeeding module (note that the start module has no input and the end module has no output); 3) Repeatedly pick up a pair of modules on a random basis and add a directed edge from left to right between them until we reach the given number of edges.

Given the workflow structure and the number of mapping nodes with randomly generated processing power, we randomly generate the mapping scheme but with some specific topological constraints. In observation of the topological structures of real networks such as ESnet [2] and Internet2 [3], we first topologically sort all modules and then map the modules from each layer of the workflow to the nodes that are within the proximity of the corresponding layer in the network. In the same layer with multiple modules/nodes, a greedy approach is adopted for node assignment.

Note that in the network, we do not need to specify the number of links as a parameter because there must exist a link between two mapping nodes where two adjacent modules are mapped to ensure the feasibility of the given mapping scheme. A random bandwidth is then selected from a suitable range of values and assigned to each link. Since other links without any dependency edges mapped to will not affect the simulation results, they are simply ignored.

Simulation Results. To evaluate the performance of the proposed CPPS algorithm, we randomly generate 4 groups of test cases with 4 different numbers of nodes, i.e. $n = 5, 10, 15,$ and 20. For each number of nodes (i.e. each group), we generate 20 problem sizes, indexed from 1 to 20, by varying the number of modules from 5 to 100 at an interval of 5 with a random number of edges.

For each problem size $(m, |E_w|, n)$, we generate 10 random problem instances for workflow scheduling with different module complexities, data sizes, node processing powers, link bandwidths, and mapping schemes, and then calculate the average of the performance measurements under both fair-share (FS) and CPPS scheduling. The MED performance improvement or speedup of CPPS over FS, defined as $\frac{T_{FS}-T_{CPPS}}{T_{FS}} \times 100\%$, is tabulated in Table 2. Note that for the problem size indexed by 1 with $m = 5$, when $n > m$ (i.e. $n = 10, 15,$ and 20), the mapping scheme maps each module one-to-one to a different node without any resource sharing, and hence the scheduling is not applicable (marked by "–" in the table); so are the cases for the problem size indexed by 2 with $m = 10$ and $n = 15$ and 20, and the problem size indexed by 3 with $m = 15$ and $n = 20$. The average performance improvement measurements together with their corresponding standard deviations are plotted in Fig. 6.

Table 2. MED improvement percentage of CPPS over FS

Prb Idx	Num of Mods m	MED Improvement Percentage (%) n=5	n=10	n=15	n=20
1	5	0.4023	–	–	–
2	10	9.3800	0.0834	–	–
3	15	12.2844	4.1384	1.5431	–
4	20	12.3537	6.9193	9.2536	5.2417
5	25	12.8543	7.9172	11.3095	5.3134
6	30	13.6095	11.3572	11.7418	5.7544
7	35	14.0034	11.7431	13.4203	7.8624
8	40	15.4378	12.8595	13.5076	8.9894
9	45	15.3575	13.0498	13.8334	11.4692
10	50	18.8860	14.2790	11.1567	12.6608
11	55	17.8824	12.8271	13.1352	14.1371
12	60	14.6911	14.6843	14.1821	12.6241
13	65	17.3636	20.5909	16.7078	13.6390
14	70	17.1467	17.5231	17.9146	15.1107
15	75	15.9394	19.3904	16.6014	13.4181
16	80	21.3327	18.2815	18.5939	13.8302
17	85	17.8155	22.2571	18.8757	17.4691
18	90	19.0516	19.2820	17.6146	17.1279
19	95	18.7070	19.0163	17.6803	16.8793
20	100	20.2888	19.0798	18.4336	19.3523

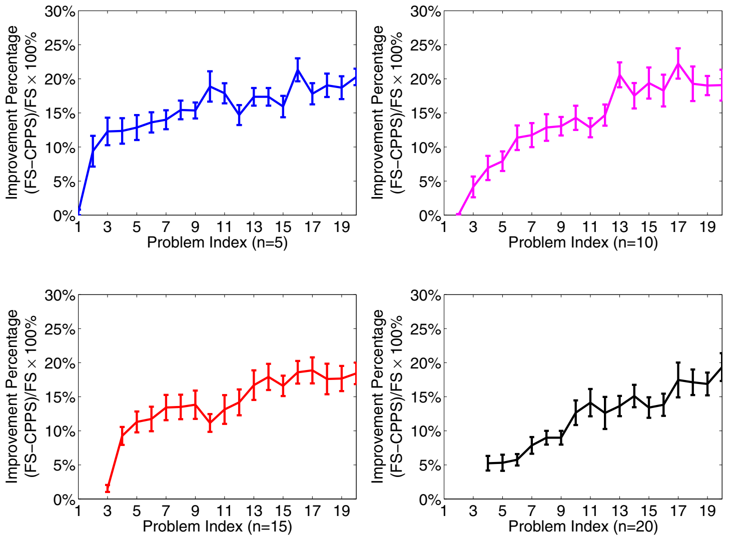


Fig. 6. MED improvement of CPPS over FS

The space for performance improvement largely depends on the given mapping scheme. In small problem sizes, or when the number of modules is comparable to the number of nodes, the modules are likely to be mapped to the nodes in a uniform manner, resulting in a low level of resource sharing, unless there exist some nodes whose processing power are significantly higher than the others in the network. Hence, in these cases, the MED improvement of CPPS over FS is not very obvious. However, as the problem size increases, more modules might be mapped to the same node with more resource sharing, hence leading to a higher performance improvement. This overall trend of performance improvement is clearly reflected in Fig 6.

5.2 Proof-of-Concept Experimental Results Using Climate Modeling Workflow

Weather Research and Forecasting (WRF). The Weather Research and Forecasting (WRF) model [29] has been widely used for regional to continental scale weather forecast. It is also one of the most widely used limited-area model for dynamical downscaling of climate projection by global climate models to provide regional details. The workflow for typical applications of WRF model takes multiple steps, including data preparation and preprocessing, actual model simulation, and postprocessing. Each step could be computationally intensive and/or involve a large amount of data transfer and processing. For a specific climate research project, such procedures have to be performed repeatedly, in the context of either routine short-term weather forecast, or periodic re-initialization of model in dynamical downscaling over the length of a climate projection. Moreover, because of the chaotic nature of the atmospheric system and unavoidable errors in the input data and the imperfection of the model, ensemble approaches have to be adopted with a sufficiently large number of simulations, with slight perturbations to initial conditions or physical parameters, to enhance the robustness of the prediction, or to provide probabilistic forecast for problems of interest. Each single simulation may require a full or partial set of preprocessing and postprocessing, in addition to the model simulation. Collectively, a project in this nature may involve the execution of an overwhelmingly large number of individual programs. A workflow-based management and execution is hence extremely useful to automate the procedure and efficiently allocate the resources to carry out the required computational tasks.

Climate Modeling Workflow Structure. The WRF model [4] is able to generate two large classes of simulations either with an ideal initialization or utilizing real data. In our workflow experiments, the simulations are generated from real data, which usually require preprocessing from the WPS package [5] to provide each atmospheric and static field with fidelity appropriate to the chosen grid resolution for the model.

As shown in Fig. 7, the WPS consists of three independent programs: geogrid.exe, ungrib.exe, and metgrid.exe. The geogrid program defines the simulation domains and interpolates various terrestrial datasets to the model grids. The user can specify information in the namelist file of WPS to define simulation domains. The ungrib program "degrib" the data and writes them in a simple intermediate format. The metgrid program horizontally interpolates the intermediate-format meteorological data that are extracted by the ungrib program onto the simulation domains defined by the geogrid program.

The interpolated metgrid output can then be ingested by the WRF package, which contains an initialization program real.exe for real data and a numerical integration program wrf.exe. The postprocessing model consists of ARWpost and GrADs. ARWpost reads-in WRF-ARW model data and creates output files for display by GrADS.

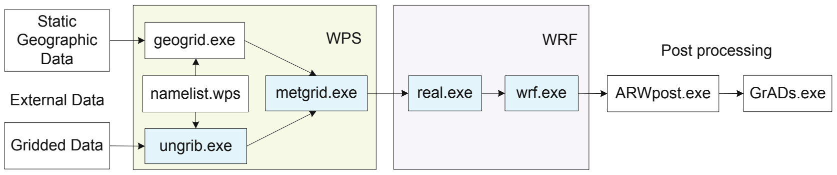


Fig. 7. The WPS-WRF workflow structure for climate modeling

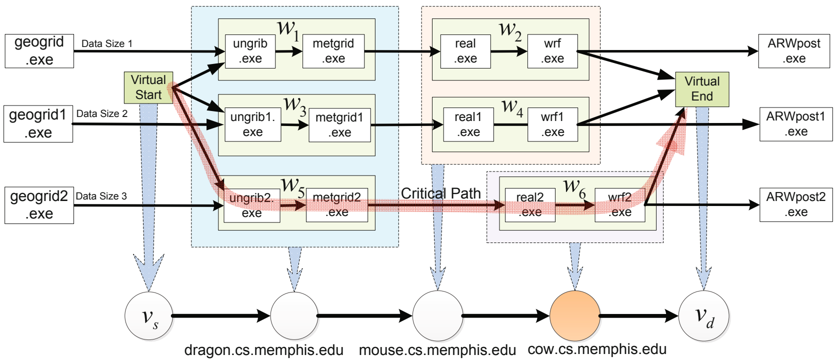


Fig. 8. The workflow mapping scheme for job scheduling experiments

Experimental Settings and Results. As shown in Fig. 8, in our experiments, we duplicate the entire WPS-WRF workflow to generate three parallel pipelines processing three different instances of input data of sizes 106.03 MBytes, 106.03 MBytes, and 740.89 MBytes, respectively. The testbed network consists of five Linux PC workstations equipped with multi-core processors of speed ranging from 1.0 GHz to 3.2 GHz. The computing requirement (CR) of each module and the processing power (PP) of each computer are measured or estimated using the methods proposed in [32]. In view of the computational complexity of each program, the scheduling experiments consider a subset of the programs in the original workflow, i.e. ungrib.exe, metgrid.exe, real.exe and wrf.exe, which serve as the main data processing routines. In the WPS part of each pipeline, we bundle ungrib.exe and metgrid.exe into one module denoted as w_1, w_3, and w_5, respectively; while in the WRF part of each pipeline, we bundle real.exe and wrf.exe into one module denoted as w_2, w_4, and w_6, respectively. We map modules w_1, w_3, and w_5 to dragon.cs.memphis.edu, modules w_2 and w_4 to mouse.cs.memphis.edu, and modules w_6 to cow.cs.memphis.edu. The virtual start and end modules are mapped to the other two machines. The preprocessing program geogrid and postprocessing program ARWpost are executed as part of the piplines, but are not considered for scheduling.

We conduct two sets of scheduling experiments on the above mapped workflow using fair-share (FS) and CPPS, respectively. In FS, each module is assigned by the system the default running priority, while in CPPS, we simply use the Linux command "nice" to adjust the level of priority to achieve coarse-grained control of execution. The module execution time and Minimum End-to-end Delay (MED) along the critical path using FS and CPPS are tabulated in Table 3. We observed that the MED performance improvement in this particular case is about 10.67%. The scheduling results in other cases with different workflow structures and mapping schemes are qualitatively similar. We would particularly like to point out that these small-scale workflow experiments with application-level coarse-grained control are conducted mainly for proof of concept. It is predictable that the performance superiority of CPPS over FS would be manifested much more significantly as the scale of computing workflows and the scope of distributed network environments continue to grow in real-life scientific applications.

Table 3. Performance measurements using FS and CPPS

FS		CPPS	
Module	Exec Time (sec)	Module	Exec Time (sec)
w1	31.506	w1	55.077
w2	79.893	w2	79.537
w3	31.821	w3	55.051
w4	79.728	w4	78.296
w5	43.397	w5	26.34
w6	125.551	w6	124.579
Critical Path	168.948	Critical Path	150.919

6 Conclusion and Future Work

In this paper, we formulated an NP-complete workflow scheduling problem and proposed a Critical Path-based Priority Scheduling (CPPS) algorithm. Extensive simulation results show that CPPS outperforms the traditional fair-share scheduling policy commonly adopted in real systems. We also conducted proof-of-concept experiments based on real-life scientific workflows deployed and executed in a testbed network.

However, finding a good on-node scheduling policy and finding a good mapping scheme are not totally independent. CPPS is able to improve the workflow performance over a fair-share algorithm for any given mapping scheme. We recognized that a better performance might be achieved if the interaction between the mapping and the scheduling is considered in the optimization, which will be explored in our future work.

We also plan to further refine the cost models by taking into consideration user and system dynamics and decentralize the proposed scheduling algorithm to adapt it to time-varying network environments. It is of our interest to investigate different ways to implement the scheduling algorithm (at either the application or kernel level) and compare their performances and overheads. We would also like to explore the possibilities to integrate this scheduling algorithm into existing workflow management systems and test it in wide-area networks with a high level of resource sharing dynamics.

Acknowledgments. This research is sponsored by U.S. Department of Energy's Office of Science under Grant No. DE-SC0002400 with University of Memphis.

References

1. DAGMan, `http://www.cs.wisc.edu/condor/dagman`
2. Energy Sciences Network, `http://www.es.net`
3. Internet2, `http://www.internet2.edu`
4. `http://www.wrf-model.org/index.php`
5. WRF Preprocessing System (WPS), `http://www.mmm.ucar.edu/wrf/users/wpsv2/wps.html`
6. Annie, S., Yu, H., Jin, S., Lin, K.C.: An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Trans. on Para. and Dist. Sys. 15, 824–834 (2004)
7. Bajaj, R., Agrawal, D.: Improving scheduling of tasks in a heterogeneous environment. IEEE Trans. on Parallel and Distributed Systems 15, 107–118 (2004)
8. Benoit, A., Robert, Y.: Mapping Pipeline Skeletons onto Heterogeneous Platforms. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007, Part I. LNCS, vol. 4487, pp. 591–598. Springer, Heidelberg (2007)
9. Boeres, C., Filho, J., Rebello, V.: A cluster-based strategy for scheduling task on heterogeneous processors. In: Proc. of 16th Symp. on Comp. Arch. and HPC, pp. 214–221 (2004)
10. Braun, T., Siegel, H., Beck, N., Boloni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B., Hensgen, D., Freund, R.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. JPDC 61(6), 810–837 (2001)
11. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I.: Programming scientific and distributed workflow with triana services. Concurrency and Computation: Practice and Experience, Special Issue: Workflow in Grid Systems 18(10), 1021–1037 (2006), `http://www.trianacode.org`
12. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M., Vahi, K., Livny, M.: Pegasus: mapping scientific workflows onto the grid. In: Proc. of the European Across Grids Conference, pp. 11–20 (2004)
13. The office of science data-management challenge, report of the DOE Office of Science Data-Management Workshop. Technical Report SLAC-R-782, Stanford Linear Accelerator Center (March-May 2004)
14. Fazar, W.: Program evaluation and review technique. The American Statistician 13(2), 10 (1959)
15. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman and Company, San Francisco (1979)
16. Goyal, D.: Scheduling processor bound systems. Tech. rep., Computer Science Department, Washington State University (1976)
17. Gu, Y., Wu, Q., Rao, N.: Analyzing execution dynamics of scientific workflows for latency minimization in resource sharing environments. In: Proc. of the 7th IEEE World Congress on Services, Washington DC, July 4-9 (2011)
18. Henry, G.: The fair share scheduler. AT&T Bell Laboratories Technical Journal (1984)
19. Kay, J., Lauder, P.: A fair share scheduler. Communications of ACM 31(1), 44–55 (1988)
20. Kelley, J., Walker, M.: Critical-path planning and scheduling. In: Proc. of the Eastern Joint Computer Conference (1959)
21. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: a 32-way multithreaded sparc processor. IEEE Micro Magazine 25(2), 21–29 (2005)

22. Kwok, Y., Ahmad, I.: Dynamic critical-path scheduling: An effective technique for allocating task graph to multiprocessors. IEEE Trans. on Parallel and Distributed Systems 7(5), 506–521 (1996)
23. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience 18(10), 1039–1605 (2006)
24. Mcnairy, C., Bhatia, R.: Montecito: A dual-core, dual-thread itanium processor. IEEE Micro. Magazine (2005)
25. Mezzacappa, A.: SciDAC 2005: scientific discovery through advanced computing. J. of Physics: Conf. Series 16 (2005)
26. Rahman, M., Ranjan, R., Buyya, R.: Cooperative and decentralized workflow scheduling in global grids. Future Generation Computer Systems 26, 753–768 (2010)
27. Raman, R., Livny, M., Solomon, M.: Resource management through multilateral matchmaking. In: Proc. of the 9th IEEE Int. Symp. on High-Perf. Dist. Comp. (August 2000)
28. Ranaweera, A., Agrawal, D.: A task duplication based algorithm for heterogeneous systems. In: Proc. of IPDPS, pp. 445–450 (2000)
29. Skamarock, W., Klemp, J., Dudhia, J., Gill, D., Barker, D., Duda, M., Huang, X., Wang, W., Powers, J.: A description of the advanced research wrf version 3. Tech. Rep. NCAR/TN-475+STR, National Center for Atmospheric Research, Boulder, Colorado, USA (June 2008)
30. Topcuoglu, H., Hariri, S., Wu, M.: Performance effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS 13(3) (2002)
31. Wassermann, B., Emmerich, W., Butchart, B., Cameron, N., Chen, L., Patel, J.: Sedna: a BPEL-based environment for visual scientific workflow modeling. In: Workflows for e-Science: Scientific Workflows for Grids, pp. 427–448. Springer, London (2007)
32. Wu, Q., Datla, V.: On performance modeling and prediction in support of scientific workflow optimization. In: Proc. of the 7th IEEE World Congress on Services, Washington DC, July 4-9 (2011)

Dynamic Kernel/Device Mapping Strategies for GPU-Assisted HPC Systems

Jiadong Wu, Weiming Shi, and Bo Hong

School of Electric and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{jwu65,weimingshi,bohong}@gatech.edu

Abstract. With their high computation throughput and outstanding performance-per-watt figures, the graphics processing units (GPU) are becoming increasingly important for high-performance computing (HPC) systems. Existing GPU execution environment restricts the GPU usage to local host node. This is suitable for standalone computer nodes, but becomes inefficient for HPC systems that consist of a large number of GPU-assisted nodes. In this paper, a novel framework is proposed to support dynamic GPU kernel/device mapping strategies for HPC systems. Adaptive mapping policies are designed to mitigate the impact of network transfer overhead. The performance of the framework is studied through extensive simulations. The results show that compared with existing local-only static mapping method, the proposed framework is capable of improving the system-wide GPU utilization rate and computation throughput, especially when the concurrent workloads exhibit different GPU usage intensities.

1 Introduction

The last two decades witnessed the evolution of graphics processing units (GPUs) from the graphics accelerators to the coprocessors that are becoming increasingly important for high-performance computing (HPC) systems. Thanks to the rapid advancement in GPU programming frameworks such as CUDA[11] and OpenCL[7], GPU computing has been successfully deployed for a wide range of applications in both desktop and HPC settings [12,10]. These applications cover a wide distribution of GPU usage intensities.

Existing GPU-assisted HPC systems often have a cluster structure where multiple GPU-assisted compute nodes are interconnected with high speed networks such as the InfiniBand. For such emerging GPU clusters, their resource management systems often adopt existing scheduling systems such as PBS [13]. These scheduling systems were originally designed for CPU-only systems, and are augmented to treat GPUs as one more type of resource on the compute nodes. When these job scheduling systems allocate user processes to the compute nodes, the execution of each process is controlled by the GPU execution environment of its host node. The user process is subsequently restricted to utilize the local GPU devices on the host node.

The per-node static kernel/device mapping method, while working well for standalone computer nodes, has significant shortcomings in HPC settings. Inefficiency would arise when the physical node configuration mismatches the workload pattern of the user processes:

1. *GPU underutilization* would be observed as GPU cards may be idle between kernels, especially when GPUs are used sporadically. Additionally, algorithmic requirements may restrict a host process to utilize only a subset of the locally available GPUs, thus wasting other GPUs. For example, an HPC application may be designed to use 2 GPUs on each node but is wastefully deployed to a 4-GPU-per-node system.
2. *GPU oversubscription* would be observed as the user processes of computation intensive applications may launch kernels faster than what the local GPUs can process. This is especially the case when the application consists of a large number GPU intensive tasks. With the static kernel/device mapping, the host processes may be starving for GPUs to process the tasks.

The overall system performance may therefore suffer from great performance degradation if applications with different GPU utilization run concurrently in the system, which will cause some GPUs to be underutilized while others oversubscribed.

Such static mapping method also affects programmability. With the existing method, a GPU-accelerated application may be (painfully) hand-optimized for a particular HPC deployment. But such optimization relies on the static kernel/device mapping and is therefore customized to the hardware configuration of that HPC system. When porting to a new/upgraded system with different configurations, those optimizations will become impaired and the code will underperform in the new system.

In this paper, we argue that these flaws of the current GPU-assisted HPC clusters can be alleviated and that the overall system performance and utilization can be improved when running unbalanced mixed workloads if a dynamic mapping strategies could be established between the GPU devices and the GPU kernels of the user applications. We present a novel idea of GPU resource management module (GREMM) that incorporates with existing remote GPU kernel execution technique and allows dynamic GPU kernel/device mapping. In particular, our study focuses on the dynamic kernel/device mapping policies that can proactively assign GPU kernels to remote GPUs that would otherwise be underutilized if the communication overhead is lower than the local waiting time. The main objective of the dynamic mapping framework is to refine the granularity of resource management and to explore both CPUs and GPUs to bridge the mismatch between the fixed physical node configurations and the varied workload requirement.

We demonstrate the efficiency of the proposed strategies by comparing against native systems (with static local kernel/device mapping). The results show that the dynamic kernel/device mapping outperforms the existing static execution environment in terms of the GPU utilization ratio and the computation throughput, especially for unbalanced mixed workloads. We expect the proposed framework to improve the efficiency of GPU-assisted HPC systems.

The rest of the paper is organized as follows. In Section 2, we provide the background information on our work and survey the related works. In Section 3, we introduce the dynamic kernel/device mapping framework and categorize its overheads, which lays down the foundation for our design of mapping strategies. In Section 4, we present the design of three mapping policies. In Section 5, we develop discrete event simulation to evaluate the performance. Some concluding remarks and future work are given in Section 6.

2 Background and Related Works

Existing GPU execution environments such as the Nvidia CUDA framework [11] assume the user processes to be bound to the local GPUs. A GPU kernel request is handled by the local GPU driver, which then loads the kernel on a local GPU device, executes it, and returns the results to the requesting process. As a way to resolve the scarcity of GPUs in many computer systems, GPU sharing has attracted intensive research attention. The existing techniques employed in GPU sharing is briefly summarized as follows.

PBS[13] and Slurm[14] are two widely adopted resource management systems for HPC systems. They were originally designed for CPU-based systems and have been upgraded to support GPU-assisted nodes. PBS and Slurm track user requests and system status, and map user processes to the compute nodes. In current PBS and Slurm systems, the process/kernel mapping is static, and the execution of the processes, once mapped, is governed by the compute nodes. For GPU-accelerated applications, the execution of each process is therefore subject to existing GPU execution environment on the compute nodes, which restricts user processes to utilize local GPU devices.

rCUDA [4] is proposed to enable the compute nodes not equipped with local GPUs to access the remote GPUs hosted on remote compute nodes. It employs API remoting technique to reroute the GPU calls to a remote GPU-assisted compute node. With rCUDA, the remote GPUs are statically specified in a configuration file on the requesting node. rCUDA works between a pair of designated nodes, and is particularly useful in a cluster environment where only a few nodes are equipped with GPUs. In such settings, rCUDA allows other non-GPU nodes to execute their GPU kernels on the GPU-assisted nodes, but the kernels in rCUDA-based system remains statically bound to devices, since the users have to hard-code the remote rCUDA server IP into their application. rCUDA is not designed to manage GPUs in an GPU-assisted HPC system.

GViM[6] is an API level solution to virtualize GPU systems. GViM is not designed to access remote GPUs since it can only virtualize GPUs on a standalone computer. Shadowfax[9] is proposed to address the access limit and to support unmodified applications in multiple virtual machines in order to share both local and remote GPUs. Similar to the static designation of remote GPUs in rCUDA, all virtual GPUs in Shadowfax need to be manually mapped to a physical GPU,

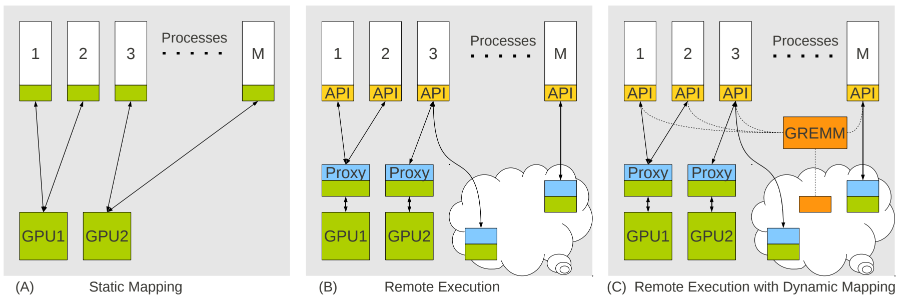


Fig. 1. Illustration of GPU kernel/device mapping models

which is unsuitable for managing GPUs in HPC systems where user application requests are not known as a priori.

The capability of remote GPU kernel execution is also explored in several other projects such as SnuCL[8], MGP[1], and gVirtuS [5]. Both SnuCL and MGP target to improve the programmability of GPU-assisted applications on a GPU cluster by providing a single system image. gVirtus focuses on providing a virtualization service which supports the remote GPU sharing. However, to the best of our knowledge, little research has been done on how to efficiently schedule the remote GPU device accesses in HPC systems.

3 Dynamic Kernel-Device Mapping

A novel HPC system framework is presented to facilitate dynamic kernel/device mapping strategies, and thereby improving the system-wide GPU resource utilization. The framework is essentially a combination of the existing remote kernel execution infrastructure and the decision maker of the kernel/device mapping.

3.1 The Framework

As we noted before, the prevailing GPU invocation method is restricted to access the GPUs on the board of the local compute node only - the kernel calls are routinely directed to the binding local GPUs as illustrated in Figure 1(A). With the development of the middleware such as rCUDA[4] and gVirtus[5], the scope of invocable GPUs is expanded to all the GPUs across the cluster system, as is illustrated in Figure 1(B). In these existing remote execution infrastructures, the mapping between the GPU kernel and device is still statically bound. But the restriction can be removed by extending such existing infrastructures.

While our work mainly focuses on the dynamic kernel-device mapping policy, it is informative to have an idea on how the framework would be constructed. To put it into perspective, such a framework can be decomposed into three components: (1) the front-end library of user API, (2) the GPU Resource Management Module (GREMM), and (3) the GPU execution proxy.

1. The front-end API library should provide equivalent interface as existing GPU programming environment such as CUDA or OpenCL, and implement functionalities that communicate to GREMM. By linking to this library, programmers can write conventional GPU codes for their applications without considering how the GPU calls are handled. Once the executable is linked with this library instead of the stock one, GPU related functions will be automatically wrapped into task messages and dynamically forwarded to proper proxy based on GREMM's decision.
2. The GPU resource management module is the middle layer of our framework that connects GPU API calls and the execution proxies. As the heart of the system, the GREMM is responsible for making the kernel mapping decisions. A variety of policies can be included in our design. Based on a specific policy, the modules will work either independently or cooperatively to assign GPU kernels.
3. The GPU execution proxy is the bottom layer of the dynamic mapping framework responsible for the host/device memory copying, kernel launching, and other device control functions. Each proxy controls one local GPU device and communicates with the local and remote API callers. Guided by the GREMM, every GPU task message will eventually be served at a execution proxy.

The described framework constitutes a direct and easy extension of the existing remote kernel execution infrastructure. More importantly, among all the design choices, we contend that the decision maker can be put into a separate module, referred to as GREMM and also illustrated in Figure 1(C), so that not too much change would be made on the side of remote kernel execution infrastructure to install various mapping policies.

3.2 Categorization of Overheads

Applications running on existing GPU-assisted HPC systems are subjected to the overhead of queuing for the statically mapped GPU devices. At the system level, this overhead is expected to be lowered through balanced allocation of GPU devices in dynamic mapping framework.

However, remote kernel execution does introduce a new type of overhead: the network overhead. Network overhead will not incur for the traditional kernel/device mapping method since it only uses local GPUs, but will incur when the GPU kernel needs to be executed on a remote node. The amount of this overhead is directly related to the volume of transfered data and network performance. Data intensive workloads will lead to negative performance gains. But the performance degradation could be avoided if an appropriate policy is available to track workload data/computation ratio and decide when to activate remote execution and when to fall back to the local-only method.

We will study the impact of network overhead and also the benefits of reduced queuing overhead in the following sections.

4 Dynamic Kernel/Device Mapping Policies

In this section, an abstraction of GPU-assisted HPC clusters is presented, followed by the design and evaluation of three mapping policies for the dynamic kernel/device mapping framework.

4.1 System Abstraction

We consider the following abstraction of GPU-assisted HPC clusters. There are N homogeneous compute nodes in the system, each configured with M processor cores per node and K GPU devices per node. And we assume $M \geq K$. A user application consists of multiple processes. Processes from all the applications are mapped to the compute nodes by a job scheduling system that employs the following rules: (1) a compute node will not be split among multiple applications, (2) processes do not migrate once mapped, (3) each compute node receives less than M processes, and (4) compute nodes receive balanced workload for each application. Such a job scheduling policy represents the typical practice of many popular scheduling systems (e.g. PBS).

We assume that each process executes a program code consisting of multiple iterations, where each iteration consists of a CPU code segment followed by a GPU code segment – the GPU kernel.

We further assume that the programmer will explicitly copy back any useful data from GPU after a kernel is finished, so the GPU context associated with certain process becomes volatile when its new kernel is not launched on the same GPU device as before. Discussions on this limitation will be presented in the final section.

4.2 Global Reservation Policy

In Global Reservation (GR) Policy, a FIFO queue is set up for the GPU cluster. GPU tasks launched by any process will be registered in this queue, which will later be served by a total number of $N \times K$ GPU devices. The actual data transfer occurs directly between the requesting process and the serving GPU, and is not transferred via the queue. Theoretically, if an infinite fast network interconnection is given, the global reservation policy is expected to achieve the best system-wide GPU utilization.

However, because the GPU device needs to be reserved while data/kernel is being transferred from a remote node, the efficiency of this policy is highly sensitive to the network overhead. For the proposed dynamic kernel/device mapping to perform well under environments of varied workload, adaptive policies are then explored.

4.3 Adaptive Greedy Policy

Adaptive Greedy (AG) Policy aims to map the kernel call to the GPU device that requires the least total waiting time every time a new kernel call is initiated.

Denote all the GPUs in the system be $\mathcal{G}$, the set of local GPUs be $\mathcal{L}$. The number of all GPUs in the system is $|\mathcal{G}| = NK$.

AG examines every GPU device g in the system, estimates the total waiting time τ_g if the kernel call is mapped to that GPU. The total waiting time τ_g is composed of the queueing delay τ_g^q and the data transfer delay τ_g^d.

$$\tau_g = \tau_g^q + \tau_g^d \tag{1}$$

The queueing delay τ_g^q is estimated by the number of queued kernel calls on that GPU device N_g and the average execution time of last k kernel calls on that GPU device τ_g^k.

$$\tau_g^q = N_g \cdot \tau_g^k \tag{2}$$

The data transfer delay τ_g^d is zero if g is a local GPU device and is estimated by the amount of data transferred from the host node to the remote node D_{out}, the amount of data transferred back from the remote node to the host node D_{in} and the outbound (resp. inbound) bandwidth B_g^{out}(resp. B_g^{in}) if g is a remote device.

$$\tau_g^d = \begin{cases} 0 & \text{if } g \in \mathcal{L} \quad (3) \\ \dfrac{D_{out}}{B_g^{out}} + \dfrac{D_{in}}{B_g^{in}} & \text{if } g \notin \mathcal{L} \quad (4) \end{cases}$$

B_g^{out} is estimated by the nominated inter-node bandwidth BW and the number of out-bound kernel calls on the host node, namely O_l and the number of in-bound kernel calls on g, namely I_g when the kernel call is to be assigned.

$$B_g^{out} = \frac{BW}{\max_{g \in \mathcal{G}-l}(O_l, I_g) + 1} \tag{5}$$

B_g^{in} is estimated by the nominated inter-node bandwidth BW and the system-wide average number of queued kernel calls per node. Notice that B_g^{in} is different from B_g^{out} as the bandwidth may change with the progress of the kernel execution.

$$B_g^{in} = \frac{BW}{\sum_{g \in \mathcal{G}-l} \max(O_g, I_l)/|\mathcal{G}|} \tag{6}$$

AG chooses the node g^* with the least total waiting time as the candidate node that the kernel call is to be assigned. The computational complexity of AG is $O(|\mathcal{G}|) = O(NK)$.

$$g^* = \arg\min_{g \in \mathcal{G}} \tau_g \tag{7}$$

4.4 Adaptive Random Policy

Adaptive Random (AR) Policy is a randomized policy. It tries to construct and maintain a table which records the probability that a particular GPU device should be chosen to serve the kernel call. It assigns the kernel call based on the probabilities in the maintained table. It resorts to the GPU driver to handle the contention for the GPU device on a particular node if there is any.

The probability of being chosen is calculated based on a weight table that is associated with the system-wide GPU availability. In this table, each GPU device is assigned a weight indicating the relative probability of being chosen.

Denote the nominated inter-node bandwidth be B, the weight of a remote idle (resp. busy) GPU device be w_{ri} (resp. w_{rb}), the weight of a local idle (resp. busy) GPU device be w_{li} (resp. w_{lb}).

Assume that the inertia towards choosing the busy GPU devices over the idle ones is characterized by a 'penalty' factor $\alpha(< 1)$, and that the preference towards choosing the local GPU devices over the remote ones is characterized by a 'bonus' factor $\beta(> 1)$. Hence we have

$$\alpha = \frac{w_{lb}}{w_{li}} = \frac{w_{rb}}{w_{ri}}, \tag{8}$$

$$\beta = \frac{w_{lb}}{w_{rb}} = \frac{w_{li}}{w_{ri}}. \tag{9}$$

Without loss of generality, if we set the $w_{ri} = 1$, then $w_{li} = \beta$, $w_{rb} = \alpha$, $w_{lb} = \alpha\beta$.

The 'penalty' factor α can be quantified by the average execution time of received kernel calls τ_g^k and the node configuration of the host node.

$$\alpha = \frac{M}{K} \cdot \frac{1}{\tau_g^k} \tag{10}$$

The design philosophy of α is that the relative probability ratio of choosing a busy node over choosing a idle node should be proportional to the relative ratio of the time ticks that a node is idle, and that the larger the ratio of the number of GPUs versus the number of CPUs on a host node, the less chance the kernel calls should be assigned to remote nodes.

The 'bonus' factor β can be quantified by the amount of transferred data D, the average execution time of received kernel calls τ_g^k and the number of nodes in the system N. The design philosophy of β is that the higher the ratio of the communication time to the computation time, or the higher the data consumption rate, or the more nodes in the system, the more chance the local nodes are favored over the remote nodes.

$$\beta = \left(\frac{D/B}{\tau_g^k}\right) \cdot \left(\frac{D}{\tau_g^k}\right) \cdot N = \left(\frac{D}{\tau_g^k}\right)^2 \cdot \frac{N}{B} \tag{11}$$

The computational complexity of AR is also $O(|\mathcal{G}|) = O(NK)$ while the information it needs to keep is less than AG. When a GPU task arrives, GREMM makes the randomized assignment decisions based on the weights in this table.

More sophisticated mapping policies can be designed, for example, to explore execution history of the system and to accept users' hint about the pattern of their jobs. We leave the exploration of the advanced mapping policies for our future work. In this paper, we focus on the benefits of dynamic GPU kernel/device mapping and its effectiveness under different workload and system conditions. Greater performance improvement is expected when more advanced mapping policies are adopted.

5 Performance Evaluation

In this section we develop a discrete event simulator to simulate the runtime behavior of large-scale GPU-assisted clusters. The performance of dynamic kernel/device mapping strategies is then verified through extensive simulations.

It is desirable to evaluate the dynamic kernel/device mapping framework in a large-scale production GPU-assisted HPC using real benchmark workloads. But because GPU-based HPC computing is an emerging field, there does not exist well-established workload traces for this type of systems. Available GPU benchmarks (e.g. RODINIA[2] and SHOC[3]) are designed to stress micro-architectural features of GPUs, which are unsuitable to describe multiple concurrent workloads at the system level for our study. To address this issue, we synthesized our workload traces, which are designed to be representative of GPU-HPC workloads.

5.1 Experimental Setup

The four GPU mapping policies tested are: 1) ST, the static kernel/device mapping policy; 2) GR, the global reservation policy; 3) AR, the adaptive random mapping policy with $k = 10$; and 4) AG, the adaptive greedy mapping policy with $k = 10$. The ST policy, as our baseline, is the conventional policy in GPU execution environment which shows the GPU utilization of the native system without remote execution or dynamic mapping. GR, AR, and AG are dynamic kernel/device mapping policies.

The two major performance metrics evaluated are GPU Utilization Rate and Mean Waiting Time. GPU Utilization Rate is the ratio of the GPU busy time to the total GPU time available. This rate directly reflects the utilization efficiency of the entire GPU cluster. The Mean Waiting Time measures the average time that a GPU task spends on data transfer and queuing for GPU devices. It reflects the average overhead for each kernel execution.

5.2 GPU-Assisted Cluster Simulator

The simulated cluster consists of N computing nodes. Each node consists of M CPU cores, K GPU devices, and a full-duplex network interface card (NIC) with max bandwidth B. The CPU cores are characterized by the processing capabilities. GPU devices are also characterized by their processing capabilities. The latency of remote execution API functions is modeled based-on data observed in previous researches[4,6,9].

The NIC on each node has two independent ports: the inbound port and the outbound port. A max bandwidth B is enforced on each port. Once any data is to be transmitted from one node to another, a connection will be established from the outbound NIC port of the source to the inbound NIC port of the sink. Concurrent connections on a single port share the port's bandwidth evenly. However, the effective bandwidth of a connection is limited by the busier one of the two participating ports. So, if any one of the concurrent connections fails to fully utilize its share, the remaining bandwidth will be utilized by other concurrent connections. According to this scheme, the system-wide bandwidth allocation changes when a new connection is established or a current connection is completed. We adopt this simplified network model as our focus is on the impact of network transfer overhead, rather than on how the overheads are generated. Therefore, the detail characteristics of a typical network such as the topology and the routing are not taken into account in this work.

Unless explicitly noted later, the cluster in following simulations is configured as $N = 24$, $M = 12$, $K = 3$. The bandwidth is set to $B = 100KB/ms$ for GbE and $B = 1000KB/ms$ for IB. The simulated time span is $10,000s$.

5.3 Generation of Workload Traces

The input to the simulator is the workload trace, which is organized as groups of consecutive *tasks*. Each group is associated with one software process. We characterize a CPU task by the amount of time T_c delayed on the CPU core, and a GPU task by three parameters: the amount of data D_u uploaded to the GPU device; the amount of execution time T_g on GPU device; and the amount of data D_d downloaded from the GPU device to host process.

The workload used for the evaluation of the dynamic mapping framework is generated based on following assumptions:

- Each process executes CPU and GPU tasks alternatively.
- The execution time of CPU and GPU task is random variable of exponential distribution with parameter $\lambda = 1/T_c$, $\mu = 1/T_g$ respectively.
- The size of the input and output data sets of a GPU kernel is proportional to the kernel execution time.

For example, a process P generated with parameters $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$, and $T_c = 750ms$ will have the following characteristics: the average length of GPU kernel is $2250ms$; the average data uploaded to GPU each time is $22500KB$; the average data downloaded from GPU each time is $11250KB$; and the average time spent on CPU before next GPU kernel launch is $750ms$.

Since the policies are designed to address unbalanced GPU utilizations of concurrent GPU workloads in an HPC system. The traces we used are mixed combinations of a heavy-GPU application and a light-GPU application. We assume that the system runs these two applications with full capacity: there are n_i (assuming n_i is a multiple of $\frac{M}{K}$) processes in Workload i, and $n_1 + n_2 = N \cdot M$. In such case, the GPUs in the system are subject to the different computation

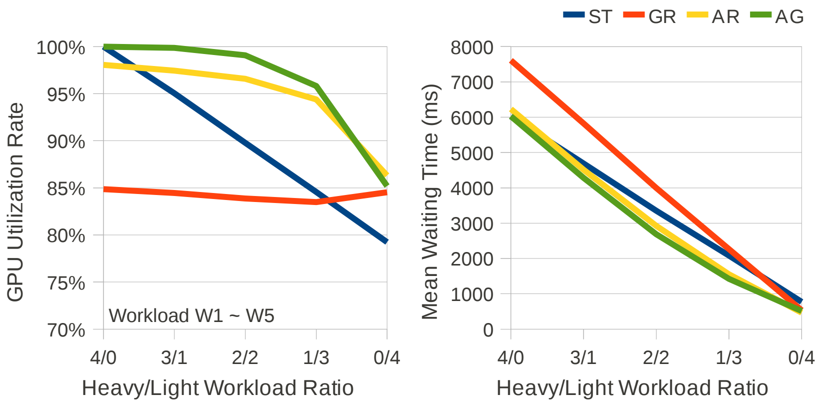


Fig. 2. Impact of workload mix

intensity. The benefit of routing a kernel from a stressed node to an idle remote node can potentially overweigh the extra overhead of network transfer. Our analysis can be extended to scenarios with more applications.

5.4 Workload Mix

Our first set of experiments examines a set of mixed workloads. Traces W_1 to W_5 are synthesized from two client applications submitted to the cluster. Client H's application consists of tasks with heavy GPU usage (with $T_c = 750ms$, $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$) and client L's application consists of tasks with light GPU usage (with $T_c = 2250ms$, $T_g = 750ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$). The five traces are synthesized to represent the mix of two workloads with different GPU demands. The process population ratio of H/L is 24/0 in W_1, 18/6 in W_2, 12/12 in W_3, 6/18 in W_4, and 0/24 in W_5.

The system is simulated with the network bandwidth set to $100KB/ms$ (GbE). As shown in the left subplot of Figure 2, the system-wide GPU utilization rate can be improved by dynamic mapping policies in most of the cases. Since there are underutilized GPU devices on the nodes, transferring GPU tasks from heavily occupied local devices to remote idle devices is beneficial. It is worth noting that significant improvement can be observed for the adaptive policies even for such low bandwidth network. This indicates that the dynamic kernel/device mapping is particularly useful for mixed workloads that have different GPU demands. Meanwhile, the mean waiting time is also improved as is shown in the right subplot of Figure 2.

Figure 3 shows the number of completed GPU kernels under different policies on Traces W_1 to W_5. Taking W_3 as an example, in the simulated time span, the conventional ST policy finishes about $12K$ kernels for client H and about $28K$ kernels for client L. When the dynamic policies are applied, the overall

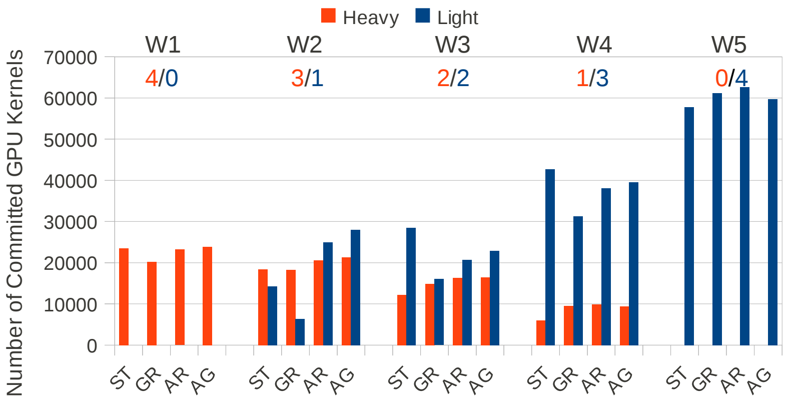


Fig. 3. Number of completed GPU kernels with different policies

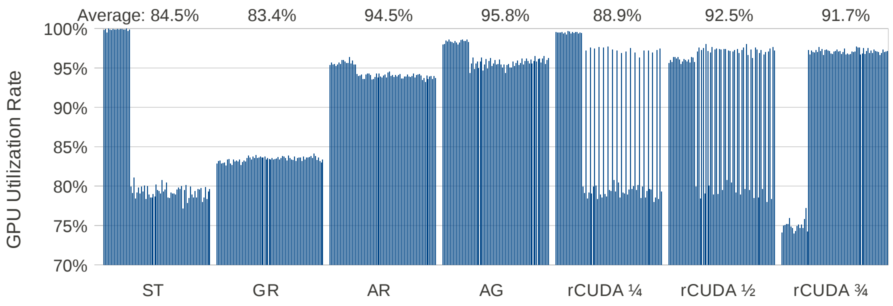


Fig. 4. Detailed GPU Utilization of the Cluster with static and dynamic policies

system-wide GPU utilization rate is improved. It is also interesting to note that client L is affected by the other policies, i.e. client L completes less kernels if remote GPU mapping is allowed. This is because the GPUs previously dedicated to client L are now executing client H's kernels too. This set of experiment suggests that if certain client's application is mission-critical, it is desirable to exclude other applications from utilizing its GPU devices, even though this will reduce the GPU utilization rate of the system. We plan to investigate the prioritized policy in our future study.

5.5 Load Balance

Figure 4 lists the detailed GPU utilization of the cluster with different policies on the mixed workload trace W_4, since W_4 is a very good example to demonstrate the performance improvement of the dynamic kernel/device mapping. The bandwidth is set to $100KB/ms$ in this and the following experiments as well.

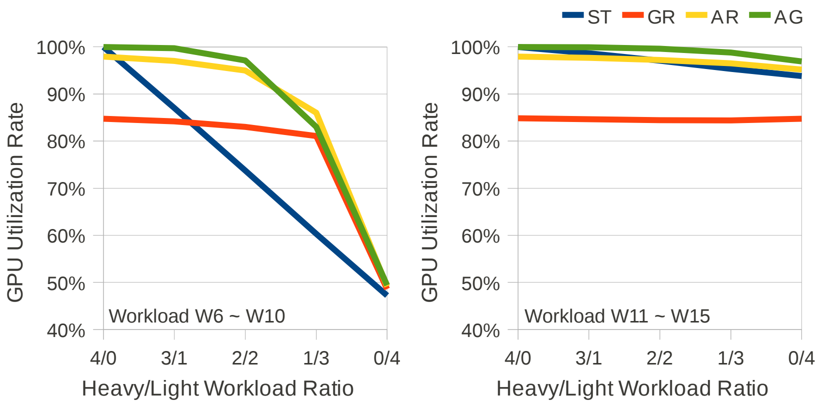


Fig. 5. Impact of GPU utilization intensity

It shows that the utilization with ST is negatively affected by the unbalanced node utilizations. The GR policy is capable of balancing GPU utilization. The AR and AG policy outperforms the other policies for this set of experiments.

As mentioned in the background section, techniques such as rCUDA allow a process to send all its GPU kernels to a statically designated remote node, but they do not support run-time kernel/device mapping. For fair comparison, we tested three static schedulers for rCUDA on workload W_4: 1/4, 1/2, and 3/4 of the client H's GPU kernels were directed to client L's GPUs. The results show that when 1/2 of client H's processes can use rCUDA, the system achieves GPU utilization rate of 92.5%, which is still worse than the performance of the dynamic mapping policies. Nevertheless, the results also demonstrate the difficulty in optimizing the performance by the static scheduler of rCUDA: ratios 1/4 and 3/4 are less efficient, finding the better ratio of 1/2 is non-trivial. Furthermore, since the rCUDA mapping decision needs to be made before launching user applications, it is infeasible to use rCUDA for actual HPC applications since there does not exist a single static mapping policy that will be suitable for all kinds of workloads.

5.6 Workload Intensity

The impact on the GPU utilization intensity is demonstrated in Figure 5. Two new groups of workloads (W_6-W_{10} and W_{11}-W_{15}) are used in the experiment. The generating parameters of these workloads are the same as that of W_1 to W_5, except that the (T_c, T_g) of light workload is set to $(2625, 325)$ in W_6-W_{10} and $(1815, 1125)$ in W_{11}-W_{15}. As the results show, the dynamic policies are significantly effective only if enough underutilized GPUs exist. In the lighter group (W_6-W_{10}), up to 26% improvement can be observed, but in the heavier group (W_{11}-W_{15}) the improvement is limited by the existence of over-utilized GPUs.

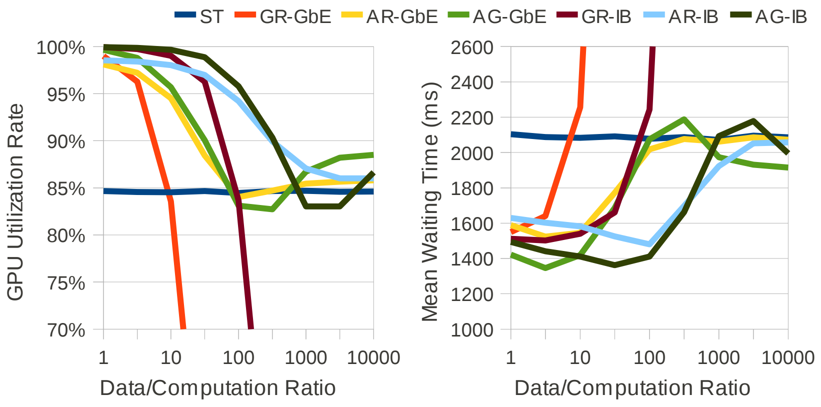


Fig. 6. Impact of network bandwidth

5.7 Network Overhead and Efficacy of Adaptation

In this experiment, we examine the sensitivity of the policies to the network bandwidth and the data/computation ratio of the GPU kernels, which are two key factors that affect the network transfer overheads introduced by the remote execution of GPU kernels.

Figure 6 shows the system-wide GPU utilization of different policies and the underlying interconnect with varied D_u/T_g (data/computation ratio). Here D_u of the workload W_4 is sampled exponentially from $D_u = 1 \times T_g$ to $D_u = 10000 \times T_g$. As D_u/T_g increases, the overhead of remote-execution increases, which negatively affects the performance of the dynamic mapping policies (and especially of the GR policy). This indicates that the amount of transferred data or the network bandwidth plays an important role in making dynamic mapping policies effective and efficient. However, thanks to the adaptation mechanism, the performance of AR and AG can still be as good as ST when the ratio is extremely high.

The benefit of the adaptation mechanism can be clearly demonstrated with Figure 7. In this experiment, we explicitly assign several fixed values to α and β, and compare these fixed-weight random policies to AR. The result reveals that the fixed-weight may favor either the low data/computation ratio workload or the high data/computation workload. Only the adaptive-weight in AR can track the best performance over the entire range of data/computation ratio.

5.8 Scalability

The scalability of the dynamic mapping policies is evaluated in the following two experiments: scalability with respect to the number of GPUs per node, and with respect to the number of nodes. Trace W_4 is used for the first set of experiments. For the second set of experiments, the four tested traces are half-sized, normal-sized, double-sized, and quadruple-sized versions of W_4. The network bandwidth

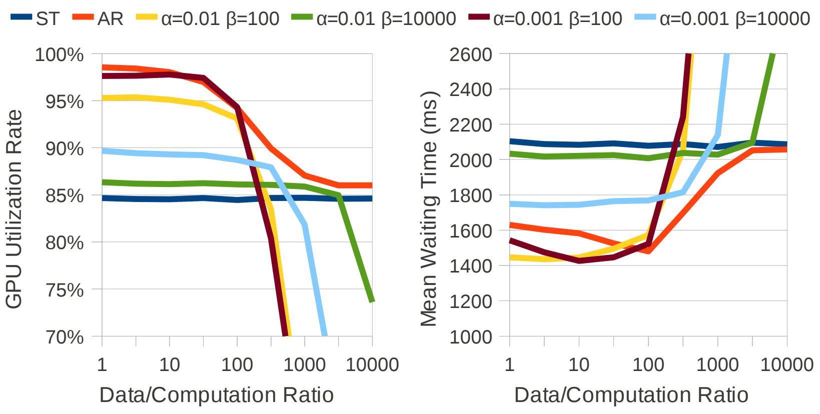


Fig. 7. Benefit of the adaptation mechanism

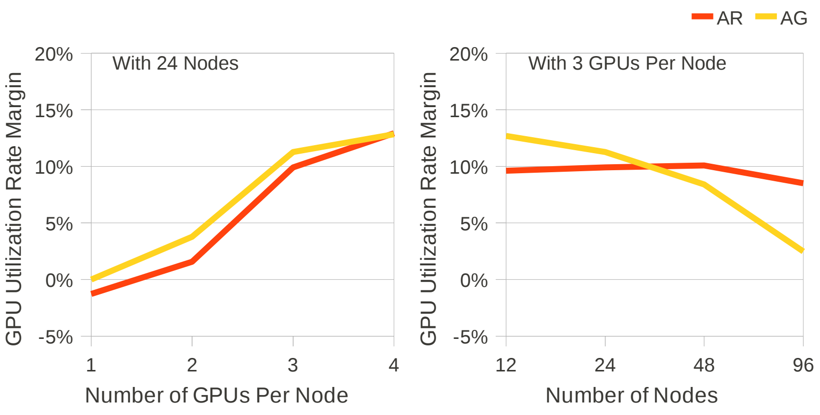


Fig. 8. Scalability of dynamic mapping policies over static mapping

is set to $100KB/ms$. The GR policy is excluded in this experiment due to its poor performance over lower-bandwidth network.

In the experiments, we observe higher possibility of underutilization by static mapping when more GPUs are installed in the cluster. In such cases, the necessity of an efficient GPU resource management policy becomes more significant.

The values reported in Figure 8 are the GPU utilization rate margin of the dynamic mapping policies over the ST policy. According to the results, both AR and AG exhibit good scalability over the number of GPUs per node. The AR policy also exhibits good scalability over the number of nodes. However, the AG policy doesn't scale well with the number of nodes. The key reason is that estimating the delay times in a larger-scale system becomes harder and less accurate. The larger amount of collaborative communication incurred during AG's decision making process also impairs its scalability over the number of nodes.

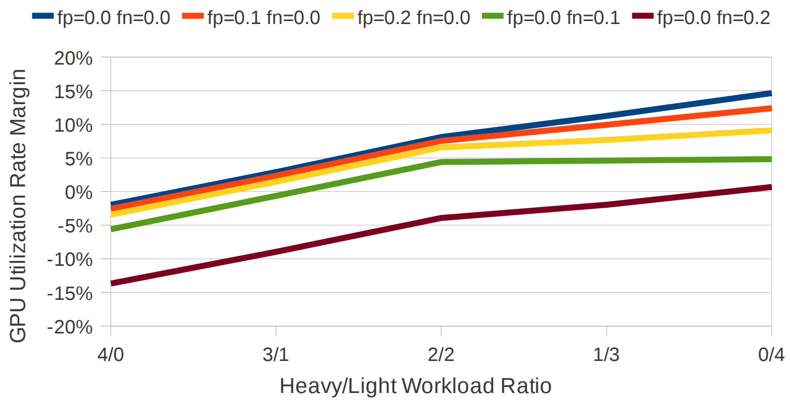


Fig. 9. The impact of false positive and false negative ratio on AR

Since both AG and AR rely on certain amount of global information to make scheduling decisions, their performance could be significantly compromised if the system scales up to thousands of nodes. To accommodate such large systems, one effective way is to group the nodes into subsets and schedule remote GPU accesses within each subset. In future research, an alternative policy relying on distributed information and local estimation will be studied.

5.9 Design Choices for the AR Policy

This set of experiments evaluates the performance of AR policy over certain design choices. As demonstrated in the previous experiments, AR is a balanced policy with several distinct advantages. One important choice in the implementation of this policy is how to maintain the distributed table about the GPU status. Real-time update is less desirable since it may incur extra network overhead. On the other hand, if the table is updated less frequently, outdated information may be used for GPU kernel/device mapping. We define a case to be false positive if an idle GPU is identified as busy, and false negative if a busy GPU is identified as idle. We used traces W_1 to W_5 to evaluate the performance of AR policy over different false positive ratio/false negative ratio. The values reported in Figure 9 are the GPU utilization rate margin of AR over ST. As shown in the figure, the performance is more sensitive to the false negative ratio than the false positive ratio. This implies that the status should be updated as soon as possible when a certain GPU becomes busy and the update is less urgent when a GPU becomes idle if the performance of the AR policy is valued.

6 Conclusion and Future Work

To address the performance degradation of GPU-assisted HPC system due to the mismatch between the physical node configuration and the GPU utilization of mixed workloads, we present the idea of dynamic kernel/device mapping, which relaxes the static binding between GPU kernels and local GPUs as in existing systems, and provide a sample design with the functionalities of remote kernel execution and GPU resource management, based on which the dynamic GPU allocation policies are further designed to balance the utilization of GPUs. The benefit and efficiency of the strategies is demonstrated through simulation-based studies, which show that the dynamic mapping strategies outperforms the existing static kernel/device binding in terms of the GPU utilization and the mean waiting time for processes to acquire GPUs.

As we noted, communication intensive workload does pose challenges for the dynamic kernel/device mapping. However, if an advisable adaptive policy is adopted such as the proposed AR and AG policies, the dynamic mapping strategies will outperform existing methods for suitable workloads, and (effectively) fall back to the existing method for unsuitable workloads (e.g. communication intensive or very short kernels, both of which are untypical for GPU-assisted HPC applications). The dynamic mapping strategies provide the mechanism to improve GPU utilization for HPC systems when possible.

Additionally, existing GPU supports the concept of context where all the kernels launched from a user process are able to reuse the data that reside in GPU's global memory. Consequently, utilizing the same device for multiple kernels can save considerable amount of time for data movement. We plan to explore such context-based locality and design policies to re-utilize a remote GPU device for consecutive kernel calls from a process in order to reduce the cost of network transfer. We also plan to study the impact of process synchronization (e.g. MPI barriers) on the dynamic mapping kernel/device policy.

Acknowledgment. This work is supported by the US National Science Foundation under award number CNS-0845583.

References

1. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A package for opencl based heterogeneous computing on clusters with many gpu devices. In: 2010 IEEE International Conference on Cluster Computing Workshops and Posters (Cluster Workshops), pp. 1–7 (September 2010)
2. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54. IEEE (2009)
3. Danalis, A., Marin, G., McCurdy, C., Meredith, J., Roth, P., Spafford, K., Tipparaju, V., Vetter, J.: The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 63–74. ACM (2010)
4. Duato, J., Pena, A., Silla, F., Mayo, R., Quintana-Ortí, E.: rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing and Simulation (HPCS), pp. 224–231. IEEE (2010)

5. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 379–391. Springer, Heidelberg (2010)
6. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: Gvim: Gpu-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17–24. ACM (2009)
7. Khronos-Group. Opencl - the open standard for parallel programming of heterogeneous systems (2011)
8. Kim, J., Kim, H., Lee, J., Lee, J.: Achieving a single compute device image in opencl for multiple gpus. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, pp. 277–288. ACM (2011)
9. Merritt, A., Gupta, V., Verma, A., Gavrilovska, A., Schwan, K.: Shadowfax: scaling in heterogeneous cluster systems via gpgpu assemblies. In: Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing, pp. 3–10. ACM (2011)
10. Nickolls, J., Dally, W.: The gpu computing era. IEEE Micro. 30(2), 56–69 (2010)
11. Nvidia. Gpu computing sdk (2011)
12. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: Gpu computing. Proceedings of the IEEE 96(5), 879–899 (2008)
13. PBS-Works. Scheduling jobs onto nvidia tesla gpu computing processors using pbs professional (2011)
14. Trofinoff, S.: Scheduling gpus with slurm (2011)

Optimal Co-Scheduling to Minimize Makespan on Chip Multiprocessors

Kai Tian[1,⋆], Yunlian Jiang[2], Xipeng Shen[3], and Weizhen Mao[3]

[1] Microsoft
kaitian@microsoft.com
[2] Google
yunlian@google.com
[3] Computer Science Department, The College of William and Mary
{xshen,wm}@cs.wm.edu

Abstract. On-chip resource sharing among sibling cores causes resource contention on Chip Multiprocessors (CMP), considerably degrading program performance and system fairness. Job co-scheduling attempts to alleviate the problem by assigning jobs to cores intelligently. Despite many heuristics-based empirical explorations, studies on *optimal* co-scheduling and its inherent complexity start only recently, and all have concentrated on the minimization of total performance degradations. There is another important criterion for scheduling, makespan, which determines the finish time of a job set. Its importance for job co-scheduling on CMP is increasingly recognized, especially with the rise of CMP-based compute cloud, data centers, and server farms. However, optimal co-scheduling for makespan minimization still remains unexplored.

This work compares makespan minimization problem with previously studied cost minimization (or degradation minimization) problem, revealing these connections as well as significant differences. It uncovers the computational complexity of the makespan minimization problem, and proposes a series of algorithms to either compute or approximate the optimal schedules. It proves that the problem is NP-complete in a general setting, but for a special case (dual-core without job migrations), the problem is solvable in $O(n^{2.5} \cdot \log n)$ time (n is the number of jobs). In addition, this paper presents a series of algorithms to compute or approximate the optimal schedules in the general setting. Experiments on both real and synthetic problems verify the optimality of the optimal co-scheduling algorithms, and demonstrate the reasonable accuracy and scalability of the approximation algorithms. The findings may advance the current understanding of optimal co-scheduling, facilitate the evaluation of real co-scheduling systems, and provide insights for the development of practical co-scheduling algorithms.

Keywords: Multicore Co-Scheduling, Optimal Co-Scheduling, Makespan Minimization, A-star algorithm, Migration.

1 Introduction

In a Chip Multiprocessors (CMP) system, multicores typically share certain resource (e.g., last-level cache) on a chip. The sharing, although shortening the communication

⋆ This work was done when Kai Tian and Yunlian Jiang were in The College of William and Mary.

among cores, causes resource contention among co-running jobs. Many studies have reported considerable, and sometimes significant, effects of the contention on program performance and system fairness [4, 7, 9, 14, 22]. The urgency for alleviating the contention keeps growing as the processor-level parallelism increases continuously.

Recent years have seen many interests in using job co-scheduling to alleviate the contention [7, 10, 19]. The basic strategy of job co-scheduling is to assign jobs to cores in a way that the overall influence from resource contention is reduced.

The prior explorations fall into two categories. The first includes the research that aims at constructing practical on-line job scheduling systems. It concentrates on heuristics-based lightweight scheduling techniques. A typical example is the symbiotic co-scheduling by Snavely and Tullsen [19]. The co-scheduler in the system samples the performance of different schedules during runtime and selects a good one. Other examples include the fair-miss-rate-based co-scheduling by Fedorova et al. [7], thread clustering by Tam et al. [23], and so on.

The second category includes the studies on optimal co-scheduling. The goal is to uncover the complexity in finding optimal co-schedules and develop algorithms to either compute or approximate optimal co-schedules. Optimal co-scheduling typically happens offline, requiring considerable computation, certain knowledge obtained through profiling runs of jobs, and other conditions. It is not for direct uses in on-line job scheduling systems, but for exposing the limit to facilitate the evaluation of practical schedulers. Without knowing optimal schedules (or a reasonable approximation), it is hard to precisely determine how good a scheduling algorithm is—how far a solution given by the scheduling algorithm is from the optimal solution and whether further improvement will enhance performance significantly, both of which are important for the design and deployment of practical co-scheduling systems.

Research in optimal co-scheduling is still in a preliminary stage. Although some studies are relevant to optimal co-scheduling (e.g., co-run cache performance prediction [2, 4] may simplify the profiling requirement), direct attacks to the problem start only recently [10, 11, 24]. The objectives of the previous explorations are all on the minimization of co-run cost (i.e., the sum of each job's co-run performance degradation).

But besides cost, there is another important criterion in job scheduling, makespan. *Makespan* refers to the time between the start of a job set and the finishing of the last job in the set. Minimizing makespan is important in situations where a simultaneously received batch of jobs is required to be completed as soon as possible. For example, a multi-item order submitted by a single customer needs to be delivered in the minimal time. This kind of situation is especially common in server farms, data centers, and compute cloud (e.g., the Amazon Elastic Compute Cloud). With the rapid rise of these modern computing forms and their wide adoption of CMP, a good understanding to makespan minimization in multicore job co-scheduling becomes increasingly important. But to the best of our knowledge, this problem has remained unexplored.

Makespan minimization differs from cost minimization. The optimal schedules for the two criteria are typically different. In traditional job scheduling literature, the two criteria have led to drastically different algorithms and complexity analyses [12]. As to be shown in this paper, for multicore job co-scheduling, the implication of their difference is pronounced as well, ranging from complexity analysis to algorithm designs to the ultimate scheduling results (summarized in Section 7).

Motivated by the contrast of the increasing importance and the preliminary understanding of makespan minimization in multicore job co-scheduling, we initiate explorations on four aspects.

– First, we prove that makespan minimization in job co-scheduling is NP-complete on systems with more than 2 cores per chip. The proof is based on a reduction from the problem of Exact Cover by 3-Sets. We are not aware of any previous analysis of the computational complexity.
– Second, by offering an $O(n^{2.5} \cdot \log n)$ algorithm (n is the number of jobs), we prove that on dual-core systems with no job migrations, the problem is polynomial-time solvable. To the best of our knowledge, this algorithm is the first polynomial-time solution for this optimal co-scheduling problem.
– Third, we present a set of A*-search–based algorithms and a greedy algorithm to tackle optimal co-scheduling for makespan minimization in the general setting—with two or more cores per chip and with or without job migrations. A*-search has been applied for job co-scheduling [24], but not for makespan minimization. Our description focuses on the issues specific to makespan minimization, including the formulation of the search process, the design of the heuristic function, and the empirical exploration of the tradeoff between the scheduling overhead and quality.
– Finally, we evaluate the algorithms on both real and synthetic problems, verifying the optimality of the co-scheduling algorithms (under certain conditions), meanwhile showing that the algorithms may save orders of magnitude overhead over the brute-force search. Results of the approximation algorithms demonstrate their capability to achieve near optimal solutions with reasonable scalability.

The analysis and algorithms contributed in this paper help reveal (or approximate) the lower bound of makespan in multicore job co-scheduling, essential for the assessment of practical scheduling systems. The algorithms may shed insights to the development of effective lightweight co-scheduling systems as well.

We organize the rest of this paper as follows. Section 2 describes the problem setting and assumptions. Section 3 proves the NP-completeness of the optimal co-scheduling problem, and presents the polynomial-time algorithm and a set of A*-search algorithms as optimal solutions. Section 4 describes a set of approximation algorithms. Section 5 reports evaluation results. Section 6 discusses the limitations of this work and future extensions. After reviewing some related work in Section 7, we conclude the paper with a short summary in Section 8.

2 Problem Definition

Roughly speaking, the optimal job co-scheduling tackled in this work is to decide the placement of a set of jobs on a number of cores so that the makespan of the schedule is minimized.

Finding optimal co-schedules in a general setting is extremely difficult: A program's fine-grained behaviors may change constantly, a program may migrate to any cores, and programs may start, terminate, or go through context switch at any time. It is necessary to define the problem settings first.

2.1 Problem Settings

To make the problem tractable and meanwhile keep the analysis useful, we specify the following settings. Some of these settings may differ from certain practical scenarios. However, as we will show (after presenting the settings), they do not prevent the use of the computed co-schedules from serving for its main goal: facilitating the evaluation of practical co-schedulers.

Machines. The computing system assumed in this exploration contains m uniform chips, and each chip has u uniform cores[1]. There is a certain amount of cache on each chip that is shared by the u cores on the chip. Only one job can run on a core at each time point. The execution speed of a job running on a chip depends on what jobs are placed on the same chip, but has negligible dependence on how the rest of the job set are placed on other chips. The architecture is a generalized form of CMP architectures on the market, such as IBM Power5 and the Intel Core2 family.

Jobs. The number and starting time of jobs are set to be as follows. The number of jobs (denoted as n) is equal to the number of cores, $n = m * u$. This setting is to help focus on the placement of jobs on cores. When $n < m*u$, the problem can be converted to the defined setting if we consider that there are $(m*u-n)$ extra dummy jobs that consume no resources. If $n > m * u$, the problem is more complex, requiring the consideration of temporal complexity (e.g. context switch) besides the spatial placement of jobs. The temporal complexity is out of the scope of this paper. But we note that this work will be still useful for that setting, as spatial placement still exists as a sub-problem in it.

All the jobs must start at the same time. This is a typical assumption in both traditional job scheduling [12] and recent job co-scheduling [10]. This setting may differ from the scenarios in real-time scheduling. However, recall that the main targeted scenario of makespan minimization is batch job processing, in which, all jobs typically arrive at the system at the same time.

Job Migrations. A job can migrate from one core to another, but the migration only happens when any of the jobs terminates. This setting comes from the following reason. As well known, keeping a process on a processor is good for locality. As a result, in practical systems like Linux, occurrences of job migrations are mostly triggered by load imbalance [1]. In our setting, as the number of jobs equals the total number of cores, load changes only when some job finishes. Therefore, allowing job migration only at those times does not cause large departure from real scenarios.

This work focuses on job co-scheduling inside a multicore machine, which is the primary component of the scheduling in any large multicore-based systems. So it assumes

[1] We use the term "cores" for simplicity of discussion. As shown in Section 5, the techniques can also be applied to thread scheduling in SMT systems.

that all processor chips are in the same machine and the migrations of a job among different chips have similar overhead. (With certain extensions, the developed algorithms may be applicable to clusters consisting of multiple nodes. The extensions are mainly on the consideration of the different overhead of migration within and across cluster nodes.)

Performance Data. As assumed in previous work [10], the following performance information is given: the time for a job to finish if it runs alone (i.e., no other jobs running on the chip), and the performance degradation (defined as the rate between the co-run time and the single-run time of the job) of the job when it co-runs with k ($0 < k < u$) other jobs in the job set. These performance data can be obtained through offline profiling runs or predictive models [2,4]. The overhead in gathering the data is not an issue for optimal co-scheduling: Finding optimal co-schedules is not for direct real-time scheduling, but for providing a reference for the evaluation of practical co-schedulers. For a given u, the overhead to gather the single run and co-run times is polynomial in the number of jobs. It is typically negligible compared to the overhead in brute-force search for optimal co-schedules, which is exponential in the number of jobs.

Because a program execution may vary constantly, the performance degradation of a program in a co-run may vary across intervals. In our setting, we use the average degradation through the entire co-run. A future enhancement is to combine with program phase analysis [17,18]. As previous studies do [10,19], we currently ignore phase changes to concentrate on co-scheduling itself.

In our setting, jobs may relate with one another, but all degradations are greater than 1. As co-runs are typically slower than single-runs because of cache and bus contention, this setting holds in most cases.

Short Discussion. The settings described in this section do not prevent the use of the optimal co-scheduling for evaluating practical schedulers. For example, the evaluation of a scheduler S on a machine with m chips and u cores per chip can proceed as follows. The developers first find $m * u$ applications that are typical for the target system. They start the applications at the same time on the machine with the scheduler S running to get the makespan, T. They then run the applications a number of times to obtain the single-run times and co-run degradations of those applications. After that, all the information needed by the problem setting is ready. By applying the optimal co-scheduling algorithms (to be described), they will get the minimum makespan, $\hat{T}$. The comparison between T and $\hat{T}$ will indicate the room for improvement of the scheduler S.

2.2 Problem Definition and Terminology

With the problem settings defined, the definition of the optimal co-scheduling problem to be tackled in this work is straightforward. It is to find a schedule that maps each job to a core under the settings defined in the previous subsection, so that the makespan of the schedule is minimized.

For the sake of clarity, we define several terms. The allowance of job migration suggests the opportunities for rescheduling the remaining jobs when some job finishes. In the following description, we call each scheduling or rescheduling point as *a scheduling stage*. So, if no job migrations are allowed, there is only one scheduling stage; when migrations are permitted, there are up to n scheduling stages. We use *an assignment* to refer to a group of u jobs that are to run on the same chip. *A sub-schedule* is a set of assignments that cover all the unfinished jobs and do not overlap with one another. *A schedule* is a set of all sub-schedules that are used from the start to the end of the executions of all jobs.

3 Complexities and Solutions of Makespan Minimization

In this section, we analyze the inherent complexity of the makespan minimization in job co-scheduling. We classify the problem instances into four cases: $u \geq 3$ with or without job migration allowed, or $u = 2$ with or without job migration allowed. Here, u is the number of cores per chip. We prove that the first two cases are NP-complete problems, but the fourth is polynomial solvable by a perfect-matching-based algorithm. The complexity of the third case is to be studied in the future. In addition, we present heuristic algorithms for all the four cases.

3.1 Complexity Analysis ($u \geq 3$, With or Without Job Migration)

When more than two cores share a cache on a chip ($u \geq 3$), the makespan minimization is an NP-complete problem. We prove this result by reducing a known NP-complete problem, *Exact Cover by 3-Sets* (X3C) [8], to our problem.

First, we formulate our co-scheduling problem as a decision problem. Given a system with m chips, each with $u \geq 3$ cores, there is a set J containing $n = m \cdot u$ jobs, which are to be scheduled on the cores. Consider all possible subsets of J with cardinality u, denoted by $J_1, \cdots, J_{\binom{n}{u}}$. For each J_i, which represents a group of u jobs that may be co-scheduled on the same chip, let w_i be the maximum co-run time of all the u jobs in J_i. The question in the decision problem is whether there are m disjoint subsets $J_{p_1}, \cdots, J_{p_m}$, where $p_1, \cdots, p_m \in \{1, \cdots, \binom{n}{u}\}$, to form a partition of J such that $\max_{i=1}^{m}\{w_{p_i}\} \leq B$ for any given bound B.

Note that the partition of J into m subsets of cardinality u is actually the construction of a schedule of n jobs on $m \cdot u$ cores and that $\max_{i=1}^{m}\{w_{p_i}\}$ is in fact the makespan of the schedule.

The problem is clearly in NP. We prove that it is NP-complete via a reduction from X3C, in which given a set X with $|X| = 3m$ and a set $C = \{C_i | C_i \subseteq X \text{ and } |C_i| = 3\}$, the question to ask is whether C contains an exact cover for X, i.e., m disjoint members of C, say $C_{p_1}, \cdots, C_{p_m}$, that makes a partition of C.

The reduction from X3C to our co-scheduling problem is straightforward. Given any instance of X3C, namely X and C, we define an instance for co-scheduling, where (1) $J = X$ with $n = 3m$ and $u = 3$, (2) for any $J_i \subseteq J$ with $|J_i| = 3$, if $J_i \in C$ then let $w_i = 1$, and if $J_i \notin C$ then let $w_i = 2$, and (3) $B = 1$.

The construction of the instance for co-scheduling can be done in $O(n^3)$ time. Furthermore, it is easy to show that C contains an exact cover for X if and only if there is a schedule of jobs in J to the $3m$ cores with a makespan no more than 1. Therefore, the co-scheduling problem with $u = 3$ is NP-complete.

The above proof holds regardless of whether job migration is allowed or not. This is because in both settings, finding a schedule with makespan no more than 1 is equivalent to finding an exact cover.

3.2 Polynomial-Time Solution ($u = 2$, No Job Migration)

We prove that, when $u = 2$ and no job migrations are allowed, the optimal co-schedules can be found in polynomial time. We describe an $O(n^{2.5} \cdot \log n)$ algorithm as follows.

The algorithm uses a fully-connected graph, namely a *co-run makespan graph*, to model the optimal co-scheduling problem. Each vertex represents a job; the weight on an edge is the longer running time of the two jobs (represented by the two vertices connected by the edge) when they co-run together.

Before describing the algorithm, we introduce the concept of a perfect matching. A *perfect matching* in a graph is a subset of edges that cover all vertices of the graph, but no two edges share a common vertex. We define the *bound* of a perfect matching as the largest weight of all the edges it covers. It is easy to see that the perfect matching of a co-run makespan graph with the minimum bound corresponds to a solution to the makespan minimization problem: Each edge corresponds to an assignment (i.e., co-run group) and the makespan equals to the bound of the perfect matching.

There are some algorithms for finding the minimum-weight perfect matching on a weighted graph [6,8]. However, they cannot apply to our problem directly because their objective functions are typically the sum of edge weights, rather than the maximum of edge weights in our problem.

We develop an algorithm to determine a minimum-bound perfect matching as shown in Figure 1. We first construct a sorted list containing all the edges of a co-run makespan graph in an ascending order of their weights; the edge with the smallest weight resides on the top of the list. We then use a binary search to determine the smallest top portion of the sorted edge list that contains a perfect matching (regardless of weights) covering all vertices. The binary search starts with the top half of the edge list and checks whether a perfect matching can be found in those edges. A negative answer would suggest that more edges are needed, so the algorithm would try the top three quarters of the edge list. A positive answer would suggest that a smaller portion of the list may be enough to contain a perfect matching, so the algorithm would try the top quarter of the edge list. This binary search continues until it finds the smallest top portion of the edge list that contains a perfect matching.

We claim that the resulted perfect matching is an *optimal* perfect matching on the original co-run makespan graph—that is, no perfect matchings on the original co-run makespan graph have bounds smaller than the bound of the resulted perfect matching. The proof is as follows.

Let M be the perfect matching produced by the algorithm, T be the makespan of the corresponding schedule, and S be the smallest top portion of the edge list that contains M. According to the algorithm, S is the smallest among all top portions that contains a perfect matching.

Assume that there is a perfect matching M' whose makespan T' is smaller than T. Let E' be the set of edges included in M'. Let S' be a set containing all the edges in the sorted edge list from the top to the heaviest edge in E'. Because the edge list is sorted in the ascending order of edge weights, $E' \subseteq S'$. So, S' contains a perfect matching. Because $T' < T$, the weights of all the edges in E' and thus in S' must be smaller than T. While T is the weight of some edge in S, hence $S' \subset S$. This contradicts with the assumption that S is the smallest top portion of the edge list that contains a perfect matching, thus the proof completes.

The time complexity of the perfect matching detection subroutine, *findPerfMatch(G)*, is $O(\sqrt{n} \cdot m)$ [8], where n and m are the numbers of vertices and edges in the graph. In the algorithm, the binary search process contains $O(\log n)$ invocations of perfect matching detection. The value of m can be no greater than n^2. The time complexity of the algorithm is $O(n^{2.5} \cdot \log n)$.

```
/* $V$: vertex set; $E$: edge set */
/* $S$: generated perfect matching */
$L$ ← sortEdges($E$);
lbound ← 1; ubound ← $|L|$;
$G.vertices$ ← $V$; $S$ ← $\emptyset$;
while (1) {
  curPos ← ⌊ (ubound+lbound)/2 ⌋;
  if (curPos == ubound) return $S$;
  $G$.edges ← $L$[1:curPos];
  $S$ ← findPerfMatch(G);
  if ($S \neq$ NULL)
   ubound ← curPos;
  else
   lbound ← curPos;}
```

Fig. 1. Algorithm for minimum-bound perfect matching

3.3 Search-Based Optimal Co-Scheduling

The polynomial-time algorithm described in the previous section works only for dual-core systems without job migrations. This section presents a search-based approach, which is applicable to larger systems and supports job migrations.

Background on A*-Search. A*-search, stemming from artificial intelligence, is designed for fast graph search. It is optimally efficient for any given heuristic function—that is, no other search-tree-based optimal algorithm is guaranteed to expand fewer nodes than A* search, for a given heuristic function [15]. Its completeness, optimality, and optimal efficiency lead to the adoption for the search of optimal schedules.

For a tree search, where the goal is to find a path from the root to an arbitrary leaf with the total cost minimized, A* search defines a function $f(v)$ to estimate the lowest cost of all the paths passing through the node v. A* search maintains a priority list, initially containing only the root node. At each step, A* search removes the top element—that is, the node with the highest priority—from the priority list, and expands that node. After the expansion, it computes the $f(v)$ values of all the newly generated nodes, and put them into the priority list. The priority is proportional to $1/f(v)$. This expansion process continues until the top of the list is a leaf node, indicating that no other nodes in the list need to be expanded any more as their lowest cost exceeds the cost of the path that is already discovered.

The definition of function $f(v)$ is the key for the solution's optimality and the algorithm's efficiency. There are two properties related to $f(v)$:

– A* search is optimal if $f(v)$ never overestimates the cost.
– The closer $f(v)$ is from the real lowest cost, the more effective A*-search is in pruning the search space.

Application to Minimize Makespan for Job Co-Scheduling. Although A*-search has been used for cost minimization problems [24], some substantial changes are necessary for applying it to makespan minimization. Specifically, we need redefine the structure of the search tree and the cost estimation function $f(v)$. This section presents our respective definitions for the scenarios with and without job migrations.

No Job Migrations. When no job migrations are allowed, the scheduling problem is essentially to partition jobs into a number of co-run groups. Figure 2 illustrates our definition of the search tree. Each non-root tree node (say v) corresponds to a set, $S(v)$, that contains u distinct jobs. The nodes in the tree are arranged as follows. Let $R(v)$ represent the set of jobs that have never been covered by any node on the path from root to v. Suppose w is a child node of v. All jobs in $S(w)$ must belong to $R(v)$ (i.e., $S(w) \subseteq R(v)$) and $S(w)$ must contain the job whose index[2] is the smallest in $R(v)$. With such an organization, each path from the root to a leaf offers a schedule. All the paths in the tree together constitute the schedule space.

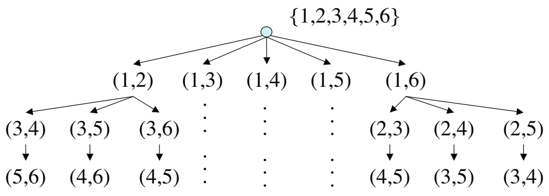


Fig. 2. An example of the search tree for the cases with no job migrations. There are 6 jobs to be scheduled to 3 dual-core chips. Each non-root tree node corresponds to a set of u (here $u = 2$) distinct jobs; the set must contain the job whose index is the smallest among all the jobs that are not covered from the root to this node. Each path from the root to a leaf therefore offers a schedule.

We define the cost estimation function $f(v)$ as follows. Let A represent the set of all n jobs, and P' be the path from the root to the node v. It is easy to see that the minimum makespan of any schedule (or path) passing node v must be either the makespan of the jobs $A - R(v)$ (i.e., the jobs covered by the path from the root to the node v) or the

[2] We assume that each job has a unique index number.

minimum makespan of the remaining jobs, $R(v)$. The former can be computed from the assignments represented by P'. The latter can be no smaller than the maximum of the minimal co-run times of the jobs in $R(v)$, which can be computed from the given co-run degradations. We then define $f(v)$ as the maximum of the two values.

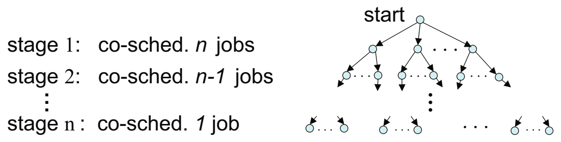

start
stage 1: co-sched. n jobs
stage 2: co-sched. n-1 jobs
stage n : co-sched. 1 job

Fig. 3. Search tree for the cases with rescheduling allowed at the end of a job. Each level corresponds to a scheduling stage. Each node, except the root, represents a sub-schedule of the jobs that have not finished yet.

With Job Migrations. Similar to the previous work [11], we use a search tree to model the co-scheduling problem when job migrations are allowed, as illustrated by Figure 3. It differs from Figure 2 in that each non-root node represents a sub-schedule (i.e., a set of assignments that cover all the unfinished jobs and have no overlap with each other) rather than a group of co-running jobs.

For n jobs, there are at most n scheduling stages; each corresponds to a time point when one job finishes since the previous stage. The nodes at a stage, say stage i, correspond to all possible sub-schedules for the $n - i + 1$ remaining jobs. There is a cost associated with each edge. Consider an edge from node a to node b. The number of unfinished jobs in b is typically one less than in a. The weight on the edge is the time for that job to finish since the scheduling stage of a.

The makespan minimization becomes to find a path from the root to any leaf node so that the sum of the weights on the path is the minimum. We define $f(v)$ as the sum of two quantities. One is the total weights from the root to the node v, the other is the longest single-run time of the remaining jobs.

Given the NP-completeness of the problem, it is not surprising that A*-search is subject to scalability issues. Our explorations aim at revealing the extent of its scalability, and shedding insights for the design of approximation algorithms.

4 Approximation Algorithms

To achieve good scalability, we develop three approximation algorithms based on the enlightenment from the optimal co-scheduling algorithms presented in the previous section. The first two algorithms are applicable generally, while the third one applies only to dual-core cases.

4.1 Combination with Clustering

The combination of A*-search and clustering may provide further flexibility for striking a tradeoff between overhead and quality of co-scheduling. We call such combined algorithms “A*-cluster” algorithms. We first describe its application to the cases with job migrations, and then outline its uses when migrations are not allowed.

At each (re)scheduling time, the unfinished jobs are clustered based on the length of their remaining portions. The algorithm controls the number of rescheduling stages by rescheduling only when a cluster of jobs finish. It also avoids the generation of sub-schedules that are similar to one another.

We say that a sub-schedule is substantially different from another one if they are not equivalent when we regard all jobs in a cluster as the same. For example, four jobs fall into two clusters as $\{\{1\ 2\}, \{3\ 4\}\}$. We regard the sub-schedule (1 3) (2 4) equivalent to (1 4) (2 3), but different from (1 2) (3 4). (Each pair of parentheses contains a co-run group.)

Among various clustering techniques, we use a simple distance-based clustering approach as the data are one dimensional. Given a sequence of data to be clustered, we first sort them in an ascending order. Then, we compute the differences between every two adjacent data items in the sorted sequence. Large differences are considered as indication of cluster boundaries. A difference is regarded as large if its value is greater than $d + \delta$, where d is the mean value of the differences in the sequence and δ is the standard deviation of the differences.

The A*-cluster algorithm reduces both the height and the width of the search tree. The number of children of a node is reduced from factorial, $\prod_{i=0}^{\frac{r}{u}-1} \binom{r-i*u-1}{u-1}$, to polynomial, $O(r^{\gamma})$, where C is the number of clusters and r is the number of unfinished jobs, and $(\gamma = C + (C^u - C)/u!)$.

For the cases without job migrations, the jobs are clustered based on their single-run time. The algorithm prunes the width of the search tree by removing the nodes that are not significantly different from their siblings, similar to the pruning in the case with migrations. The tree height remains unchanged.

4.2 Greedy Algorithm

The greedy algorithm is simpler than the A*-cluster algorithm. At each (re)scheduling stage, the algorithm iteratively determines the assignments for the remaining jobs. In each iteration, it finds the best co-run group for the job whose remaining part has the longest single-run time among all the unfinished jobs. Its intuition is that the longest jobs typically determine the makespan of a schedule.

4.3 Local Perfect-Matching Algorithm

This approximation algorithm is a generalized version of the perfect-matching-based algorithm proposed in Section 3.2. The extension makes it applicable to dual-core cases with job migrations. At each (re)scheduling point, it applies the perfect-matching-based algorithm to the unfinished jobs to obtain a locally optimal sub-schedule. As the perfect-matching-based algorithm assumes that the number of remaining jobs equals the number of cores in the computing system, we treat the jobs that have finished as pseudo-jobs, which exist but consume no computing resource. This strategy may introduce certain amount of redundant work, but it offers an easy way to generalize the perfect-matching-based algorithm. The time complexity of this algorithm is $O(n^{3.5} \cdot \log n)$.

There is a side note on the scheduling algorithms that allow migrations. A migration of different programs may have different overhead. However, because in our setting, migrations happen only when some job finishes, the total number of job migrations is small (less than the number of jobs); the total overhead covers only a negligible portion of the makespan (confirmed in Section 6). In our experiments, we use the average overhead measured on the experimental platform as the overhead of a migration when comparing the makespan of different schedules.

Table 1. Benchmarks

Benchmark	single-run time (s)	co-run degrad rate min %	max %	mean %
fmm*	5.63	0.77	11.28	3.67
ocean*	13.52	2.13	58.81	19.73
ammp	21.10	1.66	30.24	12.62
art	2.22	2.31	75.42	27.78
bzip	10.90	0.00	38.95	3.31
crafty	6.75	0.07	12.33	4.95
equake	11.05	6.42	78.00	26.46
gap	2.90	2.09	34.34	11.02
gzip	14.10	0.00	13.06	2.19
mcf	7.86	8.23	125.36	42.37
mesa	15.33	0.65	15.15	5.18
parser	3.74	1.74	37.75	13.51
twolf	5.42	0.00	15.73	5.21
vpr	4.58	3.31	42.52	18.30

$*$: from SPLASH-2. Others from SPEC CPU2000.

5 Evaluation

We evaluate the co-scheduling algorithms on two kinds of architecture. For CMP co-scheduling, the machines are equipped with quad-core Intel Xeon 5150 processors running at 2.66 GHz. Each chip has two 4MB L2 cache, each shared by two cores. Every core has a 32KB dedicated L1 data cache. For Simultaneous Multithreading (SMT) co-scheduling, the machines contain Intel Xeon 5080 processors (two 2MB L2 cache per chip) clocked at 3.73 GHz with Hyper-Threading enabled (two hyperthreads per computing unit.)

The job suite includes 14 programs: 2 parallel programs from SPLASH-2 [21] and 12 sequential programs randomly selected from SPEC CPU2000. Each of the two parallel program has two threads; so, we have 16 jobs in total. We do not use the programs from the entire benchmark suites because the large problem size would make it infeasible to compare the scheduling algorithms, especially with the brute-force search algorithm. We use the two parallel programs to examine the applicability of the co-scheduling algorithms for parallel (in addition to sequential) applications. Table 1 lists the programs with the ranges of their co-run degradations on the Intel Xeon 5150 processors. The big ranges suggest the potential for co-scheduling.

In addition, we generate some sets of jobs whose single-run time and co-run degradations are set randomly. The use of these synthetic problems helps overcome the limitations imposed by the particular benchmark set.

For each set of jobs, we test the scheduling in cases both with and without job migrations (denoted as "no rescheduling" and "rescheduling" respectively.) The difference reflects the benefits of rescheduling.

5.1 Comparison to the Optimal

This section concentrates on the verification of the optimality of the A*-search and perfect-matching-based algorithms. We stress that the optimality is under the settings defined in Section 2. We compare the results of those scheduling algorithms with the best schedules found through brute-force search.

Because of the scalability issue of the brute-force search, we use 8 jobs for the comparison. We use the top 6 programs (8 jobs considering the parallel threads) in Table 1, along with a number of synthetic job sets. Table 2 reports the results. For each synthetic setting, we generate 3 problems in that setting, referred to as the 3 trials in the table.

The data surrounded by boxes are from the optimal co-scheduling algorithms (the rest are from the approximation algorithms.) The two halves of the "matching" row correspond to the precise algorithm in Section 3.2 and the approximation algorithm in Section 4.3, respectively. Both algorithms are applicable only to 2-core cases.

Optimality. The data in Table 2 show that the optimal algorithms generate schedules with the same makespans as the schedules found by the brute-force search. For the 8 real jobs on Xeon 5150 (2-cmp), for instance, the optimal schedule found by the 3 algorithms are all as follows: (fmm-1,ocean-1), (ammp,cafty), (art,bzip), (fmm-2,ocean-2), where fmm-n and ocean-n are their nth threads, and each pair of parentheses include a co-running group. The makespan is 0.5% larger than the makespan when the programs run in isolation.

The bottom 3 rows in Table 2 reveal the minimum, median, and maximum of the makespans of 100 randomly generated schedules, corresponding to the scheduling in many existing systems, which work in a cache-sharing-oblivious manner. The minimum makespans are close to the optimal in the "no rescheduling" cases, but are mostly over 10% larger than the optimal in the "rescheduling" cases. The median and maximum are significantly larger than the optimal. For the 8 real jobs, although random scheduling is likely to produce near optimal makespan in the Xeon 5150 system, it causes over 20% makespan increase on the SMT systems. These results indicate the risks of neglecting cache sharing in job scheduling.

The comparison between the "no rescheduling" and "rescheduling" results shows that when the "no rescheduling" algorithms cause non-negligible makespan increase, rescheduling is usually able to reduce the makespan considerably.

Overhead. Table 3 reports the numbers of search-tree nodes visited and the milliseconds spent by different co-scheduling algorithms on 8 jobs on Intel Xeon 5080. In the "rescheduling" case, relative to the brute-force search, both of the two optimal co-scheduling algorithms save the search time by several orders of magnitude.

Table 2. Co-schedule makespan on 8 real jobs and a series of synthetic scheduling problems (each has 8 jobs). The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation. The real jobs run on two architectures: Intel Xeon 5150 (2-cmp) and Intel Xeon 5080 (2-smt). The synthetic scheduling problems use both dual-core (2-core) and quad-core (4-core) systems.

	no rescheduling									rescheduling								
jobs	real		synthetic							real		synthetic						
arch.	2-cmp	2-smt	2-core			4-core				2-cmp	2-smt	2-core			4-core			
trial			1	2	3	1	2	3				1	2	3	1	2	3	
brute-fc	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65		1.002	1.013	1.33	1.21	1.19	1.99	1.93	1.56	
A*	1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65		1.002	1.013	1.33	1.21	1.19	1.99	1.93	1.56	
matching	1.005	1.023	1.49	1.49	1.58	-	-	-		1.002	1.023	1.37	1.43	1.52	-	-	-	
A*-clstr	1.005	1.167	1.55	1.75	1.58	2.38	2.30	1.65		1.012	1.023	1.55	1.48	1.29	2.19	2.12	1.63	
greedy	1.005	1.170	1.49	1.90	1.80	2.77	2.34	1.85		1.005	1.170	1.43	1.90	1.80	2.32	2.08	1.87	
rand-min	1.005	1.023	1.55	1.49	1.69	2.24	2.16	1.65		1.005	1.023	1.49	1.49	1.58	2.11	2.16	1.65	
rand-med	1.016	1.255	1.81	2.70	2.22	2.55	2.34	1.88		1.016	1.196	1.81	2.70	1.92	2.54	2.33	1.87	
rand-max	1.161	1.329	2.72	3.30	2.66	3.13	2.91	2.68		1.161	1.329	2.72	3.30	2.66	3.13	2.91	2.68	

Table 3. The numbers of nodes visited and the time spent by different co-scheduling algorithms on 8 jobs on Intel Xeon 5080

	no resch.		resch.	
	nodes	time (ms)	nodes	time (ms)
brute-fc	210	41	16643446	419332
matching	1	47	4	179
A*	37	23	4405	718
A*-clstr	8	5	32	35
greedy	1	2	4	6
random	-	1	-	1

cost minimization:
schedule: (fmm-1, crafty), (fmm-2, ocean-1), (occean-2, art), (ammp, bzip)
cost (ie., total degradation): 12.13
makespan: 58.02 sec

makespan minimization:
schedule: (fmm-1, bzip), (fmm-2, art), (ocean-1, ammp), (ocean-2, crafty)
cost (ie., total degradation): 12.88
makespan: 43.56 sec

Fig. 4. Optimal schedules for cost minimization and makespan minimization on Xeon 5080 (2-smt) with no rescheduling

Comparison with Cost Minimization. As mentioned earlier in this paper, the two scheduling criteria, makespan and total cost, typically lead to different results. It is confirmed by the experimental results. For example, Figure 4 shows the optimal schedules (without rescheduling) for both criteria on the Xeon 5080 (2-smt) machine. The schedule with minimum total cost turns out to have 33% larger makespan than the schedule from the makespan minimization algorithms. On the other hand, the schedule with minimum makespan causes extra cost as well. This difference confirms the need for studies on each of the criteria and the application of the corresponding algorithms in different scenarios.

Table 4. Co-schedule makespan on 16 real jobs and a series of synthetic scheduling problems (each has 16 jobs). The numbers in the table are the makespan achieved with the respective schedule, relative to the makespan when each job runs in isolation. The real jobs run on two architectures: Intel Xeon 5150 (2-cmp) and Intel Xeon 5080 (2-smt). The synthetic scheduling problems use both dual-core (2-core) and quad-core (4-core) systems.

	no rescheduling								rescheduling							
jobs	real		synthetic						real		synthetic					
arch.	2-cmp	2-smt	2-core			4-core			2-cmp	2-smt	2-core			4-core		
trial			1	2	3	1	2	3			1	2	3	1	2	3
matching	1.005	1.033	1.26	1.14	1.2	-	-	-	1.002	1.033	1.2	1.07	1.11	-	-	-
A*-clstr	1.005	1.059	1.42	1.22	1.21	1.97	1.87	1.93	1.005	1.107	1.37	1.20	1.22	1.99	1.86	1.91
greedy	1.005	1.158	1.92	1.32	1.37	2.35	2.97	2.42	1.005	1.158	1.92	1.32	1.35	2.00	1.95	1.95
rand-min	1.005	1.062	1.70	1.48	1.38	2.11	1.92	2.05	1.005	1.056	1.58	1.32	1.38	2.08	1.95	2.00
rand-med	1.029	1.197	2.19	2.03	2.17	2.46	2.49	2.42	1.016	1.197	2.19	1.96	2.17	2.45	2.39	2.37
rand-max	1.161	1.468	3.48	2.92	2.75	3.10	3.23	2.99	1.161	1.468	3.48	2.92	2.75	3.03	3.46	2.86

5.2 Approximation Algorithms

Besides reporting the optimal co-scheduling results, Table 2 also lists the performance of the approximated schedules (outside the boxes.) On real jobs, the matching-based approximation produces near optimal results, the A*-cluster algorithm works similarly well except in the case of "no rescheduling" on "2-smt" architecture where the makespan is about 14% larger than the minimum. Because of the imprecision caused by clustering, both approximation algorithms significantly outperform the greedy and random scheduling in most real and synthetic cases. On the other hand, their distances from the optimal reflect the room for improvement.

Table 4 presents the results on 16 jobs. It does not include the brute-force and A* results because the former takes too much time (up to years with job migrations) to finish and the latter requires too much memory to run. The results of the approximation algorithms are consistent with the 8-job results. Although the minimum makespans from the random schedules occasionally get close to the results of the approximation algorithms, most random scheduling results are significantly worse than the matching-based and A*-cluster-based approximations. The greedy algorithm, although performing not as well as the other two approximation algorithms, outperforms the median results from random scheduling considerably.

Tables 3 and 5 report that the approximation algorithms take less than one second, in contrast to the up to years of time the brute-force search needs.

Table 5. The numbers of nodes visited and the time spent by different co-scheduling algorithms on 16 jobs on Intel Xeon 5080

	no resch.		resch.	
	nodes	time (ms)	nodes	time (ms)
matching	1	86	8	889
A*-clstr	76	39	122	94
greedy	1	3	8	9
random	-	1	-	2

Scalability. Figure 5 shows the scheduling overhead of the three approximation algorithms on a spectrum of problem sizes (with migrations and $u = 2$.) The greedy algorithm shows the best efficiency for its simplicity; the local matching-based algorithm shows less scheduling time than the A*-cluster algorithm.

Short Summary
We summarize the experimental results as follows:

1. The experiments empirically verify the optimality of the scheduling results of the perfect matching and the A*-search algorithm. The two algorithms are orders of magnitude more efficient than the brute-force search. They are applicable when the size of the problem is not large.
2. The local matching-based approximation algorithm is preferable when $u = 2$ (with or without job migrations) for its high scheduling quality and little scheduling time.
3. When $u > 2$, the A*-cluster algorithm offers a reasonable solution that produces good schedules in a moderate amount of time.
4. When scheduling overhead is the main concern, the greedy algorithm may be favorable, which uses slightly more time than random scheduling but offers consistently better results.

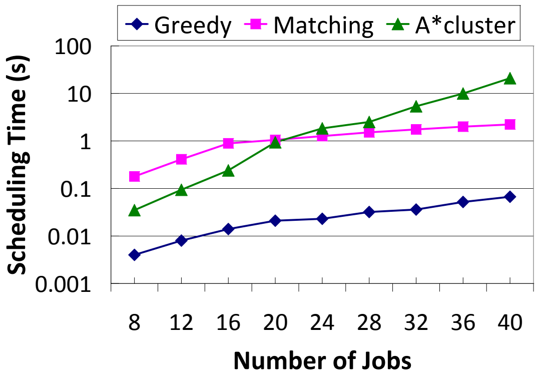


Fig. 5. Scalability of approximation algorithms

6 Discussion

This section discusses some limitations of this work and the influence on practical uses.

The requirement of all co-run degradations may seem to be an obstacle preventing the direct uses of the proposed algorithms in practical co-scheduling systems. However, that requirement does not impair the main goals of this work.

This work is a limit study. The primary goal is to offer feasible ways to uncover the optimal solutions in job co-scheduling, rather than to develop another heuristics-based runtime co-scheduler. Besides offering theoretical insights into co-scheduling, this work enables better evaluation of co-scheduling systems than before, offering the facility for efficiently revealing the potential of a practical co-scheduler in a general setting, which has been infeasible in the past for even small problems.

Furthermore, the algorithms proposed in this work may provide the insights for the development of more effective online scheduling algorithms both in operating systems and during the runtime of parallel applications. There has been some advancement in predicting co-run performance from program single runs (e.g., [2, 4]). The research in locality analysis has continuously enhanced the efficiency and accuracy in locality characterization [3, 16]. These studies make it possible to obtain co-run performance through lightweight prediction, hence offering the opportunity for the integration of the proposed scheduling algorithms in runtime scheduling systems.

In our experiments, we conduct a measurement of the migration overhead in the experimental machines by leveraging the system call, "sched_setaffinity". The system call binds processes to cores. By invoking the call at some random locations in an application with different parameters, we can migrate the process among chips in a machine. (Migrations among machines are typically too expensive to support.) The results show that a migration may cause 0.1–1.1% changes to the execution times of the benchmarks used in our experiments, depending on the length of the original execution. Recall that in our experiments, we use the average value of migration overhead as the cost of a migration. The introduced inaccuracy is hence less than 1.1% of the computed minimum makespan.

7 Related Work

To the best of our knowledge, this work is the first systematic study on finding the *optimal* schedules that minimize the *makespan* of jobs running on CMP systems.

7.1 Comparison with Cost Minimization

As mentioned earlier, there have been some studies in optimal co-scheduling for minimum cost [10, 11, 24]. We have mentioned the connections and differences between that co-scheduling problem and our makespan minimization problem throughout this paper. We here give a summary.

Connections. The two problems do have some connections, mainly in two aspects. First, they have similar dimensions to explore: dual-core or more than two cores per chip, migration allowed or not. Second, they both model the dual-core problems with a fully connected graph, and model the general cases as a search problem and derive solutions based on A*-search.

Differences. However, these two problems are fundamentally different. Their different co-scheduling goals determine that important differences exist in almost every aspect of the explorations to the two problems.

– *Complexity Analysis.* Despite that the two problems are both proved to be NP-complete in a general setting, their proofs are substantially different. The previous work [10, 11] analyzes the computation complexity of cost minimization by formulating the problem as an Multidimensional Assignment problem (MAP). But for makespan minimization, the MAP formulation cannot be applied because of the mismatch of the objectives. We have to analyze and formulate the problem in a different way, proving the NP-Completeness through the reduction from the problem of Exact Cover by 3-Sets.
– *Algorithms on Dual-Core Systems without Migrations.* For algorithm design, the classic Blossom [6] algorithm can be directly used for finding the optimal schedule for dual-core cases for cost minimization [10, 11]. But it cannot be applied to makespan minimization because the algorithm aims to minimizing the total weights of a perfect matching on a graph, rather than the largest weight as what makespan minimization requires. The solution introduced in this work (Section 3.2) turns out to be even more efficient than the Blossom algorithm, with complexity of $O(n^{2.5} \cdot \log n)$ versus $O(n^4)$.
– *Algorithms in Other Settings.* A* is a classical search algorithm widely used in many areas. We do not claim the use of it for job co-scheduling in the general settings as a contribution of this work. In fact, previous work [24] has used it for approximating optimal schedules for cost minimization. However, the key in applying A* is in the formulation of the search problem and the design of the approximation functions $f(v)$ used in every search-tree node. Both are specific to the problem to be addressed. They are where the extensions are made by the approximation algorithms in this work. In addition, the empirical exploration of the tradeoff between the approximation efficiency and the quality of resulting schedules is also specific to the makespan minimization problem.
– *Scheduling Results.* As Section 5.1 shows, the optimal schedules for the two problems typically differ from each other, confirming the need for studies on each of them.

Overall, the previous work on cost minimization [10, 11, 24] has given some insights to this work. But because of the different goals of the two problems, this systematic exploration is imperative for achieving a good understanding of the makespan minimization problem.

7.2 Comparisons with Other Scheduling Work

Scheduling is a topic with a large body of relevant work. As summarized in the *Handbook of Scheduling* [12], previous studies on optimal job scheduling have covered 4 types of machine environments: *dedicated*, *identical parallel*, *uniform parallel*, and *unrelated parallel* machines. On all of them, the running time of a job is fixed on a machine, independent on how other jobs are assigned, a clear contrast to the performance interplay in the co-scheduling problem tackled in this current work.

Traditional Symmetric Multiprocessing (SMP) systems or NUMA platforms have certain off-chip resource sharing (e.g., on the main memory), but the influence of the sharing on program performance has been inconsiderable for scheduling and has not been the primary concern in previous scheduling studies. Some scheduling work [12] has considered dependences among jobs. But the dependences differ from the performance interplay in co-scheduling in that the dependences constrains the order rather than performance of the execution of the jobs.

The hierarchical scheduling algorithm [5] in traditional job scheduling also uses a tree-like hierarchy for job scheduling. However, it is about how to move tasks among queues along a path to feed an idle processor, considering no performance influence caused by co-running jobs.

7.3 Other Work on Shared Cache Management

Due to the importance of shared cache, recent years have seen a large number of relevant studies. Some of them try to construct practical on-line job scheduling systems. They employ different program features, including estimated cache miss ratios, hardware performance counters, and so on [7, 19, 20]. Architecture designs for alleviating cache contention have focused on cache partitioning [13], cache quota management [14], and so forth. But none of them has focused on the optimal co-scheduling to minimize makespan. In addition, some studies [2, 4] have proposed statistical models for the prediction of co-run performance. The models may ease the process for getting the data needed for optimal scheduling.

8 Conclusion

As the processor-level parallelism increases, the urgency for alleviating the resource contention among co-running jobs is continuously growing. This work concentrates on the theoretical analysis and the design of optimal co-scheduling algorithms for minimizing the makespan of co-running jobs. It proves the computational complexity of the problem, proposes an $O(n^{2.5} \cdot \log n)$ algorithm and A*-search-based algorithms to solve the makespan minimization problem, and empirically verifies the optimality of the algorithms and examines the effectiveness and scalability of several approximation algorithms. The analysis and algorithms contributed in this paper may complement previous explorations by both revealing the lower bound for evaluation, and offering insights in the development of lightweight co-scheduling systems.

Acknowledgment. We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation Grant (0811791), NSF CAREER and DOE Early Career Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

1. The linux kernel archives, http://www.kernel.org
2. Berg, E., Zeffer, H., Hagersten, E.: A statistical multiprocessor cache model. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (2006)
3. Cascaval, G.C.: Compile-time Performance Prediction of Scientific Programs. PhD thesis, University of Illinois at Urbana-Champaign (2000)
4. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multi-processor architecture. In: HPCA, pp. 340–351 (2005)
5. Dandamudi, S.: Hierarchical Scheduling in Parallel and Cluster Systems. Kluwer (2003)
6. Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards B 69B, 125–130 (1965)
7. Fedorova, A., Seltzer, M., Smith, M.D.: Improving performance isolation on chip multiprocessors via an operating system scheduler. In: PACT, pp. 25–38 (2007)
8. Hochbaum, D.S.: Approximation Algorithms for NP-Hard Problems. PWS Publishing Company (1995)
9. Hsu, L.R., Reinhardt, S.K., Lyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: PACT (2006)
10. Jiang, Y., Shen, X., Chen, J., Tripathi, R.: Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: PACT, pp. 220–229 (October 2008)
11. Jiang, Y., Shen, X., Chen, J., Tripathi, R.: The complexity and approximation of optimal job co-scheduling on chip multiprocessors. IEEE Transactions on Parallel and Distributed Systems 22(7) (2011), doi: 10.1109/TPDS.2010.193
12. Leung, J.Y.-T.: Handbook of Scheduling. Chapman & Hall / CRC (2004)
13. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. Micro, 423–432 (2006)
14. Rafique, N., Lim, W., Thottethodi, M.: Architectural support for operating system-driven CMP cache management. In: PACT, pp. 2–12 (2006)
15. Russell, S., Norvig, P.: Artificial Intelligence. Prentice Hall (2002)
16. Shen, X., Shaw, J., Meeker, B., Ding, C.: Locality approximation using time. In: POPL (2007)
17. Shen, X., Zhong, Y., Ding, C.: Locality phase prediction. In: ASPLOS, pp. 165–176 (2004)
18. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS, pp. 45–57 (2002)
19. Snavely, A., Tullsen, D.: Symbiotic jobscheduling for a simultaneous multithreading processor. In: ASPLOS, pp. 66–76 (2000)
20. Snavely, A., Tullsen, D., Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In: SIGMETRICS, pp. 66–76 (2002)
21. SPLASH. Stanford parallel applications for shared memory (SPLASH) benchmark, http://www-flash.stanford.edu/SPLASH/
22. Suh, G., Devadas, S., Rudolph, L.: A new memory monitoring scheme for memory-aware scheduling and partitioning. In: HPCA, pp. 117–128 (2002)
23. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. SIGOPS Oper. Syst. Rev. 41(3), 47–58 (2007)
24. Tian, K., Jiang, Y., Shen, X.: A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In: Proceedings of ACM Computing Frontiers, pp. 41–50 (2009)

Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters

Yiannis Georgiou[1] and Matthieu Hautreux[2]

[1] BULL S.A.S
Yiannis.Georgiou@bull.fr
[2] CEA-DAM
Matthieu.Hautreux@cea.fr

Abstract. The Resource and Job Management System (RJMS) is the middleware in charge of delivering computing power to applications in HPC systems. The increasing number of computational resources in modern supercomputers brings new levels of parallelism and complexity. To maximize the global throughput while ensuring good efficiency of applications, RJMS must deal with issues like manageability, scalability and network topology awareness. This paper is focused on the evaluation of the so-called RJMS SLURM regarding these issues. It presents studies performed in order to evaluate, adapt and prepare the configuration of the RJMS to efficiently manage two Bull petaflop supercomputers installed at CEA, Tera-100 and Curie. The studies evaluate the capability of SLURM to manage large numbers of compute resources and jobs as well as to provide an optimal placement of jobs on clusters using a tree interconnect topology. Experiments presented in this paper are conducted using both real-scale and emulated supercomputers using synthetic workloads. The synthetic workloads are derived from the ESP benchmark and adapted to the evaluation of the RJMS internals. Emulations of larger supercomputers are performed to assess the scalability and the direct eligibility of SLURM to manage larger systems.

1 Introduction

The advent of multicore architectures and the evolution of multi-level/multi-topology interconnect networks has introduced new complexities in the architecture as well as extra levels of hierarchies. The continuous growth of cluster's sizes and computing power still follows Moore's law and we witness the deployment of larger petascale supercomputing clusters [1]. Furthermore, the continuous increasing needs for computing power by applications along with their parallel intensive nature (MPI, OpenMP, hybrid,...) made them more sensitive to communication efficiency, demanding an optimal placement upon the network and certain quality of services.

The work of a Resource and Job Management System (RJMS) is to distribute computing power to user jobs within a parallel computing infrastructure. Its goal is to satisfy users demands for computation and achieve a good performance in overall system's utilization by efficiently assigning jobs to resources. This assignment involves three principal abstraction layers: the declaration of a job where the demand of resources and job characteristics take place, the scheduling of the jobs upon the resources given their organization and the launching of job instances upon the computation resources along with the job's control of execution.

The efficient assignment of large number of resources to an evenly large number of users jobs arises issues like job launchers and scheduling scalability. The increase of network diameter and the network contention problem that can be observed in such large network sharing scenarios demand a certain consideration so as to favor the placement of jobs upon groups of nodes which could provide optimal communication characteristics. Since the RJMS has a constant knowledge of both workloads and computational resources, it is responsible to provide techniques for the efficient placement of jobs upon the network.

BULL and CEA-DAM have been collaborating for the design, construction or installation of 2 petascale cluster systems deployed in Europe. The first one, "Tera100"[1], with a theoretical computing power of 1.25 petaflops, is in production since November 2010 and represents Europe's most powerful system and 9Th more powerful in the world, according to November's 2011 top500 list [1]. The second, the French PRACE Tier0 system, "Curie"[2], with a theoretical computing power of 1.6 petaflops is currently under deployment and planned to be in production on March 2012. The Resource and Job Management System installed on both systems is SLURM [2] which is an open-source RJMS specifically designed for the scalability requirements of state-of-the-art supercomputers.

The goal of this paper is to evaluate SLURM's scalability and jobs placement efficiency in terms of network topology upon large HPC clusters. This study was initially motivated by the need to confirm that SLURM could guarantee the efficient assignment of resources upon user jobs, under various realistic scenarios, before the real deployment of the above petascale systems. We wanted to reveal possible weaknesses of the software before we come up with them into real life. Once the desired performance goals were attained, the studies continued beyond the existing scales to foresee the efficiency of SLURM on even larger clusters.

Inevitably the research on those systems implicate various procedures and internal mechanisms making their behavior complicated and difficult to model and study. Every different mechanism depends on a large number of parameters that may present interdependencies. Thus, it is important to be able to study the system as a whole under real life conditions. Even if simulation can provide important initial insights, the need for real-scale experimentation seems necessary for the study and evaluation of all internal functions as one complete system. On the other hand, real production systems are continuously overbooked for scientific applications execution and are not easily available for this type of experiments in full scale. Hence, emulation seems to be the best solution for large-scale experimentations.

In this paper we present a real-scale and emulated scale experimental methodology based upon controlled submission of synthetic workloads. The synthetic workloads are derived from the ESP model [3,4] which is a known benchmark used for the evaluation

[1] `http://www.hpcwire.com/hpcwire/2010-05-27/tera_100_europes_most_powerful_supercomputer_powers_up.html`

[2] `http://www.prace-ri.eu/CURIE-Grand-Opening-on-March-1st-2012?lang=en`

of launching and scheduling parameters of Resource and Job Management Systems. In this study the default ESP workload was modified in order to adapt to the characteristics of real large cluster usage. Hence, the experimental methodology makes use of two derived variations of ESP which are introduced in this paper: the Light-ESP and the Parallel Light-ESP. The experiments took place upon a subset of Tera-100 cluster during maintenance periods and upon an internal BULL cluster dedicated for research and development.

The remainder of this article is presented as follows: The next section provides the Background and Related Work upon RJMS and performance evaluation for these type of systems. Section 3 describes the experimental methodology that has been adopted and used for the evaluation of the RJMS. Section 4 provides the main part of this study where we present and discuss our evaluation results upon the scalability and efficiency of SLURM RJMS. Finally the last section presents the conclusions along with current work and perspectives.

2 Background and Related Work

Since the beginning of the first Resource and Job Management Systems back in the 80s, different software packages have been proposed in the area to serve the needs of the first HPC Clusters. Nowadays, various software exist either as evolutions of some older software (like PBSPro or LSF) or with new designs (like OAR and SLURM). Commercial systems like LSF [5], LoadLeveler [6], PBSPro [7] and Moab [8] generally support a large number of architecture platforms and operating systems, provide highly developed graphic interface for visualization, monitoring and transparency of usage along with a good support for interfacing standards like parallel libraries, Grids and Clouds. On the other hand their open-source alternatives like Condor [9], OAR [10], SLURM [2], GridEngine [11], Torque [12] and Maui [13] provide more innovation and a certain flexibility when compared to the commercial solutions.

2.1 Evaluating the Internals of Resource and Job Management Systems

Numerous studies have been made to evaluate and compare different Resource and Job Management Systems [14], [15].

One of the first studies of performance evaluations for RJMS was presented in [16]. In this study Tarek et al. have constructed a suite of tests consisted by a set of 36 benchmarks belonging to known classes of benchmarks (NSA HPC, NAS Parallel, UPC and Cryptographic). They have divided the benchmarks into four sets comprising short/medium/long and I/O job lists and have measured average throughput, average turn-around time, average response time and utilization for 4 known Resource and Job Management Systems: LSF, Codine, PBS and Condor.

Another empirical study [10] measured throughput with submission of large number of jobs and efficiency of the launcher and scheduler by using a specific ESP benchmark. The study compared the performance of 4 known RJMS: OAR, Torque, Torque+Maui and SGE. ESP benchmark has similarities with the previously described suite of tests [16]. However one of its advantages is that it can result in a specific metric that reflects the performance of the RJMS under the specific workload and the selected scheduling parameters. The approach used in this work is an extension of the methodology used for the performance evaluation in [10], since we propose a similar method using variations of the ESP benchmark workload model.

Performance evaluation is effectuated by having the system's scheduler schedule a sequence of jobs. This sequence is the actual workload that will be injected to the system. Different researches [17], [18], [19], [20] has been effectuated upon workload characterization and modeling of parallel computing systems. In order to model and log a workload of a parallel system, Chapin et al. [19] have defined the *standard workload format* (swf) which provides a standardized format for describing an execution of a sequence of jobs. In the internals of our experimentation methodology we are based on the swf format for the workload collection mechanisms and make use of particularly developed tools for swf workload treatment and graphic representations.

Based on previous work [20], [21] it seems that to give precise, complete and valuable results the studies should include the replay of both modeled and real workload traces. In our studies we have considered only the usage of synthetic workloads for the moment but once the deployment of "Curie" platform has been made we will collect the real workload traces of the petaflop system and either replay directly parts of them or try to extract the most interesting information out of them to construct new models which will be closer to the real usage of this system. Nevertheless, we believe that in those cases the reproduction of experiments and the variations of different factors results into reliable observations.

2.2 Network Topology Aware Placement

Depending on the communication pattern of the application, and the way processes are mapped onto the network, severe delays may appear due to network contention, delays that result in longer execution times. Nodes that are connected upon the same switch will result in better parallel computing performance than nodes that are connected on different switches. Mapping of tasks in a parallel application to the physical processors on a machine, based on the communication topology can lead to performance improvements [22]. Different solutions exist to deal with those issues on the resource management level.

We are especially interested on fat-tree network topologies which are structures with processing nodes at the leaves and switches at the intermediate nodes [23]. As we go up the tree from the leaves, the available bandwidth on the links increases, making the links "fatter". Network topology characteristics can be taken into account by the scheduler [24] so as to favor the choice of group of nodes that are placed on the same network level, connected under the same network switch or even placed close to each other so as to avoid long distance communications.

This kind of feature becomes indispensable in the case of platforms which are constructed upon pruned fat-tree networks [25] where no direct communication exist between all the nodes. This reduces the number of fast communication group of nodes to only those connected under the same switch. Hence, the efficient network placement is an important capability of a Resource and Job Management System that may improve the application performance, decrease their execution time which may eventually result into smaller waiting times for the other jobs in the queue and an overall amelioration of the system utilization. SLURM provides topology aware placement techniques, treated as an extra scheduling parameter and based upon best-fit algorithms of resources selection. As far as our knowledge, even if other RJMS provide techniques for topology aware placement, SLURM is the only RJMS that proposes best-fit selection of resources according to the network design. In this study we make an in-depth performance evaluation of the topology aware scheduling efficiency of SLURM.

2.3 Scalability

Scalability is one of the major challenges of large HPC systems. It is related with different issues on various levels of the software stack. In the context of Resource and Job Management Systems we are mainly interested into scalability in terms of job launching, scheduling and system responsiveness and efficiency when increasing the number of submitted jobs and the systems scale. Previous related work in the field [26,27] have produced a flexible, lightweight Resource Management called STORM which provided very good scalability in terms of job launching and process scheduling and produced experiments showed very good results in comparison with SLURM or other RJMS. However, on the one hand the particular software was more a research tool and not a production RJMS and it did not provide any backfilling or more elaborate scheduling techniques like network topology consideration.

Another scalable lightweight software used for fast deployment of jobs is Falkon [28]. This system is used as a meta-scheduler by connecting directly upon the RJMS and taking control over the resources by simplifying the procedure of jobs deployment. Hence again in this case the focus is centered just on the fast deployment of jobs without any concern about optimized task placement or prioritization between jobs. A group of computer scientists in CERN have performed scalability experiments of LSF scheduler[3] where they experiment with large scale deployment of LSF scheduler (up to 16000 nodes) using virtual machines. They resulted into good scaling performance of the LSF scheduler but there was no reported analysis of the experiments in order to be able to reproduce them.

In our context of large scale HPC systems we are interested into scalability but also keeping the elaborate scheduling capabilities of the RJMS in order to guarantee a certain quality of service to the users. Hence in this study we are trying to explore the scalability of SLURM by keeping into consideration that efficiency and user quality of services should not be disregarded while we increase in larger scales. As far as our knowledge no other studies have been found in the literature that evaluate the scalability and topology aware scheduling efficiency of Resource and Job Management Systems upon large HPC clusters.

[3] http://indico.cern.ch/contributionDisplay.py?contribId=7&sessionId=7&confId=92498

3 Experimentation Methodology for RJMS Scalability and Efficiency

3.1 Evaluation Based Upon Synthetic Workloads

To evaluate the scheduling performance of Resource and Job Management Systems, we have adopted the Effective System Performance (ESP) model [3,4]. The initial version of this benchmark not only provides a synthetic workload but proposes a specific type of parallel application that can be executed to occupy resources as simply as possible. In our case we just make use of the ESP synthetic workload model and execute simple sleep jobs in place of the proposed parallel application. Indeed, in the context of our experiments which is to evaluate the efficiency of the scheduler and not the behavior of the clusters' runtime environment, the choice of simple sleep jobs is enough.

The ESP [4] test was designed to provide a quantitative evaluation of launching and scheduling parameters of a Resource and Job Management System. It is a composite measure that can evaluate the system via a single metric, which is the smallest elapsed execution time of a representative workload. In ESP, there are 230 jobs derived from a list of 14 job types, as shown in detail in table 1, which can be adjusted in a different proportional job mix . The test is stabilized to the number of cores by scaling the size of each job with the entire system size. Table 1 shows the fraction of each class's job size along with the the number of jobs and the respective duration.

Table 1. Synthetic workload characteristics of ESP benchmark [3,4] and its two variations Light ESP and Parallel Light ESP x10

Benchmarks	Normal-ESP	Light-ESP	Parallel Light-ESP
Job Type	Fraction of job size relative to system size (job size for cluster of 80640 cores) Number of Jobs / Run Time (sec)		
A	0.03125 (2520) / 75 / 267s	0.03125 (2520) / 75 / 22s	0.003125 (252) / 750 / 22s
B	0.06250 (5040) / 9 / 322s	0.06250 (5040) / 9 / 27s	0.00625 (504) / 90 / 27s
C	0.50000 (40320) / 3 / 534s	0.50000 (40320) / 3 / 45s	0.05000 (4032) / 30 / 45s
D	0.25000 (20160) / 3 / 616s	0.25000 (20160) / 3 / 51s	0.02500 (2016) / 30 / 51s
E	0.50000 (40320) / 3 / 315s	0.50000 (40320) / 3 / 26s	0.05000 (4032) / 30 / 26s
F	0.06250 (5040) / 9 / 1846s	0.06250 (5040) / 9 / 154s	0.00625 (504) / 90 / 154s
G	0.12500 (10080) / 6 / 1334s	0.12500 (10080) / 6 / 111s	0.01250 (1008) / 60 / 111s
H	0.15820 (12757) / 6 / 1067s	0.15820 (12757) / 6 / 89s	0.01582 (1276) / 60 / 89s
I	0.03125 (2520) / 24 / 1432s	0.03125 (2520) / 24 / 119s	0.003125 (252) / 240 / 119s
J	0.06250 (5040) / 24 / 725s	0.06250 (5040) / 24 / 60s	0.00625 (504) / 240 / 60s
K	0.09570 (7717) / 15 / 487s	0.09570 (7717) / 15 / 41s	0.00957 (772) / 150 / 41s
L	0.12500 (10080) / 36 / 366s	0.12500 (10080) / 36 / 30s	0.01250 (1008)/ 360 / 30s
M	0.25000 (20160) / 15 / 187s	0.25000 (20160) / 15 / 15s	0.02500 (2016) / 150 / 15s
Z	1.00000 (80640) / 2 / 100s	1.00000 (80640) / 2 / 20s	1.00000 (80640) 2 / 20s
Total Jobs / Theoretic Run Time	230 / 10773s	230 / 935s	2282 / 935s

The typical turnaround time of normal ESP benchmark is about 3 hours, but in case of real production systems, experiments may take place only during maintenance periods and the duration of them are limited. Hence this typical problem, enabled us to define a variation of ESP called Light-ESP, where we propose a diminution of the execution time of each different class with a scope of decreasing the total execution time from 3 hours to about 15 minutes.

Table 1 provides the various jobs characteristics of ESP and Light-ESP. Each job class has the same number of jobs and the same fraction of job size as ESP but has a decreased execution time which results in a decrease of the total execution time. The execution time of each class was decreased by a factor of 0.08 except class Z which was decreased by a factor of 0.5 so that it can give a reasonable duration. The diminution of the total execution time allowed us to perform multiple repetitions of our measurements, during a relatively small duration, in order to increase our certitude about our observations.

In the case of a large cluster size like Curie (80640 cores), Light-ESP will have the 230 jobs adapted to the large system size as we can see in table 1. In order to cover different cases of workloads for the usage of such large clusters, another variation of Light-ESP seemed to be needed with a bigger number of jobs. Hence, a new variation, named Parallel Light-ESP is defined by increasing the number of all the job classes (except the Z) with a factor of 10. However, since we still want to have the same total execution time of the whole benchmark like in Light-ESP, the fraction of job size relative to system size is devided by the same factor 10. This will also allow the system to have a more adapted system utilization and probably similar fragmentation like in ESP and Light-ESP cases. Another particularity of Parallel Light-ESP is that each job class is launched by a different host allowing a simultaneous parallel submission of all type of jobs in contrast with the sequential submission of the Light-ESP. This characteristic allow us to be more closer to reality where the jobs do not arrive sequentially from one host but in parallel from multiple hosts.

The jobs in all the ESP cases are submitted to the RJMS in a pseudo-random order following a Gaussian model of submission which may be also parametrized. The tuning of the various parameters impact the inter-arrival times of the workload so various experiments allowed us to select the best fitted parameters in order to have a fully occupied system as soon as possible. In more detail, Light-ESP variation launches 50 jobs in a row (22% of total jobs) and the rest 180 jobs are launched with inter-arrival times chosen randomly between 1 and 3 seconds. In Parallel Light-ESP case each class of jobs is launched independently with similarly adapted parameters like in Light-ESP case.

Even if better alternatives than the Gaussian model of submission exist in the litterature [29,20], we decided to keep the basis of ESP benchmark intact in this first phase of experiments. Nevertheless, our goal is to continue our studies and explore the real workload traces of "Curie" platform (when those become available). Detailed analysis of the real workload traces of the production site will allow us to perform more adapted parametrization of our proposed workloads, so that they can be as closer to reality as possible.

3.2 Experiment by Combining Real-Scale and Emulation Techniques

In order to measure the various internal mechanisms of the RJMS as one complete system we are based on real-scale experiment upon a production supercomputer and an emulation technique to validate, reproduce and extend our measurements as performed upon the real machine. Hence, at the one side we have been using subsets of "Tera-100" system during its maintenance phase to perform scalability and efficiency experiments using the described workloads; and in the same time we have been studying, analyzing and reproducing the different observations using emulation upon a much smaller fraction of physical machines.

Emulation is the experiment methodology composed by a tool or technique capable for executing the actual software of the distributed system, in its whole complexity, using only a model of the environment. The challenge here is to build a model of a system, realistic enough to minimize abstraction, but simple enough to be easily managed. Various tools currently exist for network [30] or system [31] emulation but in our case we are using simply a SLURM internal emulation technique called *multiple-slurmd*. The normal function of SLURM consists of a central controller/server daemon (slurmctld) executed upon one machine, where all the scheduling, launching and resource and management decisions take place and a compute daemon (slurmd), executed upon each computing node of the cluster. The *multiple-slurmd* technique bypasses this typical functioning model and allows the execution of multiple compute daemons slurmd upon the same physical machine (same IP address) but on different communication ports.

Hence, the number of different deployed daemon on different port will actually determine the number of nodes of the emulated cluster. Therefore the central controller will indeed have to deal with different daemons slurmd and as far as the controller's internal functions concern this emulation technique allows to experiment with them in various scales regardless the real number of physical machines. Nevertheless, this technique has various drawbacks such us the fact that we cannot evaluate most of the functions related to the computational daemon since there may be system limitations due to the multiple number of them upon the same physical host. Furthermore, the number of internal computing resources (sockets, cores, etc) may be also emulated through the configuration files which means that task placement and binding upon the resources will not have the same performances as upon the real system and there may be limitations depending on the physical machines and file systems characteristics. In addition, we may not execute any kind of parallel application, which forces us to be limited on simple sleep jobs in order to provide the necessary time and space illusion to the controller that a real job is actually under execution. In some cases particular simple commands execution with some print into file are used in the end of the sleep period in order to be sure that the job has indeed been executed.

In order to use this type of emulation with big number of multiple-slurmd daemons upon each host some particular tuning was needed upon the physical machines especially concerning the various limits, like increasing the maximum number of user processes and the open files, using the `ulimit` command.

Finally, an important issue that had to be verified is the similarities of the experiment results between real and emulation as long as the dependence of the results into the scale of emulation (in terms of number of emulated hosts per physical machine). Theoretically speaking, concerning the RJMS scalability in terms of job launching and scheduling efficiency, these functions take place as internal algorithms of the controller daemon (slurmctld). Hence the differences between real and emulated results were

System scale	2012 nodes (32 cpus/node)	
Experiment method	Real NO Emulation	Emulation upon 30 nodes
Total ESP Execution time (sec)	1197	1201

System scale	4096 nodes (16 cpus/node)	
Experiment method	Emulation upon 200 nodes	Emulation upon 400 nodes
Total ESP Execution time (sec)	1283	1259

Fig. 1. Real-scale and Emulation Measurements comparisons

expected to be trivial. Figure 1 shows the results obtained when executing Light-ESP benchmark upon real and emulated platforms of various nodes. In these experiments we obtained similar Total execution times for the same workloads which confirmed our assumptions. In addition multiple experiment repetitions made us more confident about the validity of these results.

Our experiments have been deployed upon different systems: a BULL testing cluster for internal usage and a subset of "Tera-100" system during its maintenance periods. The systems have the following hardware characteristics:

- Cuzco Cluster (BULL internal) in Clayes/France site Intel Xeon bi-CPU/quad-CORE with 20 GB memory and network structured by 1Gigabit Ethernet + Infiniband 20G.
- Subset of CEA-DAM "Tera 100" Cluster in Bruyeres-le-Chatel/France with Intel Xeon series 7500 processors (quad-CPU/octo-CORE) with 32GB of Memory and Ethernet + Infiniband networks.

4 Performance Evaluation Results

In this section we present and analyze the results of the conducted experiments in three different subsections. The results concerning the scalability of the RJMS in term of jobs number are presented first. Then, the results and feedback of the topology aware scheduling experiments are detailed. The last subsection explains the results and the experiments realized to evaluate the scalability of SLURM in term of compute resources.

All the experiments are conducted with the consumable resources algorithm of SLURM. This algorithm enables to allocate resources at the core level and thus to share nodes among multiple jobs as long as no more than one job use a same core at the same time.

4.1 Scalability in Terms of Number of Submitted Jobs

The goal of the experiments presented in this section is to evaluate the effect of a large number of jobs on SLURM's behavior. Having a good knowledge of the system behavior in that scenario is necessary to properly configure the related limits and protect the system in its day-to-day usage.

The first set of experiments aims to stress the launch mechanism of SLURM. It enables to define the maximum submission throughput that characterizes the ability to perform a sustained rate of requests. The second set of experiments stresses the cancellation mechanism of SLURM and evaluate the level of control that can be kept on the system in borderline situations when a large amount of events has to be managed by the RJMS. The simulation of a sudden termination of the production is the exercise used for that purpose.

The experiments of the first set are realized using submission bursts of small batch jobs with a global utilization that fit in the available resources. The experiments are conducted at real-scale (no emulation) with a subset of 2020 Tera-100 nodes. The scheduler allocates resources to the jobs and starts them while receiving new requests. According to the documentation, a SLURM parameter called `defer` can be used to improve the management of this kind of workloads. The impact of this parameter is thus compared to the default behavior.

The execution time of each submitted job has to be set to a value large enough to prevent the termination of jobs before the end of the submission stage. The value used in the experiments is 500 seconds.

The results are presented in Figure 2.

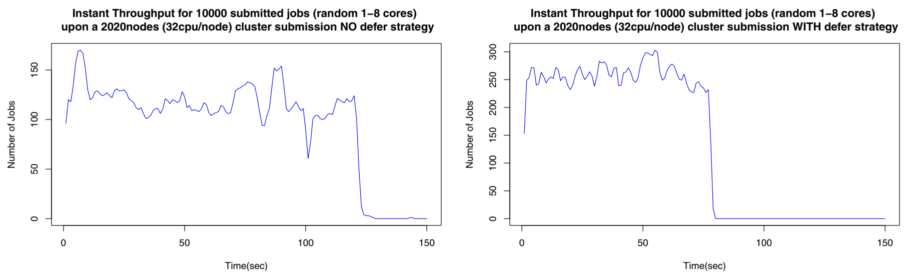


Fig. 2. Instant Throughput for 10000 simple sleep jobs (random 1 to 8 cores each) in submission upon a real cluster of 2020 nodes (32 cpus/node) with and without the usage of `defer` scheduling parameter

We see that the average submission throughput is greatly improved using the `defer` parameter. This parameter ensures that resources allocations are processed in batch by the scheduler and not sequentially at submission time. We observe an average throughput of 100 jobs per second in the standard configuration and more than 200 jobs per second when `defer` mode is activated.

We witness a gain of about 50sec for the complete submission of all the 10,000 jobs between the two cases. The main drawback of the `defer` mode is that individual jobs are slightly delayed to start. Clusters that do not have high submission bursts into their real-life workloads should not activate this parameter to provide better responsiveness.

When increasing the size of the jobs involved in these experiments, we discovered a unexpected impact on the submission throughput. As available resources were not enough to execute all the submitted jobs, the submitted jobs were put in queue in a pending state resulting in a slower pace of submission. During the slowdown, the scheduler was overloaded by the processing of the incoming submission requests. A modification of the SLURM controller was performed in order to eliminate a call to the scheduler logic while processing in parallel the incoming requests in defer mode.

The result of Figure 3 presents the instant throughput with and without this modification. In both cases, the `defer` parameter is used.

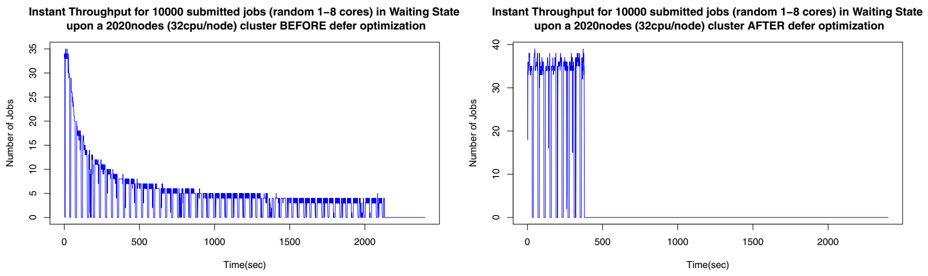


Fig. 3. Instant Throughput for 10000 simple sleep jobs (random 1 to 8 cores each) in submission in Waiting State upon a real cluster of 2020nodes (32 cpus/node) with and without an important optimization in the algorithm of `defer` scheduling parameter

We see that the modification helps to reach a constant throughput. It guarantees that only one instance of the scheduler is called regardless of the number of incoming requests. The number of jobs is thus no longer a factor of the submission rate. It can be noted that the results for waiting jobs are below the throughput of eligible jobs as evaluated above. This low limit has no link with the evaluated aspect and was corrected in a version of SLURM superior to the one used during these tests.

The experiments concerning the termination of a large workload are then performed. The initial goal is to guarantee that a large system is manageable even in the worst cases scenarios. Stopping a large number of jobs on the 2020 nodes cluster outlines a total loss of responsiveness of the system. Looking at the system during the slowdown, it is noted that the induced load is located in the consumable resources selection algorithm when dealing with the removal of previously allocated cores from the SLURM internal map of used cores. The same observation was made by a SLURM community member that proposed a patch to reduce the complexity of the algorithm used for that purpose. This patch is now part of the core version of SLURM.

The results of the termination of 10,000 jobs on the real clusters with and without this patch are presented in Figure 4.

As with the previous experiment, the patch helps to provide a constant termination throughput, no longer dependent of the number of jobs nor resources involved at the time of execution. The responsiveness of the system is now sufficient during the global termination of all the jobs.

Increasing number of both resources and jobs exhibits the scalability thresholds of the RJMS. The results of our experiments concerning scalability in term of jobs enable to safely consider the execution of 10,000 jobs on a production cluster using SLURM. It illustrates different issues that outline the interest of algorithms adapted to the management of high numbers of elements.

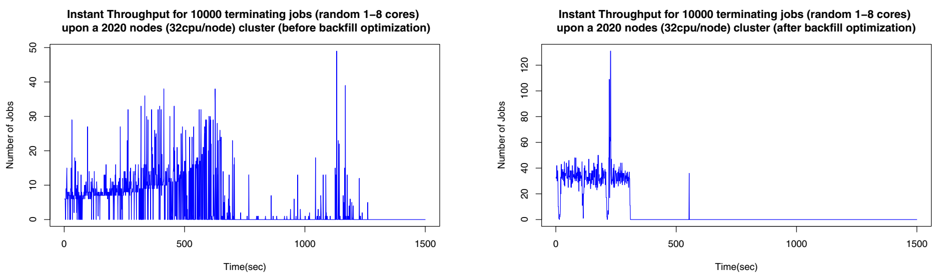


Fig. 4. Instant Throughput for 10000 simple sleep jobs (random 1 to 8 cores each) in termination upon a real cluster of 2020 nodes (32 cpus/node) before and after an important optimization upon the algorithms of backfill scheduler

4.2 Scheduling Efficiency in Terms of Network Topology Aware Placement

The goal of the second series of experiments is to evaluate SLURM's internal mechanism to deal with efficient placement of jobs regarding network topology characteristics, and more specifically regarding a tree network topology. SLURM provides a specific plugin to support tree topology awareness of jobs placement. The advantage of this plugin is its best-fit approach. That means that the plugin not only favors the placement of jobs upon the number of switches that are really required, it also considers a best-fit approach to pack the jobs on the minimal number of switches maximizing the amount of remaining free switches.

The experiments are focused on the evaluation of different tree topology configurations in order to select the most appropriate for "Curie" before its availability for production purposes.

The network topology used on "Curie" is a 3 levels fat-tree topology using QDR Infiniband technology. Current Infiniband technology is based on a 36 ports ASIC that made fat-tree topologies constructed around a pattern of 18 entities. The leaf level of the corresponding Clos-Network is thus made of groups of 18 nodes, the intermediate level is made of groups of 324 nodes (18*18) and the last level can aggregate up to 11664 nodes (18*18*36). The "Curie" machine is physically made of 280 Infiniband leaves switches aggregated by 18 324 ports physical switches grouping the 5040 nodes in 16 virtual intermediate switches.

The experiments are conducted using the emulation mode of SLURM on a few hundreds of Tera-100 physical nodes. These nodes were used to host the 5,040 fat-tree connected SLURM compute daemons of "Curie'. A topology description has to be provided to SLURM in order to inform the scheduler of the layout of the nodes in the topology. In emulation mode, this description is exactly the same as the description of the associated supercomputer. The real topology of the emulated cluster is different as it corresponds to the physical nodes topology. However, as the goal is to evaluate the behavior of SLURM scheduling and not the behavior of the workload execution, it is not at all an issue.

Four different topology descriptions are compared in order to evaluate the impact of the topology primitives of SLURM on the execution of a synthetic workload. The default scheduler is thus compared to the tree topology scheduler with no topology information, a medium topology information, and a fine topology configuration. The medium description only provides information down to the intermediate level of 324 ports virtual switches. The fine description provides information down to the leaf level of ASIC of the Infiniband topology, including the intermediate level. The leaf level of ASIC is also defined here by the term "lineboards",

Global execution time as well as placement effectiveness are then compared. The placement effectiveness is calculated based on a complete description of the fat-tree topology (fine topology). This enables to measure the side effect of a non-optimal selection logic to provide optimal results.

When evaluating placement results, a job, depending on its size, can have an optimal number of lineboards, an optimal number of intermediate switches or both. For example, a job running on a single lineboard will automatically have an optimal number of lineboards and an optimal number of intermediate switches. A job running on three different lineboards of the same intermediate switch could have an optimal number of intermediate switches but a sub-optimal number of lineboards if the requested number of cores could be served by only two lineboards. A job requiring n lineboards could have a sub-optimal number of intermediate switches if the allocated lineboards are spread among the intermediate switches without minimizing the associated amount.

Fragmentation of the available resources over the time is one of the biggest issue of topology placement and prevents from having the theoretical placement effectiveness in practice.

The results of the topology experiments are presented in Table 2 and in Figure 5.

Table 2. Light-ESP benchmark results upon the emulated cluster of "Curie" with 5040 nodes (16 cpus/node), for different types of topology configuration

SLURM NB cores-TOPO Cons / Topo-ESP-Results	Theoretic- Ideal values	No Topo	Basic Topo	Medium Topo	Fine Topo
Total Execution time(sec)	925	1369	1323	1370	1315
# Optimal Jobs switches	228	53	13	180	113
# Optimal Jobs on switches + lineboards	228	21	6	120	106
# Optimal Jobs on lineboards	228	64	31	120	209

Ten runs of each configuration are executed and show stable results. The global execution time of the different runs are similar, having even the most detailed flavor of the tree topology strategy finishing a little bit sooner. CDF on execution times and submission times follow the same trend and show a little gain for fine topology and a little loss for the medium topology. This first result is really interesting as it validates the stability of the global throughput of the simulated clusters in regard to the different topology configurations evaluated. No trade-off between global throughput and independent jobs performances is thus required. Improvements in the jobs performances due to a better placement should directly and positively impact the system utilization.

The difference in term of placement effectiveness between the experiments without detailed topology information and the experiments with the details are huge.

In its standard behavior, the SLURM scheduler surprisingly acts better than a tree topology having all the nodes at the same level with our synthetic workload. We expected to have a same poor level of effectiveness but the best-fit logic of both schedulers treats

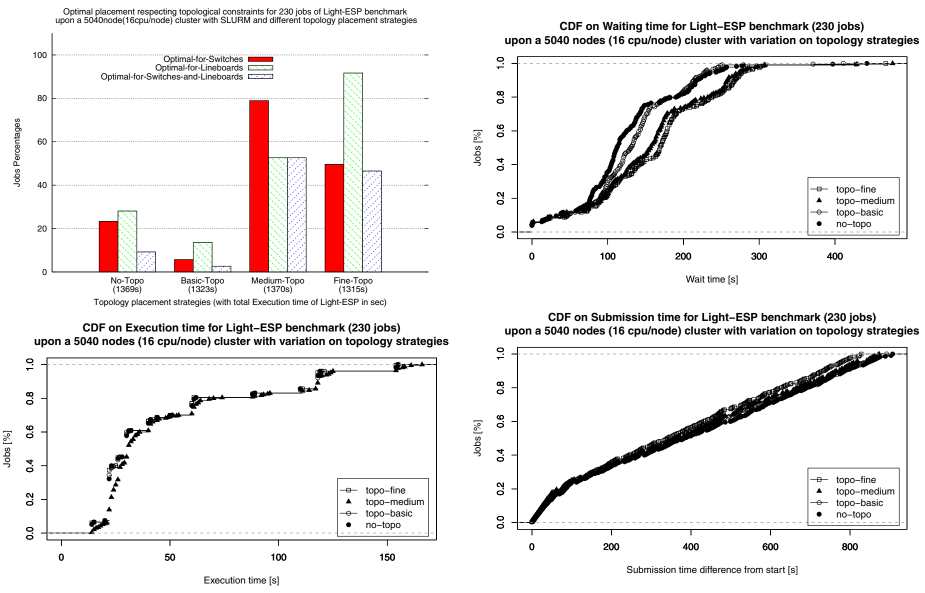


Fig. 5. System Utilization and Cumulative Distribution Functions of jobs submission and waiting times, for different cluster sizes and Light-ESPx10 with 2282 jobs

the row of nodes differently. The original consecutive nodes allocation of the default SLURM scheduler may be at the origin of the difference.

The best placement results of the default scheduler are about 20% to 25% of right placement when only considering one of the two levels of significant topology information of the tree, and less than 10% with both simultaneously.

The medium topology configuration provides better results with about 80% of right placement for its targeted medium level and about 50% for the lineboard level and both simultaneously.

The fine topology configuration provides even better results for the lineboard level with more than 90% of right placement. The results for optimal placement at the intermediate level and for both levels simultaneously are below the previous configuration, standing between 45% and 50%.

At first sight, the fine topology results are surprising as one could expect the fine strategy to be a refined version of the medium configuration only increasing the chance of having the right amount of lineboards when possible. Code inspection was necessary to understand that difference. The tree topology logic of SLURM is a two steps logic. First, it looks for the lowest level switch that satisfies the request. Then it identifies the associated leaves and do a best-fit selection among them to allocate the requested resources. Intermediate tree levels are thus only significant in the first step. The differences between the medium and the fine configuration are explained by to that logic.

When the system is fragmented and no single intermediate switch is available to run a job, the top switch is the lowest level that satisfies the request. In medium configuration, all the intermediate switches are seen as the leaves and the best-fit logic is applied against them. Chances to have the optimal number of intermediate switches are high. In fine configuration, all the lineboards are seen as the leaves and the best-fit logic is applied against them without any consideration of the way they are aggregated at intermediate levels. Chances to have the optimal number of lineboards are high but chances to have the optimal number of intermediate switches are quite low.

SLURM aims to satisfy the leaves affinity better than the intermediate levels affinity when the system is fragmented. As a result, multiple level trees are subject to larger fragmentation issues than standard 2-levels tree. Indeed, the leaves being equally treated when fragmentation occurs, jobs are spread among the intermediate level switches without any packing strategy to increase the chance of recreating idle intermediate switches. Considering that fact, the choice between using a medium versus a fine topology configuration is driven by both the hardware topology and the workload characteristics.

Pruned fat-tree topologies, like the one used in Tera-100, have smaller bandwidths between switches located at the intermediate level. Using a fine configuration with these topologies could result in the incapacity to provide enough bandwidth to jobs requiring more than one switch because of too much fragmentation. A medium configuration, only focusing on detailed information about the pruned level is certainly the best approach.

For systems where the workload characteristics let foresee idle periods or large job executions, then the fine configuration is interesting. Indeed, it will favor the optimal number of lineboards and even switches when the system is not highly fragmented. If the workload characteristics let foresee jobs having the size of the intermediate switches or if a high 24/24 and 7/7 usage would inevitably conduct to a large fragmentation, using the medium configuration appears to be a better approach.

The results of the topology experiments are really interesting to help configuring the most adapted topology configuration for tree based network topology. A same approach, using a synthetic workload tuned to reflect a site typical workload should be used in addition to confirm the choice before production usage. It could also be used on a regular basis to adapt the configuration to the evolution of jobs characteristics.

SLURM behavior with multiple-level trees needs to be studied deeper in order to determine if a balanced solution between medium and fine configuration could bring most of the benefits of each method and enable to have better effectiveness with a fine description. A best-fit selection of intermediate switches in the first step of SLURM tree topology algorithm will have to be evaluated for that purpose.

4.3 Scalability in Terms of Number of Resources

The light-ESP benchmark [32], described in section 3, is used in these experiments. This synthetic workload of 230 jobs with different sizes and target run times is run on different emulated clusters from 1,024 nodes to 16,384 nodes. In this benchmark, the size of the jobs automatically grows with the size of the targeted cluster. As a result, a job size always represent the same share of the targeted system.

To evaluate the job size effect on the proper execution of the workload, a modified version of the Light-ESP benchmark having 10 times the number of 10x smaller jobs is used. This benchmark results in the same amount of required resources and total execution time as with the previous workload.

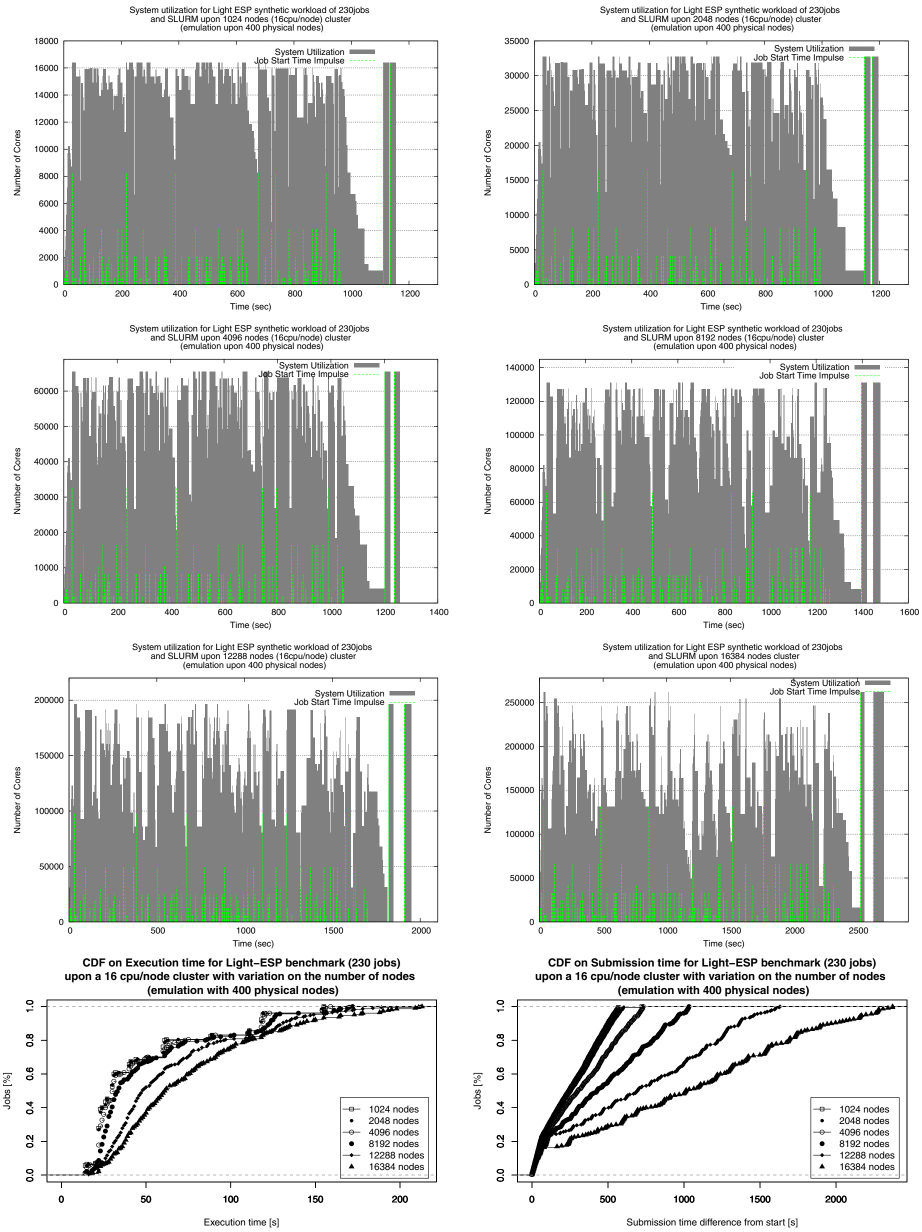


Fig. 6. System Utilization and Cumulative Distribution Functions of jobs submission and execution times, for different cluster sizes and Light-ESP with 230 jobs

In all cases, a single encapsulated SLURM client call is used in every job in order to evaluate the responsiveness of the system. The more the central controller is loaded, the more the execution time is increased by the delay of the associated reply reception. The results gathered during the executions of the Light-ESP for different cluster sizes are detailed in Figure 6.

We see an increase in jobs execution times as well as a stretching of CDF on submission time for clusters of 8k nodes and more. The results suggest a contention proportional to the number of compute nodes in the central controller process. Looking closer at the system during a slowdown, the central SLURM controller is overloaded by a large amount of messages to process. The messages are mostly internal messages that are transmitted by the compute nodes at the end of each job.

Current design in SLURM protocol (at least up to version 2.3.x) introduces the concept of epilogue completion message. Epilogue completion messages are used on each node involved in a job execution to notify that the previously allocated resources are no longer used by a job and can be safely added to the pool of available resources.

This type of message is used in a direct compute node to controller node communication and result in the execution of the SLURM scheduler in order to evaluate if the newly returned resources can be used to satisfy pending jobs. In order to avoid denial of service at the end of large jobs, SLURM uses a dedicated strategy to smooth the messages throughput over a certain amount of time. The amount of time is capped by the number of nodes involved in the job multiplied by a configuration variable defining the estimated process time of a single message by the controller. This strategy induces latencies at the end of the jobs before the global availability of all the associated resources for new executions.

In SLURM vocabulary, these latencies correspond to the time the previously allocated nodes remain in the `completing` state. The decreasing system utilization densities of the workloads when cluster size increases, as displayed in Figure 6, outline these latencies.

Each Light-ESP workload is finished when two jobs using all the resources are processed. As a result, the end of each workload enables to have a direct picture of the effect of the completing latency. While the number of nodes increases on the emulated clusters, the idle time between the last two runs increases proportionally. `EpilogMsgTime`, the parameter used to define the amount of time of a single epilogue complete message processing, has a default value of 2ms. This result in a completing window of at least 32s when a 16,384 nodes job is finishing. Looking at the results of our experiments, it sounds that the delay between the last two jobs is even greater than the expected completing time. In order to understand that behavior, a new set of experiments are conducted. The associated results are shown in Figure 7.

The idea was to first reduce the EpilogMsgTime in order to see if the completing time would be proportionally reduced and in a second phase to reduce the epilogue message processing complexity to see if the time to treat each message would be reduced too. For the second target, a patch was added in order to remove the scheduling attempt when processing an epilogue complete message on the controller. The epilogue message time was kept smaller for this run too in order to have a better idea of the additional effect of the two propositions. As shown by Figure 7, the two strategies enable to reduce the

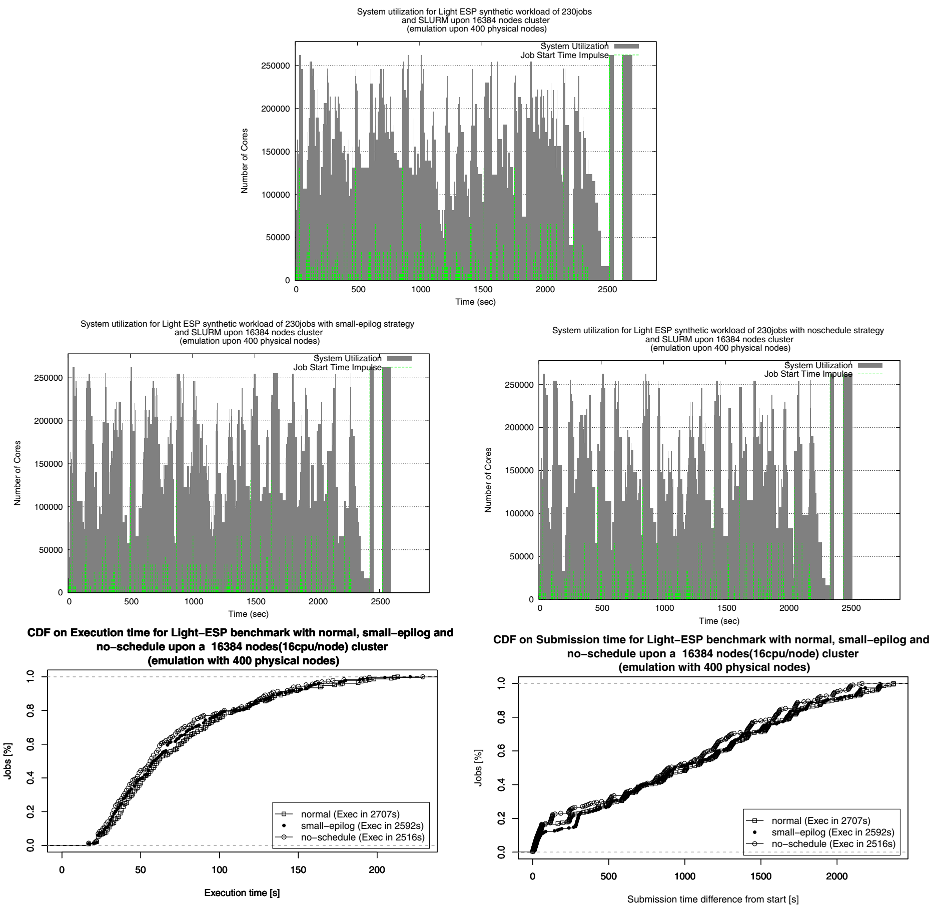


Fig. 7. Cumulative Distribution Functions of jobs submission and execution times, for different termination strategies upon a 16384 nodes(16 cpus/node) cluster and Light-ESP with 230 jobs

global computation time consecutively and provide reduced execution time in comparison with the standard run. However, the results are still no longer as effective as for small clusters. The CDF on submission times on the same figure still shows the stretching of the curves that reflects delay of submissions of the ESP launcher. The CDF on execution time shows an increase of execution times that suggests an unresponsive controller during the jobs execution for large clusters. The epilogue completion mechanism in SLURM, and more specifically its processing time and its peer-to-peer communication design appears to be the main bottlenecks at scale.

Further investigations must be done in order to evaluate the feasibility of a modification of this protocol in order to achieve better performances. Reversed-Tree communications, like the one used for the aggregation of jobs statistics of each node right before the epilogue stage could be a good candidate. This would result in spreading the overhead of messages aggregation to the compute nodes and let the central controller only manage one single message informing of the completion of the epilogue at the end of each job. Side effects, like compute nodes taking too much time in processing the epilogue stage will have to be taken into account. This issue may be one of the origins of the peer-to-peer design choice that is made for current version of SLURM. However, the previous results outline the fact that large clusters require a new balance between epilogue completion aggregation and quick return to service of previously allocated nodes.

Large jobs induce long completing periods and a high load of the controller because of large number of epilogue completion message to process. It is interesting to evaluate the behavior of SLURM with the same different cluster sizes but with a larger workload composed of smaller jobs to determine if this behavior is still observed. The results of the experiments are presented in Figure 8.

For small clusters, up to about 4,096 nodes, the global throughput of such a workload is really great and even close to the theoretical value of 935 seconds. An interesting result is that the responsiveness of the controller is also greater. Indeed, there is no longer a large variation on the CDF on execution times for the different cluster sizes. This has to be related to the smoothing of the job endings induced by the larger workload of smaller jobs. However, large clusters, having the ability to run a larger numbers of jobs simultaneously, are subject to the same kind of stretching of system utilization than with the original Light-ESP workload. The effect is even worst as the system utilization is not only stretched but collapses over the time.

Based on the feedback of the previous studies on large number of job submissions and the effect of removing the scheduler call in the epilogue complete processing of the controller, a potential candidate to explain the effect would be the impact of SLURM internal scheduler when called at submission and completing time. Indeed, the CDF on submission time shows that the submissions on the larger clusters are delayed during the execution of the workloads on the large supercomputers. As the responsiveness stays similar across the different sizes, the collapses should be due to the scheduling complexity at submission time that decrease the submission rate.

Further experiments are required to identify the reasons of this issue. The usage of the `defer` parameter, as well as the patches made to remove the various useless scheduler call when this mode is activated, will have to be tested to validate this assumption.

To sum up, the results on the scalability in terms of number of resources show that good efficiency and responsiveness can be obtained for clusters up to 4,096 nodes. The scalability threshold is placed between 4,096 and 8,192 nodes in the evaluated configuration with the consumable resources selection algorithm of SLURM. A first way of enhancement has been identified and should help to provide better result in term of scalability for small workload (in number of jobs). A second issue has been identified and could prevent from extending the positive effect of the first one to larger workload (in number of jobs).

Improvements and new experiments are required to prepare SLURM to manage larger clusters. Current production clusters should not suffer of the outlined issues but must be monitored.

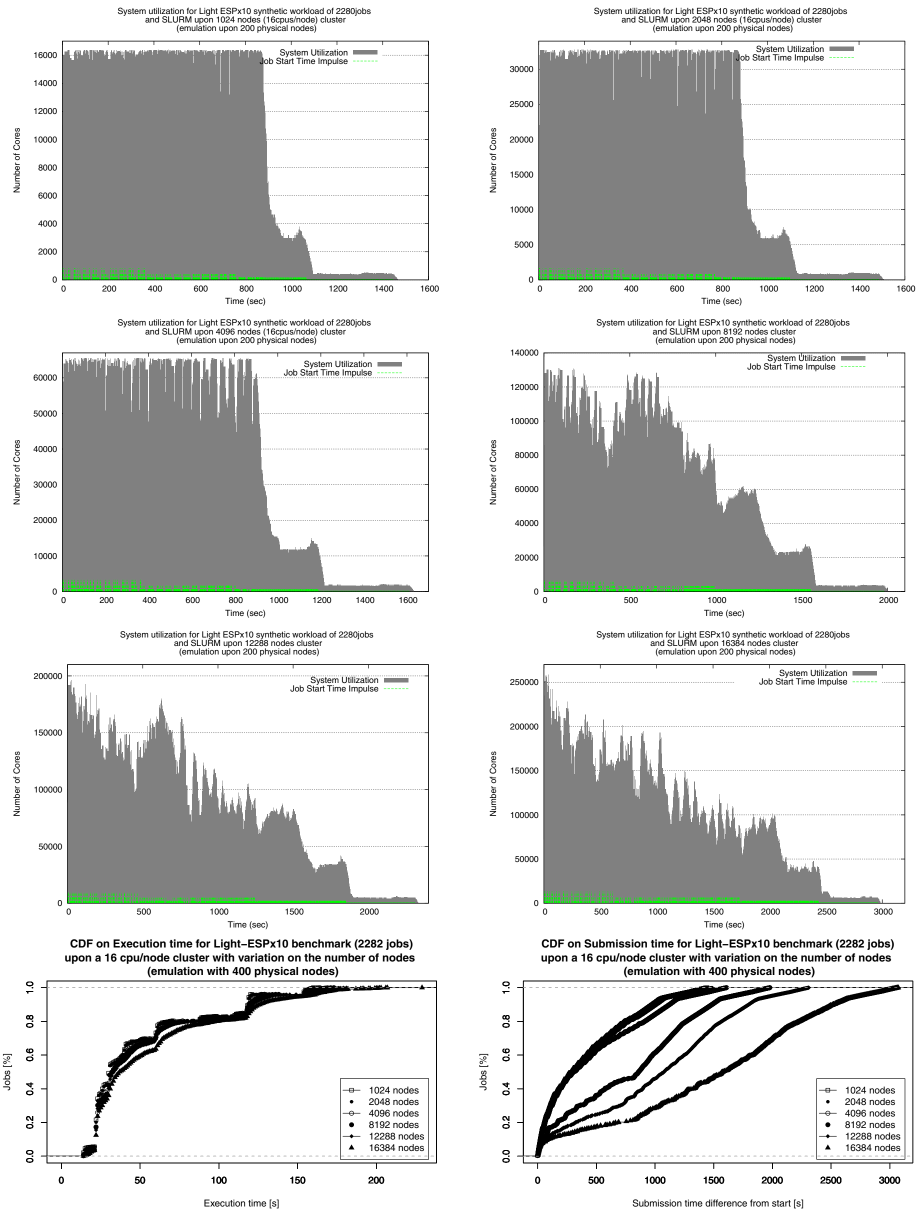


Fig. 8. System Utilization and Cumulative Distribution Functions of jobs submission and execution times, for different cluster sizes and Light-ESPx10 with 2282 jobs

5 Conclusions and Future Works

In this paper we have presented an evaluation methodology for the Resource and Job Management System SLURM based upon combined real-scale and emulation experiments using synthetic workloads.

We have validated our methodology by observing similar results for identical workloads on both the real-scale and emulated systems. Using emulated clusters is a real alternative to cost-effective real-scale experiments that are not only hard to be approved but also suffers from the inherent hardware faults that make evaluating a system in its complete availability a pretty hard task. The validation of this methodology enabled us to conduct experiments on emulated clusters with a strong confidence on their correctness in real situation.

The choice of using synthetic workloads was initially performed to ease the modification of their characteristics and address a broader spectrum of scenarios. Furthermore, the PRACE targeted machine, “Curie”, not being in production at the petaflop scale at the time of the study, no real workload traces were available as an input for a similar approach.

The conducted experiments helped to guarantee the capabilities and the efficiency of the open source RJMS compared to the requirements of the targeted petaflop machines, “Tera-100” and “Curie”. In a second time, experiments at larger scales were realized to not only have a precise idea of the behavior of the product on current machines but also evaluate the capability of SLURM to manage clusters up to 16,384 nodes.

The conclusions raised during the evaluations are significant for the day-to-day usage of the installed machines and let us have a better view of the work that will be required in the following months and years to prepare a valid RJMS candidate for the next generation of supercomputers. Our plans are now to extend our methodology with real workload traces replay to confirm our first assumptions concerning the configuration parameters to use on our system. In addition, we will work on the correction of the identified topology awareness and scalability limitations in order to enhance SLURM for the proper management of clusters up to 16,384 nodes.

The emulation experience acquired during this study let us consider that an emulated cluster of up to 65,536 nodes, the theoretical limit of SLURM, could be correctly emulated on the currently available petaflop machines. We will certainly face new issues and thresholds associated to such a new scale of experiments.

References

1. Top500 supercomputer sites, http://www.top500.org/
2. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003)
3. Wong, A., Oliker, L., Kramer, W., Kaltz, T., Bailey, D.H.: System Utilization Benchmark on the Cray T3E and IBM SP. In: Feitelson, D.G., Rudolph, L. (eds.) IPDPS-WS 2000 and JSSPP 2000. LNCS, vol. 1911, pp. 56–67. Springer, Heidelberg (2000)
4. Kramer, W.T.C.: PERCU: A Holistic Method for Evaluating High Performance Computing Systems. PhD thesis, EECS Department. University of California, Berkeley (November 2008)

5. Zhou, S., Zheng, X., Wang, J., Delisle, P.: Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical report (1993)
6. Ibm loadleveler, http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf
7. Henderson, R.L.: Job scheduling under the portable batch system. In: IPPS 1995: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, pp. 279–294. Springer, London (1995)
8. Moab workload manager, http://www.adaptivecomputing.com/resources/docs/mwm/7-0/help.htm
9. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency - Practice and Experience 17(2-4), 323–356 (2005)
10. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: 5th Int. Symposium on Cluster Computing and the Grid, pp. 776–783. IEEE, Cardiff (2005)
11. Grid engine, http://gridscheduler.sourceforge.net/howto/howto.html
12. Torque resource manager, http://www.adaptivecomputing.com/resources/docs/torque/4-0/help.htm
13. Maui scheduler, http://www.adaptivecomputing.com/resources/docs/maui/index.php
14. Kaplan, J.A., Nelson, M.L.: A comparison of queueing, cluster and distributed computing systems. NASA TM-109025 (Revision 1), NASA Langley Research Center, Hampton, VA 23681-0001 (June 1994)
15. Baker, M.A., Fox, G.C., Yau, H.W.: Cluster computing review (1995)
16. El-Ghazawi, T.A., Gaj, K., Alexandridis, N.A., Vroman, F., Nguyen, N., Radzikowski, J.R., Samipagdi, P., Suboh, S.A.: A performance study of job management systems. Concurrency - Practice and Experience 16(13), 1229–1246 (2004)
17. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: 4th Workshop on Workload Characterization, pp. 140–148 (December 2001)
18. Frachtenberg, E., Schwiegelshohn, U.: New Challenges of Parallel Job Scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 1–23. Springer, Heidelberg (2008)
19. Chapin, S.J., Cirne, W., Feitelson, D.G., Jones, J.P., Leutenegger, S.T., Schwiegelshohn, U., Smith, W., Talby, D.: Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999. LNCS, vol. 1659, pp. 67–90. Springer, Heidelberg (1999)
20. Frachtenberg, E., Feitelson, D.G.: Pitfalls in Parallel Job Scheduling Evaluation. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 257–282. Springer, Heidelberg (2005)
21. Feitelson, D.G.: Metric and workload effects on computer systems evaluation. IEEE Computer 36(9), 18–25 (2003)
22. Bhatele, A., Bohm, E.J., Kalé, L.V.: Topology aware task mapping techniques: an api and case study. In: PPOPP, pp. 301–302 (2009)
23. Leiserson, C.E.: Fat-trees: Universl networks for hardware-efficient supercomputing. IEEE Transactions on Computers c-34(10) (1985)
24. Navaridas, J., Miguel-Alonso, J., Ridruejo, F.J., Denzel, W.: Reducing complexity in tree-like computer interconnection networks. Parallel Computing 36(2-3), 71–85 (2010)
25. Bay, P., Bilardi, G.: Deterministic on-line routing on area-universal networks. JACM: Journal of the ACM 42 (1995)
26. Frachtenberg, E., Petrini, F., Fernández, J., Pakin, S.: Storm: Scalable resource management for large-scale parallel computers. IEEE Trans. Computers 55(12), 1572–1587 (2006)

27. Fernández, J., Frachtenberg, E., Petrini, F., Sancho, J.C.: An abstract interface for system software on large-scale clusters. Comput. J. 49(4), 454–469 (2006)
28. Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M.: Falkon: a fast and light-weight task execution framework. In: IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2007) (2007)
29. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63, 2003 (2001)
30. Vishwanath, K.V., Vahdat, A., Yocum, K., Gupta, D.: Modelnet: Towards a datacenter emulation environment. In: Schulzrinne, H., Aberer, K., Datta, A. (eds.) Peer-to-Peer Computing, pp. 81–82. IEEE (2009)
31. Canon, L.-C., Jeannot, E.: Wrekavoc: a tool for emulating heterogeneity. In: IPDPS. IEEE (2006)
32. Wong, A.T., Oliker, L., Kramer, W.T.C., Kaltz, T.L., Bailey, D.H.: ESP: A system utilization benchmark. In: SC 2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, pp. 52–52. ACM Press and IEEE Computer Society Press (2000)

Partitioned Parallel Job Scheduling for Extreme Scale Computing

David Brelsford[1], George Chochia[1], Nathan Falk[1], Kailash Marthi[1], Ravindra Sure[1], Norman Bobroff[2], Liana Fong[2], and Seetharami Seelam[2]

[1] IBM Systems and Technology Group
{brels4d,chochia,nfalk,kmarthi}@us.ibm.com, ravisure@in.ibm.com
[2] IBM T.J. Watson Research Center
{bobroff,llfong,sseelam}@us.ibm.com

Abstract. Recent success in building extreme computing systems poses new challenges in job scheduling design to support cluster sizes that can execute million's of concurrent tasks. We show that for these extreme scale clusters the resource demand at a centralized scheduler can exceed the capacity or limit the ability of the scheduler to perform well. This paper introduces *partitioned scheduling*, a hybrid centralized and distributed approach in which compute nodes are assigned to the job centrally, while task to local node resources assignments are performed subsequently at the assigned job nodes. This reduces the memory and processing growth at the central scheduler, and improves the scaling behavior of scheduling time by enabling operations to be done in parallel at the job nodes. When local resource assignments must be distributed to all other job nodes, the partitioned approach trades central processing for increased network communications. Thus, we introduce features that improve communications such as pipelining that leverage the presence of the high speed cluster network. The new system is evaluated for jobs with up to 50K tasks on clusters with 496 nodes and 128 tasks per node. The partitioned scheduling approach is demonstrated to reduce processor and memory usage at the central processor and improve job scheduling and job dispatching times up to an order of magnitude.

1 Introduction

The primary goal of job scheduling for high performance computing (HPC) is to assign parallel jobs to compute nodes, matching the resource requirements of the job to available and capable nodes. Traditionally, job to node matching is performed by a centralized scheduling architecture in which a resource manager module monitors the state of the compute nodes in the cluster, sharing this data with the collocated scheduling module. However, this centralized scheduling model is stretched in meeting the demands of extreme scale HPC supercomputer clusters.

For example, an exemplary supercomputer that is part of the IBM DARPA HPCS [1] program, contains up to 16K nodes, 2 million tasks and targets sustained performance in excess of a petaflop for applications such as computational biology and chemistry, fluid mechanics, and galaxies formation studies.

Each compute node consists of a quad Power7 chip module with 8 cores per chip for a total of 32 cores [21]. Each core supports simultaneous multi-threading (SMT) up to 4, allowing up to 128 job tasks to execute on a compute node. Compute nodes are packaged in modular groups of 32 termed super-nodes. Internal to each super-node is a full point-to-point switched network with minimum sustained bandwidth of 6 GB/s. The supercomputing system is constructed by inter-connecting tens to hundreds of super nodes with multiple 10 GB/s links, up to a total of 16K nodes and capable of running 2 million tasks.

While a centralized scheduler scales linearly in memory and processor demand with the number of job nodes and tasks, it still may not deliver adequate scheduling performance and can be overloaded in demand for memory. In fact, we have found this to be the case while doing performance studies aimed at extreme scale systems using the IBM TWS-LoadLeveler (LL) scheduler [20,4]. A root cause of the problem is that centralized schedulers do not leverage the large number of compute nodes assigned to the job that can assist in the scheduling process. Increasing the parallelism allows a transition of large parts of the scheduling and dispatching problem from linear scaling to more constant order, with substantially reduced resource demands at the central scheduler. This transition to a distributed and parallel model is enabled by the higher speed and lower latency networks that are now present on HPC clusters since distributed models invariably require more communication than centralized ones.

To this end we introduce a hybrid approach to the scheduling architecture which adopts key elements of both parallel and distributed system features that we call *partitioned scheduling*. The main observation that leads to partitioned scheduling is that by deferring the assignment of local compute node resources to tasks - such as DMA windows and processor affinity - until after the node selection process allows the compute nodes themselves to do scheduling of these local resources. Of course, certain local scheduling decisions such as network adapter assignments still have to be propagated to all other nodes in an all-gather type of communication paradigm [17]. A further contribution to the work is the many improvements in system communication developed to distribute the local node scheduling decisions and improve dispatching.

Once local resource scheduling is moved out of the central scheduling component, additional improvement to the linear aspects of global job to nodes selection are possible by parallelization. Concurrently threads in node assignment function are leveraged using multicores while avoiding slowdowns arising from lock contention.

This paper presents the partitioned scheduling design and implementation, and shows how it has led to significant speedups in job scheduling while simultaneously reducing the resource demand at the central manager. We show that even in small clusters of 248 nodes this trade-off in costs is considerable, leading to order of magnitude improvements in many performance aspects of job scheduling and dispatching. From an overall system evaluation perspective the total gain does not quite reach a factor of 10, but exceeds a factor of five on our smaller cluster. But the improvements provide a capacity and scaling safety

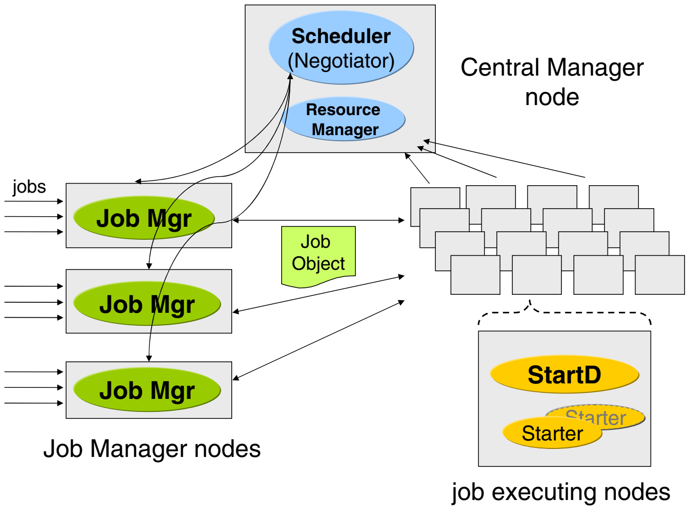


Fig. 1. A high level view of LoadLeveler Architecture

margin that is also expected to be significant in absorbing more load as the system scales up to 16K nodes.

The paper starts with a description of the centralized IBM LL architecture that existed prior to this work. After an analysis of its scaling limitations, the adoption of the partitioned scheduling and the numerous scalability enhancements are described. Measurements on up to 496 nodes and 128 tasks per node are used to illustrate the gains in of scalability that have been achieved.

2 LoadLeveler Architecture

IBM LoadLeveler (LL) is a job management product for high performance computing (HPC) clusters. LL focuses on batch job management for enterprise systems with commercially pertinent features, such as detailed accounting for charge-back, priority & work flow control, high availability with parallel checkpointing, and job recovery. It supports a wide range of systems from small clustered workstations to large IBM BlueGene [6] supercomputers.

The basic architecture of LL (Figure 1) consists of three key components: a central manager (CM), one or more job managers (JM), and a cluster of compute nodes. The JM provides a portal for job submission and is responsible for job lifecycle management including: persisting job state to disk, launching and coordinating the execution of the job on the nodes assigned by the scheduling component, and recovering the job in case of a failure. The CM runs on a single machine and includes the collocated resource manager (RM) and job scheduler (aka Scheduler). The CM has a simple failover scheme based on a recovery model that consists of standby CMs. In case of failure, job states are supplied by the JMs and resource states reestablished by compute nodes via the RM.

The RM is an in-memory repository of cluster information such as the state of every node, including dynamic and static attributes. A daemon process (`StartD`) on each of the compute nodes pushes the resource information to the RM. `StartD` reports both the static and dynamic attributes of the node, the former at node start-up and reconfigurations, and the latter at configurable sampling intervals or job state changes. Static configuration data include hardware attributes, memory size, types and numbers of network adapters, while examples of dynamic metrics are CPU utilization and active job states.

The Scheduler allocates compute nodes and other resources to a job by matching the requirements of the job to compute nodes using the state data provided by the RM. In addition to number of nodes and number of tasks per node, the job may specify computational processor power, memory space, system architecture type (e.g., Intel x86, IBM POWER), disk storage, high-speed network resources, software environmental entities (e.g. OS, software license), etc. When available and supported in a clustered systems, jobs can specify a number of *network windows* that provide an efficient mechanism to exchange data between job tasks via direct memory access (DMA).

It is useful to characterize the resource types considered by the Scheduler into categories according to the job matching conditions:

1. **Global or Floating Resources:** A global resource is a cluster-wide consumable of finite number N such as cluster-wide licenses for a particular software. No more than N jobs requiring such a resource can run concurrently on the cluster.
2. **Node Resources:** A node resource has static or/and dynamic attributes that are defined on a per node or per job basis, irrespective of the number of tasks of job running on that node. For example, a job class may specify for a compute node with a specific number of slots. Once a job assigned to such a job class slot, the slot would not be available to other job until that job releases the class slot.
3. **Node Local Resources:** Node local resource are a resources tied to a node. They are typically consumed on a per-task basis and the association between the task and the corresponding resource being consumed must be preserved for the duration of the job. Two common examples here are *task to core* mapping and *task to network window* mapping. In HPC, a particular task often runs on a particular core for the duration of the job using the *task to core* mapping. Similarly, in *task to network window* mapping, each task will be assigned a particular network window for its use. Every other task of this job must use this window to communicate with this task. For this, every task must know the task to network window association of all tasks of a job. This requires an all-to-all communication of this information.

The job-to-resource matching complexity is different for each category of resources. Global and node resources are matched on a per job basis while node local resources are assigned to individual tasks of each job. The complexity of matching leads to considerable processing, memory consumption, and a scalability

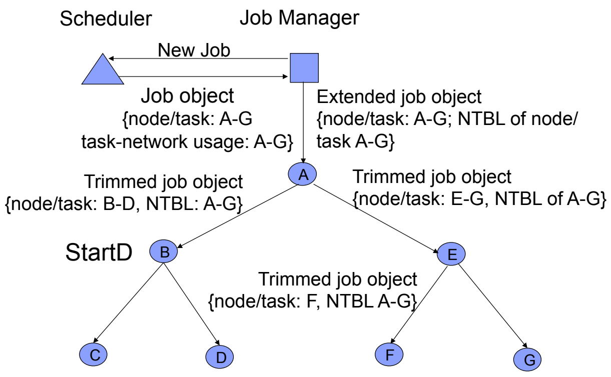

Scheduler
Job Manager
New Job
Job object {node/task: A-G task-network usage: A-G}
Extended job object {node/task: A-G; NTBL of node/ task A-G}
Trimmed job object {node/task: B-D, NTBL: A-G}
Trimmed job object {node/task: E-G, NTBL of A-G}
StartD
Trimmed job object {node/task: F, NTBL A-G}
A
B
C
D
E
F
G

Fig. 2. Hierarchical Job Object and Network Table Propagation

bottleneck at the Scheduler. The subsequent section expands on the limits of processor and memory scalability at the CM machine.

When scheduling for a job is successfully, the CM returns to the JM the resource assignments, including global, node and node local resources. As shown in Figure 2, nodes A through G represent the assigned compute nodes. If network window to task assignments are provided for a job by CM, as the task-network usage information, then JM takes the per task network adapter information and builds it into the Network Table (NTBL) which contains necessary information to enable all-to-all inter task communication. JM then merges the job node assignments with the NTBLs and wraps it into a single large job object (JO) that is sent to the `StartD` on the compute nodes. JO dispatching is done using the N-ary tree, where the structure of the tree is encoded into the JO itself. As shown in the figure, the JO is first passed to the root node where data for that node is extracted. The node then revised the JO contents and forwards the JO to its children and the process continues to the leaf nodes.

Job execution at compute nodes is managed by the local `StartD` which forks a `Starter` process – a job executing agent – to initiate and control the job execution. Concurrent execution of tasks at a compute node is enabled by forking multiple `Starter` processes. Changes of job status will be reported by `StartD`'s back to the corresponding JM that manages the job's lifecycle.

3 Scalability Issues in LL Architecture

The centralized nature of job scheduling and the demand in processing cycles and memory is a scalability concern for extreme scale clusters. CPU bottlenecks are examined first by looking at the scheduling stages, and identifying where they consume cycles that can be moved to local compute nodes. Then we consider how the memory footprint, which becomes unsustainable, in a central architecture, is reduced by node level scheduling.

LL uses a two stage scheduling process in which a set of nodes *capable* of running the job (based on specific job attributes) is selected first, followed by a priority based selection of a subset of these nodes according to *dynamic capacity.* These steps are further described below:

– Stage 1: Designate capable nodes as having the *static capabilities* to match the requirements of the job; capabilities include node architecture types and features, job class definitions, and high-speed network adapter types and counts.
– Stage 2: Select capable nodes from an ordered list of nodes that have the *dynamic capacities* to assign to the job; exemplary capacities include unused job class instances, unfilled multiprogramming level, and spare network windows. The nodes are ordered based on an administrator defined priority such as descending order of free CPU utilization.

Stage 1 is performed once for a job to produce a list of capable nodes in a cluster. An update to the list is triggered only for the addition or removal of nodes from the cluster, an infrequent event in HPC environments. Stage 2 is repeated at each scheduling cycle, as existing jobs terminate and free up resources, until enough nodes are available to the requesting job. The two stage scheduling process is an optimization that is useful if typically incoming jobs being scheduled must wait for termination of running jobs for resources.

The processing time to complete these stages is bi-linear in the number of job nodes and node local resources that need to be assigned to the job. In fact, node local resource scheduling time is proportional to the number of tasks because local resources are assigned on a per task basis. A typical example is allocation of network windows for DMA to each task on a node. Here the processing of stage 2 is logically an outer loop on the N nodes with an inner loop assigning network windows to T tasks. This CPU intensive work is quantified using the observed scheduling time (upper curve of Figure 8(B) in Section 6) which shows that scheduling a job of 248 nodes and 64 task per node takes 2.4 seconds on the centralized Scheduler. Assuming at best a linear scaling in nodes and tasks, this projects to at least 160 seconds on the 16K node cluster.

One could argue that, performed centrally, there is an opportunity to unwrap the scheduling loop, for example by having each of a group of hardware threads perform the local task scheduling for each node. However, a second major scaling issue arises which is that memory usage is becoming unmanageable. According to our data on a 496-node, 64 task job (upper curve of Figure 7 in Section 6), memory consumption would scale out to over 4.6GB on the 16K node cluster. In fact, as long as the cluster is fully occupied this much memory must be consumed to keep track centrally of resource assignments of all currently running jobs. The scalability issue with memory is not simply the size, but the fact that it is not a monolithic allocation, but comprised of a very large number of smaller data objects reflecting the scheduling details. These memory allocations and releases add significant overhead to the CM.

The solution to the scalability problem is to leverage the compute nodes to perform the scheduling of node local resources such as network windows. This reduces CPU utilization at the central manager by separating the scheduling into two sequential steps. First, the Scheduler selects job nodes that have sufficient free local resources for the number of tasks per node. This is done by maintaining an aggregate counts of resources, for example, each node's free network windows at the CM. Once the job nodes are selected, nodes assign the windows to tasks and report these assignments to peer nodes that require communication. More detail of how this partitioned scheduling works in practice is described in the next section. The main point is that a large amount of the required processing of the Scheduler is now performed as a short serial process followed by a parallel and distributed phase.

The distributed aspect of partitioned scheduling also solves the memory usage problem, dividing the required allocation among a large number of nodes. It improves other scale driven concerns which although less important, contribute in a measurable way to the overall system performance. For example, in the basic design of the prior section, the JO that is distributed to all nodes is very large and each node must read it into memory and scan it to find the data specific to that node. With a much smaller object optimizations in the processing and distribution of the JO transmission and handling are possible as described in Section 5.

Finally, we note that even the remaining sequential process of selecting the nodes which remains on the CM can be parallelized to leverage the multi-core hardware. The challenge in achieving a linear speedup in using a modest number of threads is to avoid locking conflicts on the per node data. Section 4.2 describes our multi-threading optimizations of these scheduling stages.

4 Partitioned Scheduling

The main observation that leads to a reduction in both processing and memory consumption in the Scheduler is that the node local resource assignment to each task can be performed in parallel by the compute nodes that are selected for the job. In order for the Scheduler to know job tasks can be dispatched to a node based on availability of sufficient node local resources, the RM only has to keep a count of total and free local resources for each node.

This motivates several new architectural aspects of LL, enhancing the base model of Section 2. The most significant is the partitioning of the scheduling function of per task local node resources to individual nodes. While not as broad in scope, the multi-threaded job scheduler improves scalability for large clusters and jobs. Finally, introduction of pipelining (in Section 5) to the distribution of JO and NTBLs has a significant impact on total schedule and dispatch timing.

4.1 Partitioned Scheduling of Local Resources

In the partitioned scheduling design, a scheduler agent (`Schd Agent`) is introduced to `StartD` to perform the detailed assignment of node local resources to tasks of job scheduled on the node, as shown in Figure 3. In the example of

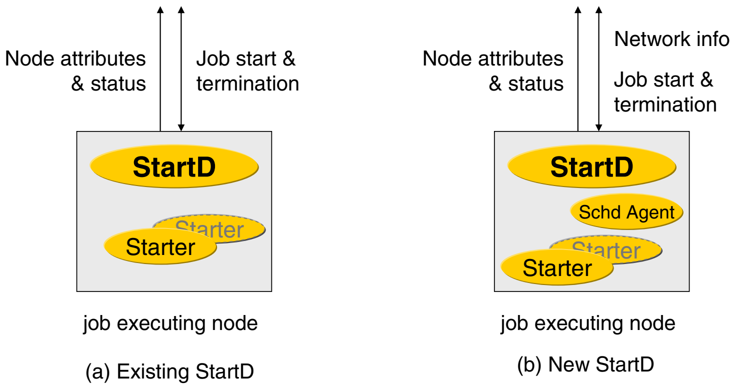


Fig. 3. Designs of StartD functions

network window scheduling, only the static attributes of the network adapters and the count of available network windows are reported by `StartDs` to the RM. During the scheduling cycle, Scheduler matches only the available count of windows and the type of network adapters to the job request.

The majority of the node local resource scheduling assignments made by the Scheduler Agents are propagated to the JM to persist complete job information for failure recovery. Also, to enable all-to-all inter-task communication using network windows, these assignments need to be distributed to all other nodes.

In the new model the JO is first distributed to the centrally selected job nodes using the hierarchical tree of Figure 4. Upon receiving the JO, each node schedules local resources including network windows and starts the process of building the entries for the NTBL (Figure 5). Here, LL uses a variant of the well known *all_gather* implementation in which the unique information (adapter window) from each node is sent up the tree and consolidated at the root, after which the full set of information is distributed to all nodes through the same tree. This process starts from the leaf nodes where the assignments are passed to the parent and merged with those of its siblings. This continues up the tree as each node builds a partial NTBL from its own assignment and those of its child subtrees and so on up the tree, as shown in Figure 5. One optimization is that instead of passing all NTBL data to the root node, building a full NTBL and then sending the full NTBL down the tree to all nodes, each parent (including the root) sends the NTBL for each child subtree down the sibling subtrees. This reduces by half the amount of data exchanged to build the NTBLS for a binary tree.

The potential downside to the partitioning scheduling model is this necessity to communicate the detailed local scheduling decisions such as NTBLs to the all job task peers. If this distributed building of the NTBLs is significantly slower than the centralized in memory creation at the JM and subsequent distribution, then no net gain in scalability is achieved. In other words, based on network latency and bandwidth and processor speed at the CM, there is a cluster size

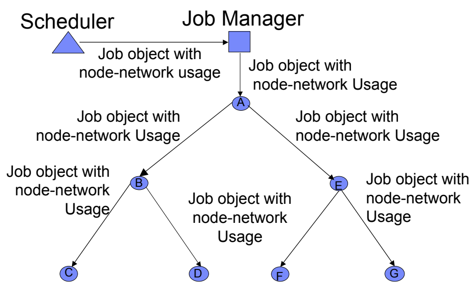

Scheduler
Job Manager
Job object with node-network usage
Job object with node-network Usage
Job object with node-network Usage
Job object with node-network Usage
Job object with node-network Usage
Job object with node-network Usage
Job object with node-network Usage
A
B
C
D
E
F
G

Fig. 4. Job & Network Usage Distribution

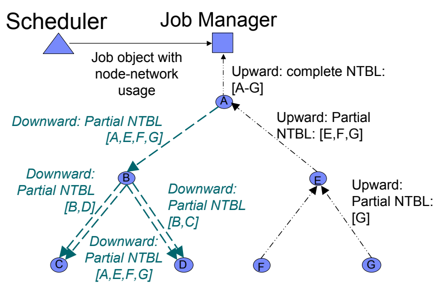

Scheduler
Job Manager
Job object with node-network usage
Upward: complete NTBL: [A-G]
Downward: Partial NTBL [A,E,F,G]
Upward: Partial NTBL: [E,F,G]
Downward: Partial NTBL [B,D]
Downward: Partial NTBL [B,C]
Upward: Partial NTBL: [G]
Downward: Partial NTBL [A,E,F,G]
A
B
C
D
E
F
G

Fig. 5. NTBL Building & Distribution

at which performance will be better in central scheduling as opposed to this partitioning model. We have found with modern hardware that even a modest cluster size of 248 nodes sees substantial gain from the changes, even as the number of tasks per node approaches unity.

4.2 Multithreading in Resource Matching

Scheduling is typically compute intensive and it is performed in a single thread of execution. Single threaded execution fails to exploit the multi-core design of current generation nodes. Therefore, it is redesigned here to leverage multicore

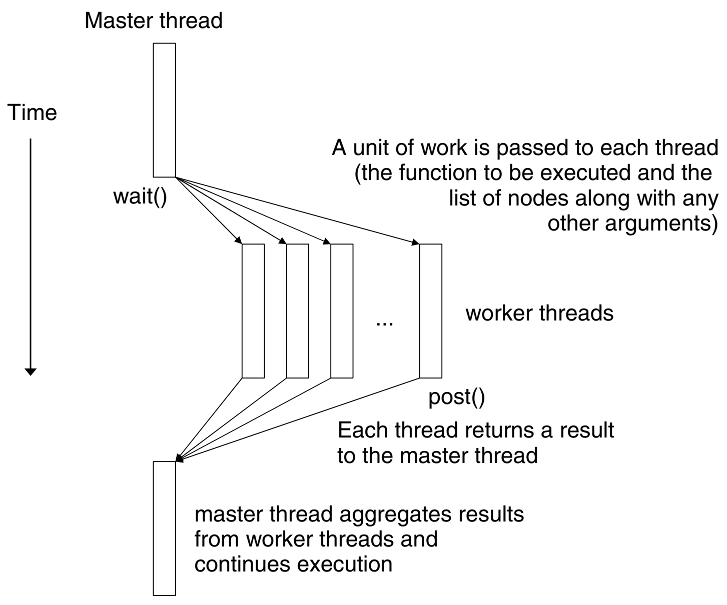

Master thread
Time
A unit of work is passed to each thread (the function to be executed and the list of nodes along with any other arguments)
wait()
...
worker threads
post()
Each thread returns a result to the master thread
master thread aggregates results from worker threads and continues execution

Fig. 6. Application of multi-threading to Scheduling

and SMT capability using a master-worker design with multiple worker threads, as shown in Figure 6. The master thread assigns units of work, such as perform matching and selection from a block of nodes, and the workers pull the work from a queue. As each worker operates on a subset of the node list, it generates new data that must be reflected in global counters, for example, the number of software licenses consumed or the number of nodes successfully scheduled. These updates are handled by passing that data back to the master thread upon thread completion. The master is also responsible for error recovery and re-dispatching work from failed threads. This design ensures there are no potential access conflicts between threads to avoid the overhead of lock contention and lock recovery for failed threads.

The improvement of the multithreading design is borne out by the data showing the total time spent in matching and in selection process reduced from 14 seconds to 4 seconds on a four core system for 10K nodes and 32 tasks per node. All of the experiments reported in the evaluation Section 6 include this optimization.

5 Pipelined Communication

Once a job is scheduled, the job specific resource information for all tasks is encapsulated in the Job Object (JO) which is distributed to the assigned compute nodes using the n-ary tree of Figure 4. Each job has a unique tree which is dynamically constructed from a structured list of nodes contained in the JO and organized by parents and children starting at the root. The tree is persistent for the job life-cycle and used for all subsequent LL communication for that job, both down and up the tree.

There are two modes of JO distribution down the tree: store-forward, and pipelined. In store and forward, a message is completely received at a parent node prior to forwarding to the children. The original design of the communication tree attempts to minimize data sent down the tree and always uses store and forward for messages with node dependence such as the JO. In this design the JO is opened at each node and new JOs are constructed specific to the sub-tree of each child, each trimmed JO being approximately half the size of its parent for a binary tree. In between the store and forward operations, a starter process is initiated to begin job launch on the node. Even though the starter is on its own thread, it is executing during the store and forward and is therefore on the critical path for messaging.

However, through performance analysis it is determined that the CPU time required for message trimming dominates the network transmission times [12]. The trimming is complicated as JO are XDR [2] encoded messages to support communication between heterogeneous systems. Encoding and decoding XDR is compute intensive. This is true even for fast CPUs and slow networks (100Mb Ethernet) and the gap between the processing bottleneck and network communication time widens with higher speed networks. Since processing time is the bottleneck, it is preferable to send larger messages to other nodes as fast as possible so they can begin message processing in parallel. Thus, a second communication mode, pipelining, is added. In pipelining, the JO message is logically divided into 'chunks' and a node receiving data from its parent starts forwarding to its children upon receipt of each chunk. The XDR processing is performed concurrently with the pipelining by a separate thread on the parent node without interference to the communication threads connected to the child nodes.

The immediate forwarding of partial message data in pipelining greatly reduces the latency, with leaf nodes beginning to receive the JO after approximately $log_2(N)$ chunks have been sent from the root to its immediate children. Pipelining achieves almost O(constant) scaling for a large message, where store and forward would be $log_2(N)$ times the total transmission time. This is already an order of magnitude improvement with a 1024 node job. Pipelining also benefits NTBL distribution where it is used to send the partial NTBLs from a tree root to leaf nodes. More details of the performance gain and its overall contribution to the dispatching budget are given in Section 6.

6 Evaluation

In this section, we present our evaluation of the partitioned scheduling and optimizations discussed in the previous sections. We discuss the experimental setup, performance improvements in the memory growth at JM, scheduling time improvements at the CM, and the job dispatching time improvements.

We report on results from two versions of LoadLeveler: 1. a baseline version before our enhancements (V3.5.1.0), and 2. a new version with our design for partition scheduling and enhancements described in this paper (V4.1.0.1) [5]. We refer to V3.x as "Old" and V4.x as "New" design in the description below. We present the data from the old design to substantiate our motivations outlined in the introduction section. We juxtapose this with the data from new design – not so much to show the limitations of the previous design – but to highlight the potential improvements and the trends associated with the performance of the new design in job scheduling and dispatching operations on large scale systems.

6.1 Experimental Setup

The design and implementation work spanned over two years and in this time we had access to multiple clusters with different hardware and operating system configurations. We used different clusters for experimental validation of the different components of new design. In this paper, we report on the results from two representative large scale Power6 clusters: one 497-node cluster and another 249-node cluster.

The 497-node Power6 cluster is connected with Infiniband interconnection network. This cluster is used to obtain the experimental results for the memory usage analysis at the Job Manager, and the partitioned scheduling design that trades off some of the functionality from CM and JM to the compute nodes. Of the 497 nodes, one node is used for CM and JM and the remaining 496 nodes for job execution (compute nodes). The node with CM and JM consists of 8 Power6 cores and 32 GB memory, which provides sufficient memory space to study the memory usage with the old and new designs. Each of the 496 compute nodes consists of 4 Power6 cores and 14 GB memory. So, the cluster consists of 1980 Power6 cores for job execution and 8 cores for LL operations. All 497 nodes consist of a single network adapter per node. All of these nodes are configured with AIX 6.1 and used exclusively for the performance evaluation.

The 249-node Power6 cluster also connected with InfiniBand interconnect is used to study the job dispatching optimizations. In this cluster, the CM and JM node consists of 8 cores and 32 GB memory and the remaining 248 compute nodes consist of 4 cores and 14 GB memory each. All 249 nodes are configured with SUSE Enterprise Linux Server (SLES) version 11.

6.2 Job Manager: Memory Consumption Data Capture and Analysis

In this section, we study the JM memory consumption for jobs requiring 496 nodes with different number of tasks per node: from 1 task per node to 128 tasks per node. We study memory consumption with different number of tasks per node because the amount of state information is directly proportional to the number of tasks assigned to a node. Applications that exploit hybrid MPI/OpenMP programming models tend to use different number of MPI tasks per node depending on the MPI and OpenMP parallelization characteristic of the application. The 128 tasks per node is interesting because it corresponds to an extreme end execution model on Power7 node: typical Power7 compute nodes consists of 32 cores and with SMT= 4, 128 hardware threads therefore a single Power7 node could potentially execute 128 MPI tasks one per hardware thread.

As we discussed before, the memory growth occurs at the JM as well as at the CM. At the CM, the memory is consumed not only for task scheduling but also for other activities such as RM which processes incoming traffic of status and utilization updates so measuring the memory consumption for task scheduling cleanly at CM is a bit tricky. However, the memory consumption at the JM is only due to task scheduling and there is no other memory consumption as long as the task queue is kept constant. So, since both CM and JM consume proportionate amounts of memory for a given job and since it is easier to isolate and measure memory consumption per job for JM process compared to CM process, we capture the data from JM process while scheduling jobs to the 496-node cluster.

For each test case the JM is restarted and initial memory usage by the JM process $M_{initial}$ is recorded before any jobs are submitted to the cluster. Then a job is submitted to JM, is sent to the CM for resource matching. After the resource matching it will be returned to JM at which point, the JM will dispatches the job. The memory usage at this point is recorded again for the JM process as M_{final}. The difference between these two memory usages: $M_{job} = M_{final} - M_{initial}$ is the memory consumption by the JM process associated with a single job. These tests are repeated for LL old and new designs for comparison.

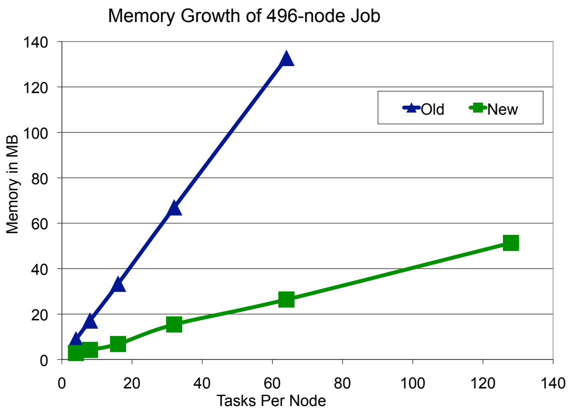


Fig. 7. Job Manager Memory Growth

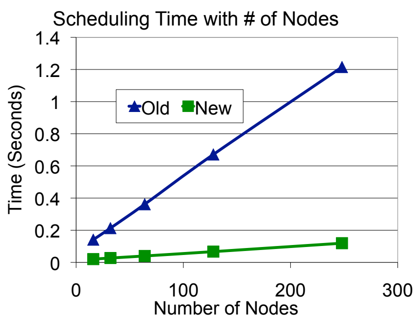
Figure 7 shows the JM memory usage for a 496-node job with increasing number of tasks per node from 1 to 128. For the old design, the data is captured for up to 64 tasks per node. From the data, it is obvious to see that the new design results in substantial memory saving at the JM node – in fact about 5.5x lower memory usage in the new design compared to the old design. Recall that this cluster had a single network adapter while HPC clusters can typically have two or more adapters and the memory consumption in the old design increases linearly with the number of adapters while it increases only by a constant with the new design. The memory savings will be even more substantial for systems with multiple network adapters.

Now, what would be the JM memory for original and new design for the largest scale Power7 HPC system with 16K nodes? A linear projection of the data from Figure 7 from 496 nodes, 64 tasks per node to 16K nodes give us about 4.6 GB per job with old design and 0.83 GB per job with the new design. Keep in mind that job scheduling not only depends the memory usage but also on memory allocation/de-allocation time.

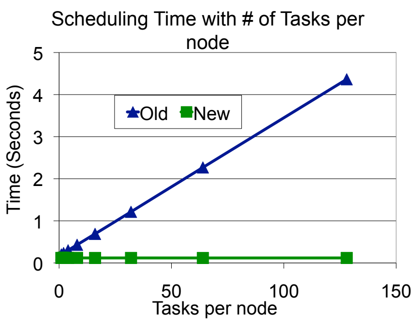
6.3 Central Manager: Job Scheduling Time Capture and Analysis

Job scheduling time is defined as the time between when a job request arrives at the CM from JM and when it is ready to be sent back to the JM with the task resource assignments. In this time, eligible resources are identified, allocated, and the job object is created with the assigned resource information in the Scheduler. The scheduling time does not include the job dispatching time, which is the time between when the job arrived at JM and when the JM gets notification from all compute nodes that they ready to execute the job. All evaluation measurements from our experiments were taken after the clustered system initialized to the idle state, i.e., they system is up and operational but it has no workload.

Figure 8(A) shows the scheduling time for jobs requiring different number of nodes with 32 tasks per node. Significant improvement in scheduling time is



(A)



(B)

Fig. 8. Partitioned scheduling performance evaluation. (A) Scheduler scheduling time per job with different number of nodes. Each job requests resources for 32 tasks per node. (B) Scheduler scheduling time for a 248-node job with different number of tasks.

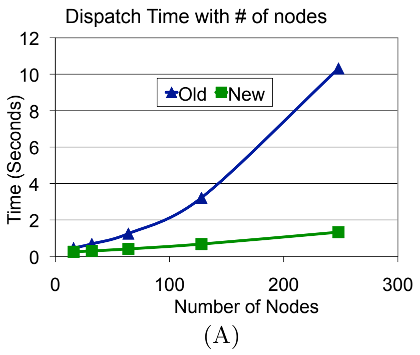
noticed for jobs running under the LL new design. There are two contributing factors to the improvement. First, the Scheduler is relieved from matching node local resources to individual task of the job, as described in Section 4.1. Second, the Scheduler is exploiting the multi-threading enhancements on Power6, as described in Section 4.2.

Figure 8(B) shows the scheduling times for 248-node job with varying number of tasks per node from 2 to 128. In the old design, because task specific resources are allocated at the Scheduler, the scheduling time increases linearly with the number of tasks per node. In the new design, this time becomes a constant because this task specific matching is offloaded from the Scheduler to the compute nodes. The data here further confirms this advantage of relieving Scheduler from matching node discrete resources to job tasks such that the scheduling time is independent with respect to number of tasks per node in the LL new design, as comparing to the increasing in scheduling time with increase number of tasks per node in the original design.

Data in Figures 8(A) and 8(B) together shows that the centralized scheduling performance can be improved significantly by carefully partitioning the resource matching between the Scheduler and the compute nodes. Thus partitioned scheduling can achieve performance and scalability while avoiding scheduling complexity and load balance issues associated with distributed scheduling.

6.4 Job Dispatching Time Analysis

Our partitioned scheduling moves the network table construction from JM to compute nodes. Network tables are constructed among the compute nodes by exchanging the node specific information as discussed in Section 4.1. Communicating node specific information requires exchanges between job nodes to disseminate the local adapter window assignments to all other nodes. In this section, we will analyze the job dispatching time of the old and new designs.



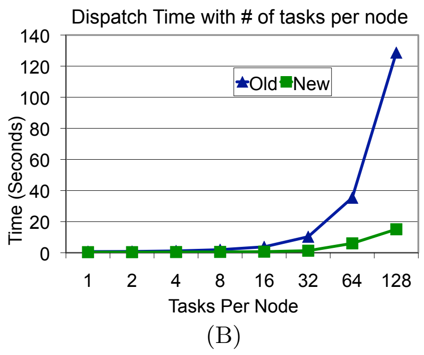


Fig. 9. A. Job Dispatch time as a function of number of nodes. B. Job Dispatch Time as a function of number of tasks node of a 248 node job.

The job dispatching time for the new design includes: 1. the time to send job object from JM to all compute nodes, 2. the time the for Scheduler Agents allocating network resources to individual local tasks, 3. the time to building partial Network Tables for local tasks, 4. the time to hierarchically build the partial network tables and distribute them to all nodes, and 5. the time to start the MPI master task.

Similarly the job dispatching time for the old design only includes steps 1 and 5 of the new design, i.e., the time to send job object including network tables from JM to the compute nodes, and the time to start the master task.

Figure 9(A) shows the job dispatching time as a function of number of nodes requested by the job for the old and new designs. In this example, the job requests 32 tasks per node. As shown in the figure, the dispatch time for the old design grows approximately as a quadratic function whereas the scaling improves to a linear function of the number of nodes in the new design. For a 248 node job, job dispatching improves from over 10 seconds in the old design to under 2 seconds with the new design. Similarly, Figure 9(B) shows the scaling of job dispatch time with the number of tasks/node for a 248 node job. Here, the job dispatch time improves from 130 seconds to under 20 seconds at 128 tasks per node. Although the data shows quadratic growth it is bounded as the number of tasks per node used in practice is unlikely to exceed 128.

These substantial improvements in the new design – despite the extra steps associated for the network table construction and dispatching – are attributed to efficient exploitation of parallelism in allocating and distributing network resources by the agent schedulers at the compute nodes.

7 Job Dispatch Optimization: Pipelining

Although the new design achieves a 5X speed-ups in job dispatching time and fundamentally alters the quadratic scaling to linear, the dispatch time is still

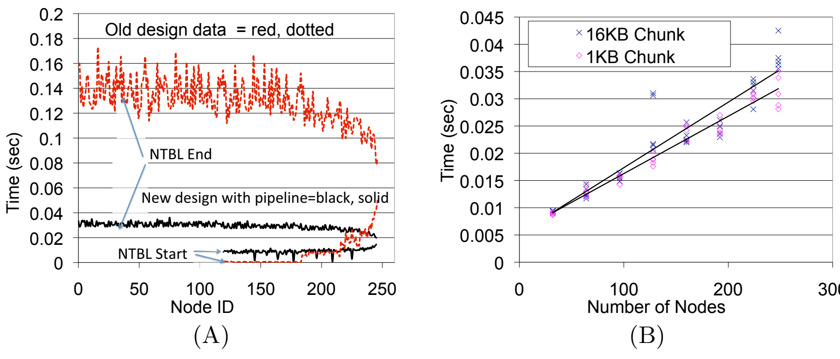


Fig. 10. A. Communication Performance for Network Tables. B. Pipeline tuning for NTBL distribution, no data size scaling.

deemed excessive, especially as we project it to a 16K node system with 32 to 128 tasks per node. To further optimize the dispatch time we implemented additional optimizations, some just restructuring the existing code and others to more efficiently exploit system resources. We report on the performance of the pipelining optimization in dispatching the job object and network tables. In the figures below "new" refers to the partitioned scheduling design which uses the store-forward method and "pipelined" refers to the optimized partitioned scheduling with pipelining.

7.1 Impact of Pipelining on Network Table Dispatch

Figure 10(A) shows the start and completion of Network table (NTBL) construction on a per node basis for a binary tree. The horizontal axis is the node number where the nodes are ordered by level in the tree with the leaf nodes first, then the level above the leaf nodes, and so on. Since the tree has 248 nodes, nodes 1 thru 120 are leaf nodes, while the immediate children of the root are at the far right, and the root is not shown. The vertical axis is the time that the NTBL is received at that node. The new design data set is colored as black and in solid line, while the old design data is in color red and in dotted line.

Since the partial tables are built on the way up the tree and the full NTBL is sent down the tree, each data set contains two lines corresponding to the direction of data flow: 1) A lower line for the time a non-leaf nodes receive the partial tables from its children and 2) An upper line (later time) when that node receives the combined network tables for the other sub-trees from the root as they are passed back down the tree. Since leaf nodes don't receive partial NTBLs on the way up, they only have one value and do not appear on the lower trace of each data set. The gap at the far right measures the time it takes for the immediate children of the root to forward the NTBLs for their subtree to the root, and for the root to turn around and start passing the NTBL for the down complementary sibling's subtree. Thus, time on the upper curve is when

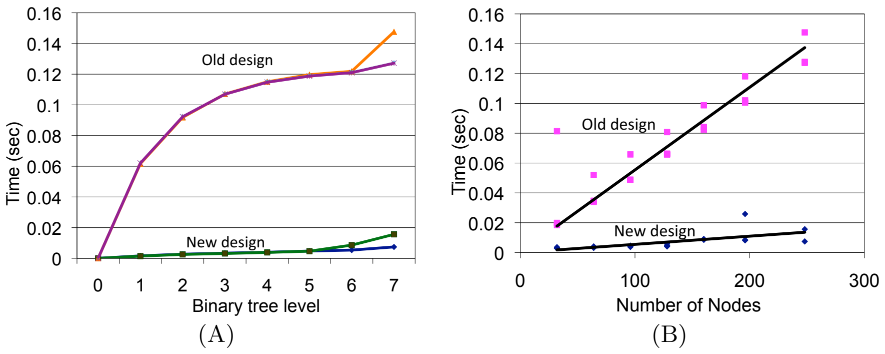


Fig. 11. A. JO distribution with depth of tree, B. JO distribution with number of nodes

that node receives the complete NTBL and the latest time on this trace marks completion of the distribution of NTBLs. This type of per node data chart is used to analyze performance in addition to general trends to show variability in the communication chain.

The pipelined communication causes NTBL distribution to complete about 5 times faster than the store and forward approach. The coefficient of variation in the upper trace is also reduced indicating better predictability and stability in the new system, an important consideration in scaling to 16K nodes.

Detailed performance data for pipelined distribution of NTBLs is provided in Figure 10(B). It shows the time to complete NTBL construction and distribution on the 248 node scale test cluster with different chunk sizes in the pipelining process. There are 32 tasks per node so the NTBL size is of the order of 200KB. The data series labeled 1KB and 16KB reflect the chunk size of the pipeline, i.e., the size of the message received at a node before starting to forward. The effect of the chunk size is noticeable as the scaling is much better with 1KB chunk size as expected by the design. Small chunk size benefits the pipelining design, however, keep in mind that smaller chunks increase number of communication steps so there is a trade off in the chunk size selection that must be considered carefully depending on the job size in terms of the number of nodes and the number tasks per node.

7.2 Impact of Pipeline on Job Object Distribution Performance

Figures 11(A) and 11(B) show the JO distribution using store-forward and pipeline schemes. In Figure 11(A) the horizontal axis is the depth of the binary tree used in the 248 node test cluster. The vertical axis is the time for the JO to be received at that node. In the data series labeled old design the store-forward communication mode is used. The inter-arrival time between each tree level decreases, as depth increases because the JO is trimmed by half and takes less time to process. Here the JO is about 200KB, and LL can drive the InfiniBand network at about 100MB/s to 200MB/s so the raw network time is around 1-2ms. Figure 11(B) compares the two communication modes as a function of the size of the parallel job. The new design curve uses pipelining and provides a nearly linear result, showing a greatly improved scaling in comparison with the original design of store-forward and achieving almost 7x improvement in JO dispatching.

8 Related Work

Decentralized or distributed scheduling has been proposed and evaluated for a long time [15]. [16] used the hierarchical scheduling as a way to aggregated resources from lower level to higher level in making scheduling decision. [14] provided a comprehensive taxonomy of distributed scheduling and touched upon the cooperative peer schedulers. However, the concept of partitioned scheduling and the way it is used in the paper has not been proposed to the best of our knowledge. The partitioned scheduling is a cooperative scheduling of node discrete resources after the centralized scheduling of node to a job is done.

Frachtenberg et al. [18] present Flexible Coscheduling that classifies processes based on their communication and computation requirements to increase system utilization by improving the job compaction in gang scheduled systems. It is a hybrid scheduling algorithm in the sense that there is a global scheduler for the system and a local scheduler per node. The local scheduler makes decisions on when to run a local task beyond its allocated time slot. In contrast, the LoadLeveler local scheduler schedules local resources for the task and alleviates the burden from the global scheduler to improve scalability of the scheduling system.

The terms partitioned scheduling [10] or semi-partitioned scheduling [7,19] are used in real-time scheduling community, in contrast to global scheduling, for scheduling job tasks in multiprocessing systems. More specifically, the author's *partitioned scheduling* refers to having job tasks assigned to specific processors with a system, and then executed on those processors without migrations. The use of partitioned scheduling in real-time discipline is similar in concept to distributed scheduling [9] as the parallel job community. Our use of partitioned scheduling is different, and we refer to the scheduling assignments of node discrete resources to job tasks.

The paper of Bobroff [12] presented a lightweight virtualization approach of LoadLeveler and applied to study the scalability of job scheduling and dispatching in large scale parallel systems using a modest number of physical nodes. The results provided insights on scalability issues and inspired the re-design of LoadLeveler product.

Balaji et al. [11] and Butler et al. [13] describe API's for scalable process management of parallel jobs in large scale systems. A parallel job scheduler has two primary roles: one to match parallel jobs with required resources and two to manage the life cycle of the parallel jobs including launching of the tasks of the job and providing for inter-task communication. The second part is broadly called resource management. The API's enable a unified resource management.

However, to fully exploit systems with unique hardware features, these API's could result in limitations. IBM systems have proprietary resources and they provide resource management software so jobs can exploit these resources. The LoadLeveler process manager coordinates the fine grained resource scheduling such as DMA allocations with the underlying IBM hardware platform resource manager. Layers such as IBM Parallel Operating Environment [3] provide the additional mechanisms for process boot strapping.

Although performance data of the old version of LoadLeveler showed significant scalability problems, it is used on several large scale systems such IBM BlueGene without these problems. Part of the reason for this is that the LoadLeveler design on IBM BlueGene works differently from how it works on general purpose large scale systems. For example, there are no LoadLeveler components that actually run on every node of the BlueGene system, which means it does not have to scale with the number of nodes of the system (see e.g., [8]). More importantly, each node of the DARPA HPCS system consists of many more resources compared to nodes of current generation HPC systems which cause the increased complexity and result in scalability issues for LoadLeveler. Our solutions in this paper address this complexity by carefully dividing the scheduling responsibilities between the global and the local scheduler.

9 Concluding Remarks

This paper presents significant enhancements to IBM LoadLeveler to achieve performance and scalability objectives in job scheduling for extreme scale computing systems. A set of architectural changes are introduced that bring distributed scheduling concepts to LL while preserving the benefits of centralized scheduling.

The most important of these is *partitioned scheduling* that performs task to local node resource matching at the job nodes instead of the central scheduler. This reduces the processing and memory footprint at the CM, leverages the ability to use the job nodes in parallel to assist with scheduling, and changes the scaling of scheduling time with tasks per node n from O(n) to approximately O(constant). This outcome is demonstrated on a large system (50K tasks and 128 tasks per node) requiring local scheduling of adapter window assignments, a particularly challenging case because the local scheduling results have to be propagated to the other job tasks to enable all-to-all communication.

Closely related is the observation that trading off increased parallel processing by using job nodes with additional network data transmission is often favorable with modern high speed, low latency networks in large scale clusters. This tradeoff is leveraged in the dispatching phase by sending the full JO containing data for all nodes to every job node so they can each concurrently extract their locally relevant data and begin processing in parallel. In looking to further reduce network time and increase concurrent processing during the JO and NTBL distribution, the underlying LL communication algorithm was reexamined leading to a change from store and forward to pipelining.

Finally, the scheduling phase of the central manager is architected to take advantage of the underlying multicore and multithreaded hardware by avoiding lock contention in the algorithm. This contribution reduced the CM scheduling time from 14 second to 4 second on a four core compared to single core system achieving about the ideal 4X scaling for a job with 10 K nodes and 32 task per node.

This paper presented data from relatively smaller clusters compared to our target system. In addition to measurement we applied modeling and projections to reason about the scalability at target system scale. Based on this, we expect the efficiency gain in the new design would be even more significant than it is shown in this paper. However, this will need validation with real data, and that is a key part of our future work.

Acknowledgments. This material is based upon work supported by the US Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002.

References

1. DARPA High Productivity Computing Systems project, http://www.darpa.mil/IPTO/programs/hpcs/hpcs.asp
2. External Data Represenation Standard, http://tools.ietf.org/html/rfc1014
3. IBM Parallel Environment (PE), http://www-03.ibm.com/systems/software/parallel/index.html
4. IBM Tivoli Workload Scheduler LoadLeveler, http://publib.boulder.ibm.com/-infocenter/clresctr/vxrx/index.jsp
5. IBM Tivoli Workload Scheduler LoadLeveler Version 4.1, http://www-01.ibm.com/common/ssi/rep_ca/5/897/ENUS210-145/ENUS210-145.PDF
6. Adiga, N.R., Alm'asi, G., Aridor, Y., et al.: An overview of the BlueGene/L Supercomputer. In: Proceeding of Supercomputing, pp. 1–22 (2002)
7. Anderson, J.H., Bud, V., Devi, U.C.: An edf-based scheduling algorithm for multiprocessor soft real-time systems. In: ECRTS (2005)
8. Aridor, Y., Domany, T., Goldshmidt, O., Kliteynik, Y., Moreira, J., Shmueli, E.: Open Job Management Architecture for the Blue Gene/L Supercomputer. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 91–107. Springer, Heidelberg (2005)
9. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E.: Effective distributed scheduling of parallel workloads. In: SIGMETRICS, pp. 25–36 (1996)
10. Baker, T.P.: A comparison of global and partitioned edf schedulability tests for multiprocessors. In: Proceeding of International Conf. on Real-Time and Network Systems (2005)
11. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Krishna, J., Lusk, E., Thakur, R.: PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 31–41. Springer, Heidelberg (2010)

12. Bobroff, N., Coppinger, R., Fong, L., Seelam, S., Xu, J.: Scalability Analysis of Job Scheduling Using Virtual Nodes. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 190–206. Springer, Heidelberg (2009)
13. Butler, R., Gropp, W.D., Lusk, E.: A Scalable Process-Management Environment for Parallel Programs. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) EuroPVM/MPI 2000. LNCS, vol. 1908, pp. 168–175. Springer, Heidelberg (2000)
14. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Software Eng. 14(2) (1988)
15. Casey, L.M.: Decentralised scheduling. Australian Computer Journal 13(2) (1981)
16. Chandra, A., Shenoy, P.J.: Hierarchical scheduling for symmetric multiprocessors. IEEE Trans. Parallel Distrib. Syst. 19(3) (2008)
17. Demaine, E.D., Foster, I.T., et al.: Generalized communicators in the message passing interface. IEEE Trans. Parallel Distrib. Syst. 12(6) (2001)
18. Frachtenberg, E., Feitelson, D.G., et al.: Adaptive parallel job scheduling with flexible coscheduling. IEEE Trans. Parallel & Distributed Syst. 16 (2005)
19. Kato, S., Yamasaki, N., Ishikawa, Y.: Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: ECRTS (2009)
20. Prenneis, A.: Loadleveler: Workload management for parallel and distributed computing environments. In: Super Computing Europe, SUPEREU (1996)
21. Rajamony, R., Arimilli, L.B., Gildea, K.: PERCS: The IBM Power7-IH high-performance computing system. IBM J. Res. Dev. 55(3), 233–244 (2011)

High-Resolution Analysis of Parallel Job Workloads

David Krakov and Dror G. Feitelson

School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract. Conventional evaluations of parallel job schedulers are based on simulating the outcome of using a new scheduler on an existing workload, as recorded in a log file. In order to check the scheduler's performance under diverse conditions, crude manipulations of the whole log are used. We suggest instead to perform a high-resolution analysis of the natural variability in conditions that occurs within each log. Specifically, we use a heatmap of jobs in the log, where the X axis is the load experienced by each job, and the Y axis is the job's performance. Such heatmaps show that the conventional reporting of average performance vs. average load is highly oversimplified. Using the heatmaps, we can see the joint distribution of performance and load, and use this to characterize and understand the system performance as recorded in the different logs. The same methodology can be applied to simulation results, enabling a better appreciation of different schedulers, and better comparisons between them.

1 Introduction

The performance of a computer system obviously depends on the workload it handles. Reliable performance evaluations therefore require the use of representative workloads. This means that the evaluation workload should not only have the same marginal distributions as the workloads that the system will have to handle in production use, but also the same correlations and internal structure. As a result, logs of real workloads are often used to drive simulations of new system designs, because such logs obviously contain all the structure found in real workloads. But replaying a log in a simulation only provides a single data point of performance for one workload.

The most common approach to obtaining data for different conditions is to manipulate the log, and run multiple simulations using multiple manipulated versions. As an alternative, we suggest to exploit the natural variability that is inherent in real workloads. For example, if we are interested in performance as a function of load, we can distinguish between high-load periods in the log and low-load periods in the log.

In developing this idea, we first partition the jobs in each log into a small number of classes, according to the load that they each experience. Following the pioneering work of Rudolph and Smith [8], we find the number of jobs in each class and their average performance, in order to create a "bubbles plot" describing the performance as a function of the load. We analyze the characteristics of such plots, and suggest that a higher resolution may be desirable; in particular, the distribution of performance often has a long tail, and thus the average is not a good representative value. This leads to the idea of creating heatmaps that show the full distribution of performance vs. load. Applying this idea to existing logs reveals several phenomena that have not been known before. The heatmaps can also be used to compare the performance of simulated schedulers, and reveal interesting differences between the behavior of EASY and FCFS — and between simulation results and the behavior of the production schedulers as recorded in the original logs.

2 Evaluating Parallel Job Schedulers with Log-Based Simulations

Log-based simulations have emerged as the leading methodology for evaluating parallel job schedulers. The logs used are actually accounting logs, which contain data about all the jobs that ran on some large-scale machine during a certain period of time. Such logs are available from the Parallel Workloads Archive [7], where they are converted into the Standard Workload Format (SWF) [1]. This makes them easier to use, as the simulators needs to know how to parse only this one format.

Given a log, the simulator simulates job arrivals according to the timestamps in the log. As each job arrives, the simulated scheduler is notified of the number of processors it requires, and possibly also of the user's expectation regarding the runtime. It then decides whether the job should run immediately (in the simulated system), or be queued and run later. Job terminations are simulated based on the scheduling decisions of the simulated scheduler, together with the runtime data provided in the log. Finally some overall average performance metric is computed over all the jobs in the simulation, such as the average response time or the average slowdown. This can be associated with the average load (utilization) of the log.

The main problem with the methodology described thus far is that it provides a single data point: the average performance for the average load. But an important aspect of systems performance evaluation is often to check the system's performance under *different* load conditions, and in particular, how performance degrades with increased load. Given a single log, crude manipulations are typically used in order to change the load. These are

– Multiplying all arrival times by a constant, thus causing jobs to arrive at a faster rate and increasing the load, or causing them to arrive at a slower rate and decreasing the load. However, this also changes the daily cycle, potentially causing jobs that were supposed to terminate during the night to extend into the next day, or causing the peak arrival rate to occur at night.

An alternative approach that has essentially the same effect is to multiply all runtimes by a constant. This doesn't change the arrival pattern, but may cause jobs that were previously independent to clash with each other. Worse, it creates an artificial correlation between load and response time, which essentially invalidates the use of response time as a performance metric.

– Multiplying all job sizes (here meaning the number of processors they use) by a constant, and rounding to the nearest integer. This has three deficiencies. First, many jobs and most machine sizes are powers of two. After multiplying by some constant in order to change the load, they will not be powers of two, which may have a strong effect on how they pack, and thus on the observed fragmentation. This effect can be much stronger than the performance effects we are trying to measure [6]. Second, small jobs cannot be changed with suitable fidelity as the sizes must always be integers. Third, when large jobs are multiplied by a constant larger than 1 in order to increase the load, they may become larger than the full machine.

  A variant on this method is to combine scaling with replication. This allows larger jobs to be generated without losing smaller jobs, and has been suggested as a method to adapt simulations to different machine sizes [2].

  An alternative approach that has essentially the same effect is to modify the simulated machine size. This avoids the problem presented by the small jobs. However it may suffer from changing the inherent fragmentation when packing jobs together. Also, when the machine size is reduced to increase load, the largest jobs in the log may no longer fit.

  A possible alternative is not to multiply job sizes by a constant but to change the distribution of job sizes. Consider the CDF of the distribution of job sizes. To increase the load we want more larger jobs. Multiplying job sizes by a constant will cause the CDF to shift to the right, with all the ill-effects noted above. The alternative is to shift the top-right part of the CDF downwards. As a result, the relative proportion of large jobs is increased and the total load increases too. However, the idea of changing the distribution in this way has only received limited empirical support [12], and more work is needed.

Another alternative is to use multiple logs that have different loads. The problem then is that the workloads in the different logs may have completely different characteristics, so comparing them to each other may not be meaningful. Also, the number of available logs and the available load values may not suffice.

The most common approach used is to artificially change the load of a log by multiplying arrival times by a constant as described above, despite this method's deficiencies. Our goal is to find an alternative to this approach.

3 Evaluations Based on the Variability in a Single Run

As an alternative to evaluating a parallel job scheduler using simulations with a job log that has been subjected to various manipulations, we suggest to exploit the natural load fluctuations that occur in any log. In other words, by observing periods of low load separately from periods of high load, we may try to uncover the effects of load on performance. This has the following benefits:

– It is more realistic, because it is based on real load conditions that had occurred in practice when the log was recorded, with no artificial manipulations, and
– It is easier in the sense that a single simulation can be used instead of multiple simulations.

However, it also has its drawbacks. For example, in a given log the range of load conditions that have occurred may be limited. Nevertheless, we feel that this approach is worthy of investigation.

Note that the suggested approach is different from the conventional approach at a very basic level. In the conventional approach, the average performance is found as a function of the average load. This is similar to the outcome of queueing analyses, such as the well-known M/M/1 queue. The suggested approach does not concern itself with different average load conditions. It is actually about understanding the variability and dispersion of performance, and the possible correlation between this dispersion and load — not about performance under different average loads. We discuss this further in the conclusions.

The approach of analyzing a single log and dissecting it according to load conditions was pioneered by Rudolph and Smith in the context of evaluating large-scale systems in the ASCI project [8]. Their goal was to establish whether these machines were being used efficiently. By analyzing workload logs, they attempted to show that performance as a function of load exhibits a "knee" at some load level, and beyond that point performance deteriorates markedly. Then if most of the jobs execute under a load that is just below the knee, the machine is being used efficiently.

The procedure employed by Rudolph and Smith to analyze the logs was somewhat involved. The analysis was performed at the level of individual jobs. For each job, they first found the average system utilization experienced by that job during its tenure in the system (this is explained in more detail below). The jobs were then binned according to the load into deciles: those jobs that experienced around 0 load, those that experienced around 10% load, those that experienced around 20% load, and so on up to those that experienced 90% and 100% load. Then a "bubble plot" was drawn. The X axis in these plots is the load, and the Y axis is the performance metric, e.g. average slowdown. Each class of jobs is represented by a disk. The coordinates of the center of the disk are at the average load experienced by jobs in the class and the average slowdown of jobs in the class. The size of the disk represents the number of jobs in the class.

The load experienced by a job was calculated as follows. The load (or utilization) at each instant is simply the fraction of processors that are allocated to running jobs (due to fragmentation, there are often some unused processors even if additional jobs are waiting in the queue). This only changes when the scheduler decides to start running a job, or a running job terminates. Assume a certain job arrives at time t and terminates at time t'. Consider the set of time instants from t to t' at which any job either starts to run or terminates, and number them from 0 to n (such that $t_0 = t$ and $t_n = t'$). Denote the utilization during the interval t_i to t_{i+1} by $U(t_i, t_{i+1})$. The load experienced by the job is then

$$\text{load} = \sum_{i=1}^{n} \frac{t_i - t_{i-1}}{t_n - t_0} U(t_{i-1}, t_i)$$

(Inexplicably, the instantaneous utilization calculation sometimes leads to values greater than 1, which implies data quality problems in the logs. We discuss such issues in a separate paper [4]; in the current context they are rare enough to be largely meaningless.)

In the following we use the Rudolph-Smith bubble plots as our starting point, and use them to further analyze parallel job logs. But the motivation is not only to understand the performance as it was on specific machines in the past. Rather, we contend that the same analysis can be applied to simulation results. In other words, when running a simulation of a parallel job scheduler on a given workload, a *new* log recording the performance of the simulated system can be recorded. This (single) log can then be analyzed in the same way as real logs are analyzed, to uncover the performance as a function of load.

4 Evaluating Parallel Job Logs with Bubble Plots

Example bubble plots are shown in Fig. 1. The plot for the CTC SP2 log looks approximately as expected: the average slowdown tends to grow from around 10 to about 30 with increased load, and most of the jobs observed what appears to be the maximum sustainable load, which is around 70% in this case. However, the few jobs that enjoyed near zero load suffered a slowdown of around 50, and those that suffered a load of 90-100% enjoyed a slowdown of less than 10. Also, the plots for other logs are messier. For example, the KTH log exhibits a zig-zag pattern, the SDSC Paragon shows marked decrease in slowdown with increased load, and in SDSC Blue the jobs are evenly distributed across all loads.

Rudolph and Smith also observed some strange patterns like this. Part of their solution was to filter some of the jobs in the log. In particular, they filtered jobs that were shorter than 1 minute, and jobs that had a very high slowdown. We also did so (the graphs in Fig. 1 are after such filtering, where the threshold for "high slowdown" was 1000). However, while the strange behavior is reduced, it is not eliminated.

The high slowdowns with low utilizations are somewhat of a mystery. The phenomenon seems to happen because jobs are not scheduled to run while processors are in fact available. Possible excuses are unrecorded down time, when the processors were actually not available but we don't know it, or special scheduler

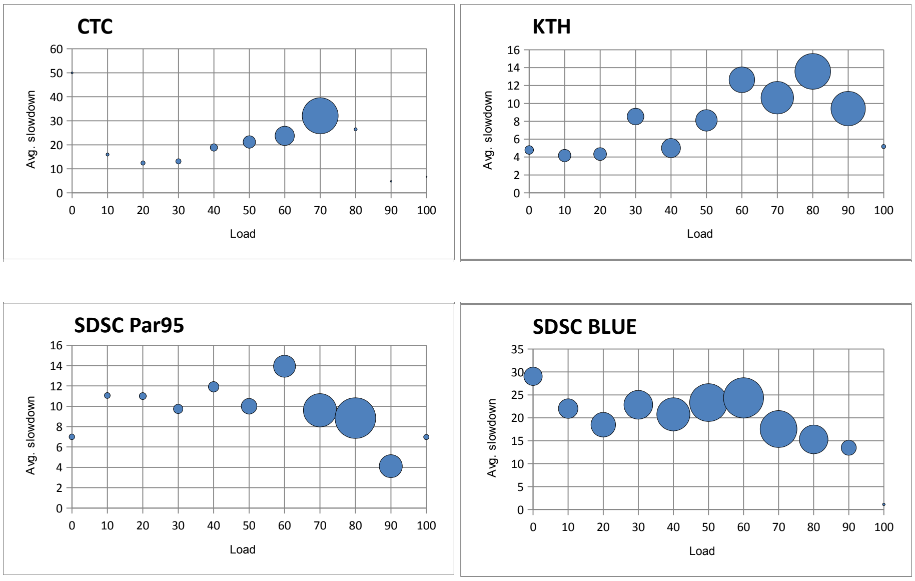


Fig. 1. Examples of bubble plots derived from different logs. Short jobs (< 1 minute) and jobs with high slowdown (> 1000) were filtered out.

considerations, such as reserving processors for some other use. Another possible explanation is that processors are not the only important resource, and perhaps also not the most important one. Thus jobs may be delayed if sufficient memory is not available, or if they need to use a floating software license that is being used by another job. Such considerations do not affect the utilization metric.

The low slowdowns observed with high utilizations can be explained as follows. Consider a sequential job. When such a job arrives, it will be able to run immediately if the utilization in less than 100%. Moreover, when the utilization is high, *only* such jobs will be able to run immediately (because very few processors if any are available). So many jobs that see a high utilization throughout their lifetime can be expected to be serial jobs that run immediately, and therefore have slowdown 1 — the lowest (and best) slowdown possible.

Conversely, consider a large job that requires many processors. Such a job will most probably have to wait until the processors become available. Moreover, it will block other jobs and cause processors to become idle while it waits. Therefore it will see a lower average load during its lifetime, but suffer a high slowdown due to the waiting.

Slowdown is sensitive to short jobs that may have very high slowdown values [11,3]. This may explain the variability in the bubble plots: if in a certain group of jobs one happened to have a very high slowdown, this could affect the average of all of them and cause the bubble to float upwards. Such effects could lead to the uneven behavior seen in Fig. 1. Additional support for this hypothesis comes from looking at the median slowdown instead of the average. The plot of medians and plot of averages turn out to be quite different from each other.

The conclusion is that bubbles may be too coarse, as they represent potentially large groups of unrelated jobs. As an alternative, we suggest looking at all the jobs at a much higher resolution.

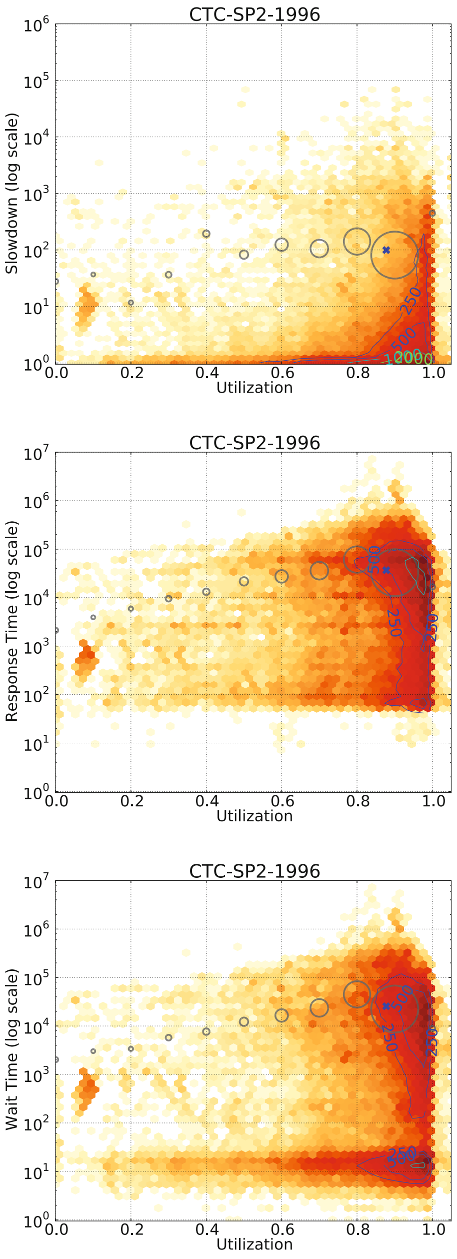
5 Evaluating Parallel Job Logs with Heatmaps

The way to look at the performance vs. load data in more detail is to use heatmaps. The axes remain the same: the X axis represents load, and the Y axis represents performance (this can be slowdown, as used in the Rudolph-Smith bubble plots, but also response time or wait time). But instead of using a coarse classification of loads and lumping all the jobs that saw approximately the same load together, we now use a relatively fine classification according to both load and performance. The number of jobs that experienced approximately the same load and same performance is then represented by the shading: a darker shading corresponds to a higher numbers of jobs. The actual numbers differ according to the log's length, so contours are used to give an indication of the number of jobs in the high-density areas.

Applying this to the different logs leads to the heatmaps shown in Figs. 2 to 9. For each log, heatmaps of slowdown, response time, and wait time vs. load are shown. Note that the Y axis is logarithmic. Slowdowns of 1 are at the bottom of the scale, adjacent to the X axis. As wait times can be 0 and this cannot be shown on a logarithmic scale, we artificially change 0 values to 1, so they too are shown at the bottom adjacent to the X axis. Bubble plots calculated from the same data are superimposed on the heatmaps for comparison.

These graphs allow for several general insights and some specific ones. The main general insights are as follows:

– The average values are unrepresentative. In each heatmap, the point of average load and average performance is marked with an 'X'. This represents the output of conventional analysis. But as we can clearly see, this often falls in a relatively sparse area of the heatmap.
– Likewise, the bubbles, which are shown overlaid on the heatmap, are unrepresentative. (In these graphs the bubbles are based on all the jobs, with no filtering).
– The performance distribution tends to be highly skewed. Many jobs have very low wait times and a slowdown of 1. The higher averages are a result of combining this with relatively few jobs that suffer from much worse performance.
– In many cases the load distribution is also highly skewed. In particular, many jobs actually observe very high utilizations of near 100% as they run. The lower average load is a result of the low-load and idle periods, which actually affect only a small number of jobs.



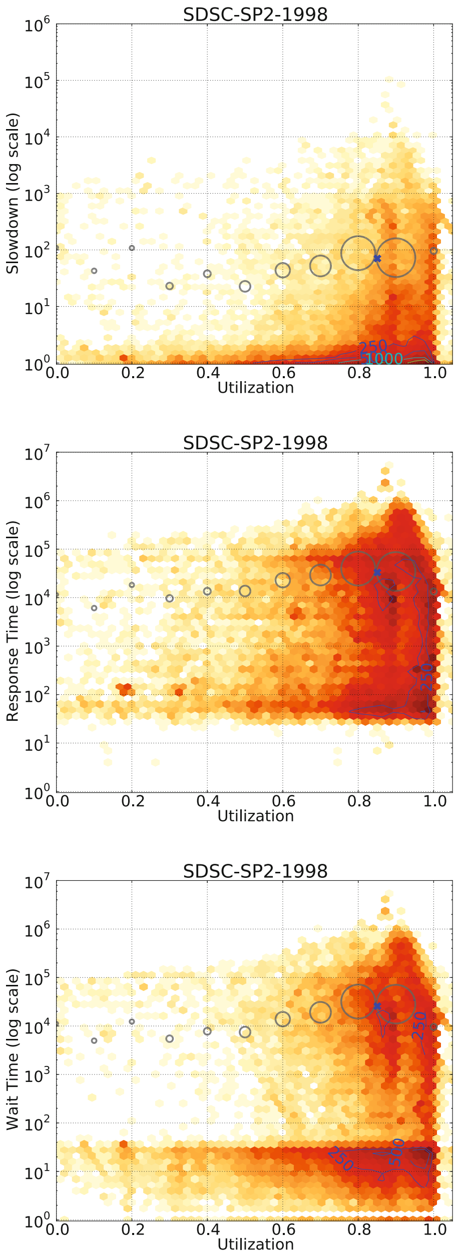


Fig. 2. Heatmaps for the CTC SP2 log **Fig. 3.** Heatmaps for the SDSC SP2 log

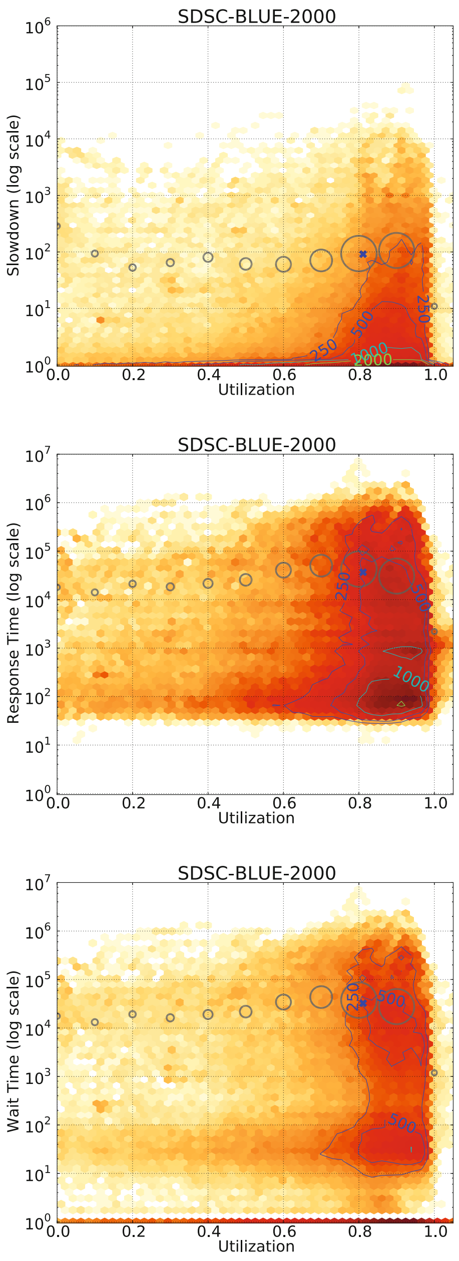


Fig. 4. Heatmaps for the SDSC Blue log

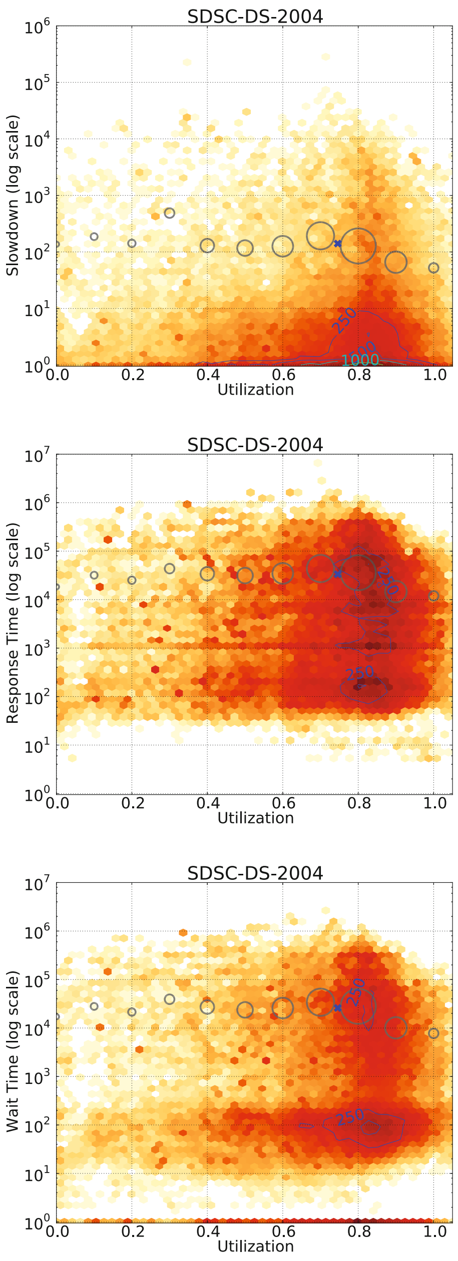


Fig. 5. Heatmaps for the SDSC DS log

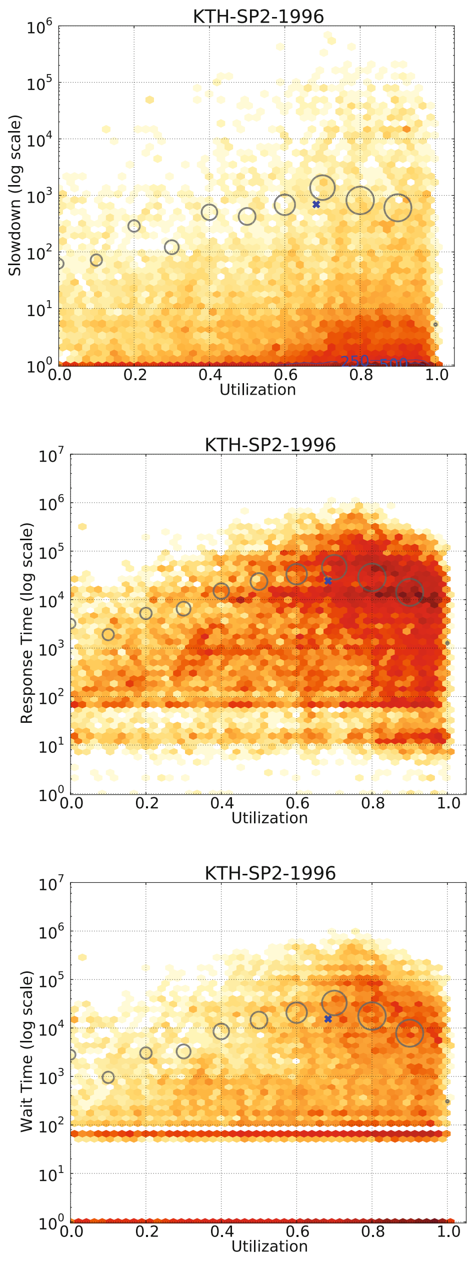


Fig. 6. Heatmaps for the KTH SP2 log

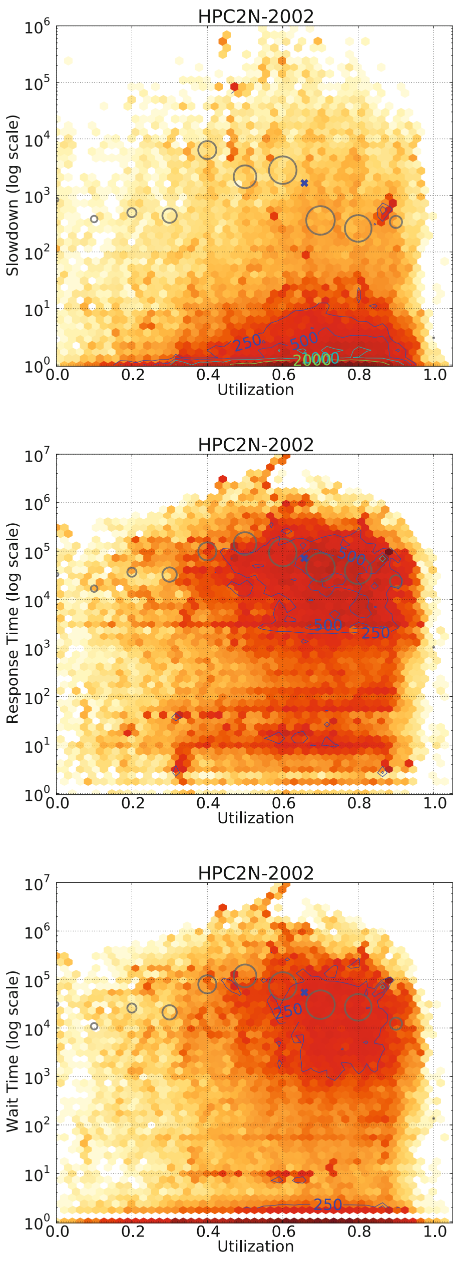
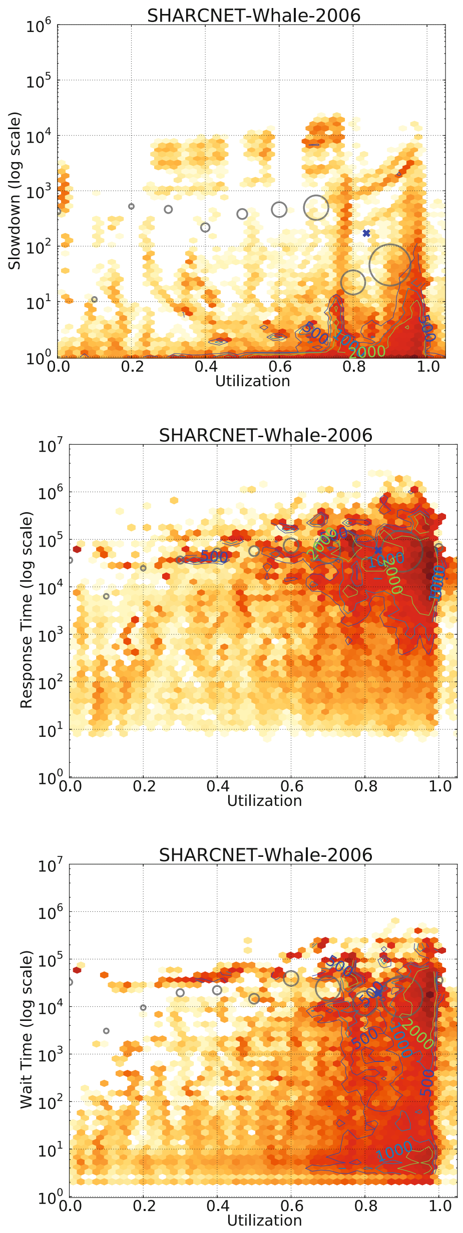


Fig. 7. Heatmaps for the HPC2N log



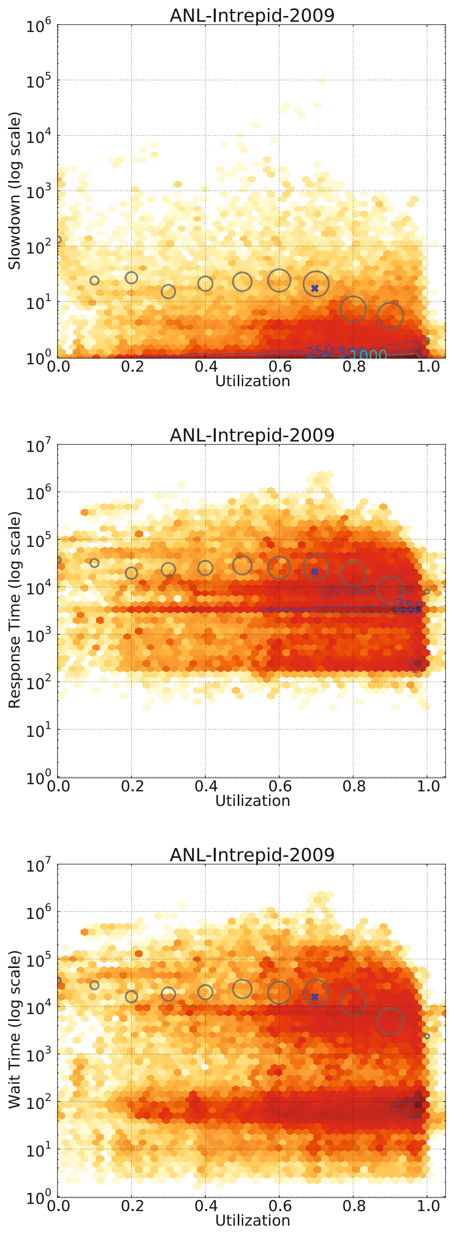


Fig. 8. Heatmaps for the SHARCNET Whale log

Fig. 9. Heatmaps for the ANL Intrepid log

– There appears to be *only a weak if any functional relationship between experienced load and performance*. In other words, it is not generally true that jobs that experience higher loads also suffer from worse performance. It is true that we often see a concentration of jobs at the right of the heatmap (high loads), and that this includes the top-right area (bad performance), but it also typically includes the bottom-right area (good performance) to a similar degree. To quantify the possible correlation of load and performance we calculated the Spearman rank correlation coefficient between them for different logs. The absolute values of the results are typically less than 0.2, and often also less than 0.05, indicating very low correlation (Table 1).

Rudolph and Smith in their paper that introduced the bubble plots were looking for the characteristic behavior of a queueing system: good performance at low loads, that deteriorates asymptotically as the system load approaches saturation. Their success varied; some of the bubble plots exhibited the expected characteristics, while others were rather messy and hard to understand. These results were replicated in our work above.

Using the heatmaps, we can take a more detailed look at performance as a function of load. Focusing on slowdown to begin with, we find that for many logs there is a strong concentration of jobs along the bottom and right boundaries of the plot. The concentration along the bottom represents the jobs that enjoyed the best possible performance, namely a slowdown of 1. This happened at all loads, and dominates low loads. The concentration at the far right represents jobs that suffered from congestion under a high load. in some logs, e.g. CTC, this only happens at near 100% utilization. In SDSC SP2 there seem to be two distinct concentrations, at 90% and at 100%. In SDSC Blue the concentration is near 90%. In SDSC DS, it happens at around 80%.

Interestingly, there were also logs that did not display the expected pattern at all. Examples are the KTH, HPC2N, and Intrepid logs. In these systems we see a wide smear, with no concentration of jobs that experience high loads. Rather, jobs seem to suffer approximately the same slowdowns regardless of load. An optimistic interpretation of this result is that the scheduler is doing something good, and manages to avoid bad performance under high load. As we show below, however, a more realistic interpretation seems to be that the scheduler is incapable (or unwilling) to exploit low load conditions in order to improve performance.

Similar observations may be made for response time. Here we do not see a concentration of low response times under low loads, because response times are more varied due to run times being varied. However, in many logs we do see a concentration of jobs at the right end of the plot, reflecting high loads. These include CTC, SDSC SP2, SDSC Blue, and SDSC DS.

The concentration at low values (indicating good performance) is clearly evident when we look at wait times. Several logs actually have a distinct bimodal distribution of wait times: very short wait times of up to about a minute, and long wait times of many minutes to several hours (note that in all the plots the Y axis is logarithmically scaled). Good examples are again CTC, SDSC SP2, SDSC Blue, and SDSC DS. The short wait times apparently reflect some minimal granularity of activating the scheduler, such that it only runs say once a minute and therefore does not schedule newly arrived jobs immediately [4].

Potentially interesting patterns that probably deserve further study appear in the heatmaps of specific logs. These include

– A distinct blob of jobs at the left side of the CTC heatmap. This is a concentration of jobs that saw very low load, but nevertheless suffered non-trivial slowdowns.
– Strange patterns in the SHARCNET Whale log. These seem to reflect sets of jobs that suffered from some congestion conditions.
– A horizontal band in the Intrepid response time map, that probably reflects many jobs with the same runtime.

6 Comparing Heatmaps of Logs with Heatmaps from Simulations

We also ran straightforward simulations using the EASY [5] and FCFS schedulers on the logs, and compared the resulting heatmaps to the heatmaps produced based on the original log data. Examples are shown in Figs. 10 to 12. The leftmost column in these figures reproduces the data shown previously for the original log. The middle column is the result of an EASY simulation, and the rightmost one is FCFS. The comparison leads to two main observations.

Table 1. Spearman's rank correlation coefficient of performance vs. load, showing much higher values for simulation results than for the original logs

log	*metric*	*data*	*EASY*	*FCFS*
CTC SP2	slowdown	0.17	0.62	0.51
	response time	-0.03	0.13	0.20
	wait time	0.08	0.56	0.43
KTH SP2	slowdown	-0.01	0.55	-0.10
	response time	0.15	0.07	-0.28
	wait time	0.09	0.44	-0.26
SDSC SP2	slowdown	0.16	0.60	0.40
	response time	0.02	0.26	0.28
	wait time	0.11	0.55	0.37
HPC2N	slowdown	0.08	0.72	0.68
	response time	0.05	0.41	0.48
	wait time	0.08	0.70	0.69
SDSC Blue	slowdown	0.04	0.58	0.48
	response time	0.02	0.32	0.39
	wait time	0.00	0.57	0.47
ANL Intrepid	slowdown	-0.05	0.66	0.44
	response time	-0.07	0.32	0.32
	wait time	-0.05	0.63	0.41

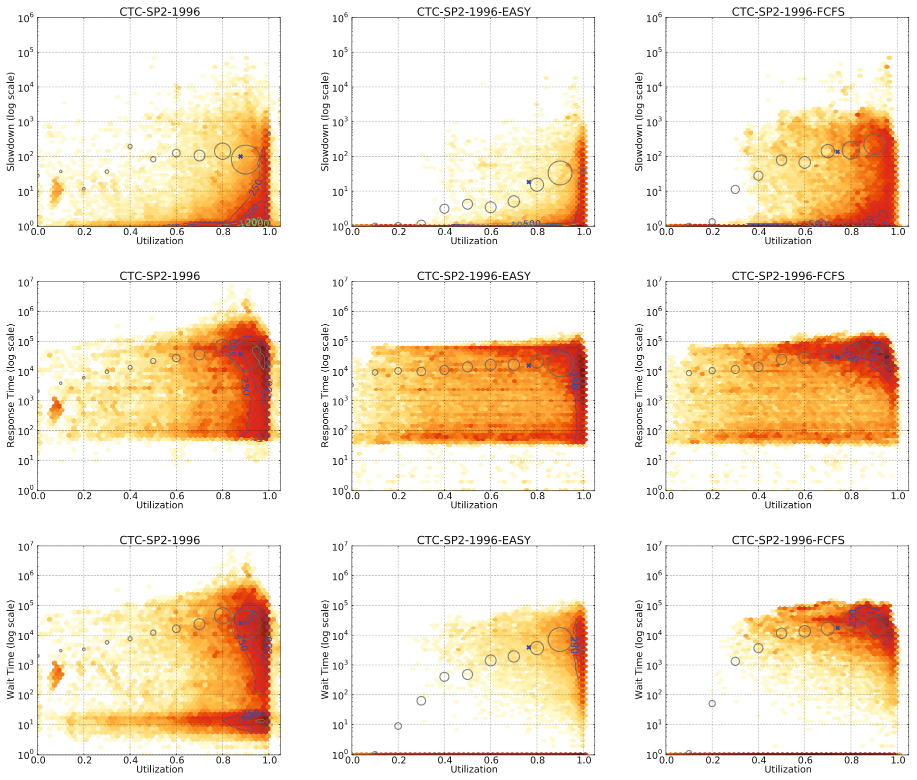


Fig. 10. Heatmaps produced by running EASY and FCFS on the CTC log

– The simulations tend to produce “nicer” results. Specifically,
  * More jobs have a slowdown of 1,
  * High slowdowns and wait times occur only when load is near 100%, and
  * There are no strange patterns.
– The simulations do not reflect reality! It seems that the schedulers on the real systems are often restricted in some way, and cannot achieve efficient packing of the executed jobs. In addition, in the simulations there is a much stronger correlation between experienced load and the resulting performance. With EASY in particular, high slowdowns and wait times are seen exclusively for jobs that suffered from high load conditions. With FCFS this happens to a somewhat lesser extent. Spearman rank correlation coefficients for several logs are shown in Table 1.

In more detail, consider the CTC workload shown in Fig. 10. From the heatmaps it appears that the original CTC scheduler is closer to FCFS than to EASY. But in fact it is even worse than FCFS. Looking at the scale, we find that for EASY simulations the response times are evenly smeared from around 30 seconds to around 80,000 seconds, with rather sharp boundaries. The top limit probably reflects a runtime limit imposed by the system administrators. But in the original

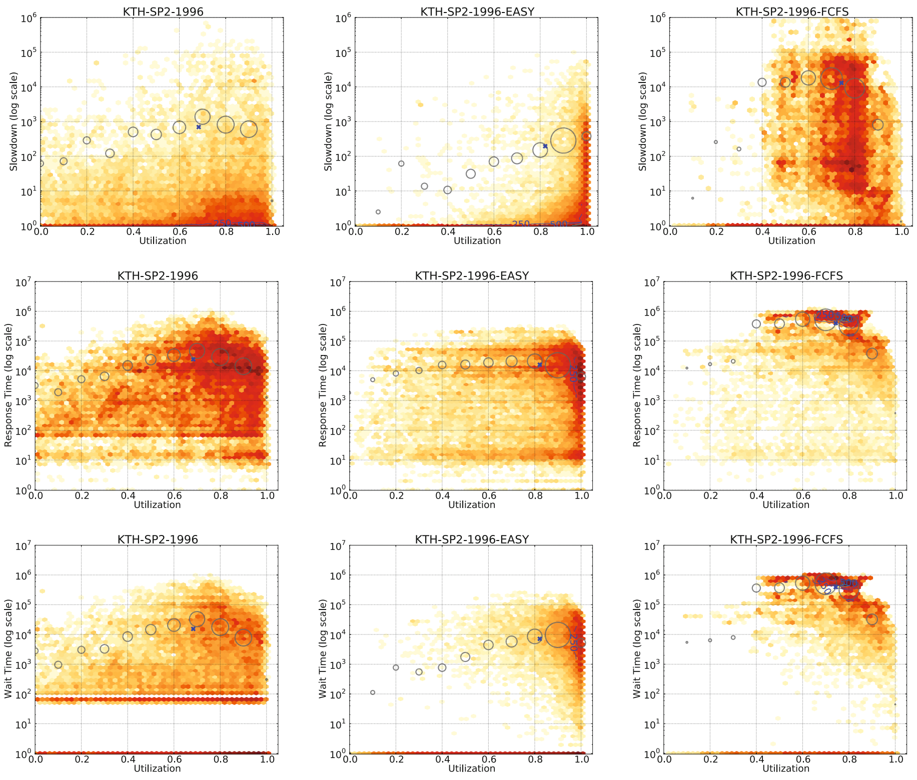


Fig. 11. Heatmaps produced by running EASY and FCFS on the KTH log

log we don't see any such boundary, and response times may be as high as a million seconds.

Looking at the KTH log we see a different picture (Fig. 11). Here it seems that the heatmaps produced from the original log are somewhat more similar to the heatmaps produced by EASY, implying that the original scheduler behaves more like EASY than like FCFS. FCFS produces much higher response times and wait times, and they are all concentrated at the same high values. Regarding slowdowns, EASY produces much lower slowdowns except at the very highest loads.

Another example comes from the HPC2N log, shown in Fig. 12. Here the distribution of response times is approximately the same for both the original scheduler and the simulated ones. However, with the simulated schedulers many fewer jobs see low loads, and most jobs are concentrated at the extreme right, indicating near 100% utilization. In the original log, in contradistinction, they were scattered from about 30% to about 90%.

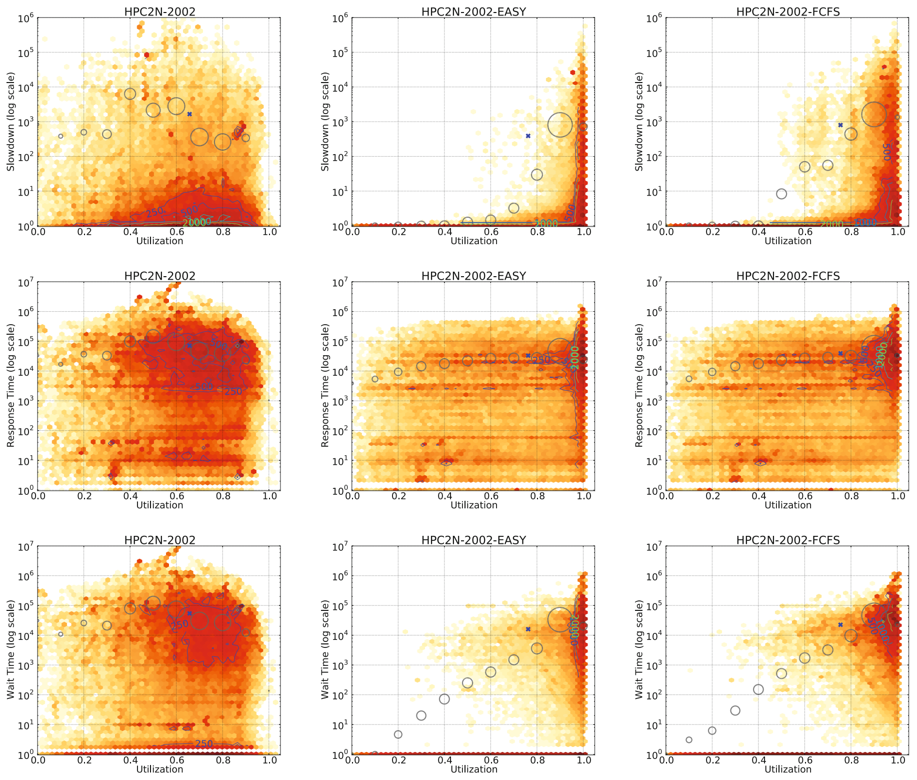


Fig. 12. Heatmaps produced by running EASY and FCFS on the HPC2N log

7 Conclusions

We set out to devise a new way to use data from accounting logs for performance evaluations. The idea was that the long-term load on a system exhibits natural fluctuations, and these can be exploited in order to evaluate performance under different load conditions. This idea can be applied directly to the available logs in order to analyze the system in production use. It can also be applied to the output of simulations, whether driven by real logs or by synthetic workloads.

To implement this idea we use heatmaps, where the X axis represents load and the Y axis represents performance. The shading at each point reflects the number of jobs that experienced this load level and enjoyed this level of performance.

This analysis led to two main outcomes. The first was the observation that our heatmaps expose a wealth of information that has been glossed over till now. In particular, the common practice of reporting average performance as a function of average load seems ill-advised, as both load and performance have skewed distributions. Thus the average values do not reflect system behavior.

The second was the observation that conventional simulations do not reflect what is going on on real systems. Simulations using EASY, and sometimes also simulations using FCFS, produce behaviors that are markedly different and often much better than those observed in the original logs. This seems to indicate that real schedulers employ various considerations that limit their options, and lead to sub-optimal packing of jobs. It is not clear at this point whether this reflects deficiencies in production schedulers, or maybe deficiencies in simulations. It is certainly possible that simulations like those we performed are over simplified, and do not take all the real world considerations into account. For example, real schedulers need to consider memory requirements, software licenses, and heterogeneous configurations, and do not just count processors.

Analyzing the variability in real systems or single simulation runs as we suggest represents a significant departure from current practice. This immediately leads to the question of whether this can indeed be used to gauge performance as a function of load, or maybe it is necessary to actually change the overall average load on the system. In defense of our approach, we note the recent interest in generative user-based workload models. In such models the simulation includes not only the system, but also the processes by which users generate the workload [9,10]. An important element in such models is the feedback from the system to the users. In particular, when performance is bad users may elect to leave the system. Such feedback leads to a self-regulating effect, and may counteract attempts to increase the average load.

In any case, we suggest that evaluations of parallel job schedulers will do well to utilize heatmaps like the ones we produced in order to better understand the behavior of the systems under study. However, this is only the first step. Additional research is needed in order to make better use of the heatmaps. In particular, we suggest the following.

– In our work we interpret "load" as the average system utilization observed by each job (as was done by Rudolph and Smith). This ignores the backlog that may accumulate in the scheduler's queue (except for the fact that a large backlog may cause a job to be delayed in the queue, and therefore the load calculation will cover a longer interval). But it can be argued that the overload represented by this backlog is also an important component of the system load. The question is how to incorporate this information explicitly in the load calculation.
– Our heatmaps enable patterns to be observed for a specific log and scheduler. An important extension would be to find a good way to compare such heatmaps to each other. In particular, is there a good metric for evaluating whether one heatmap represents "better performance" than another?
– It may also be useful to consider subsets of jobs, and draw independent heatmaps for them. For example, this can be done for jobs with a certain range of degrees of parallelism, or jobs belonging to a certain user.
– Finally, the heatmaps may also be used to characterize and evaluate synthetic workload models. By comparing a heatmap representing the behavior of a given scheduler on a synthetic workload with a heatmap of that scheduler's behavior on a real log we can see whether the synthetic workload leads to reasonable behavior.

Acknowledgments. Many thanks to all those who have made their workload data available through the Parallel Workloads Archive. Thanks are also due to the anonymous reviewers who contributed many ideas and observations that improved upon the original paper.

References

1. Chapin, S.J., Cirne, W., Feitelson, D.G., Jones, J.P., Leutenegger, S.T., Schwiegelshohn, U., Smith, W., Talby, D.: Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999. LNCS, vol. 1659, pp. 67–90. Springer, Heidelberg (1999)
2. Ernemann, C., Song, B., Yahyapour, R.: Scaling of Workload Traces. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 166–182. Springer, Heidelberg (2003)
3. Feitelson, D.G.: Metric and workload effects on computer systems evaluation. Computer 36(9), 18–25 (2003)
4. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with the parallel workloads archive (2012) (in preparation)
5. Lifka, D.: The ANL/IBM SP Scheduling System. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
6. 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.) JSSPP 1998. LNCS, vol. 1459, pp. 25–46. Springer, Heidelberg (1998)
7. Parallel workloads archive, `http://www.cs.huji.ac.il/labs/parallel/workload/`
8. Rudolph, L., Smith, P.H.: Valuation of Ultra-scale Computing Systems. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2000. LNCS, vol. 1911, pp. 39–55. Springer, Heidelberg (2000)
9. Shmueli, E., Feitelson, D.G.: Using site-level modeling to evaluate the performance of parallel system schedulers. In: 14th Modeling, Anal. & Simulation of Comput. & Telecomm. Syst., pp. 167–176 (September 2006)
10. Shmueli, E., Feitelson, D.G.: On simulation and design of parallel-systems schedulers: Are we doing the right thing? IEEE Trans. Parallel & Distributed Syst. 20(7), 983–996 (2009)
11. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective Reservation Strategies for Backfill Job Scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 55–71. Springer, Heidelberg (2002)
12. Talby, D., Feitelson, D.G., Raveh, A.: A co-plot analysis of logs and models of parallel workloads. ACM Trans. Modeling & Comput. Simulation 12(3) (July 2007)

Identifying Quick Starters: Towards an Integrated Framework for Efficient Predictions of Queue Waiting Times of Batch Parallel Jobs

Rajath Kumar and Sathish Vadhiyar

Supercomputer Education and Research Center, Indian Institute of Science, Bangalore, India
rajath@ssl.serc.iisc.in,vss@serc.iisc.in

Abstract. Production parallel systems are space-shared and hence employ batch queues in which the jobs submitted to the systems are made to wait before execution. Thus, jobs submitted to parallel batch systems incur queue waiting times in addition to the execution times. Prediction of these queue waiting times is important to provide overall estimates to the users and can also help metaschedulers make scheduling decisions. Analyses of the job traces of supercomputers reveal that about 56 to 99% of the jobs incur queue waiting times of less than an hour. Hence, identifying these quick starters or jobs with short queue waiting times is essential for overall improvement on queue waiting time predictions. Existing strategies provide high overestimates of upper bounds of queue waiting times rendering the bounds less useful for jobs with short queue waiting times. In this work, we have developed an integrated framework that uses the job characteristics, and states of the queue and processor occupancy to identify and predict quick starters, and use the existing strategies to predict jobs with long queue waiting times. Our experiments with different production supercomputer job traces show that our prediction strategies can lead to correct identification of up to 20 times more quick starters and provide tighter bounds for these jobs, and thus result in up to 64% higher overall prediction accuracy than existing methods.

Keywords: Queue Wait Times, High Performance Computing, Batch Systems, Prediction, Scheduling.

1 Introduction

Production parallel systems in many supercomputing sites are batch systems that provide space sharing of available processors among multiple parallel applications or jobs. Well known parallel job scheduling frameworks including IBM Loadleveler [1], PBS [2], Platform LSF [3] and Maui scheduler [4] are used in production supercomputers for management of jobs in the batch systems. These frameworks employ batch queues in which the jobs submitted to the batch systems are queued before allocation by a batch scheduler to a set of available processors for execution. Thus, in addition to the time taken for execution, a job submitted to a batch queue incurs time due to waiting in the queue before allocation to a set of processors for execution.

Predicting queue waiting times of the jobs on the batch systems will be highly beneficial for users. The predictions can be used by a user for various purposes including planning management of his jobs and meeting deadlines, considering migrating to other queues, systems or sites at his disposal for application execution when informed of possible high queue waiting times on a queue, and investigating alternate job parameters including different requested number of processors and estimated execution times. Such predictions can also be efficiently used by a metascheduler to make automatic scheduling decisions for selecting the appropriate number of processors and queues for job execution to optimize certain cost metrics, and help reduce the complexities associated with job submissions for the users. The decisions by the user and metascheduler using the predictions can in turn result in overall load balancing of jobs across multiple queues and systems. Such predictions are also highly sought after in the production batch systems. For example, predictions of queue waiting times are available in production systems of TeraGrid [5]. These show the importance of accurate queue wait time prediction mechanisms for the users submitting their jobs to batch systems.

Analyses of widely used job traces in supercomputer sites reveal the presence of large number of jobs that incur short queue waiting times. Table 1 shows statistics for eight different supercomputer job traces we use in this work. All the eight traces were cleaned versions obtained from Feitelson's workload archive [6]. The last column of the table shows the percentages of the number of jobs with queue waiting times of less than or equal to 1 hour. We find that most of the jobs, particularly 56-99% of the total number of jobs, submitted to a system incur queue waiting times of less than or equal to 1 hour. We refer to these jobs as *quick starters*. Correct identification and good predictions of these quick starters that form a majority are hence essential for overall accuracy of the prediction system.

Table 1. Supercomputing Log Details

Trace	**Duration (in months)**	**Total no of jobs**	**% quick starters**
CTC SP2	11	77222	56
ANL Intrepid	8	68936	65
LANL CM5	24	122060	94
HPC2N	42	202876	57
SDSC Paragon 95	12	53970	88
SDSC Blue	32	243314	70
SDSC SP2	24	59725	66
DAS2 fs0	12	225710	99

It is important to note that these quick starters do not necessarily correspond to testing/debugging jobs that are associated with short execution times, and whose predictions are relatively less important. Many systems have separate debug queues for testing/debugging jobs. Our experiments were conducted on general/production queues in which production runs are performed, and where predictions of queue waiting times of the production jobs are required. A significant number of quick starters in these production queues have high execution times. For example, in CTC and ANL production queues, about 30% of the quick starters have runtimes greater than 1 hour and some of them have runtimes as high as 120 hours. Prediction of queue waiting times is challenging due to various factors including diverse scheduling algorithms followed by the job scheduling frameworks, time-varying policies applied for a single queue, and priorities for the jobs. High values of predictions will have more severe impact on the predictions for quick starters than for jobs with long wait times. High overestimations for quick starters can have detrimental effect even on the job submissions to the system. For example, an upper bound of 8 hours for a job that executes for 15 minutes and whose actual waiting time will be 30 minutes can discourage the user from submitting the job to the system that the user would have found suitable for his job in the absence of such overestimation. Hence it is essential to give tight upper bounds especially for these quick starters. In our work, we fix this upper bound as 1 hour for all the quick starters. The assumption is that even if a quick starter's actual waiting time is 5 minutes, this upper bound of 1 hour will not severely discourage the user since the user typically expects waiting times of at least few minutes to an hour in a multi-user system.

The objective for predictions of quick starters is two fold:

- Maximizing *true positives*, i.e., increasing the number of correct identifications of quick starters
- Minimizing *false positives*, i.e., decreasing the number of incorrect identification as quick starters of jobs with long queue waiting times.

The former objective is important for improving the overall accuracy of predictions, while the latter is essential to avoid "misguiding" (or colloquially, "cheating") the user into using the system with the promise of short queue waiting times.

In this work, we have developed an integrated framework, **PQStar** (**P**redicting **Q**uick **Star**ters), for identification and prediction of quick starters. An important aspect of our prediction strategy for quick starters is that it considers the processor occupancy state and the queue state at the time of the job submission in addition to the job characteristics including the requested number of processors and the estimated runtime. The processor and queue states include the current number of free nodes, number of jobs with large request sizes currently executing in the system, and relative difference between the current job and other jobs in the queue in terms of request size and estimated run time. These states are obtained by using a simulator that updates the states during job arrivals and departures. For jobs identified as those with potential long queue waiting times, our framework uses existing strategies [7,8] for predictions. Our experiments with different production supercomputer job traces given in Table 1 show that our prediction strategies can lead to correct identification of up to

20 times more quick starters and provide tighter bounds for these jobs, and thus result in up to 64% higher overall prediction accuracy than existing methods. Our model was designed not to use dynamic and variable parameters including scheduling algorithms and job priorities. In many cases, it is not practical to obtain/infer job priorities and scheduling algorithms. Scheduling algorithms on batch systems are usually not published, and are not easy to model. Our model primarily uses the job traces, and the job submission states (queue and processor occupancy states). This way, our system can be generic and can be applied to different batch systems with different scheduling and priority policies.

Section 2 presents existing strategies for predictions of queue waiting times. In Section 3, our prediction methodology is described in detail. Section 4 describes the simulation experiments with the supercomputer job traces and presents results related to accuracy in identifying quick starters and overall predictions. This section also compares the performance of our predictions with the existing methods. Section 5 presents a summary of our work and plans for future work.

2 Related Work

There have been two primary efforts in prediction of queue waiting times. They can be broadly classified into *Non-Statistical* and *Statistical* methods.

Non-statistical methods try to simulate the exact scheduling algorithm and decisions which would be made by the scheduler in real time. However, in most production systems, the scheduling algorithms are usually not published and are also difficult to model.

In the works by Smith et al. [9] [10], runtime predictions are derived using similar runs in the history, and these estimates are further used to simulate the scheduling algorithms like FCFS, LWF (Least Work First) and Backfilling [11] to obtain the queue wait times predictions. Another work by Li et al. [12] tries to improve the runtime prediction and simulate the batch system for the Maui scheduler [4]. These efforts consider specific scheduling algorithms for predictions while our effort considers only job traces and hence can work with multiple scheduling algorithms. Moreover, their efforts use runtime estimates for the prediction of queue wait times. The runtime estimates reported in these efforts [9] [10] have high prediction errors from 33% to 74% in many cases, and hence using these estimates to predict queue wait times will lead to large errors in wait time predictions.

The statistical method by Downey [13] used the observation that the cumulative distributions of the execution times of the jobs in the workload can be modeled by using a logarithmic function. After the distribution functions are calculated, two different methods are used to predict when a certain number of nodes will become free and thus when the job waiting at the head of the queue can start. This work considers FCFS Scheduling.

Some statistical methods use time series analysis of queue waiting times for jobs in the history to predict waiting times for submitted jobs. QBETS [14–16] is a system that predicts the bounds on the queue wait times with a quantitative confidence level. QBETS uses a quantile-based approach in which a given quantile in the distribution of the queue waiting times of the jobs is used as an upper bound for the target job's queue waiting time. Since the distribution is not known, a confidence level has to be provided. QBETS uses a predictor based on non-parametric inference, an automated change-point detector, machine-learned, model-based clustering of jobs having similar characteristics, and an automatic downtime detector to identify systemic failures that affect job queuing delay. Thus QBETS handles the effects of varying workloads and customized local queuing discipline. However, QBETS gives conservative upper bound predictions, which leads to large prediction errors especially for quick starters. Also it does not consider the state of the system, and consider only the job characteristics, which we show is insufficient for efficient predictions of queue wait times.

The efforts by Li et al. [7,8] consider the system states for the prediction of queue waiting times. In their method known as Instance Based Learning (IBL), they use weighted sum of Heterogeneous Euclidean-Overlap Distances between different attributes of two jobs to find the similarities between the jobs. They consider both job characteristics and system parameters for job attributes. They then find such similarity values between the target job and all jobs in the history and choose a subset of most similar jobs in the history, and use their queue waiting times in methods like 1-NN (nearest neighbor) or the n-WA (weighted average of n nearest neighbors) to predict the waiting time of the target job. Their work assumes linear relationship between attribute values and queue waiting times. They also assume fixed weights for predictions of all target jobs. Our work explicitly considers quick starters for predictions. We show that our method gives better predictions of quick starters. By providing tighter estimated bounds for *quick starters* that form a majority of the jobs, our work attempts to improve the overall accuracy of predictions.

3 Methodology

The basis of our method, *PQStar*, for identifying a quick starter job is to establish boundaries in the history of prior job submissions, and to use the similar jobs within the boundaries for prediction. Thus, the most relevant history is used for predicting the target quick starter job. Specifically, PQStar splits the history for a target job into near, mid and long term history based on processor occupancy states. A processor occupancy state at a given instance denotes the allocation of the processors to the jobs executing at that instance. It consists of the number of processors used by each executing job.

3.1 Predictions Using Near-Term History

For finding the near-term history, PQStar traverses the jobs starting from the target job in the reverse chronological order of job submission times as long as the processor occupancy state at the time of submission of the job in the history is similar to the processor occupancy state at the time of submission of the target job. The earliest job in the history having similar processor occupancy state denotes the near-term boundary and the set of jobs between the target job and the boundary forms the near-term history. For identifying if the target job is a quick starter, PQStar finds jobs in the near-term history with similar characteristics in terms of processor request sizes and estimated run time, and which have started executions. It also checks if none of the jobs in the waiting queue which have arrived after the near-term boundary and which have similar characteristics to the target job, have waited for more than an hour in the queue. If these two conditions are met, PQStar identifies the target job as a quick starter and establishes an upper bound of 1 hour for the target job.

For the purpose of predictions using near-term history, two processor occupancy states are considered similar if the number of jobs with large request sizes that are executing in the two states are the same. We use the executing large jobs to define processor occupancy similarity since jobs that can be backfilled in the remaining processor space and thus incur small queue waiting times are candidate quick starters. Thus the basis of identifying quick starters using near-term history is that by looking at jobs with similar characteristics in the near-term history with similar processor occupancy states and checking if those jobs have potentially been backfilled, it can be predicted if the target job can be backfilled and hence marked as a quick starter. Note that by our definition, two processor occupancy states are also considered similar if there are no large jobs in both the states. For our work, we denote a job as a large job if the request size of the job is at least the next power of two greater than or equal to the square root of the total system size . We define job characteristics of two jobs as similar if their processor request sizes are equal and if the difference between their estimated run times are within an hour. As our experiments will show, this method of using near-term history for predictions yields a large percentage of identifications of quick starters.

3.2 Predictions Using Mid-term History

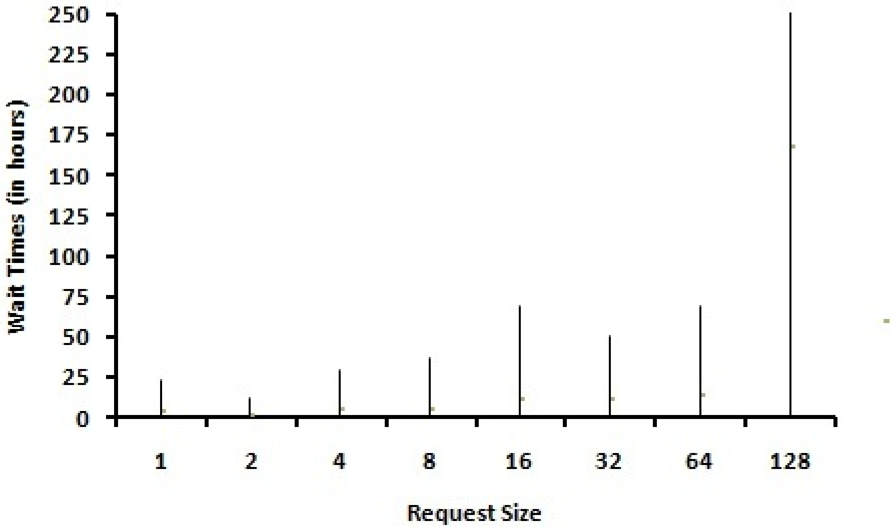
For those jobs for which near-term history cannot be used due to the above mentioned criteria not being met, PQStar traverses the jobs in the reverse chronological order of job submissions from the near-term boundary until the processor occupancy state becomes completely different in terms of executing jobs. In other words, suppose A = {set of jobs in execution at the time of target job's submission} and B = {set of jobs in execution at the time of the history job's submission}, then we draw the mid term boundary at a point where $(A \cap B) = \emptyset$.

PQStar marks the *mid-term boundary* after the job when the states become completely different and denotes the set of jobs between the mid and near-term boundaries as mid-term history. For identifying if the target job is a quick starter, PQStar finds jobs in the mid-term history with similar characteristics in terms of processor request sizes and estimated run time, and which have started executions. It also checks if none of the jobs in the waiting queue which have arrived after the mid-term boundary and which have similar characteristics to the target job, have waited for more than an hour in the queue. However, unlike for predictions with near-term history, these conditions alone are not sufficient for predictions with mid-term history since the processor states in mid-term are less similar to target job than those in near-term. We found that jobs satisfying the conditions had widely varying queue waiting times. Hence we introduce three extra criteria for predictions with mid-term history: *request size criterion*, *estimated run time (ERT) criterion*, and *free nodes criterion.*

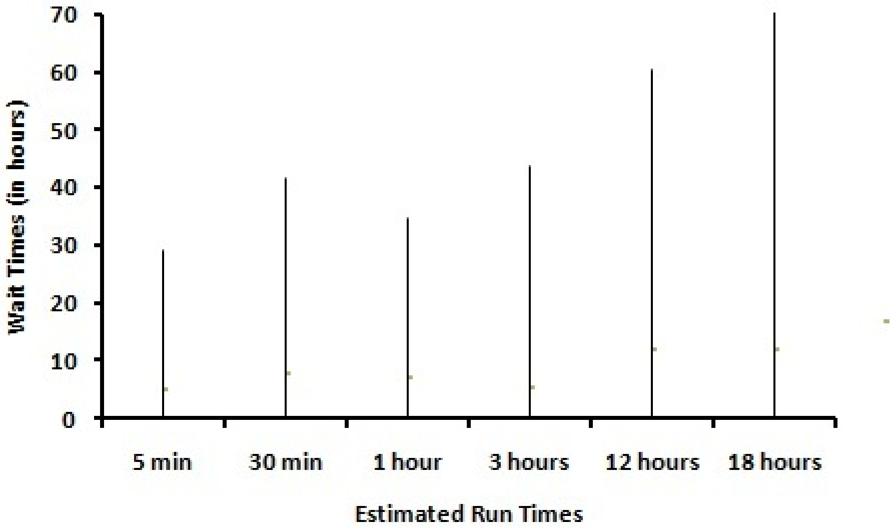
Request Size Criterion. One useful criterion we found is to rank the target job in terms of its request size among the jobs in the queue at the time of the job submission. Specifically, we calculate the metric $jobrank_{reqsize}$ using the position of the target job in the list of jobs in the queue at the time of its entry sorted in increasing order of request sizes. $jobrank_{reqsize}$ is calculated by normalizing this position with respect to the total number of jobs in the queue.

Most of the existing strategies group jobs only in terms of request sizes, and find jobs with similar request sizes for predictions of queue waiting times. However, a single request size can correspond to widely varying queue waiting times as shown in Figure 1(a) that shows statistics for 1000 sample jobs from the CTC trace. For example, corresponding to the request size of 8 processors in the figure, we find that queue waiting times can vary from 1 minute to 32 hours. Hence, in addition to finding the similar jobs for a target job in terms of request sizes, we consider the rank of the job in the queue in terms of request size, thereby implicitly considering both the request size and the queue state for predictions. We consider target jobs with $jobrank_{reqsize}$ values less than a threshold, $threshold_{reqsize}$, as candidate quick starters. For fixing $threshold_{reqsize}$, we consider jobs in the near-term history, find two thresholds: $threshold_{reqsize1}$ as the maximum of $jobrank_{reqsize}$ values of the jobs with queue waiting times less than or equal to 1 hour, and $threshold_{reqsize2}$ as the minimum of $jobrank_{reqsize}$ values of the jobs with queue waiting times greater than 1 hour, and use the minimum of the two thresholds for $threshold_{reqsize}$. By using near-term history to find thresholds, PQStar uses the most recent history and hence also takes into consideration only those jobs having similar processor occupancy.

ERT Criterion. Similar to the request size criterion, we also use estimated run time (ert) criterion for identification of the target job as a quick starter. Specifically, PQStar finds a metric, $jobrank_{ert}$, using the list of jobs in the queue, at the time of entry of the target job, sorted in the ascending order of estimated run times, and finding the normalized position of the target job in the list with respect to the total number of jobs in the queue. PQStar marks the target job as a quick starter if its $jobrank_{ert}$ is less than a threshold, $threshold_{ert}$. $threshold_{ert}$ is found similarly to $threshold_{reqsize}$ by using the queue waiting times of jobs in the near-term history. Figure 1(b) considers only the ERTs and their impact on queue waiting times. Similar to considering only request sizes, we find that a single ERT can correspond to wide variation of queue waiting times. Thus the existing strategies that use only ERTs to find similar jobs for predictions can give high upper bound values for predictions. By considering $jobrank_{ert}$, PQStar defines similarity using both the job and the queue state characteristics. The assumption behind the request size and the ERT criterion is that target



(a) Wait Time Ranges for Different Request Sizes



(b) Wait time ranges for different ERTs

Fig. 1. Wait Time Ranges of Jobs for different Request Sizes and ERTs (CTC Trace)

jobs with small request sizes or ERT relative to the other jobs in the queue have higher chances of backfilling and hence can be quick starters.

Free Node Criterion. The final criterion we use for mid-term history is based on the number of free nodes available to accommodate the target job. Thus this criterion explicitly takes into account the processor occupancy state in addition to the queue waiting state for identifying quick starters. Specifically, at the time of submission of the target job, PQStar finds the difference between the total number of free nodes available and the number of nodes requested by the jobs in the queue that have smaller request sizes or smaller estimated run times than the target job. If this difference is larger than the number of nodes requested by the target job, PQStar marks the target job as a quick starter. The assumption behind this criterion is that jobs in the queue with smaller request sizes or estimated run times have higher chances of backfilling and hence start earlier than the target job, thereby consuming some subset of free nodes.

For predictions with mid-term history, PQStar marks a target job as a quick starter if it meets any one of the three criteria, namely, request size, ert, or free node criteria.

3.3 Predictions Using Far-Term History

For those jobs for which mid-term history also cannot be used due to the above mentioned criteria not being met, we use the far-term history, which is the rest of the jobs in the history beyond the mid-term boundary. Among these far-term history jobs, PQStar extracts a subset of jobs with similar characteristics in terms of processor request sizes and estimated run time, and which have started executions. If all the jobs in this subset have queue waiting times of less than one hour, then it indicates that the target job will most likely have a wait time of less than or equal to one hour. Hence PQStar marks the target job as a quick starter.

3.4 Overall Predictions and Using IBL

In summary, PQStar tries to find similar jobs in the near, mid or far-term history, and uses a set of criteria to mark a job as a quick starter. In addition to considering only the job characteristics of request sizes and estimated run times, PQStar explicitly or implicitly considers the system states, namely processor occupancy and queue states, for defining similarity and for obtaining predictions. In our evaluations we found that the near-term history typically consisted of about 50 jobs spanning around 1-3 hours and the mid-term history typically consisted of more than 500 jobs spanning around 10-25 hours. For target jobs that are either not marked as quick starters or for which similar jobs cannot be found in the near, mid or far-term history, PQStar uses an existing strategy for predicting queue waiting times. We use the IBL method by Li et al. [7] for these predictions, since we found in our experiments that IBL gives better predictions than QBETS [14]. IBL (Instance Based Learning) uses weighted sum of

Heterogeneous Euclidean-Overlap Distances between job attributes to calculate similarities of jobs, and use similar jobs in the history to give point predictions for the target jobs based on 1-nearest neighbor or weighted average of k-nearest neighbors methods.

The entire algorithm followed in PQStar is illustrated in the flowchart shown in Figure 2.

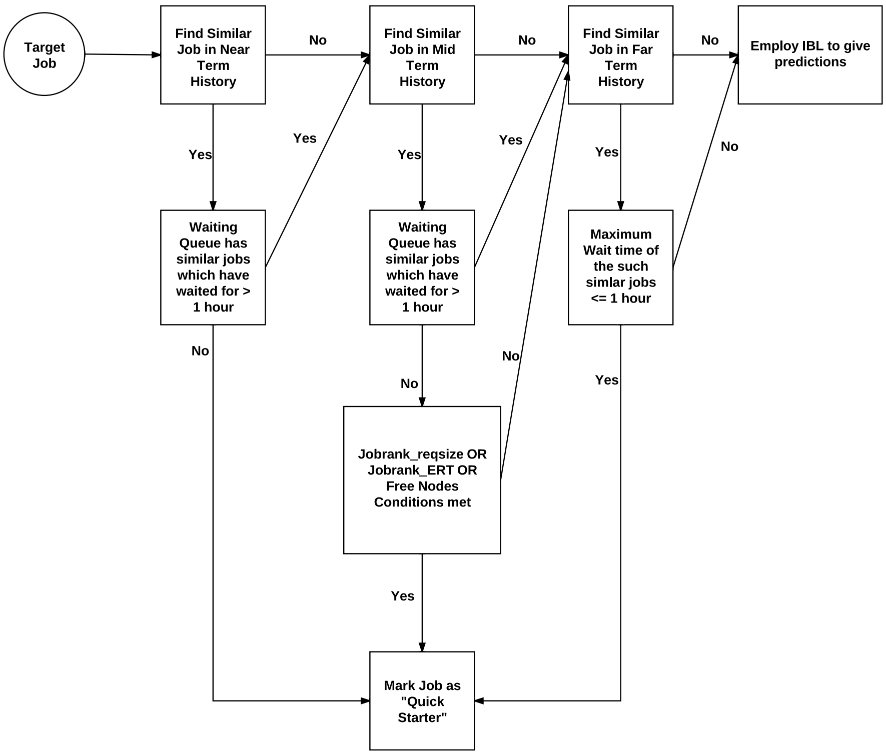

Target Job
Find Similar Job in Near Term History
No
Find Similar Job in Mid Term History
No
Find Similar Job in Far Term History
No
Employ IBL to give predictions
Yes
Yes
Yes
Yes
Yes
No
Waiting Queue has similar jobs which have waited for > 1 hour
Waiting Queue has similar jobs which have waited for > 1 hour
Maximum Wait time of the such simlar jobs <= 1 hour
No
No
No
Yes
Jobrank_reqsize OR Jobrank_ERT OR Free Nodes Conditions met
Yes
Mark Job as "Quick Starter"

Fig. 2. PQStar Methodology

4 Experiments and Results

4.1 Experimental Setup

The experiments were conducted using a discrete event simulator that we have developed. It creates a simulated environment of the jobs waiting in the queue and running on the system at different points of time. It is important to note that the simulator will only be keeping track of the jobs submitted to the system, and maintain their attributes including arrival times, wait times, actual runtimes and request sizes. It will not be simulating the actual scheduling algorithm used, thus avoiding assumptions about the underlying scheduling algorithm.

The simulator can be operated in two modes. In the first mode, the user can invoke the simulator with a supercomputing job trace/log in the Standard Workload Format (SWF) [17] as input, and obtain predicted queue waiting time of a new job. This mode is followed in the QBETS prediction system [14]. In this mode, the simulator creates the simulated environment of jobs in the system using the statistics available in the log. In the second mode, the simulator can be executed on the front end node of a batch system for which predictions are required. It will then track the arrivals, executions and exit of the real jobs submitted to the system, and will create the simulated environment using these real jobs. In this second mode, the job attributes maintained and used by the simulator can be obtained by queue and job management commands. For example on PBS based batch systems, the *qstat* command (with $-f$ option) will give all of the job parameters required by PQStar. Thus in the second mode, the predictions are obtained "live" at real time. The simulator is triggered by three primary events corresponding to job arrival, job beginning to execute and job termination. Whenever a job arrives, it is added to a waiting queue maintained by the simulator. As soon as a job's wait time is over and it starts executing, it is removed from the waiting queue and added to a running list in the simulator. Also at this time, the free nodes available in the system is decremented by the value equal to the job's request size. Once a job which is running completes its execution, it is removed from the running list and the free nodes available in the system is incremented by the value equal to the job's request size. This process is repeated for each job and thus a simulated system state is created using which we extract the processor state and the queue properties that are needed for our algorithm.

For each supercomputing trace in our experiments, we performed predictions for all the jobs starting from the 10001^{th} job up to a maximum of 50000 jobs or the end of the log. Each of the jobs in this set constitutes the evaluation data for which predictions were made. For a given target job for which waiting time is predicted, all the jobs submitted prior to it constitute the history. Out of these history jobs, we use a subset of jobs and use their waiting times for predicting for the target job. The subset is formed based on similarity to the target job and using near, mid and far term boundaries as described earlier. Once the target jobs start their execution and their wait times are known, they are added to the set of history jobs. We compared the predictions of our PQStar method with the results of IBL, QBETS and a parametric model, namely, using *log-uniform* distribution of the waiting times for predictions. We used the log-uniform model since it was found to give competitive results with QBETS in some cases [14].

4.2 Results

Predictions for Quick Starters. We first show the effectiveness of using near-term history in PQStar for predicting quick starters. Table 2 shows the percentage of quick starters identified using near-term history and also the average number of Jobs in the Near Boundary. The results show that predictions based on near-term history contribute significantly to the identification of large number of quick starters.

Table 2. Percent of Quick starters marked using Near Boundaries in PQStar

Logs	% Quick starters marked using Near Boundary	Average number of Jobs in the Near Boundary
CTC	53	53
ANL	17	48
LANL	39	39
HPC2N	62	54
SDSC Paragon	61	43
SDSC Blue	51	46
SDSC SP2	50	45
DAS	66	59

Table 3. % of the QuickStarters Successfully Marked

Logs	Log-Uniform	QBETS	IBL	PQStar
CTC	2	5	43	84
ANL	11	42	61	78
LANL	8	67	85	88
HPC2N	3	19	49	80
SDSC Paragon	6	48	81	89
SDSC Blue	8	45	63	89
SDSC SP2	1	4	49	79
DAS	71	95	96	98

Table 3 shows the percentage of the quick starters, that are successfully identified by log-uniform, QBETS, IBL and our method, PQStar. Successful identification corresponds to estimating the upper bound of the quick starters as less than or equal to one hour. We can see that our method, PQStar, performs better than the next best strategy, IBL, by successfully identifying up to 1.95 times more quick starters. It also successfully identifies up to 20 times more quick starters than QBETS. Our method is also more consistent and identifies more than about 80% of the quick starters irrespective of the log.

Misguiding Predictions. These results show that our PQstar prediction system is highly successful in obtaining large number of *true positives*, i.e., identifying large number of quick starters. Our other objective is to minimize the number of *false positives*, i.e., number of jobs with long queue waiting times falsely identified as quick starters. We refer to these predictions as *misguiding predictions*. Table 4 shows that for the supercomputing traces used in our experiments, PQStar incurs such misguiding predictions for only 1-10% of the total number of jobs. The last column of the table also shows that 32-72% of the those misguided jobs have actual wait times of less than 4 hours. This indicates

Table 4. Misguiding Predictions

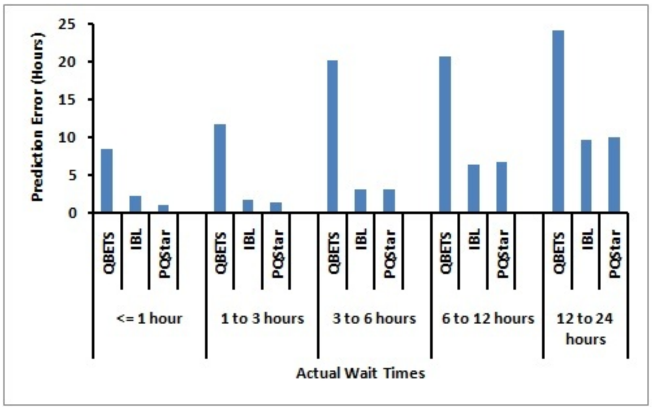
Logs	**% of Total Jobs Corresponding to Misguiding Predictions**	**% of Misguiding Predictions with Actual Wait Times of Less than 4 Hours**
CTC	9	59
ANL	4	72
LANL	1	61
HPC2N	10	46
SDSC Paragon	1	45
SDSC Blue	6	57
SDSC SP2	6	57
DAS	0.25	32

that for half of the misguiding predictions, the amount of misguidance is within reasonable limits.

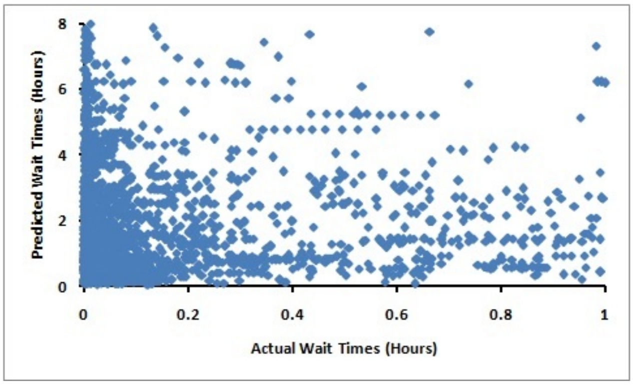
Overall Predictions. Since IBL was found to be a better strategy than QBETS as shown in Table 3, our PQStar system uses IBL for predictions of non quick starters. We illustrate the comparisons of predictions of QBETS, IBL and PQStar in Figure 3, for 5000 jobs of CTC trace. Figure 3(a) shows the mean prediction error (absolute difference in predicted and actual wait times) for the different ranges of the actual wait times. From this, we can see that PQStar gives the least prediction error for the quick starters, and gives the same prediction error as IBL for the rest of the jobs, since PQStar uses IBL for jobs predicted as non-quick starters. In order to further elaborate the advantage of PQStar over IBL,we show a scatter plot of actual v/s predicted wait times for quick starters, as shown in Figures 3(b) and 3(c). We can clearly see that PQStar provides tight upper bounds, while IBL provides high upper bounds for a large number of quick starters.

Figure 4 shows the distribution of predicted waiting times for different ranges of actual waiting times for all the jobs in the ANL (Figure 4(a)) and CTC (Figure 4(b)) traces for QBETS, IBL and PQStar. The graphs show that for jobs with actual waiting times of less than or equal to 1 hour, i.e., quick starters, PQStar is able to identify most of them as quick starters. For the other quick starters, the predictions by PQStar correspond to low ranges of queue waiting times. With QBETS and IBL, high predicted ranges are provided for a large number of these quick starters. For jobs with actual quick waiting times of 1 to 12 hours, PQStar gives smaller ranges of predicted queue waiting time for more jobs than QBETS and IBL.

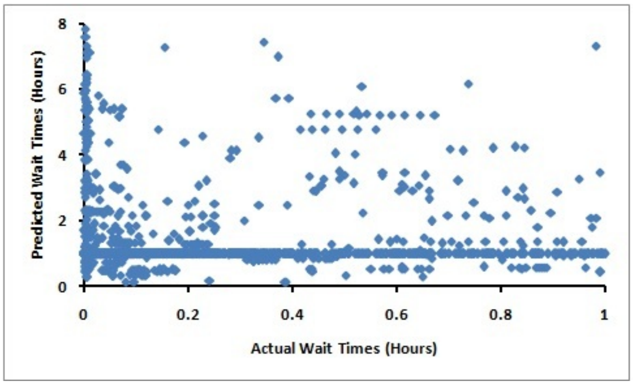
The prediction time per job in our PQStar method is under a second and the total time take for the simulation to run for entire datasets was almost similar to that of both IBL and QBETS. Hence, there is minimal or no overhead added in terms of prediction time by PQStar to the existing IBL method.



(a) Prediction Error for Different Actual Queue Waiting Time for QBETS, IBL and PQStar

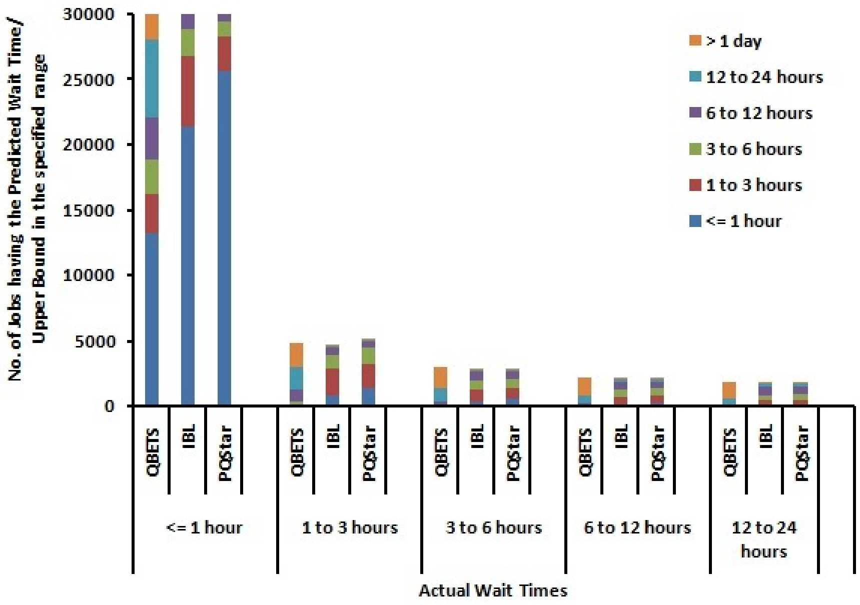


(b) Actual Wait Time v/s Predicted Wait Time for IBL

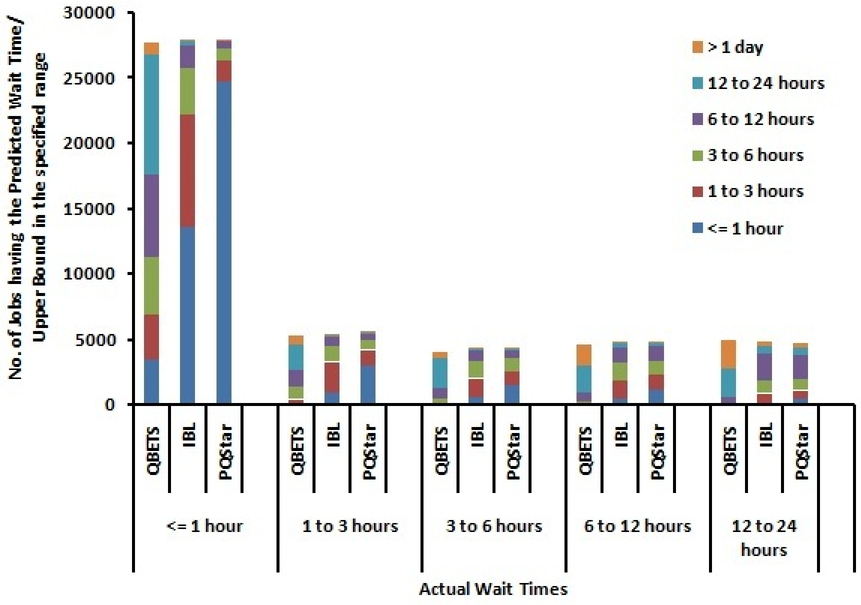


(c) Actual Wait Time v/s Predicted Wait Time for PQStar

Fig. 3. Prediction Comparison for QBETS, IBL and PQStar using 5000 jobs of CTC trace



(a) ANL Trace



(b) CTC Trace

Fig. 4. Distributions of Predicted Queue Waiting Times for Different Actual Queue Waiting Time for QBETS, IBL and PQStar using ANL and CTC Traces

RMS Error and Response Time Predictions. In order to evaluate the effect of the predictions of quick starters by PQStar on the overall accuracy of predictions, we calculate the RMS (Root Mean Square) value between the actual and predicted queue waiting times for all the jobs. We compute the percentage decrease in RMS error for PQStar from the RMS errors of the other methods. For example, for comparison of RMS errors of PQStar and QBETS we compute $\frac{rmserror_{qbets} - rmserror_{pqstar}}{rmserror_{qbets}}$.

We denote the percentage decrease as $rmsdec_{fromlu}$, $rmsdec_{fromqbets}$ and $rmsdec_{fromibl}$, for comparisons with log-uniform, QBETS and IBL, respectively. Positive values for the percentage decrease indicate better predictions by PQStar. Table 5 shows the decrease in RMS error due to PQStar for all predictions. We find that PQStar gives better prediction accuracy than the other methods. Our method results in up to 90% average improvement in overall prediction accuracy of all the jobs over log-uniform and up to 64% average improvement over QBETS predictions. The average improvement due to PQStar is only about 2% when compared to IBL since PQStar uses IBL for predictions of non quick-starters. The actual waiting times and the prediction errors for these non quick-starters have large values, and these large prediction errors dominate the overall RMS error considering all the jobs. Hence the difference in RMS error between PQStar and IBL is small. The last column of the table shows $nrmsdec_{fromibl}$, the percentage decrease in normalized RMS error due to PQStar when compared to IBL. The normalized RMS errors are calculated by normalizing the individual prediction errors using the actual waiting times. As the results in this column show, PQStar results in significant gain up to 42% in overall prediction accuracy when compared to IBL.

Table 5. Prediction Accuracy: RMS Errors

Logs	**rmsdec$_{\mathbf{fromlu}}$**	**rmsdec$_{\mathbf{fromqbets}}$**	**rmsdec$_{\mathbf{fromibl}}$**	**nrmsdec$_{\mathbf{fromibl}}$**
CTC	80	58	2	42
ANL	75	61	2	22
LANL	90	57	1	5
HPC2N	84	59	2.25	42
SDSC Paragon	79	64	1.5	6
SDSC Blue	77	58	2	33
SDSC SP2	71	56	1.75	38
DAS	19	7	0	0

We also compute the percentage difference in predicted and actual response times for each job, where response time is the sum of queue waiting time and execution time. For the execution time, we consider the predicted execution time to be equal to the actual execution time. Hence the percentage predicted error in response time is calculated as $PPE_{rt} = \frac{|predictedwaitingtime - actualwaitingtime|}{actualresponsetime}$. This metric determines the amount of impact of the prediction errors on jobs

Table 6. Bins and the corresponding Intervals

Bins	**Intervals**
less than or equal to 1 hour	15 min
1 hour to 3 hours	30 min
3 to 6 hours	1 hour
6 to 12 hours	3 hours
12 to 24 hours	6 hours
1 day to 2 days	12 hours
greater than 2 days	24 hours

of different lengths or execution times. A prediction in queue waiting time with an error of 1 hour will have higher impact on a job whose execution time is 15 minutes than for a job whose execution time is 2 days.

Further, to define good predictions, we divide the waiting and run times into different *bins*. Each bin represents a range of wait/run times and is associated with an interval. The wait/run time of a job is rounded off to the nearest interval values associated with the bin to which the wait/run time belongs. Table 6 refers to the bins and interval size for each bin used for our experiments. For example, if the wait time of a job is 37 minutes, the interval size that will be used is 15 minutes (first row). Hence the wait time is rounded off to 45 minutes, which is the nearest next 15 minute interval. As can be seen, the idea of using bins is to give different tolerance limits in prediction errors for different predicted and actual wait times. Prediction error of 15 minutes is large for a job whose actual waiting time is 20 minutes, while it is small for a job whose actual waiting time is 2.5 days.

We consider a prediction for a job as a *good prediction* if the rounded values of actual and predicted queue waiting times lie in the same bin or if its PPE_{rt} value is within 10%. Tables 7 shows the average PPE_{rt} values and percentage good predictions obtained by the various methods. The table shows that the average PPE_{rt} is up to 35 times less and the number of good predictions is up to 58% more with PQStar when compared to the other methods.

Thresholds for Quick Starters. For all the above results, we have used a queue waiting time threshold of 1 hour for the definition of quick starters. Jobs with actual queue waiting times less than this threshold are marked as quick starters. This threshold is based on the assumption that waiting time of less than one hour may be short and prediction errors up to that limit may be acceptable for the user. We used the value of one hour as a threshold to target jobs with queue waiting times less than this threshold to improve/tighten the bounds of these jobs, since this class of jobs form a majority of the jobs as we had shown in Table 1. However, there have been a series of works [18] [19] which show that the response time of a job should be less than 20 minutes to consider a job submission session as interactive. In this experiment, we analyse the effect of the changing thresholds for the quick starters by using different

Table 7. Prediction Accuracy: % difference in Predicted and Actual Response Times of Different Prediction Methods

Logs	Log-Uniform		QBETS		IBL		PQStar	
	Average PPE_{rt}	**% Good predictions**	**Average PPE_{rt}**	**% Good predictions**	**Average PPE_{rt}**	**% Good predictions**	**Average PPE_{rt}**	**% Good predictions**
CTC	41.1	3	19.6	6	1.4	39	0.5	60
ANL	20.7	5	17.5	32	2.21	36	1.4	57
LANL	35.2	9	22.4	61	0.12	91	0.06	94
HPC2N	44.7	5	16.5	17	1.96	46	0.68	66
SDSC Paragon	45.7	6	18.3	45	0.45	79	0.24	85
SDSC Blue	31.7	8	13.1	35	2.01	53	0.75	70
SDSC SP2	55.2	3	32.3	6	2.86	45	1.2	64
DAS	2.6	73	0.25	96	0.01	98	0.007	99

thresholds in PQStar for predicting quick starters. Table 8 shows the impact of changing the thresholds for quick starters from 10 minutes to 1 hour on the PQStar predictions of quick starters for the CTC trace. We find that PQStar consistently identifies more than 80% quick starters for all the thresholds, and the variation in threshold does not have an impact on the predictions of quick starters.

Table 8. Impact of changing the thresholds for quick starters (CTC Trace)

Quick Starter Threshold (in minutes)	**% of Quick Starters correctly identified**
10	84.22
20	83.99
30	83.74
40	83.69
50	83.62
60	84.28

In summary, PQStar performs better than both IBL and QBETS, and it also outperforms the parametric model, using log-uniform distribution, in all the above shown aspects. From these results, we can clearly see that our method is providing much more aggressive bounds for the quick starters compared to rest of the methods, and also the under predictions is kept to limited amounts.

5 Conclusions and Future Work

In this work, we had developed a prediction system called PQStar for identification of quick starters or jobs whose actual queue waiting times are less than or equal to 1 hour. These quick starters form a majority of the job submissions in many supercomputer traces. In this work we consider both job characteristics, namely, request size and estimated runtime time, and the state of the system, namely the queue and processor occupancy states, for predictions. By means of experiments with different supercomputer traces, we showed that that our prediction strategies can lead to correct identification of up to 20 times more quick starters and provide tighter bounds for these jobs, and thus result in up to 64% higher overall prediction accuracy than existing methods.

We currently use the IBL method for predictions of jobs with potential long queue waiting times. We plan to explore alternate strategies for predictions of such jobs. We also plan to develop techniques for predictions of execution time in order to predict total response times. Predicting execution times for jobs submitted to batch systems is challenging due to limited history. We finally plan to build scheduling and metascheduling strategies that use these stochastic predictions to select the appropriate resources for job executions.

References

1. IBM Load Leveler, http://www.redbooks.ibm.com/abstracts/sg246038.html
2. PBS Works, http://www.pbsworks.com/
3. Platform LSF, http://www.platform.com/workload-management/high-performance-computing
4. MAUI Scheduler, http://www.supercluster.org
5. Tera Grid Karnak Prediction Service, http://karnak.teragrid.org/karnak/index.html
6. Parallel Workload Archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html
7. Li, H., Groep, D.L., Wolters, L.: Efficient Response Time Predictions by Exploiting Application and Resource State Similarities. In: GRID 2005 Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, pp. 234–241 (2005)
8. Li, H., Chen, J., Tao, Y., Groep, D.L., Wolters, L.: Improving a Local Learning Technique for Queue Wait Time Predictions. In: CCGRID 2006 Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 335–342 (2006)
9. Smith, W., Foster, I., Taylor, V.: Predicting Application Run Times Using Historical Information. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 122–142. Springer, Heidelberg (1998)
10. Smith, W., Taylor, V.E., Foster, I.T.: Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In: IPPS/SPDP 1999/JSSPP 1999: Proceedings of the Job Scheduling Strategies for Parallel Processing, pp. 202–219 (1999)
11. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel Job Scheduling - A Status Report. In: JSSPP 2007 Proceedings of the 13th International Conference on Job Scheduling Strategies for Parallel Processing, pp. 1–16 (2004)

12. Li, H., Groep, D., Templon, J., Wolters, L.: Predicting Job Start Times on Clusters. In: CCGRID 2004: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (2004)
13. Downey, A.B.: Predicting Queue Times on Space-Sharing Parallel Computers. In: IPPS 1997 Proceedings of the 11th International Symposium on Parallel Processing, pp. 209–218 (1997)
14. Nurmi, D., Brevik, J., Wolski, R.: QBETS: Queue Bounds Estimation from Time Series. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 76–101. Springer, Heidelberg (2008)
15. Brevik, J., Nurmi, D., Wolski, R.: Predicting Bounds on Queuing Delay for Batch-Scheduled Parallel Machines. In: PPoPP 2006: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 110–118 (2006)
16. Brevik, D.N.J., Wolski, R.: Using Model-Based Clustering to Improve Predictions for Queueing Delay on Parallel Machines, pp. 21–46
17. Standard Workload Form, `http://www.cs.huji.ac.il/labs/parallel/workload/swf.html`
18. Shmueli, E., Feitelson, D.G.: Uncovering the Effect of System Performance on User Behavior from Traces of Parallel Systems. In: MASCOTS 2007 Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 274–280 (2007)
19. Zilber, J., Amit, O., Talby, D.: What is worth learning from Parallel Workloads?: A User and Session based Analysis. In: ICS 2005 Proceedings of the 19th Annual International Conference on Supercomputing, pp. 377–386 (2005)

On Identifying User Session Boundaries in Parallel Workload Logs

Netanel Zakay and Dror G. Feitelson

School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract. The stream of jobs submitted to a parallel supercomputer is actually the interleaving of many streams from different users, each of which is composed of sessions. Identifying and characterizing the sessions is important in the context of workload modeling, especially if a user-based workload model is considered. Traditionally, sessions have been delimited by long think times, that is, by intervals of more than, say, 20 minutes from the termination of one job to the submittal of the next job. We show that such a definition is problematic in this context, because jobs may be extremely long. As a result of including each job's execution in the session, we may get unrealistically long sessions, and indeed, users most probably do not always stay connected and wait for the termination of long jobs. We therefore suggest that sessions be identified based on proven user activity, namely the submittal of new jobs, regardless of how long they run.

1 Introduction

There has recently been increased interest in user-based workload models for parallel supercomputers [16, 11–13]. Such models are generative in nature. This means that instead of modeling the statistical properties of the workload, as was done for example by Jann et al. and Lublin and Feitelson [3, 5], they model the process by which the workload is generated. As jobs are submitted by users, this implies the need to model user behavior.

The motivation for using generative user-based workload models is that such models enable us to include feedback effects in performance evaluations. The stream of jobs submitted to a parallel supercomputer is the result of an interaction between the system and its users. If the system is responsive, users will submit more jobs. If it performs poorly, users may depart in frustration and refrain from submitting more jobs. When we evaluate the performance of a new scheduler design, we need to include such feedback and its effect on user behavior [11, 13].

An important characteristic of user behavior is its temporal pattern. Human users may work for some time, but then they stop and do something else. The periods of continuous work are called *sessions*. There are usually many more user sessions during the day than during the night or weekend, leading to the creation of an overall daily and weekly cycle of activity. Understanding how such patterns are generated is a basic component in defining a generative workload model.

Data about user behavior is contained in accounting logs from existing parallel machines, such as those that are available in the Parallel Workloads Archive [9]. Unfortunately, these logs only include data about individual jobs: when the job was submitted, when it started to run and when it terminated, how many processors it used, etc. Importantly, we usually also know the identity of the user who submitted the job (or at least anonymized identity, in the interest of preserving privacy). But we do not know when the user started or ended each session. If we want to characterize this behavior, we need to glean this data based on the pattern of job submissions.

The common approach to extracting session data is based on the assumption that users typically wait to see the results of their jobs, and then submit additional jobs. Thus the user session extends from the submittal of some job till the termination of that or some later job. Zilber et al. have suggested that breaks of 20 minutes or more between successive jobs indicate a session break [16], and others have followed this definition [12].

The problem with this definition is that parallel jobs may be very long. In some logs we even observe jobs that run for multiple days. Obviously it is unreasonable to expect the user to remain active for such a long time waiting for the job to terminate. And indeed we find that sessions defined according to the above definition may be much longer than is reasonable. We therefore suggest an alternative approach, whereby sessions are defined based on only explicit user activity, namely the submittal of new jobs. The times at which the jobs terminate are ignored.

A basic problem with this line of research is that ground truth is not available for comparison. In other words, we do not really know when users started or ended their sessions. We therefore need to make qualitative judgments. Our main criterion is to look at the distribution of session lengths that is generated by the analysis, and to reject methods that lead to distributions with obvious deficiencies (such as sessions that extend over more than a week).

The next section describes the technical details of how sessions may be defined according to different approaches. Section 3 discusses the selection of threshold values used to identify session breaks. Section 4 shows how we can use the distribution of generated sessions to select among two competing approaches for how to apply the threshold. Section 5 identifies some problems that occur when using inter-arrival times rather than think times. Finally, Section 6 introduces the notion of using the generated session lengths as a criterion for accepting or rejecting different approaches.

2 Definition of Sessions and Batches

Intuitively, a *session* is a period of continuous work by a user. This does not mean that the user was active 100% of the session's time. A user may run a job to completion, think about the result, and then run another job, all within the same session.

The above description seems to imply sequential work, where jobs in a session never overlap. Empirical evidence from parallel supercomputer job logs shows that this is clearly not always the case, and jobs may overlap. Given such an overlap, the later job cannot depend on the earlier one. Following Shmueli, we call a set of such independent, overlapping jobs a *batch* [11]. Thus a session may contain several batches in sequence, and each batch may contain a number of jobs. The interval between batches is called the *think-time*, or TT.

Finding the batches and the sessions of the users is a basic requirement in order to understand and analyze their behavior. However, activity logs do not contain explicit information about neither the sessions nor the batches. Therefore, we need to estimate them based on the data that the logs do contain. The most relevant information is the job arrival times (also called submit times) and the job end times. For job i, we will denote these as $J[i].arr$ and $J[i].end$.

2.1 Definitions Based on Think Times

Assume we scan the jobs in a log one by one. As each job is considered, the question is whether it belongs to the previous session or batch, or starts a new session or batch. The simplest and most commonly used approach makes this decision based on the think time, namely the interval from the termination of one job to the submittal of the next[1]:

1. If the think time is negative, the job overlaps the current batch and therefore belongs to this batch.
2. If the think time is positive but below the *session threshold*, the job starts a new batch in the same session as the previous batch. (We discuss the value of the session threshold in Section 3.)
3. Otherwise, the job starts a new batch in a new session.

Note, however, that we need to be precise regarding how we measure the think time, and in particular, exactly what job end time do we use as a reference point. There are two possibilities:

– The end time of the last job that was submitted. With this approach, the think time of job i will be calculated as

$$TT_{Last} = J[i].arr - J[i-1].end$$

Hereafter we denote this approach by Last.

– The maximal end time among all previous jobs. In this case, the think time is calculated as

$$TT_{Max} = J[i].arr - \max_{j<i} J[j].end$$

This approach will be denoted by Max.

[1] Recall that the conceptual model is that the user submits a job, waits for it to terminate, and then thinks about the result before submitting the next job.

To appreciate the difference, consider a sequence of 3 jobs. Job 1 is very long. Job 2 is short and ends much before job 1 ends. Job 3 arrives after job 2 ended, but still overlaps job 1. In this situation all 3 jobs will be in the same batch based on Max, but job 3 will start a new batch based on Last. This is illustrated in Figure 1.

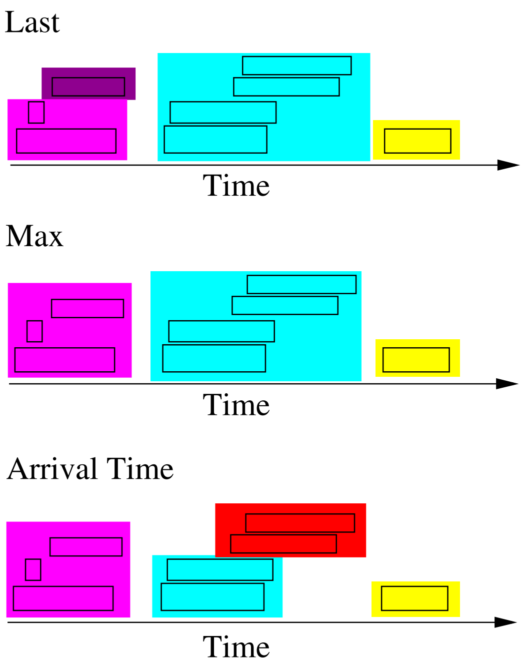


Fig. 1. Batches according to the three approaches: Last, Max, and Arrival

2.2 Definition Based on Inter-arrival Times

Another approach to define sessions is according to the arrival times of the jobs, or rather, the inter-arrivals (to be denoted by Arrival). In this approach, the current job would belong to the same session as previous jobs if it arrives up to the session threshold time after the arrival of the previous job in the session. In other words, if the inter-arrival time is longer than the session threshold, we decide that this represents a session break. Once the jobs are partitioned into sessions using this approach, we partition each such session into batches according to the Max approach.

An example showing the effect of this procedure is shown in Figure 1. The four jobs in the middle all overlap, and are considered to be the same batch by both Last and Max. But there is a relatively large gap between the arrival of the first pair and the arrival of the second pair. If this gap is bigger than the session threshold, the two pairs will be in different sessions according to Arrival, and as a result also in different batches.

The reason for using Max to partition a session into batches is as follows. Consider how the end of a batch is defined. If batch A comes a certain TT after batch B according to Max, then it will start only TT time after *all* the jobs in B are finished. But according to Last, it will start TT time after the last job in B has finished, while other jobs from B may still be running. Shmueli indeed used the last job as the critical one [13]. However, this definition is problematic, because it means that the future activity of the user depends only on the last job in each batch, while the other jobs don't effect the future activity at all. This seems very unrealistic. A simple example of the problem is that it is very easy to create a scheduler that reduces both the user's wait-times and the overall system utilization by running the last job of each user last, thereby causing the user to wait a long time before submitting more jobs. Alternatively, one can construct a scheduler that would increase both the wait-times and the utilization by handling the last job of each user first. To avoid such problems, we prefer Max.

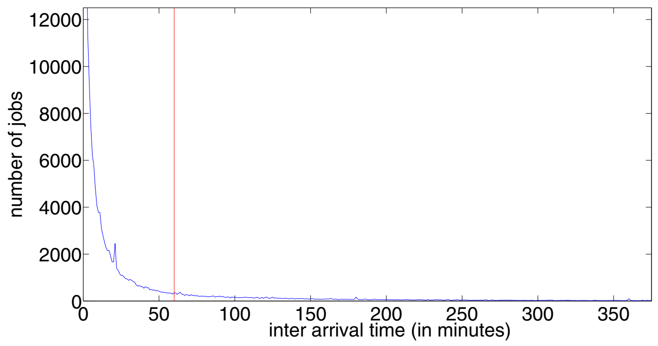
We note that Max creates a sequence of batches with no overlaps. In Last they may overlap, but the dependencies between batches are still a linear sequence. In Arrival a batch may depend on multiple earlier batches.

In the area of parallel supercomputer workloads, the common way to define sessions uses the end time (meaning Last or Max). For example Zilber et al. and Shmueli use Last [16, 12]. But in other areas, where job durations are extremely short, it is more common to define sessions based on arrivals. An example is interactive web use (surfing, searching, or e-commerce) [1, 2, 4, 7, 8, 10, 15]. Of course, due to the very short time it typically takes to process a request on the web, requests never overlap. Therefore Last, Max, and Arrival are actually equivalent in this case.

In the next sections we will discuss the session threshold for each approach and the influence of the choice of this unique value. Additionally, we will present a comparison between the Max and Last approaches. Later, we will investigate the session lengths produced by the different approaches, and conclude that Arrival is the best approach to use.

3 Selecting a Session Threshold Value

The dominant methodology to extract session data from activity logs is to postulate a certain threshold value, and assume that breaks in activity which are longer than this threshold represent a division between separate sessions. Such a threshold exists in all three approaches: Last, Max, and Arrival. The main difference between these definitions is the time interval that we compare to the threshold. In Arrival this interval starts at the arrival of the last job, in Last at the end of the last job, and in Max at the maximal end time among previous jobs. The threshold value that is chosen may have a strong effect on the resulting session properties [1]. In this section we will consider how to select the threshold value for each approach, and consider its influence on the sessions.



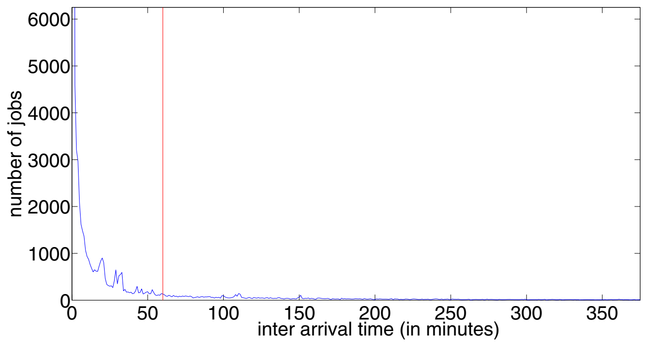
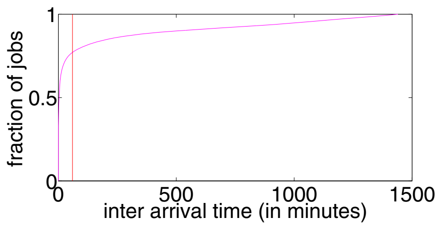


Fig. 2. The distribution of inter-arrival time in the SDSC-BLUE and SDSC-DS logs

As mentioned above, Last and Max are both popular approaches in this area. Therefore, many previous works have considered the selection of the the threshold value for them. The commonly used value is 20 minutes, because this seems to capture the majority of think times. For example, Zilber et al. and Shmueli [16, 12] used this threshold value.

As far as we know, there has been no previous work concerning the selection of a threshold on inter-arrival times for parallel workloads. Several different values have been used in the context of web workloads, including 30 minutes [2, 7], an hour [14], and even two hours [8]. To find what value would make a suitable threshold for our parallel workloads, we calculated the distribution of inter-arrival times for different logs available from the Parallel Workloads Archive [9]. Thus, for each user we found the difference between the arrival times of each pair of successive jobs. We ignored values that were above a day (1440 minutes), because such long intervals obviously defy the notion of a single session. Examples of the resulting distributions are shown in Figure 2. CDFs[2]

[2] The Cumulative Distribution Function (CDF) is the integral of the probability density function (pdf). For each value x, it gives the probability of observing values that are smaller than or equal to x. In the case of empirical data, it is the fraction of samples that are smaller than or equal to x.



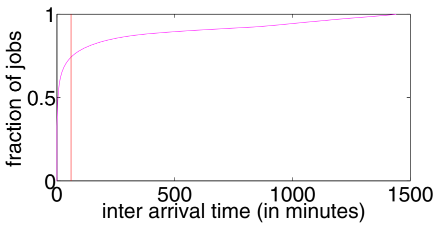
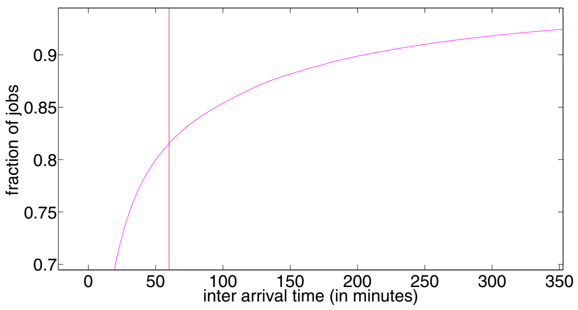


Fig. 3. CDFs of inter-arrival times in the SDSC-SP2 and KTH-SP2 logs



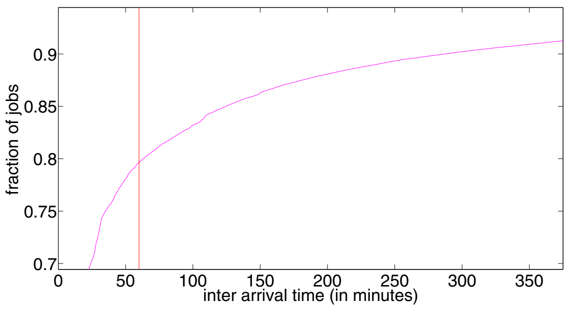


Fig. 4. Zoom in on the CDFs of inter-arrival times for the SDSC-BLUE and SDSC-DS logs

are shown in Figure 3 and Figure 4. In all these figures we added a vertical line at 60 minutes (1 hour), which is the threshold we eventually chose.

Our goal was to find the point in the distribution where the derivative doesn't changed much any more. From Figure 2 it appears that any value between approximately 25 minutes and 200 minutes will be logical. However, for values below 40 one can still observe an obvious drop in the distribution. This is even clearer in the CDF (and especially in the figures with zoom in). In the range of 100 to 200 minutes the slope is already very low, and therefore we would prefer a lower value for the threshold. We concluded that the value ought to be between 40 minutes to 100 minutes. We chose 60 minutes as it is in the middle of this range and is a round value (one hour). We do not claim this is necessarily the best value, but it seems that there is no other value that is obviously better.

Selecting a session threshold has a strong impact on the resulting analysis. If we were to select a higher threshold, jobs with longer intervals will nevertheless be grouped together. As a result the number of jobs in each session would grow and the number of sessions would decrease. In the following sections we provide an in-depth analysis of the implications of the selected session threshold values, mainly in terms of the distribution of session lengths.

4 Comparing Last and Max Using the Think-Time Distribution

As we mentioned above, the most common definition of sessions is based on think times, using Last or Max. In this section we investigate which definition leads to a more reasonable think time distribution. The problem is that we do not know what the think time distribution should be. We circumvent this problem as follows. First, we identify the batches according to both approaches separately, and calculate the think times. Then we create a list that contains the common batches (batches that are exactly the same according to both approaches). Based on this list, we extract the think times following these common batches. This provides us with two lists of think times: *CommonTTMax* and *CommonTTLast*. Note that the common batch think times may be different because the think times are defined differently in each approach. In Last, the think time is measured from the end of the last job, whereas in Max it is the maximal end time of all jobs. But we expect the distributions to be close, which indeed they are.

Given the agreement on the common batches, and the similarity of their think time distributions, we take this to represent the "real" distribution of think times. The remaining think times, that we didn't put in the common lists, represent the differences between think times of batches that were created according to Last and Max. Therefore, we would prefer the approach for which the distribution of unique think times is similar to the distribution of common think times.

The resulting distributions are shown in Figure 5. The first obvious conclusion from the graphs is that our expectation that the distributions for common batches shared by Max and Last would be very close to one another was correct. It is also quite clear that the distribution for unique batches as identified by Last is much closer to the common distributions than the distribution for unique batches as defined by Max. It is true that the distribution for Max is closer for larger values, but in most of the range, Last is a lot closer. We concluded that Last creates a more realistic distribution of think times. This supports the use of Last by Zilber et al. [16], Shmueli [12], and others.

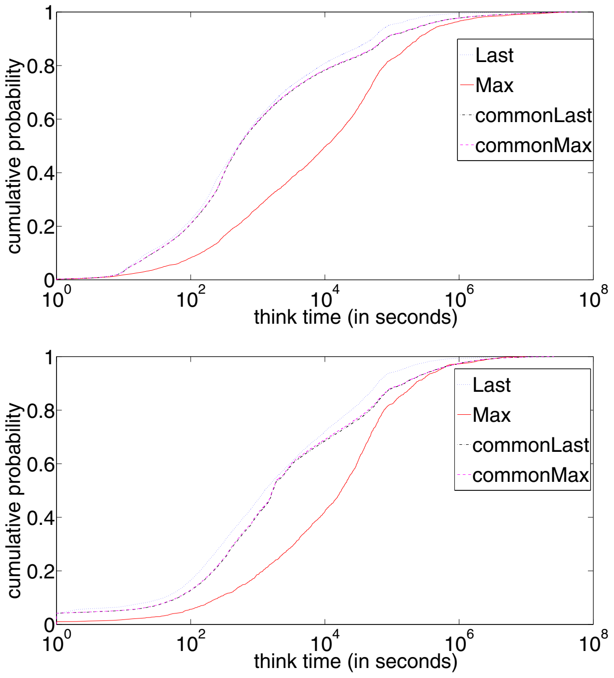


Fig. 5. Comparison between distributions of think times in the SDSC-BLUE and SDSC-DS logs

5 Artifacts in the Distribution of Session Lengths with Arrival

According to the work of Mehrzadi, using the Arrival approach may lead to artifacts in the distribution of session durations [6]. Specifically, he shows that in web search data the distribution of session lengths exhibits a pronounced drop exactly at the threshold value that was used to define the sessions. In order to check this, we examined the distribution of session durations for each of the three approaches. Due to the large number of very short sessions, we ignore sessions of up to 2 minutes. The results are shown in Figure Figure 6 for Last, Figure 7 for Max, and 8 for Arrival.

Upon examination of the graphs, we found that Last and Max behave very similarly, but Arrival is indeed different. According to the Max and Last approaches, the distribution of session lengths is essentially the same for different threshold values. The difference in heights is due to the fact that larger thresholds lead to a smaller number of sessions, but the behavior of each graph is the same. In addition, for all values, there are no obvious discontinuities.

In contrast, with the Arrival approach it is easy to notice a sharp drop in the distribution at the threshold value, exactly as had occurred in Mehrzadi's

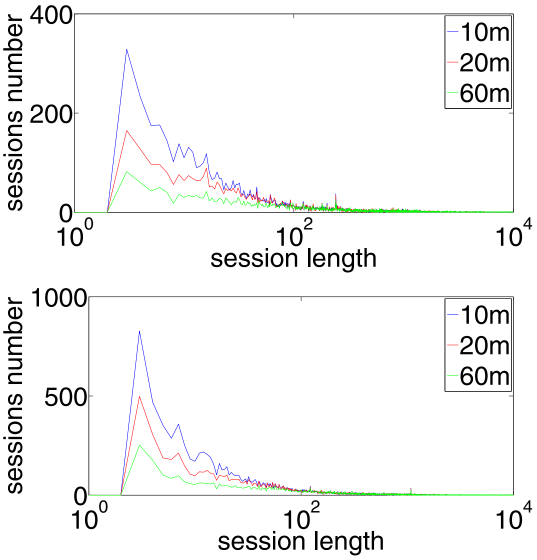


Fig. 6. Distribution of session lengths as created by Last, for the KTH-SP2 and SDSC-SP2 logs

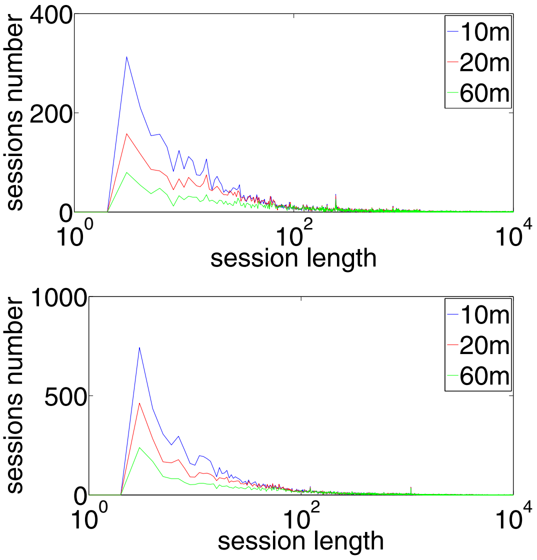


Fig. 7. Distribution of session lengths as created by Max, for the KTH-SP2 and SDSC-SP2 logs

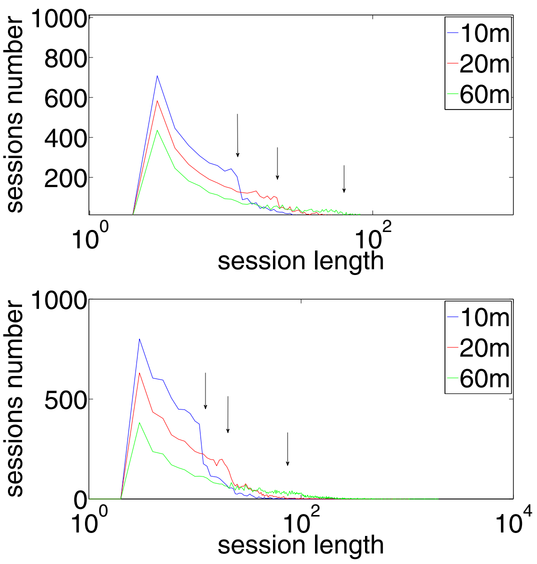


Fig. 8. Distribution of session lengths as created by Arrival, for the KTH-SP2 and SDSC-SP2 logs. Arrows denote threshold values.

data. In particular, each specific threshold value changes the distribution of session lengths in a different way (see arrows). However, this effect is reduced when we increase the value of the threshold. This is more evident in Figure 9. In this figure we ignored all session lengths below 9 minutes, and present the histogram without any connecting lines for clarity. With a 10 minute threshold the discontinuity is very significant. For 20 minutes the discontinuity is smaller (but still noticeable). With 60 minutes the drop becomes a step. It is worth mentioning that in some logs (although not in most) there is a clearer drop for 60 minutes, yet rather less dramatic than for 20.

In conclusion, we find that the Arrival approach is sensitive to the threshold value, although with large values (like the one we chose) the effect is rather small.

6 The Problem of Very Long Sessions

The graphs in Figures 6 and 7 show that with Last and Max most sessions are short, and few sessions are very long, possibly unrealistically so. However, it is impossible to see the details. In order to emphasize the long sessions we calculated the survival function[3], and present the results in Figure 10. This shows that when using the Last approach, approximately 13.5% of the session

[3] The survival function is the complement to the CDF: for each value x, it gives the probability of observing values that are larger than x.

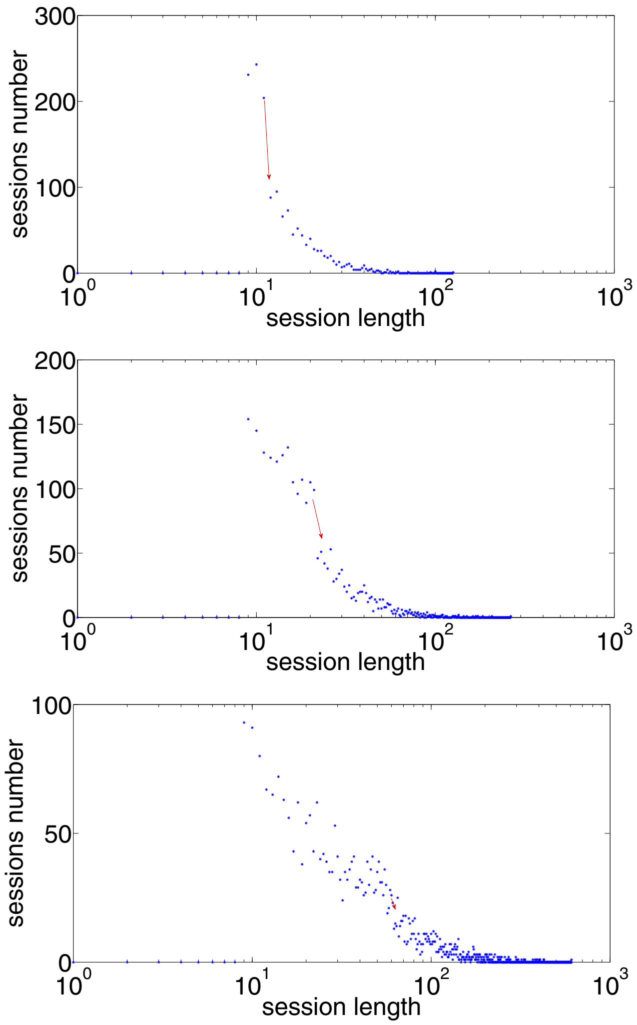


Fig. 9. Detailed view of discontinuities in the histogram of session lengths, using the Arrival approach, with thresholds of 10m (top), 20m (middle), and 60m (bottom), for the KTH-SP2 log

lengths are longer than 10^3 minutes (16 hours) in the SDSC-BLUE log, and 17.7% are longer than this value in the SDSC-DS log. With the Max approach, the percentages are a little higher: 15% in SDSC-BLUE and 19.5% in SDSC-SP2. In addition, one may notice that the maximum session length is above 10^5 minutes (approximately 70 days) in both logs.

Recall that a session is supposed to represent the time period when the user is active at the computer (the interval from when the user begins to work and until

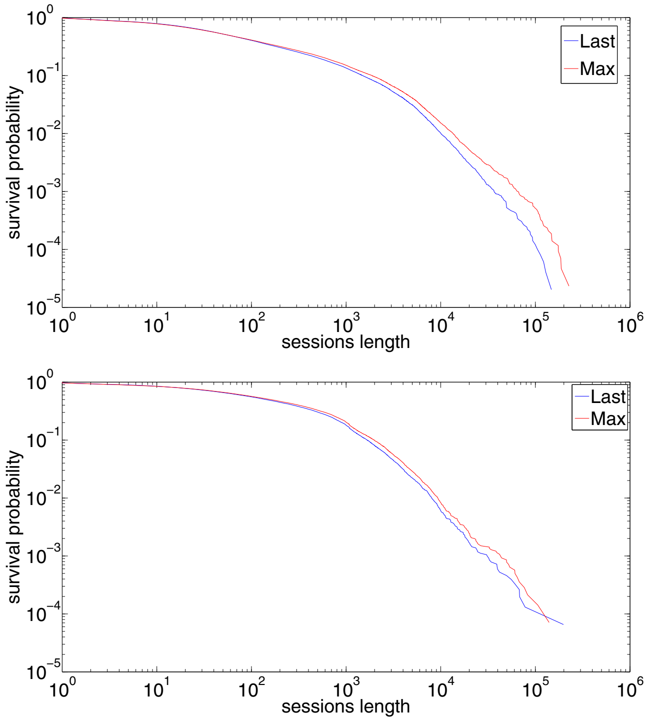


Fig. 10. The survival function of session lengths in the SDSC-BLUE and SDSC-SP2 logs, using log-log axes

he is done working). Therefore, session lengths above 10^4 minutes are impossible. In addition, even sessions of 16 hours are not reasonable. It should be very rare that a user would work this long continuously, yet still the results show that more than 13% of the sessions are that long. This is very unlikely.

The reason both approaches create sessions that are far too long so many times is that both are based on a wrong assumption. This is the assumption that all work is interactive. With interactive work, it is reasonable to assume that the users wait for the termination of each job, think for a while, and then send the next jobs. But on parallel supercomputers at least some of the work is not interactive. In particular, this is the case for very long jobs that run for many hours or even days. Including these very long jobs within the session, as is done by both Last and Max, then leads to unrealistically long sessions. For example, if a user sends out a job that takes 5 days, and after 3 days sends another job to the system, both approaches will put these two jobs into the same session, although the user most probably wasn't active in the system this whole time. The same problem may also occur on a smaller scale of a hew hours. If there are jobs that run during a break in the user's activity in the middle of the day (for example, during meetings or lunch), these jobs may overlap new work done after the user returns. Therefore, instead of a few short sessions of a couple of hours scattered along the day, we would get one long session — from the first job the user submits in the morning until after he goes home at night.

In order to avoid such problems, we suggest alternative versions of Last and Max which we call Last+Cut and Max+Cut. In these versions we define a new threshold value, called the *Cut*. Then, we use each job's arrival time plus Cut as its effective end time, instead of using the real end time, provided it is shorter. This means that if a job ends within Cut time from its arrival, we measure the think time from its end time without change. Otherwise, we use its arrival time + Cut as the start of the think time. Assuming a session threshold of T minutes we then have:

– Last+Cut: A job will belong to the current batch if it arrives before the arrival time of the last job + Cut + session-threshold (or the end time + session-threshold):

$$J[i].arr \leq \min\{\ J[i-1].end,\ J[i-1].arr + Cut\ \} + T$$

– Max+Cut: A job will belong to the current batch if it arrives before the maximum of the arrival times of all the jobs in the batch + Cut + session-threshold (or with end times):

$$J[i].arr \leq \max_{j<i}[\ \min\{\ J[j].end,\ J[j].arr + Cut\ \}\ ] + T$$

The results of using these approaches are shown in Figure 11. We checked three different values for Cut: 30 minutes, 1 hour, and 2 hours. (Last, Max, and Arrival are also included for comparison.) As expected, in all 3 cases the problem of overly long sessions is largely eliminated. Also, the difference between Max+Cut and Last+Cut with the same threshold is very marginal. Therefore we will distinguish between the Cut approaches only according to the threshold. In the SDSC-BLUE log, the fraction of sessions longer than 10^3 is a little more than 10^{-3} with a large Cut value of 2 hours, but with 1 hour or 30 minutes this fraction is only a little higher than 10^{-4} (approximately $10^{-3.9}$). In the SDSC-SP2 log, this fraction is approximately $10^{-3.4}$ with a 2 hours Cut, $10^{-3.7}$ with 1 hour, and $10^{-3.9}$ with 30 minutes. The value of the maximum session length is also dramatically decreased in the Cut approaches: down to 2600 minutes (43 hours) in the SDSC-BLUE log and less than 2000 minutes (33 hours) in the SDSC-SP2 log.

The conclusion is that the Cut approach creates more realistic session lengths. The longest sessions still seem to be too long, lasting nearly 2 days, but still this is much better than the sessions that last for more than 2 months we had before. While unreasonable for humans, such long sessions may be due to a short script or a number of people who might have replaced each other on the computer, sending the jobs through the same user name. In addition, the percentage of long sessions has dropped. Only a very small percentage of the sessions were

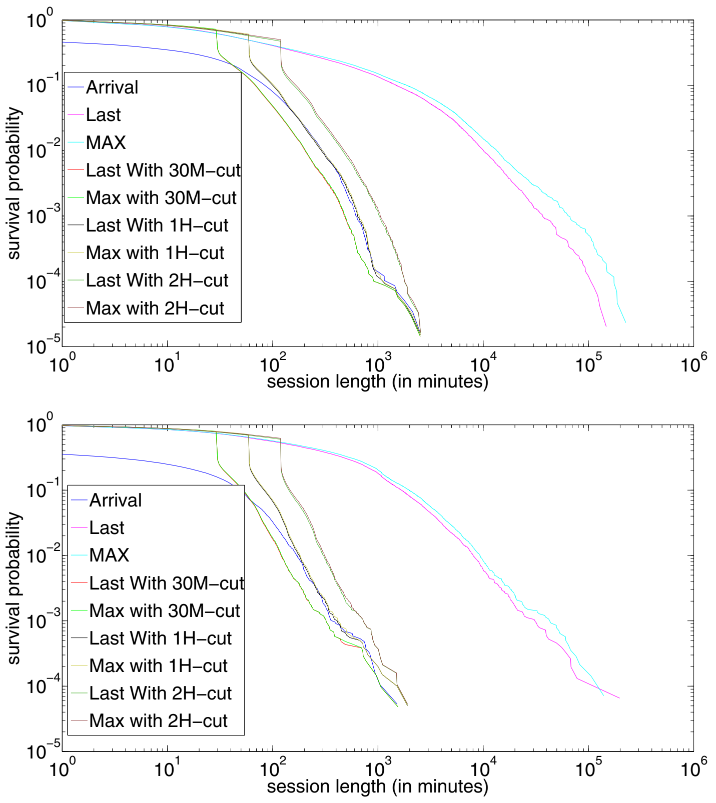


Fig. 11. The survival function of session lengths according to all the different approaches, for the SDSC-BLUE log and the SDSC-SP2 log

more than 1000 minutes (16 hours) long, in comparison to 13% or more with the original Last and Max.

However, although the length of the sessions in the Cut approaches are more realistic, the effect of the Cut value on the distribution is enormous: There is a very sharp drop in the graph at the point of the Cut value. In order to examine this effect, we created histograms of the session lengths generated by Last+Cut and Max+Cut. These are presented in Figure 12. It is easy to see that the Cut values produce a very significant mode in the distributions. The reason for these modes is as follows. For all the sessions with one job, if the job ends before the Cut value, the length will be the end time minus the arrival time. This part of the distribution will be continuous. But if the job ends after the Cut Value, the length will be the equal to the Cut value. Therefore, many sessions will receive the Cut value length.

The bottom line is that Last and Max remain problematic. In the original version, they create sessions that are way too long. Introducing the Cut heuristic

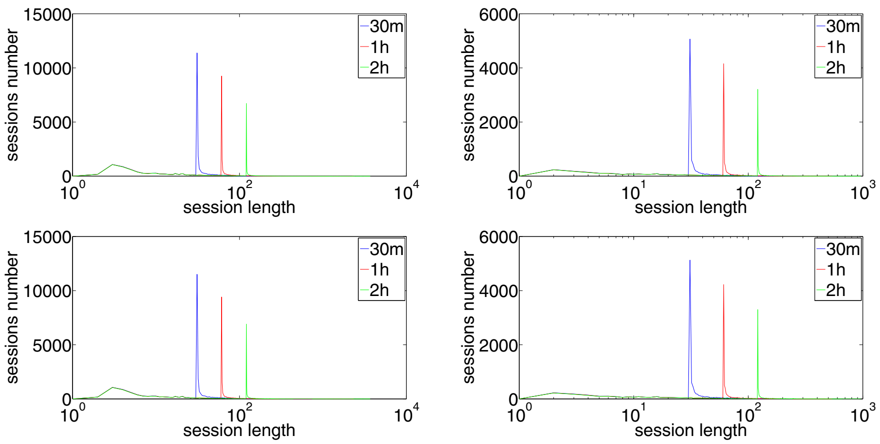


Fig. 12. Histograms of session lengths generated by Last+Cut (top) and Max+Cut (bottom) using the SDSC-DS and KTH-SP2 logs (left and right)

leads to a strong artifact in the distributions of session lengths. Hence, the only logical approach is to use the Arrival approach. It is equivalent to the Cut approach, where Cut=0, and with a larger session-threshold (60m instead of 20m). (Note that if the Cut value is 0, then Max+Cut is equivalent to Last+Cut.)

The Arrival approach produces realistic session lengths similar to the Cut approaches, But in addition, the distribution is smooth with no modes that depend on parameter values. Therefore, it seems that this approach creates the most sensible distribution of session lengths. We conclude that the Arrival approach, especially with a relatively long session threshold of 1 hour, is the most promising approach to delimit sessions.

7 Results with the Arrival Approach

Due to the fact that it is innovative and uncommon to use the Arrival approach to define sessions in parallel workloads, we present a few details and distributions of sessions and batches.

First, we present the number of jobs, batches, and sessions in Table 1. In all of the logs the ratios are very similar. On average, the number of jobs is a little

Table 1. Number of jobs, batches, and sessions in the main logs

Log	Jobs	Batches	Sessions
SDSC-SP2	54,051	32,614	18,730
SDSC-DS	85,003	41,679	24,294
KTH-SP2	28,489	16,488	10,303
SDSC-BLUE	223,407	136,460	58,311

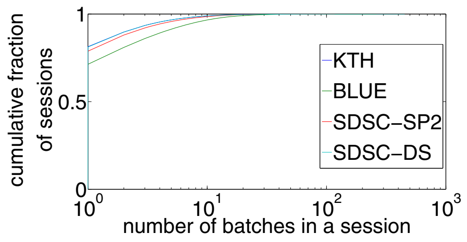


Fig. 13. CDF of the number of batches in a session

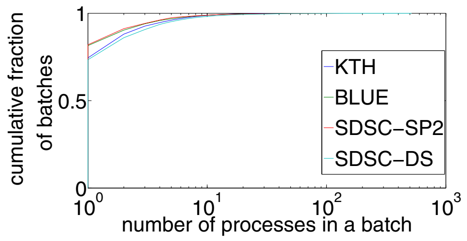


Fig. 14. CDF of the number of jobs in a batch

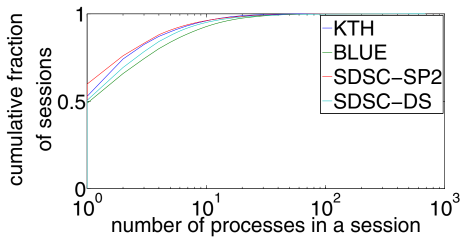


Fig. 15. CDF of the number of jobs in a session

less than twice the number of batches, and the number of batches is a little less than twice the number of sessions. Additional data on batches and sessions are presented in Figure 13, Figure 14, and Figure 15. A very important observation is that generally more than 50% of the sessions and 75% of the batches contain only one job. This means that when users work with supercomputers, most of the time they send out a single job and then stop their interaction with the computer for a while. However, it is important to note that some sessions have very many jobs, so the distribution is skewed, and most jobs do not constitute single-job sessions.

8 Conclusions

A summary of the methods that can be used to identify sessions when analyzing parallel workloads is given in Table 2.

Table 2. Summary of approaches and their effect on the session length distribution

Approach	*Issues*
Last, Max	excessively long sessions
Last+Cut, Max+Cut	strong peak at cut value
Arrival	peak at threshold value many zero-length sessions

The most common approach is to use the Last and Max approaches. These approaches are based on setting a threshold on think times: if the think time is long, this is assumed to be a break between sessions. However, these approaches occasionally cause extremely long sessions, due to the fact that some of the jobs running on such systems are extremely long.

A possible improvement is to use Last+Cut or Max+Cut. This eliminates the very long sessions, at the price of producing a strong peak in the distribution of session length at the value of the cut threshold being used. This is also undesirable.

The alternative is to use the Arrival approach, as in commonly done in other domains, such as the analysis of web workloads. In this approach, inter-arrival times are used. If the inter-arrival is longer than some threshold, a session break is assumed. The main problem with this approach is that long sessions may not be identified correctly, and again a peak in the distribution is created at the value of the threshold being used. However, the size of this peak decreases with increasing threshold values. We suggest to use a threshold of 1 hour. With such a threshold the peak in the distribution of session lengths is very small.

The obvious deficiency with the above is that it is based on common sense, not on data. A desirable avenue for future work is therefore to conduct a user study in which the actual activity patterns of users are followed, and this is correlated with their job submittal patterns.

Acknowledgments. Many thanks to all those who have made their workload data available through the Parallel Workloads Archive.

References

1. Arlitt, M.: Characterizing web user sessions. Performance Evaluation Rev. 28(2), 50–56 (2000)
2. Downey, D., Dumais, S., Horvitz, E.: Models of searching and browsing: Languages, studies, and applications. In: 20th Intl. Joint Conf. Artificial Intelligence, pp. 1465–1472 (January 2007)

3. Jann, J., Pattnaik, P., Franke, H., Wang, F., Skovira, J., Riodan, J.: Modeling of Workload in MPPs. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1997. LNCS, vol. 1291, pp. 95–116. Springer, Heidelberg (1997)
4. Jansen, B.J., Spink, A., Blakely, C., Koshman, S.: Defining a session on web search engines. J. Am. Soc. Inf. Sci. & Tech. 58(6), 862–871 (2007)
5. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. J. Parallel & Distributed Comput. 63(11), 1105–1122 (2003)
6. Mehrzadi, D., Feitelson, D.G.: On extracting session data from activity logs. In: 5th Intl. Syst. & Storage Conf. (June 2012)
7. Menascé, D.A., Almeida, V.A.F., Riedi, R., Ribeiro, F., Fonseca, R., Meira Jr., W.: A hierarchical and multiscale approach to analyze E-business workloads. Performance Evaluation 54(1), 33–57 (2003)
8. Montgomery, A.L., Faloutsos, C.: Identifying web browsing trends and patterns. Computer 34(7), 94–95 (2001)
9. Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/
10. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed: A cautionary tale. In: 3rd Networked Systems Design & Implementation, pp. 239–252 (May 2006)
11. Shmueli, E., Feitelson, D.G.: Using site-level modeling to evaluate the performance of parallel system schedulers. In: 14th Modeling, Anal. & Simulation of Comput. & Telecomm. Syst., pp. 167–176 (September 2006)
12. Shmueli, E., Feitelson, D.G.: Uncovering the effect of system performance on user behavior from traces of parallel systems. In: 15th Modeling, Anal. & Simulation of Comput. & Telecomm. Syst., pp. 274–280 (October 2007)
13. Shmueli, E., Feitelson, D.G.: On simulation and design of parallel-systems schedulers: Are we doing the right thing? IEEE Trans. Parallel & Distributed Syst. 20(7), 983–996 (2009)
14. Shriver, E., Hansen, M.: Search Session Extraction: A User Model of Searching. Tech. rep., Bell Labs (January 2002)
15. Silverstein, C., Henzinger, M., Marais, H., Moricz, M.: Analysis of a very large web search engine query log. SIGIR Forum 33(1), 6–12 (1999)
16. Zilber, J., Amit, O., Talby, D.: What is worth learning from parallel workloads? a user and session based analysis. In: 19th Intl. Conf. Supercomputing, pp. 377–386 (June 2005)

Performance and Fairness for Users in Parallel Job Scheduling

Dalibor Klusáček[1,2] and Hana Rudová[1]

[1] Faculty of Informatics, Masaryk University
Botanická 68a, Brno, Czech Republic
[2] CESNET z.s.p.o., Zikova 4, Prague, Czech Republic
{xklusac,hanka}@fi.muni.cz

Abstract. In this work we analyze the performance of scheduling algorithms with respect to fairness. Existing works frequently consider fairness as a job related issue. In our work we analyze fairness with respect to different users of the system as this is a very important real-life problem. First, we discuss how fair are selected popular scheduling algorithms with respect to different users of the system. Next, we present an extension to the well known Conservative backfilling algorithm. Instead of "ad hoc" decisions, the schedule is now created subject to evaluation and optimization. Notably, the fairness is considered as an important metric, which accompanies standard performance related metrics such as slowdown or wait time. To achieve that, an inclusion of fairness as an optimization criterion is proposed. The new extension improves the performance and fairness of Conservative backfilling with respect to other classical techniques such as FCFS, EASY backfilling or aggressive backfilling without reservations.

Keywords: Scheduling, Fairness, Metaheuristic, Backfilling.

1 Introduction

This paper is inspired by the lessons learned over the past years when analyzing the job scheduling problem in the Czech National Grid Infrastructure MetaCentrum [24]. During that time it became apparent that for satisfactory scheduling several major principles should be met. First of all, the scheduler must guarantee good performance regarding classical performance related metrics such as low job wait times and slowdowns and high system utilization. At the same time, fairness has shown to be one of the most important factors to keep users satisfied. Therefore the users should be treated in a fair fashion, such that the available computing power is fairly distributed among them [13]. Last but not least, the predictability, i.e., planning functionality [10,25] was found to be very useful as it allows users to better understand when and where their jobs will be executed. In fact, even experienced users often do not understand the scheduling decisions as delivered by existing scheduler that does not use planning functionality.

In order to deal with this situation we have proposed an optimization procedure designed to improve the performance of well known *Conservative backfilling* algorithm [4,31,22]. The choice of Conservative backfilling is straightforward as it allows predictability by establishing reservation for every waiting job. In Conservative backfilling, the plan of job reservations is created in an "ad hoc" fashion as new jobs arrive. This approach may not guarantee good solutions as previously established scheduling decisions are fixed and do not change even when it is obvious that better solution exists. At such situation it is often useful to "reconsider" previous decisions. For this purpose we apply two core strategies: the evaluation procedure and the optimization procedure. The evaluation is used to identify inefficient scheduling decisions and it guides the optimization procedure toward better schedules. Beside common performance related criteria it also focuses on the problem of maintaining *fairness among different users* of the system as this is in fact one of the most important features that a production system should guarantee. Together, the proposed extension improves the performance and fairness of the original Conservative backfilling with respect to other classical techniques such as FCFS, EASY backfilling or aggressive backfilling without reservations.

The paper is organized as follows. First, we define the studied problem and discuss suitable optimization criteria that are lately applied in our study. Next we discuss popular scheduling algorithms, closely describing their strengths and weaknesses while emphasizing fairness related issues. Section 4 presents applied optimization of the Conservative backfilling, i.e., the evaluation procedure and the optimization metaheuristic. Following section presents experimental evaluation where the proposed extension of Conservative backfilling is experimentally compared with the original Conservative backfilling as well as with other popular scheduling algorithms such as FCFS, EASY backfilling or aggressive backfilling without reservations. Finally we conclude our paper with a short summary and we discuss the future work.

2 Problem Description

Let us briefly define the considered job scheduling problem as well as the applied optimization criteria that have been used to express the general requirements on the proposed job scheduler.

2.1 System Description

We consider a classical scenario where the system is managed by one centralized scheduler, which has complete control over all jobs and system resources.

Job represents a user's application that may require one (sequential) or more CPUs (parallel). Also the arrival time and the job processing time are specified. There are no precedence constraints among jobs and we consider neither job preemptions nor migrations from one machine to another. Each job has its owner. When needed, the runtime estimates are precise (perfect) in this study.

The system is composed of one or more computer clusters and each cluster is composed of several machines. So far, we expect that all machines within one cluster have the same parameters. Those are the number of CPUs per machine and the CPU speed. All machines within a cluster use the Space Slicing processor allocation policy [6], which allows the parallel execution of several jobs at the cluster when the total amount of requested CPUs is less or equal to the number of CPUs of the cluster. Therefore, several machines within the same cluster can be co-allocated to process a parallel job. On the other hand, machines belonging to different clusters cannot be co-allocated to execute the same parallel job.

As we already briefly mentioned in Introduction, the proposed scheduler should guarantee both good performance and fairness. Therefore we now define several criteria that were considered when designing the new scheduler and lately used when evaluating its performance with respect to other existing scheduling techniques.

2.2 Classical Performance Related Criteria

There are several popular metrics used to measure the efficiency of scheduling algorithms. Frequently, makespan and machine usage are used as a general indicator of algorithm's suitability [40,39]. However, these criteria are not very suitable for systems that are running for a long time. In fact, when the considered time period is long enough, then different algorithms generate similar values of makespan and machine usage. This is not a surprising fact [8,20] because in such case, the resulting makespan — which is then used to calculate the machine usage — is not controllable by the scheduler since it can never be smaller than the arrival time of the last job plus its processing time. Then, the utilization is rather a function of user activity than of scheduler's performance [8].

Therefore, we have decided to use classical performance related metrics: the avg. response time [6], the avg. wait time [3] and the avg. bounded slowdown [6]. The avg. response time represents the average time a job spends in the system, i.e., the time from its submission to its termination. The avg. wait time is the mean time that the jobs spend waiting before their execution starts. The avg. bounded slowdown is the mean value of all jobs' bounded slowdowns. Slowdown is the ratio of the actual response time of the job to the response time if executed without any waiting. If a job has a very small runtime it often means that the job ended prematurely due to some error. As a result, its slowdown can be huge, which may seriously skew the final mean value. Therefore, so called *bounded slowdown* is often applied [4,6], where the minimal job runtime is guaranteed to be greater than some predefined time constant, sometimes called a "threshold of interactivity" [6]. However, there is no general agreement concerning the actual value of this threshold. Sometimes it is equal to 10 seconds [4,6], while different authors use, e.g., 1 minute [37]. Since the resulting value is very sensitive to the applied threshold value, we set the threshold equal to 1 second in this paper. It allows us to eliminate problems related to extremely short jobs while keeping the resulting values close (comparable) to the values of "normal" slowdown in most cases.

All three criteria are to be *minimized.* As pointed out by Feitelson et al. [6], the use of response time places more weight on long jobs and basically ignores if a short job waits few minutes, so it may not reflect users' notion of responsiveness.

Slowdown reflects this situation, measuring the responsiveness of the system with respect to the job length, i.e., jobs are completed within the time proportional to the job length. Wait time criterion supplies the slowdown and response time. Short wait times prevent the users from feeling that the scheduler "forgot about their jobs".

2.3 Fairness Related Criteria

So far, all criteria focused either on the system or the job performance. Still, good performance is not the only aspect that makes the scheduler acceptable. The scheduler must be also fair, i.e., it must guarantee that the available computing power will be fairly distributed among the users of the system. As far as we know there is no widely accepted standardized metric to measure fairness and different authors use different metrics [29,30,28,37,21,31]. A *fair start time* (FST) metric is used in [29,21]. It measures the influence of later arriving jobs on the execution start time of currently waiting jobs. FST is calculated for each job, by creating a schedule assuming no later jobs arrive. The resulting "unfairness" is the difference between FST and the actual start time. Similar metric is so called *fair slowdown* [31]. The fair slowdown is computed using FST and can be used to quantify the fairness of a scheduler by looking at the percentage of jobs that have a higher slowdown than their fair slowdown [31]. Another metric measures to what extent each job was able to receive its "share" of the resources [30,28]. The basic idea is that each job "deserves" $1/n^t$ of the resources, where n^t is the number of jobs present in the system at the given time t. The "unfairness" is computed by comparing the resources consumed by a job with the resources deserved by the job. An overview of existing techniques including discussion of their suitability can be found in [26].

Fairness is usually understood and represented as a *job related* metric, meaning that every job should be served in a fair fashion with respect to other jobs [29,30,28,37,21,31]. Such requirements are already partially covered by the common performance criteria like the slowdown and the wait time that were both discussed earlier. Moreover, job fairness has been also reflected in the design of common scheduling algorithms (see Section 3) and the results concerning selected job fairness indicators are well known [30,28,37,31]. In this work, we aim to guarantee fair performance to *different users* of the system as well. Therefore, we apply a well know fair-share principle [13] with respect to different users of the system. Fair-share tries to minimize the differences among the normalized mean wait times of all users. Let o be a given user (job owner) in the system and $\mathcal{JOBS}_o$ be the set containing jobs of user o. Let r_j and S_j denote the arrival time and start time of given job j respectively. Then the *normalized user wait time* ($NUWT_o$) for each user (job owner) o is calculated as shown by Formula 1.

$$NUWT_o = \frac{TUWT_o}{TUSA_o} \tag{1}$$

$$TUWT_o = \sum_{j \in \mathcal{JOBS}_o} (S_j - r_j) \tag{2}$$

$$TUSA_o = \sum_{j \in \mathcal{JOBS}_o} (p_j \cdot usage_j) \tag{3}$$

Normalized user wait time $NUWT_o$ is the *total user wait time* (see $TUWT_o$ in Formula 2) divided (normalized) by the so called *total user squashed area* (see $TUSA_o$ in Formula 3), which can be described as the sum of products of the job runtime (p_j) and the number of requested processors ($usage_j$). This user-oriented metric is based on more general *total squashed area* metric proposed in [3]. The normalization is used to prioritize less active users over those who utilize the system resources very frequently [13]. Such normalization is commonly used as can be seen, e.g., in the Czech National Grid Infrastructure MetaCentrum [24] where the normalization is used when monitoring and adjusting fairness in the production TORQUE scheduling system [1] as well as it was used earlier in the PBS Pro scheduling system [11]. Similar approach has been also adopted in the ASCI Blue Mountain supercomputer cluster when establishing job priorities in the fair-share mechanism [13].

The normalized user wait time ($NUWT_o$) can be used "on the fly" by the scheduling algorithm to dynamically prioritize the users. Also, it can be used in the graphs with the experimental results (e.g., Fig. 2) to reflect the resulting fairness of the applied scheduling algorithm. In this case, the interpretation is following. The closer the resulting $NUWT_o$ values of all users are to each other, the higher is the fairness. If the $NUWT_o$ value is less than 1.0, it means that the user spent more time by computing than by waiting, which is advantageous. Similarly, values greater than 1.0 indicate that the total user wait time is larger than the computational time of his or hers jobs.

3 Fairness vs. Performance in Scheduling Algorithms

In this section we recapitulate several popular scheduling algorithms that are widely used both in practice and in the literature. We will discuss their strengths and weaknesses with respect to classical performance related metrics as well as with respect to fairness related issues.

All production systems support trivial *First Come First Served (FCFS)* scheduling policy [11,23]. FCFS always schedules the first job in the queue, checking the availability of the resources required by such job. If all the resources required by the first job in the queue are available, it is immediately scheduled for the execution, otherwise FCFS waits until all required resources become available. While the first job is waiting for the execution none of the remaining jobs can be scheduled, even if the required resources are available. Despite its simplicity, FCFS approach presents several advantages. It does not require an estimated processing time of the job and it guarantees that the response time of a job that arrived earlier does not depend on the execution times of jobs that arrived later. As there is no "queue jumping" FCFS is in some sense a very fair scheduler [25,31,30]. On the other hand, if parallel jobs are scheduled then this fairness related property often implies a low utilization of the system resources, that cannot be used by some "less demanding" job(s) from the queue [29,23]. To solve this problem algorithms based on backfilling are frequently used [25].

Algorithms using *backfilling* are an optimization of the FCFS algorithm that try to maximize the resource utilization [23]. There are several variants of backfilling algorithms. The most popular one is the aggressive *EASY backfilling* [25]. It works as FCFS but when the first job in the queue cannot be scheduled immediately EASY backfilling calculates the earliest possible starting time for the first job using the processing time estimates of running jobs. Then, it makes a reservation to run the job at this pre-computed time. Next, it scans the queue of waiting jobs and schedules immediately every job not interfering with the reservation of the first job [23]. This helps to increase the resource utilization, since idle resources are *backfilled* with suitable jobs, while decreasing the average job wait time.

EASY Backfilling takes an aggressive approach that allows short jobs to skip ahead provided they do not delay the job at the head of the queue. The price for improved utilization of EASY Backfilling is that execution guarantees cannot be made because it is hard to predict the size of delays of jobs in the queue. Since only the first job gets a reservation, the delays of other queued jobs may be, in general, unbounded [25][1]. Therefore, without further control, EASY does not guarantee good fairness.

In order to prevent such situation, the number of reservations can be increased. In case of slack-based [34] and selective backfilling [31] the number of jobs with a reservation is related to their current wait time and slowdown respectively. *Conservative backfilling* [4,31,22] makes reservation for every queued job which cannot be executed at the given moment. It means that backfilling is performed only when it does not delay any previous job in the queue. Clearly, this reduce the core problem of EASY backfilling where jobs close to but not yet at the head of the queue can be significantly delayed. The price paid is that the number of jobs that can utilize existing gaps[2] is reduced, implying that more gaps are left unused in Conservative backfilling than in EASY backfilling [26]. Still, both approaches lead to significant performance improvements compared to FCFS [26,7]. As the scheduling decisions are made upon job submittal, it can be predicted when each job will run, giving the users execution guarantees. Users can then plan ahead based on these guaranteed response times. Obviously, there is no danger of starvation as a reservation is made for every job that cannot be executed immediately. Apparently, such approach places a greater emphasis on predictability [25,4] and it is a good compromise between "fair" but inefficient FCFS and "unfair" but efficient EASY backfilling.

All previously mentioned variants of backfilling require that each job specifies its estimated execution time. Therefore, existing systems also support backfilling without reservations [19,11] where estimates are not needed at all. In order to

[1] If a job is not the first in the queue, new jobs that arrive later may skip it in the queue. While such jobs do not delay the first job in the queue, they may delay all other jobs and the system cannot predict when a queued job will eventually run [25].

[2] Some authors [26] call the unused CPU time slot a "hole" while others [25,36] prefer the term "gap".

maintain fairness among users, the queue(s) can be ordered according to some priority mechanism such as fair-share. For example, the TORQUE scheduler [1] currently used in MetaCentrum [24] uses backfilling without reservations where each queue is ordered according to priorities computed using the fair-share principle. Here, the users of the system are prioritized according to the fair-share mechanism that balances the amount of consumed CPU time among users. This means that all jobs belonging to a given user get the priority that is equal to the user's priority[3]. Still, as in the case of EASY backfilling, such techniques cannot guarantee starvation-free behavior and additional mechanisms are needed to minimize the risk that the delays of queued jobs become very large.

In this section we tried to mention the most popular scheduling algorithms and we tried to demonstrate their pros and cons. None of these algorithms suits our needs perfectly. FCFS is somehow fair but inefficient. EASY backfilling improves the performance significantly but at some situations may degrade the performance for "unlucky" jobs. Similarly, backfilling without reservations do not need estimates but it cannot guarantee starvation-free behavior.

From this point of view, Conservative backfilling is a good candidate for further extension. First, as each job gets a reservation waiting jobs cannot be delayed by lately arriving jobs, which is fair, at least from the user's point of view. Second, job reservations, i.e., the plan of execution, are good for the users as they can get some sort of guarantee and they "know what is happening". Last but not least, the prepared plan of execution can be easily evaluated with respect to selected optimization criteria, covering both performance and fairness related objectives. Therefore, possible inefficiencies that can appear in classical Conservative backfilling can be identified and fixed. For this purpose some form of metaheuristic algorithm seems to be a natural solution. In the next section we describe such an extension of the Conservative backfilling.

4 Optimization of Conservative Backfilling

As we mentioned in previous text, we found Conservative backfilling to be a good initial scheduling technique for our purposes, which included requests of good performance, fair distribution of available computing power among users and predictability. In this section we describe the two fundamental techniques used to improve the overall performance of Conservative backfilling. Those techniques are *evaluation* of existing solution and an *optimization metaheuristic* that improves the quality of generated solution with respect to considered criteria. We start with a description of the evaluation method.

4.1 Evaluation Procedure

The purpose of the evaluation procedure is to compare two different schedules and decide, which one is better with respect to applied optimization criteria. As

[3] The priority is computed using the Formula 1 that represents the normalized user wait time $NUWT_o$.

we already discussed in Section 2, we focus both on the classical and the fairness related criteria. We use the avg. wait time (WT), the avg. response time (RT) and the avg. bounded slowdown (BSD) to measure the performance of the scheduling algorithm. Each such metric can be easily used when deciding, which solution is better — the one having smaller values of given metric. When considering fairness with respect to different users the situation is more complicated. The fairness related *normalized user wait time* ($NUWT_o$) described in Section 2.3 cannot be directly used as it is only a per user metric. For our purpose we needed a function that for given schedule returns a *single value*. Therefore, we have proposed a criterion called *fairness* (F), which is computed as shown by Formula 5.

$$UWT = \frac{1}{u}\sum_{o=1}^{u} NUWT_o \tag{4}$$

$$F = \sum_{o=1}^{u} (UWT - NUWT_o)^2 \tag{5}$$

First, we calculate the mean *user wait time* (UWT) using the values of $NUWT_o$ as shown in Formula 4. Then the fairness F is calculated by the Formula 5. The squares in F definition guarantee that only positive numbers are summed and that higher deviations from the mean value are more penalized than the small ones. This approach has been inspired by the widely used *Least Squares method* [38] where similar formula of squared residuals is minimized when fitting values provided by a model to observed values.

The fairness (F) criterion is used during the evaluation of performed scheduling decisions, i.e., "inside" the optimization procedure (see Section 4.2). When two possible solutions are available, then the values of F are computed for both of them. The one having smaller F value is considered as more fair. The squares used during computation of F help to highlight unfair assignments, which is very favorable when performing scheduling decisions. Sadly, the squares basically prevent us to reasonably interpret the resulting F values as it was possible in case of the $NUWT_o$ criterion (see discussion in Section 2.3). This is the reason why $NUWT_o$ is used in graphs to display how fair the solution has been with respect to different users while F is used when evaluating two different schedules "inside" the optimization algorithm.

Together, there are four criteria to be optimized simultaneously. Each criterion produces one value characterizing the solution. The final decision on which of the two solutions is better is implemented in separate function, called SELECTBETTER($schedule_A$, $schedule_B$) which is shown in Algorithm 1. It is a form of a *weight function*, which is often used when solving multi-criteria optimization problems [39,18,17]. The SELECTBETTER function is an extended version of the function that has been already successfully used in our previous works [17,18]. This extension includes the fairness criterion.

This function uses two inputs — the two solutions that will be compared. The $schedule_A$ may represent existing (previously accepted) solution while $schedule_B$

Algorithm 1. SelectBetter($schedule_A, schedule_B$)

```
1: compute $BSD_A$, $WT_A$, $RT_A$, $F_A$ according to $schedule_A$;
2: compute $BSD_B$, $WT_B$, $RT_B$, $F_B$ according to $schedule_B$;

3: $v_{BSD} := (BSD_A - BSD_B)/BSD_A$;
4: $v_{WT} := (WT_A - WT_B)/WT_A$;
5: $v_{RT} := (RT_A - RT_B)/RT_A$;
6: $v_F := (F_A - F_B)/F_A$;
7: $weight := v_{BSD} + v_{WT} + v_{RT} + v_F$;

8: if $weight > 0$ then
9:     return $schedule_B$;
10: else
11:     return $schedule_A$;
12: end if
```

represents the newly created *candidate solution*, a product of optimization. First, the values of used objective functions are computed for both schedules (lines 1–2). Using them, decision variables v_{BSD}, ..., v_F are computed (see lines 3–6). Their meaning is following: when the decision variable is positive it means that the $schedule_B$ is better than the $schedule_A$ with respect to the applied criterion. Strictly speaking, decision variable defines percentual improvement or deterioration in the value of objective function of $schedule_B$ with respect to the $schedule_A$. Some trivial correction is needed when the denominator is equal to zero, to prevent division by zero error. To keep the code clear we do not present it here. It can easily happen, that for given $schedule_B$ some variables are positive while others are negative. In our implementation the final decision is taken upon the value of the *weight* (line 7), which is computed as the (weighted) sum of decision variables. If desirable, the "importance" of each decision variable can be adjusted using a predefined weight constant. However, proper selection of these weights is not an easy task. In this particular case, all decision variables are considered as equally important and no additional weight constants are used. When the resulting *weight* is positive the (candidate) $schedule_B$ is returned as a better schedule. Otherwise, the (existing) $schedule_A$ is returned.

4.2 Optimization Algorithm

Newly arriving jobs are added into the schedule using classical Conservative backfilling algorithm. It means that the earliest suitable time slot is found in existing schedule. Such initial schedule ($schedule_{initial}$) is periodically optimized with an optimization algorithm. It is a simple metaheuristic that tries to optimize the existing $schedule_{initial}$. The algorithm includes an important feature that is typical for the *Tabu search*-based algorithms [9,27]. It is a short term memory called *tabu list* where few previously manipulated jobs are stored. If the algorithm is trying to move some job then this change is not allowed in case that this job is present in the tabu list. It has limited size and the oldest item is always removed when the list becomes full. Tabu list helps to protect the algorithm against short cycles where the same few jobs are repeatedly selected as the move candidates. The main structure of the algorithm is based on a procedure that has been already successfully used in our previous work which focused on a different problem involving minimizing the number of late jobs [17].

Algorithm 2. TabuSearch($schedule_{initial}, iterations, time\_limit$)

```
1: $schedule_{new} := schedule_{initial}$; $schedule_{best} := schedule_{initial}$; $tabu\_list := \emptyset$;
2: $i := 0$;
3: **while** ($i < iterations$ **and** $time\_limit$ not exceeded) **do**
4:    $i := i + 1$;
5:    $job$ := select random $job$ from $schedule_{new}$ such that $job \notin Tabu$;
6:    **if** $job = null$ **then**
7:       $tabu\_list := \emptyset$;    (all jobs were tested, reset the tabu list)
8:       **continue**;
9:    **end if**
10:   remove $job$ from $schedule_{new}$;
11:   compress $schedule_{new}$;
12:   move $job$ into earliest suitable gap in $schedule_{new}$;
13:   $schedule_{best}$ := SelectBetter($schedule_{best}, schedule_{new}$);
14:   $schedule_{new} := schedule_{best}$;    (update/reset candidate schedule)
15:   **if** $tabu\_list$ is full **then**
16:      remove the oldest item;
17:   **end if**
18:   $tabu\_list := tabu\_list \cup job$;
19: **end while**
20: **return** $schedule_{best}$;
```

TabuSearch($schedule_{initial}, iterations, time\_limit$) optimization algorithm is described in Algorithm 2. It has three inputs — the schedule that will be optimized, the maximal number of iterations and a time limit. In each iteration, one random non-tabu job selected (line 5). Once the job is selected, it is removed from its current position and the schedule is immediately compressed. The compression is designed to shift reservations to earlier time slots that could have appeared as a result of the job removal [14]. This procedure is an analogy to the method used in Conservative backfilling when job terminates earlier due to an overestimated runtime estimate [25]. During the compression the relative ordering of job start times is kept [14]. Next, the removed job is returned to the compressed schedule into the earliest suitable gap and this new schedule is evaluated with respect to applied criteria in the SelectBetter($schedule_{best}, schedule_{new}$) function (see Algorithm 1). If this attempt is successful SelectBetter returns $schedule_{new}$ as the new $schedule_{best}$, otherwise $schedule_{best}$ is not changed (line 13). Next, the $schedule_{new}$ is updated with the $schedule_{best}$ (line 14). Finally, the job is placed into the $tabu\_list$ (line 18) — so that it cannot be chosen in the next few iterations — and a new iteration of the Tabu search starts. If in some iteration all jobs are already in the tabu list, then the list is emptied and another iteration starts (line 7)[4]. The cycle continues until the predefined number of iterations or the given time limit is reached (lines 3). Then, the $schedule_{best}$ is returned as the newly found solution (line 20).

When applied, TabuSearch is executed every 5 minutes of simulation time. Here we were inspired by the actual setup of the scheduler [1] used in the MetaCentrum, which performs periodic priority updates of jobs waiting in the queues with a similar frequency. The maximum number of iterations is equal to the number of currently waiting jobs (schedule size) multiplied by 2. The *time_limit* variable was set to be 2 seconds, which is usually enough to try all iterations. However, when some higher priority event such as new job arrival or job completion is detected during TabuSearch execution, the *time_limit* is immediately set to 0 and the TabuSearch terminates. Therefore, the optimization phase cannot cause any significant delays concerning job processing and the potential overhead of optimization is practically eliminated [17].

5 Experiments

This section presents the experimental evaluation where the proposed Tabu Search optimization technique is experimentally compared with selected popular scheduling algorithms. Beside common performance related criteria also the fairness related issues of considered scheduling algorithms are analyzed here.

5.1 Experimental Setup

All experiments have been computed on an Intel Xeon 3.0 GHz machine with 12 GB of RAM using the GridSim [33] based *Alea* simulator we have implemented [15].

The proposed Tabu search-based optimization (TS) of Conservative backfilling has been evaluated against several existing algorithms that have been all closely discussed in Section 3. We have considered FCFS, aggressive backfilling without reservations (BF), EASY backfilling (BF-EASY), Conservative backfilling (BF-CONS) and the aggressive backfilling without reservations prioritized according to fair-share (BF-FAIR).

Six different data sets from the Parallel Workloads Archive [5] have been used in the simulation: MetaCentrum (806 CPUs, 103,656 jobs during 5 months), SDSC BLUE (1,152 CPUs, 243,314 jobs during 34 months), CTC SP2 (338 CPUs, 77,222 jobs during 11 months), HPC2N (240 CPUs, 202,876 jobs during 42 months), SDSC SP2 (128 CPUs, 59,725 jobs during 24 months) and KTH SP2 (100 CPUs, 28,489 jobs during 11 months). If available, the recommended “cleaned” versions of workload logs are always used. Detailed descriptions of these logs are available in the Parallel Workloads Archive [5].

[4] In the current implementation the tabu list has maximum size of 10 jobs. If all jobs from the current schedule are in the *tabu_list*, it means that there are at most 10 jobs in the whole schedule.

In the experiments, the avg. response time, the avg. wait time and the avg. bounded slowdown have been measured as the standard performance related metrics. A bubble chart is used to display corresponding values of wait time and slowdown simultaneously — the y-axis depicts the avg. wait time while the size of the circle represents the avg. bounded slowdown. The actual bounded slowdown value is shown as a label bellow each circle (see Fig. 1).

Concerning fairness, the resulting normalized user wait times $NUWT_o$ were collected for all users. In the next step, we have removed all $NUWT_o$ values that belonged to users who submitted only one job as this represents an extreme situation. When such user submits the first (and only) job into the system, the job gets some default priority, since the system cannot compute $NUWT_o$ that is normally used to establish the job priority[5]. At the same time, other jobs in the queue often have higher priority, therefore this single job may be delayed by other high priority jobs. However, as the user do not submit any other job, there is no way for the scheduler to "fix" this unfair assignment and the resulting $NUWT_o$ may be quite high. Once the set of $NUWT_o$ values was reduced as explained above, the $NUWT_o$ values were interpreted in two different ways. The cumulative distribution function (CDF) of users' normalized wait times presents how different scheduling algorithms affect the resulting $NUWT_o$ values for all users of the system. In this case, the CDF is a $f(x)$-like function showing the probability that the $NUWT_o$ of given user o is less than or equal to x. In another words, the CDF represents percentage of users having their $NUWT_o$ less than or equal to x. The steeper is the resulting curve the closer (i.e., more fair) were the $NUWT_o$ values of different users. As the resulting distributions often have very long tails, the maximal $NUWT_o$ shown on the x-axis is bounded by 10.0 for better visibility. The CDFs are accompanied with two additional metrics which show the arithmetic mean of all normalized user wait times and the corresponding standard deviation. The smaller the mean and the standard deviation are the lower were the $NUWT_o$ values and the closer (i.e., more fair) they were. When two or more algorithms have similar CDFs, these two metrics helps to highlight the differences among algorithms as they can highlight the influence of the distribution's long tail.

5.2 Experimental Results and Discussion

Fig. 1 presents the avg. wait time, the avg. bounded slowdown (1st and 3rd row) and the avg. response time (2nd and 4th row) for all six data sets. The fairness related results for all data sets are shown in Fig. 2. As discussed in Section 5.1, two different graphs are used to capture the resulting normalized user wait times ($NUWT_o$). The CDFs of $NUWT_o$ distributions are shown in the first and third row in Fig. 2. The bar charts with the mean of all normalized user wait times and the corresponding standard deviations are shown in the second and fourth row in Fig. 2.

[5] $NUWT_o$ cannot be computed because both $TUWT_o$ and $TUSA_o$ are not known unless at least one job of given user o completes (see Formulas 1–3).

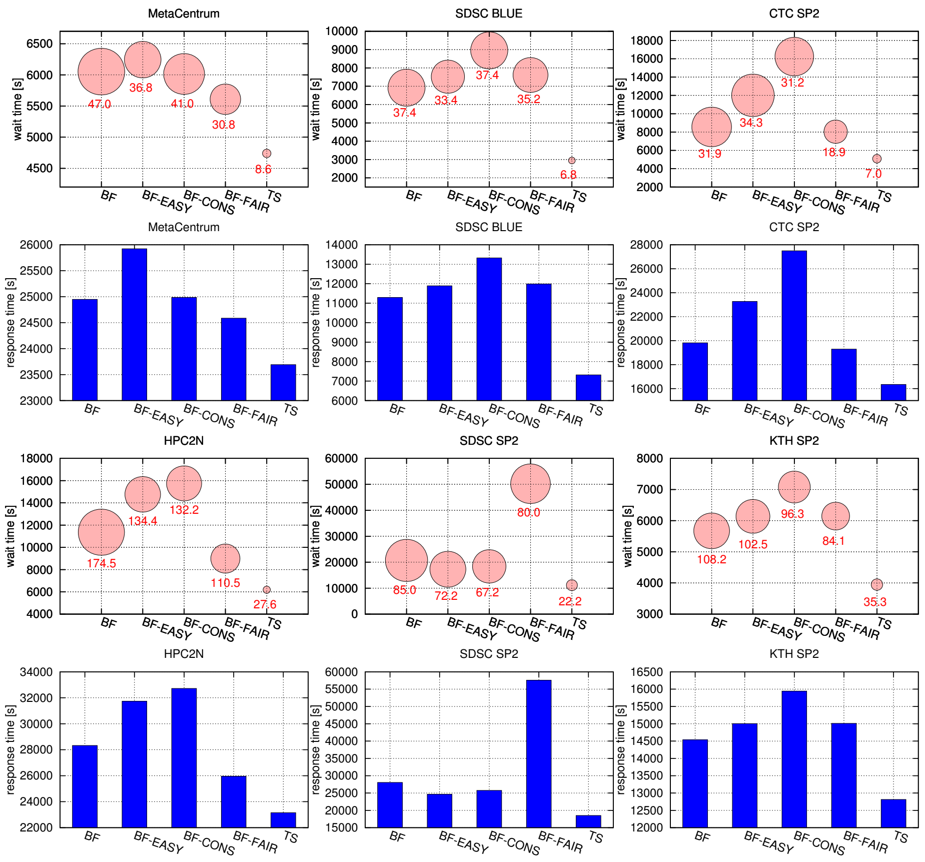


Fig. 1. The avg. wait time and the avg. slowdown (1st and 3rd row), and the avg. response time (2nd and 4th row) for all six data sets

Let us discuss the results of the experiments. In all experiments FCFS performed very bad, which is not surprising as the applied workloads represent reasonably utilized systems with parallel jobs where FCFS is known to be inefficient [12,23,25]. Therefore — with the exception made in case of CDFs — the results of FCFS are not presented in the charts for better visibility, as in all cases the results of FCFS were very bad and off-scale high.

Concerning the avg. wait time (1st and 3rd row in Fig. 1) and the avg. response time (2nd and 4th row in Fig. 1), original Conservative backfilling (BF-CONS) does not work very well with respect to other backfilling algorithms, as both BF-EASY or BF produce better results in most cases. This is not surprising as these issues have been already addressed in several works [31,32,26]. Basically, the problem here is that establishing reservation for every job can be less efficient than aggressive approaches as used in BF or BF-EASY. Reservations decrease the opportunities for backfilling, due to the blocking effect of the reserved jobs in the schedule [31,4]. On the other hand, the slowdown (see circle labels in the bubble charts in Fig. 1) is often slightly better for BF-CONS as no job can be delayed. This is a normal behavior also observed in previous works [31,4]. BF-FAIR is also competitive, however in case of SDSC SP2 it does not perform very well with respect to the avg. wait time. On the other hand, TS produces the best wait times, response times and slowdowns in all six cases. Clearly, TS significantly improves the otherwise relatively weak performance of Conservative backfilling. These results indicate that the "ad hoc" manner used to establish reservations in BF-CONS is not very efficient and can be easily improved using the evaluation and optimization techniques.

In case of fairness related criteria (see Fig. 2), the results clearly demonstrate that standard solutions such as FCFS, BF, BF-EASY or BF-CONS are not very good to guarantee good fairness with respect to different users since they do not involve any suitable mechanism for this purpose. Especially FCFS is truly unfair for users as can be seen in the CDFs (1st and 3rd row in Fig. 2). In general, BF, BF-EASY and BF-CONS produce worse results than BF-FAIR or TS in most cases. While the differences in the CDFs are not huge, there are typically several users with very high (unfair) $NUWT_o$ when BF, BF-EASY or BF-CONS is used, respectively. This unfairness significantly increases the arithmetic mean of all normalized user wait times and the corresponding standard deviation as can be seen in the second and fourth row in Fig. 2.

From this point of view, a simple extension involving fair-share based priorities as applied in BF-FAIR algorithm can significantly improve the fairness of the solution with respect to different users. In most situations BF-FAIR generates better, i.e., steeper CDFs of normalized user wait times (see 1st and 3rd row in Fig. 2) as well as better, i.e., lower mean values and standard deviations (2nd and 4th row in Fig. 2) than FCFS or other backfilling algorithms. Again, TS optimization procedure shows great potential when improving the fairness of the original BF-CONS. TS can guarantee fairness on the same or even higher level as BF-FAIR algorithm does, thanks to the applied extensions that involve schedule evaluation and optimization. Clearly, good results in standard performance metrics (see Fig. 1) are not achieved at the expenses of fairness (see Fig. 2).

As discussed in Section 2.1, if needed, the runtime estimates are precise (perfect) in this study. This setup has been used in order to provide "ideal" conditions for all algorithms that somehow use reservation(s). This approach minimizes the effect of inaccuracy that may sometimes produce "confusing" results [35,36]. One may suggest that the proposed extension of Conservative backfilling may not work that well as soon as realistic, i.e., inaccurate estimates are used. However, as we observed in our recent studies [14,16], the inaccuracy can be handled efficiently if proper "recovery" methods are applied. For example, the inefficiency caused by early job completions can be minimized by applying schedule compression as discussed in Section 4.2 or in [25,14]. Moreover, as the optimization procedure follows a "gap filling" approach (see line 12 in Algorithm 2) it can be flexibly used to further improve the schedule once the compression phase terminates [14,16]. Similarly to, e.g., EASY backfilling, it is favorable when the users' estimates are heterogeneous. When there are only few popular estimates (e.g.,

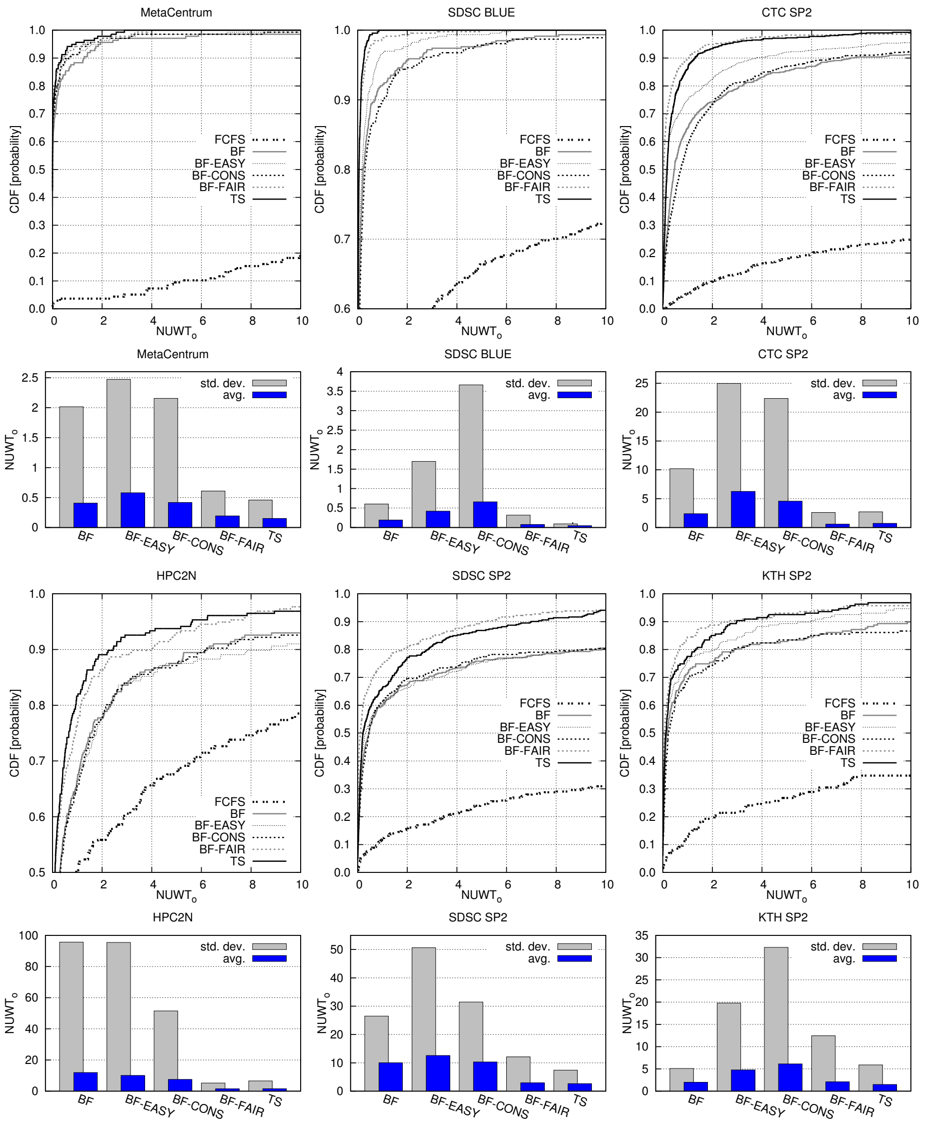


Fig. 2. The CDFs of normalized user wait times (1st and 3rd row), and the mean of normalized user wait times and the corresponding standard deviation (2nd and 4th row) for all six data sets

3 popular estimates), the performance of the proposed technique is similar to the performance of the backfilling solutions [14]. At such situation, the limited diversity of runtime estimates prevents us from building efficient schedules, since there is no good opportunity for successful optimization [14].

6 Conclusion

This paper addressed a real life-based job scheduling problem. The goals were to maintain the fairness among different users of the system while keeping good performance regarding classical criteria such as slowdown or wait time. Several existing algorithms have been analyzed with respect to these two goals. Moreover, an extension of the well known Conservative backfilling has been proposed in order to guarantee good fairness and performance. The extension uses optimization procedure, which allows to improve the quality of the schedule. Optimization is guided by the evaluation that is performed subject to applied objective functions. Experimental evaluation demonstrates that the proposed extension represents significant improvement by means of fairness and performance over several existing algorithms including FCFS, Conservative and EASY backfilling as well as aggressive backfilling without reservations.

Just like the original Conservative backfilling, the current solution still supports predictability as reservations are established for every job. However, the optimization technique can delay particular jobs with respect to their initial reservations if it improves the overall quality of the schedule. As this behavior may be problematic in some cases, we plan to solve this issue in the future. Currently, we are working on a closely related scheduling approach involving evaluation and optimization algorithms [2] within the production TORQUE scheduler that is used in the Czech National Grid Infrastructure MetaCentrum.

Acknowledgments. We appreciate the gracious support of the Grant Agency of the Czech Republic under the grant No. P202/12/0306. The access to the MetaCentrum computing facilities provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic is highly appreciated. The MetaCentrum workload log was graciously provided by the Czech National Grid Infrastructure MetaCentrum. The CTC SP2, KTH SP2, SDSC SP2 and HPC2N workload logs were graciously provided by Dan Dwyer, Lars Malinowsky, Victor Hazlewood and Ake Sandgren, respectively. The workload log from the SDSC Blue Horizon was graciously provided by Travis Earheart and Nancy Wilkins-Diehr.

References

1. Adaptive Computing Enterprises, Inc. TORQUE Admininstrator Guide, version 3.0.3 (February 2012), `http://www.adaptivecomputing.com/resources/docs/`
2. Chlumský, V., Klusáček, D., Ruda, M.: The extension of TORQUE scheduler allowing the use of planning and optimization algorithms in Grids. Computer Science 13(2), 5–19 (2012)
3. Ernemann, C., Hamscher, V., Yahyapour, R.: Benefits of global Grid computing for job scheduling. In: GRID 2004: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 374–379. IEEE (2004)
4. Feitelson, D.G.: Experimental analysis of the root causes of performance evaluation results: A backfilling case study. IEEE Transactions on Parallel and Distributed Systems 16(2), 175–182 (2005)

5. Feitelson, D.G.: Parallel workloads archive (PWA) (February 2012), http://www.cs.huji.ac.il/labs/parallel/workload/
6. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)
7. Feitelson, D.G., Weil, A.M.: Utilization and predictability in scheduling the IBM SP2 with backfilling. In: 12th International Parallel Processing Symposium, pp. 542–546. IEEE (1998)
8. Frachtenberg, E., Feitelson, D.G.: Pitfalls in Parallel Job Scheduling Evaluation. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 257–282. Springer, Heidelberg (2005)
9. Glover, F.W., Laguna, M.: Tabu search. Kluwer (1998)
10. Hovestadt, M., Kao, O., Keller, A., Streit, A.: Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 1–20. Springer, Heidelberg (2003)
11. Jones, J.P.: PBS Professional 7, administrator guide. Altair (April 2005)
12. Keleher, P.J., Zotkin, D., Perkovic, D.: Attacking the bottlenecks of backfilling schedulers. Cluster Computing 3(4), 245–254 (2000)
13. Kleban, S.D., Clearwater, S.H.: Fair share on high performance computing systems: What does fair really mean? In: Third IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2003, pp. 146–153. IEEE Computer Society (2003)
14. Klusáček, D.: Event-based Optimization of Schedules for Grid Jobs. PhD thesis, Masaryk University (2011)
15. Klusáček, D., Rudová, H.: Alea 2 – job scheduling simulator. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010), ICST (2010)
16. Klusáček, D., Rudová, H.: Handling inaccurate runtime estimates by event-based optimization. In: Cracow Grid Workshop 2010 Abstracts (CGW 2010), Cracow, Poland (2010)
17. Klusáček, D., Rudová, H.: Efficient Grid scheduling through the incremental schedule-based approach. Computational Intelligence 27(1), 4–22 (2011)
18. Klusáček, D., Rudová, H., Baraglia, R., Pasquali, M., Capannini, G.: Comparison of multi-criteria scheduling techniques. In: Grid Computing Achievements and Prospects, pp. 173–184. Springer (2008)
19. LaTorre, A., Pena, J., Robles, V., De Miguel, P.: Supercomputer Scheduling with Combined Evolutionary Techniques. In: Xhafa, F., Abraham, A. (eds.) Metaheuristics for Scheduling in Distributed Computing Environments. SCI, vol. 146, pp. 95–120. Springer, Heidelberg (2008)
20. Lee, C.B.: On the User-Scheduler Relationship in High-Performance Computing. PhD thesis, University of California, San Diego (2009)
21. Leung, V.J., Sabin, G., Sadayappan, P.: Parallel job scheduling policies to improve fairness: a case study. Technical Report SAND 2008-1310, Sandia National Laboratories (2008)
22. Li, B., Zhao, D.: Performance impact of advance reservations from the Grid on backfill algorithms. In: Sixth International Conference on Grid and Cooperative Computing, GCC 2007, pp. 456–461 (2007)

23. Lifka, D.A.: Lifka. The ANL/IBM SP Scheduling System. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
24. MetaCentrum (February 2012), http://www.metacentrum.cz/
25. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12(6), 529–543 (2001)
26. Ngubiri, J.: Techniques and Evaluation of Processor Co-allocation in Multi-cluster Systems. PhD thesis, Radboud University Nijmegen (2008)
27. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems. Prentice-Hall (2002)
28. Sabin, G.: Unfairness in parallel job scheduling. PhD thesis, The Ohio State University (2006)
29. Sabin, G., Kochhar, G., Sadayappan, P.: Job fairness in non-preemptive job scheduling. In: International Conference on Parallel Processing, ICPP 2004, pp. 186–194. IEEE Computer Society (2004)
30. Sabin, G., Sadayappan, P.: Unfairness Metrics for Space-Sharing Parallel Job Schedulers. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 238–256. Springer, Heidelberg (2005)
31. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective Reservation Strategies for Backfill Job Scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 55–71. Springer, Heidelberg (2002)
32. Srinivasan, S., Kettimuthu, R., Subrarnani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Proceedings of 2002 International Workshops on Parallel Processing, pp. 514–519. IEEE Computer Society (2002)
33. Sulistio, A., Cibej, U., Venugopal, S., Robic, B., Buyya, R.: A toolkit for modelling and simulating data Grids: an extension to GridSim. Concurrency and Computation: Practice & Experience 20(13), 1591–1609 (2008)
34. Talby, D., Feitelson, D.G.: Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In: IPPS 1999/SPDP 1999: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, pp. 513–517. IEEE Computer Society (1999)
35. Tsafrir, D.: Using Inaccurate Estimates Accurately. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 208–221. Springer, Heidelberg (2010)
36. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: Solving the mystery of why increased inaccuracy help. In: IEEE International Symposium on Workload Characterization (IISWC), pp. 131–141. IEEE Computer Society (2006)
37. Vasupongayya, S., Chiang, S.-H.: On job fairness in non-preemptive parallel job scheduling. In: Zheng, S.Q. (ed.) International Conference on Parallel and Distributed Computing Systems (PDCS 2005), pp. 100–105. IASTED/ACTA Press (2005)
38. Wolberg, J.: Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments. Springer (2006)
39. Xhafa, F., Abraham, A.: Computational models and heuristic methods for Grid scheduling problems. Future Generation Computer Systems 26(4), 608–621 (2010)
40. Xhafa, F., Carretero, J., Alba, E., Dorronsoro, B.: Design and evaluation of Tabu search method for job scheduling in distributed environments. In: International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp. 1–8. IEEE (2008)

Comprehensive Workload Analysis and Modeling of a Petascale Supercomputer

Haihang You[1] and Hao Zhang[2]

[1] National Institute for Computational Sciences,
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[2] Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville, TN 37996, USA
{hyou,haozhang}@utk.edu

Abstract. The performance of supercomputer schedulers is greatly affected by the characteristics of the workload it serves. A good understanding of workload characteristics is always important to develop and evaluate different scheduling strategies for an HPC system. In this paper, we present a comprehensive analysis of the workload characteristics of Kraken, the world's fastest academic supercomputer and 11th on the latest Top500 list, with 112,896 compute cores and peak performance of 1.17 petaflops. In this study, we use twelve-month workload traces gathered on the system, which include around 700 thousand jobs submitted by more than one thousand users from 25 research areas. We investigate three categories of the workload characteristics: 1) general characteristics, including distribution of jobs over research fields and different queues, distribution of job size for an individual user, job cancellation rate, job termination rate, and walltime request accuracy; 2) temporal characteristics, including monthly machine utilization, job temporal distributions for different time periods, job inter-arrival time between temporally adjacent jobs and jobs submitted by the same user; 3) execution characteristics, including distributions of each job attribute, such as job queuing time, job actual runtime, job size, and memory usage, and the correlations between these job attributes. This work provides a realistic basis for scheduler design and comparison by studying the supercomputer's workload with new approaches such as using Gaussian mixture model, and new viewpoints such as from the perspective of user community. To the best of our knowledge, it's the first research to systematically investigate the workload characteristics of a petascale supercomputer that is dedicated to open scientific research.

Keywords: Workload Characterization and Modeling, Petascale Supercomputer, Academic Supercomputer, High Performance Computing.

1 Introduction

As high performance computing (HPC) is becoming highly accessible, more researchers from a variety of research fields start to use supercomputers to solve large scientific problems. A supercomputer provides massive computational resources to achieve extremely fast processing speed in order to accelerate slow processes and generate outcomes with less time. The overall performance of a supercomputer depends heavily on the quality of the scheduling system. As indicated in [6], the performance of a supercomputer scheduler is greatly affected by the workload to which the supercomputer is applied, and there is no single scheduling algorithm working perfectly for all workloads. Therefore, workload characterization of a supercomputer is an important step to develop and evaluate scheduling strategies. Furthermore, a good understanding of workload characteristics can guide an HPC center to make decisions for purchasing or allocating specific hardware and software for the applications with different resource usage patterns.

Over the past decades, a variety of studies were conducted on workload analysis and modeling of parallel computers to evaluate scheduler performance [5] [11], and to predict job performance [6] [20]. Using historical data of workload traces that were recorded on real machines, statistical analysis of workloads was performed to understand the characteristics, such as distributions of job runtime and memory usage, of a single HPC system [3], a multi-cluster supercomputer [8], or a Grid computing environment [2]. Statistical workload models were also widely studied to generate abstract representation of job attributes, such as fitting distributions to job attributes [17] and modeling correlations between job attribute pairs [7]. Another type of workload models were developed on the basis of usage behavior classification. Wolter [21] experientially classified supercomputer users into three groups, and analyzed the job characteristics in each group. Song [18] used a mixed usage group model to classify jobs into predefined number of categories and investigated the workload traces in each category for job scheduling. Temporal characteristics of workloads in parallel systems were also analyzed and modeled. An infinite two-state Markov model was used to describe the characteristics of job inter-arrival in [13], which was extended to a n-state Markov modulated Poisson process to capture autocorrelations in [9]. Temporal locality of parallel system workloads was investigated in [14] to improve runtime prediction accuracy. However, the user community was relatively small in previous publications compared to a top ranking petascale supercomputers with a large user community and millions of job submissions. Previous researches failed to provide a comprehensive analysis of the characteristics of the user community, which greatly affects the patterns of the workload on a machine. For instance, the geographical distribution of users determines the temporal distribution of workload in a day.

In this paper, we present a comprehensive workload characterization of Kraken, an petascale supercomputer, which now ranks as the eleventh fastest computer in the world, and holds the title of world's fastest academic supercomputer [19]. Different from other top ranking HPC systems, Kraken is dedicated to academia, and it's user community consists of researchers from universities, research centers, institutes and laboratories. By analyzing Kraken workload, we can understand the patterns of how academia uses supercomputers to solve large scientific problems. Another distinguished characteristic of Kraken is the high utilization, which has been a sustained rate of 94 percents by average for the past year, and already contributed two billion CPU hours in total to open scientific research. The workload dataset used in this work contains about 700 thousand job traces, which were collected between November 2010 and October 2011 on Kraken. The large size of the dataset ensures that we are able to analyze the workload characteristics properly and come to solid conclusions. The job characteristics investigated in this work include general characteristics (e.g., distribution of the number of jobs submitted by a user), temporal characteristics (e.g., distribution of job inter-arrival time), and execution characteristics (e.g., distribution of job size).

The contributions of this paper are two fold. First, this paper analyzes a large workload dataset that was collected from the world's fastest petascale academic supercomputer. This is the first work, to the best of our knowledge, to systematically investigate the workload characteristics on a petascale supercomputer that is dedicated to open scientific research. Second, we provide new viewpoints and approaches with statistical models to comprehensively investigate a variety of the characteristics of Kraken workload. Besides investigating and modeling most of the workload characteristics introduced in previous research, we also analyze some characteristics which appear to be unique to Kraken, especially the characteristics of the user community, such as the user distribution and the distribution of the compute resource allocated to each research field. We also apply the Gaussian mixture models to fitting the distributions of some job attributes, such as job queuing time and actual runtime, which exhibit different patterns on a petascale supercomputer and cannot be modeled using a single distribution. We believe that this work is valuable in helping HPC centers to better understand the workload characteristics of a petascale supercomputer and the user community in academia, which is a necessary and important step to improve the overall performance of a supercomputer, and to prepare for the next generation of exascale HPC systems.

The remainder of the paper is organized as follows. Section 2 introduces the Kraken supercomputer and describes the workload dataset used in this study. Section 3 discusses the general workload characteristics of Kraken, including distribution of research fields, distribution of job size for an individual user, job distribution over queues, job cancellation rate, job termination rate, and walltime request accuracy. The temporal workload characteristics of Kraken are investigated in Section 4, which are monthly utilization, job temporal distribution in a year, a month, a week, and a day, and inter-arrival time between temporally adjacent jobs and jobs submitted by the same user. Section 5 talks about the execution characteristics, such as distributions of each job attribute, job queuing time, job actual runtime, job size, memory usage, and the correlations between these job attributes. Finally, Section 6 concludes the paper and reiterates the important characteristics of a petascale academic supercomputer.

2 Workload Traces of the Kraken Supercomputer

The historical workload traces of the Kraken supercomputer [15] was used in this study. Kraken is managed by the National Institute for Computational Sciences (NICS) at the Oak Ridge National Laboratory (ORNL) in the United States, and is funded by the National Science Foundation (NSF). It provides a petascale computing environment that is fully integrated with the Extreme Science and Engineering Discovery Environment (XSEDE). The supercomputer is a Cray XT5 system consists of 9,408 compute nodes with 112,896 compute cores, 147 terabytes of memory, and 3.3 petabytes storage. Each compute node contains 12 AMD 2.6 GHz Istanbul compute cores and 16 GB memory. The peak performance of the Kraken supercomputer is now 1.17 petaflops. Access to Kraken compute resources is managed by the Portable Batch System (PBS). The Lustre file system is used to support I/O operations, with the peak performance of 30GB/s. Moab is used to schedule jobs on Kraken, with preference to large core count jobs. Backfilling is applied on Kraken to allow smaller, shorter jobs to use remaining idle resources. The supercomputer is funded for the NSF community, which enables the scientific discoveries of nationwide researchers. In general, experienced researchers from academic or nonprofit organizations of the United States are eligible to request allocations of compute time of Kraken.

Table 1. A typical historical usage data that records the workload traces of the Kraken supercomputer. The exemplary workload dataset contains five instances. Each row represents a job, and each column denotes a job attribute.

job_id	user_name	account	nproc	mem_used	submit_time
0000001.nid00016	hao	U1-INDEX001	12	7588	2011-01-27 08:10:13
0000002.nid00016	haihang	U2-INDEX001	1	142664	2011-03-28 14:15:50
0000003.nid00016	hao	U2-INDEX001	1032	16276	2011-04-16 01:28:41
0000004.nid00016	admin	SUPPORT	288	11836	2011-05-31 09:43:51
0000005.nid00016	hao	U1-INDEX002	98304	71812	2011-07-05 17:01:08

start_time	end_time	walltime_req	walltime	cpu_hours
2011-01-27 09:26:03	2011-01-27 09:36:14	00:30:00	00:10:11	3.22
2011-03-28 14:16:14	2011-03-28 14:35:01	05:30:00	00:18:47	0.0658
2011-04-16 11:51:30	2011-04-17 11:51:56	24:00:00	24:00:27	24775.74
2011-06-04 21:00:20	2011-06-05 21:00:57	23:59:59	24:00:37	6914.96
2011-07-07 11:11:13	2011-07-08 13:11:34	32:00:00	26:00:21	2556477.44

queue	type	software	research_area	description
small	batch	—	Physics	Simulation in Physics
hpss	batch	wrf	Earth Sciences	Simulation in Earth Science
medium	batch	mpcugles	Physics	Energy conserving eddy simulation
small	interactive	—	Benchmark	Interactive benchmark
capability	batch	gadget	Earth Sciences	Large earth simulation

Table 2. Classification of jobs that are submitted to Kraken, which is determined by the number of compute nodes. Each job category is associated with a unique queue that is managed by job scheduler.

Queue	Compute Cores			Walltime$_{\text{Max}}$
	Minimum	Maximum	Percentage (%)	(hours)
Small	1	512	0–0.45	24.0
Medium	513	8,192	0.45–7.26	24.0
Large	8,193	49,536	7.26–43.88	24.0
Capability	49,537	98,352	43.88–87.12	48.0
Dedicated	98,353	112,896	87.12–100	48.0
HPSS	N/A	N/A	N/A	24.0

The dataset of the Kraken workload traces were collected between November 2010 and October 2011. It contains 693,829 job traces submitted by more than one thousand researchers from 25 research fields. This dataset was obtained by tracking all jobs that were submitted to the Kraken supercomputer and documenting the information that was related to the jobs. Some instances of the Kraken workload traces are listed in Table 1, each of which contains sixteen attributes. The attributes *job_id*, *user_name*, and *account* are the unique identifiers of a job, an user, and an account, respectively. The attributes *submit_time*, *start_time* and *end_time* record the times when a job is submitted, executed, and finished, respectively. Jobs are automatically classified into several categories according to the number of requested compute nodes, denoted by *nproc*. Each category is associated with a *queue* that indicates the priority. The definitions of the job categories and queue types are described in Table 2. Because each compute node of Kraken contains twelve cores, the attribute *nproc* must be a multiple of 12, except for the jobs in the queue for accessing the High Performance Storage System (HPSS). These jobs are executed to transfer files to a storage system using a batch file. No compute nodes on Kraken are allocated to the jobs in HPSS queue. The attribute *walltime_req* is the requested walltime of a job, which is required to be estimated and provided by a user before submitting a job. The attributes *walltime*, *mem_used* and *cpu_hours* represent actual runtime, consumed memory and CPU hours of a job, respectively. The attribute *type* takes a boolean value (i.e., interactive or batch), which indicates whether a user has interactive access to the compute resources. The attribute *software* describes the name of the software or package, if any, used by a job. For instance, the job 0000003.nid00016 used MPCUGLES, which is a software for energy-conserving large eddy simulations. The attribute *research_area* indicates the research field of the target problem, or to which the user belongs. At last, the attribute *description* explains the objective of the project.

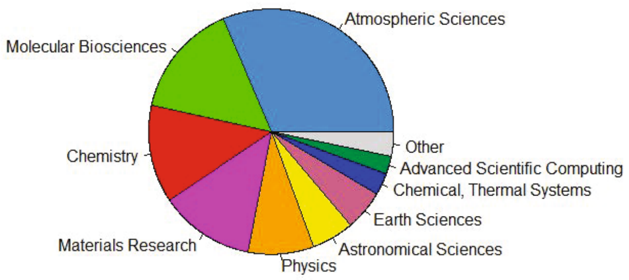


Fig. 1. Pie chart of the research fields sorted by the number of jobs submitted to Kraken in each field

3 Job General Characteristics

We first analyze the general characteristics of jobs running on Kraken and its user community. The objective of this analysis is to figure out how users from different research fields utilizing the petascale supercomputer, and how well they performed on using the supercomputer for open scientific research.

3.1 Distribution of Research Fields

There were 1,111 users from 473 different accounts submitted jobs to Kraken during the year when the workload traces were collected. The users scattered in 25 different research fields from all over the United States. The proportion of each research field sharing the supercomputer is illustrated in Figure 1. The top five research fields using Kraken were Atmospheric Sciences, Molecular Biosciences, Chemistry, Materials Research and Physics, which took over 75 percents of the job submissions to Kraken. However, the number of Kraken users in a research field was not necessarily proportional to the number of job submissions in the research field. For example, the research field of Molecular Biosciences had the most 205 Kraken users, but only 90 users came from Atmospheric Sciences who submitted the most jobs. On the other hand, the number of job submissions and the number of users in a research field were positively correlated, i.e., a larger user group often resulted in more job submissions.

3.2 Distribution of Number of Jobs for a User

Typically, a supercomputer user only submits jobs to address similar problems in the same discipline. On the other hand, it is very common that a user submits multiple jobs at one time or across multiple times. The distribution of the number of jobs submitted by a user is depicted in Figure 2 in a logarithm scale. On Kraken, there were 693,829 job submissions recorded in a year, and an average of 624 jobs were submitted by each user. However, it should be noted that the number of jobs submitted by a user was not uniformly distributed. Oppositely,

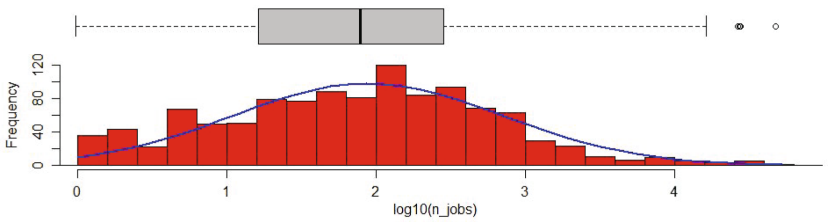


Fig. 2. Histogram (red rectangles) of the number of jobs submitted by a Kraken user, with density estimate (blue curve) and statistical summary (gray boxplot)

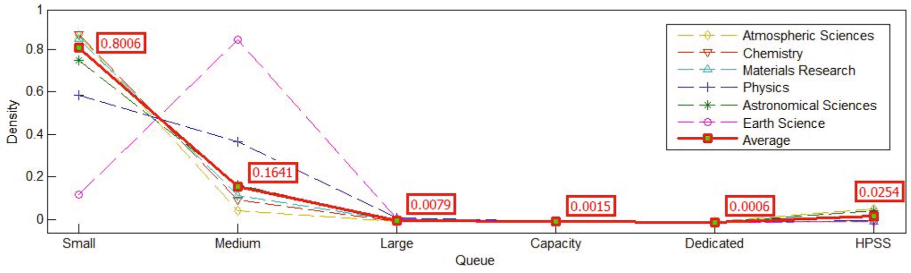


Fig. 3. Distribution of jobs over queues in different research fields. The number in a red rectangle indicates the probability that a job belongs to a queue.

the most active 10% users contributed 79.2% job submissions to the workload. The statistical summary of the jobs submitted by a user is also plotted with a boxplot in Figure 2, which indicates that 50% users had job submissions between 20 to 600 in a year. The number of jobs submitted by a user can be modeled using the log-normal distribution, i.e., the logarithm of the number of job submissions conforms to the Gaussian distribution. The parameters of the distribution can be computed using the maximum-likelihood estimation (MLE). The probability density function (PDF) is plotted with the blue curve in Figure 2.

3.3 Distribution of Jobs over Queues

The probability distribution of jobs over queues and the probability distributions of jobs over queues within different research fields are depicted in Figure 3. In general, the probability of the jobs belonging to a queue consistently decreased with the increase of the job size in the queues of *Small*, *Medium*, *Large*, *Capability*, and *Dedicated*, as shown by the average distribution in Figure 3. The probability that a job belongs to a queue is shown in a red rectangle in the figure. On average, over 80% jobs belonged to the queue of type *Small*, and only 1% jobs consumed significant compute resources, which belonged to the queues of type *Large*, *Capacity*, or *Dedicated*. As for the distributions of jobs over queues in each individual research field, although the distribution in Earth Science had great difference from the average distribution, in which case the probability that a job was in the queue of type *Medium* was greater than the probability that the

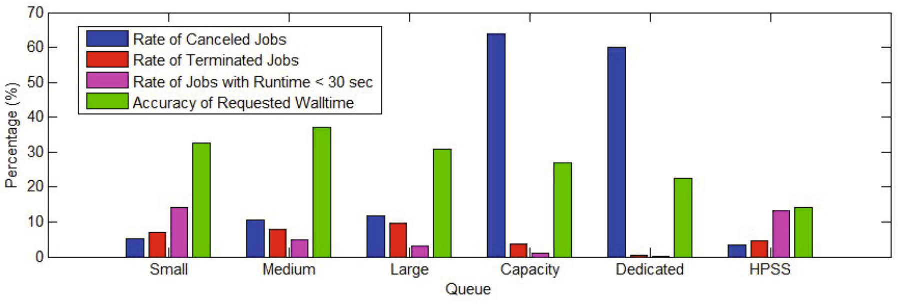


Fig. 4. General characteristics of workload traces in different queues

job belonged to the queue of type *Small*, in general, most of the per-discipline distributions were quite similar to the average distribution, such as Atmospheric Sciences and Materials Research. Consequently, we can model the distribution of jobs over queues with a single multinomial distribution for all research fields, with each model parameter equal to the value of the corresponding average probability of a job belonging to a queue.

3.4 Cancellation/Termination Rate and Walltime Request Accuracy

The general characteristics of the jobs in different queues are also investigated in this study, which include job termination rate, job cancellation rate, and average accuracy of walltime estimation. In most HPC systems, a job can be canceled manually by a user before the job starts executing, in which case the job actual runtime is zero. On the other hand, a job can be forcefully terminated by the HPC system if the job runs out of the requested walltime, in which case the job actual runtime is equal to the requested walltime. Moreover, a job might fail and exit during its execution due to some runtime error, in which case the job actual runtime is often significantly smaller than the requested walltime. If a job neither completes correctly, nor provides meaningful results, it is considered incorrect. Because the incorrect jobs are always misleading for meaningful analysis, we evaluate the accuracy of the requested walltime with correct jobs. A job is considered *correct*, if it belongs to the job set:

$$\begin{aligned}\mathcal{J}_c = \{j \mid (j.mem\_used \neq 0) \wedge (j.walltime > 30) \\ \wedge\ (j.walltime < j.walltime\_req)\}\end{aligned} \tag{1}$$

where "." represents attribute relationship, and the unit of the attribute *walltime* is second. The three literals in (1) remove the canceled jobs, terminated jobs, and a part of jobs with runtime errors at the beginning of job execution, respectively. The second literal also contributes to remove some "hello world" jobs, in which case a user often does not care about job runtime. On the other hand, it should be noted that this definition does not remove all incorrect jobs, such as jobs

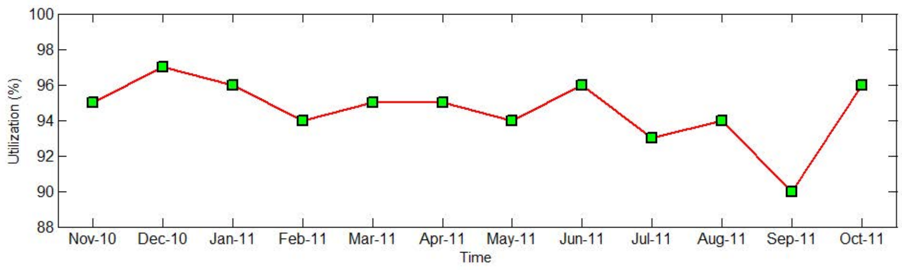
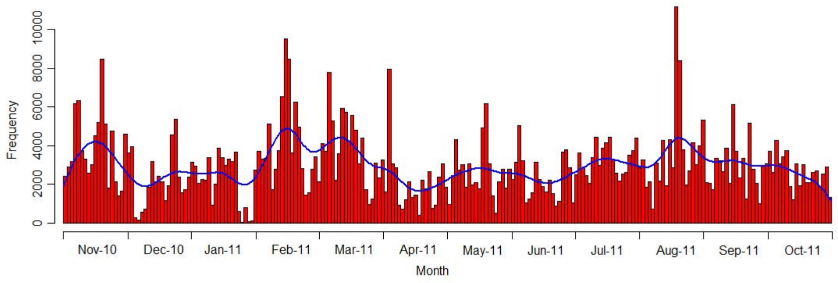


Fig. 5. Monthly utilization of the Kraken supercomputer over a year

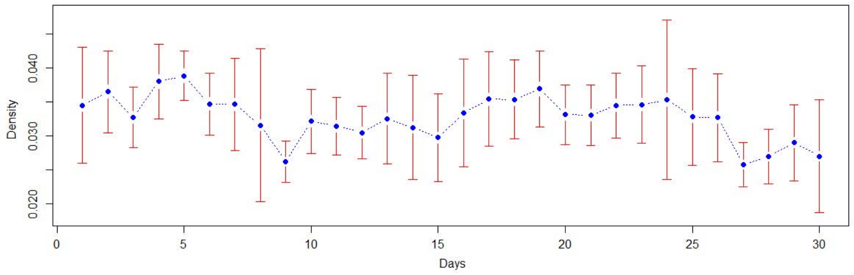
with runtime errors at the end of their execution. These remaining incorrect jobs are treated as noise in this work. To quantitatively measure the accuracy of the requested job runtime, the walltime request accuracy (WRA) is applied. For a correct job j in $\mathcal{J}_c$, WRA is defined as:

$$WRA(j) = \frac{j.walltime}{j.walltime\_req} \times 100\% \tag{2}$$

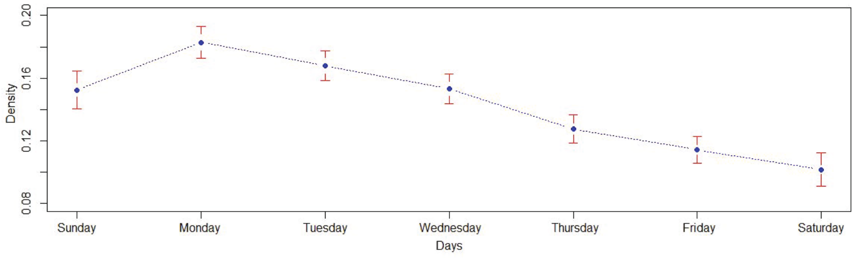
The rates of canceled jobs, terminated jobs and jobs with runtime less than 30 seconds, along with the WRA for each queue are illustrated in Figure 4. A most obvious phenomenon is that the jobs requesting significant compute resources, in the queues of type *Capability* and *Dedicated*, had the highest cancellation rate of more than 60%, which might be resulted from the long queuing time. Moreover, the jobs in the queue *Capability* and *Dedicated* had the lowest termination rate of less than 5%, and the lowest 30-second job quitting rate of less than 1%, along with the lowest WRA. This phenomenon can be partially explained by the fact that researchers submitted these jobs often had rich HPC experience to reduce errors in the code, and they also tended to request longer walltime to decreased the probability that their jobs were forcefully killed by the system. Third, a high rate of the jobs quitting within 30 seconds in the queues of type *Small* and *HPSS*, which at least doubled the same rate of other queues, indicates that new users of Kraken were more probable to submit small jobs to figure out how to access the compute and storage resources. Consequently, it is necessary for the scheduling system of an HPC system to have backfilling policy, which allows small jobs to be backfilled. The last important observation is that the overall accuracy of the requested walltime was around 33%, which was typically estimated and provided by a user. The low accuracy of the job runtime estimate indicates that many users were lack of experience with the HPC system. It is of great necessity for HPC centers to provide users with more supports, such as a recommendation system to guide users to predict the runtime of their jobs.



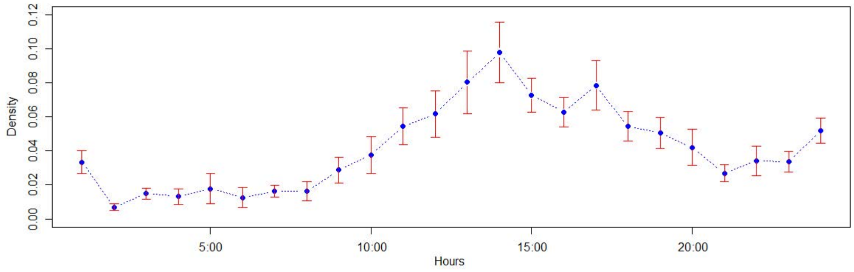
(a) Histogram (red rectangles) and the kernel density estimate (blue curve) of the number of jobs that were submitted to Kraken in the year between November 2010 and October 2011.



(b) Temporal distribution of the number of per-day jobs over a month.



(c) Temporal distribution of the number of per-day jobs over a week.



(d) Temporal distribution of the number of per-hour jobs over a day.

Fig. 6. Temporal distributions of the Kraken workload

4 Job Temporal Characteristics

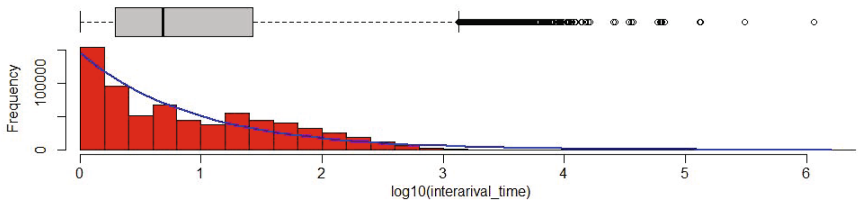
In this section, we investigate the temporal characteristics of the jobs submitted to Kraken, which include 1) the monthly utilization of the supercomputer in a year; 2) temporal distributions of jobs in a year, a month, a week, and a day; and 3) the inter-arrival time of job submissions. The goal is to unveil the workload dynamics, i.e., the time-based patterns in the workload.

4.1 Supercomputer Utilization

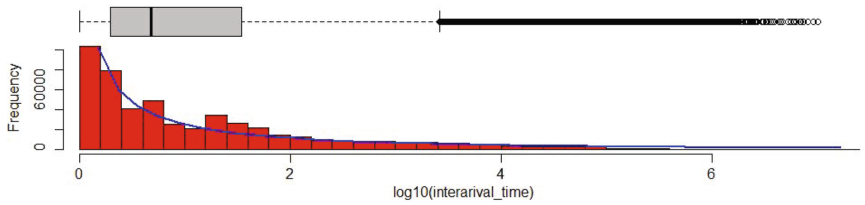
The monthly utilization of Kraken over a year is depicted in Figure 5. The system utilization for each month was very consistent on Kraken, which oscillated between 90% and 97%, and an average monthly utilization of 94.6% was observed. Kraken has been providing two thirds of cycles that are available to the national research community funded by NSF in the United States. Projects are selected by xRAC [22] through process which is designed to provide independent merit-review of proposals for the XSEDE. At NICS, teams of operations, user support, computational science and education, outreach and training provide support at various levels. The combination of project selection, management, support, and expertise provided on site at NICS result in such high utilization rate for the petaflop supercomputer, which serves as a good example to increase the overall utilization of an HPC system to open scientific research.

4.2 Temporal Distributions

The temporal distributions of the Kraken job submissions over a year, a month, a week, and a day are illustrated in Figure 6a, 6b, 6c, and 6d, respectively. In Figure 6a, the kernel density estimate [16] is computed to fit the dataset and to provide a smooth representation. An important observation from this figure is that the job submissions were not uniformly distributed over time, and the bursty behavior of job arrivals can be observed, such as the burstiness of the job arrivals in the middle of February and August, 2011. This phenomenon is possibly caused by the training activities using the supercomputer at the beginning of spring and fall semester. On the other hand, the low job arrival can be also observed, which might be caused by system upgrade, system failure, or occupation by capacity or dedicated jobs. An interesting observation is that the number of job submissions was not necessarily correlated with the utilization. For instance, running dedicated jobs might lead to very high system utilization with very low job counts. At last, the workload of Kraken did not exhibit obvious patterns over a year. The temporal distribution of the daily job arrivals over a month is computed using the 12-month Kraken workload dataset, and the standard deviations are also calculated to indicate the reliability of the estimates, which are presented with the error bars in Figure 6b. The large errors for the daily job arrival estimates indicate that the workload of the same day (e.g., the 8th day) in a month was not stationary, which varied greatly from month to month.



(a) Distribution of the inter-arrival time(second) between the temporally adjacent jobs.



(b) Distribution of the inter-arrival time(second) of jobs from the same user.

Fig. 7. Comparisons between the inter-arrival time of the continuous job stream and the inter-arrival time of the job sequence submitted by the same user

However, the workload exhibited obvious and stationary patterns in weekly cycles. As illustrated in Figure 6c, the daily workload presented obvious cyclical patterns in a week, which started at the maximum on Mondays, then constantly decreased to its minimum on Saturdays, and bounced back on Sundays towards the peak. Moreover, the small standard variations presented by the short error bars in Figure 6c also indicate a high degree of consistency in the daily workload over a week. In general, the workload at the end of a week was quite different from the workload at the beginning of a week. The maximum daily workload on Mondays was 1.8 times of the minimum daily workload on Saturdays. Similarly, the hourly workload also showed stationary, obvious, but more complicated patterns in daily cycles, which is illustrated in Figure 6d. The peak of the hourly workload arrival appeared at around 3:00 PM in the afternoon. In general, the hourly workload in the midnight was much lower than the hourly workload in the afternoon. Another interesting phenomenon is that researchers tended to submit more jobs before getting off work and going to sleep, which is well supported by the local maximum at around 5:00 PM and 12:00 PM in Figure 6d, respectively. However, it should be noted that the cyclical patterns of hourly workload over a day on a supercomputer is significantly dependent on the spatial distribution of its users.

4.3 Inter-arrival Time

We analyze the inter-arrival time of the jobs submitted in sequence to Kraken, and compare the results with the inter-arrival time of the jobs submitted by the same user. The inter-arrival time distributions are illustrated in Figure 7, in which the inter-arrival time is plotted in a log scale to reduce a wide range to a manageable size. From Figure 7a, we can observe that the time intervals between two temporally adjacent job submissions were very small. As indicated by the statistical summary in the box plot, around 25% inter-arrival times were less than three seconds, around 50% inter-arrival times less than six seconds, and around 75% inter-arrival times less than 30 seconds. The large inter-arrival time might be resulted from system upgrade or system failure, in which case, users cannot access or submit jobs to the supercomputer. The inter-arrival time distribution of the jobs submitted by the same user exhibited similar characteristics to the distribution of the overall inter-arrival time, as shown in Figure 7b. This distribution indicates the temporal locality, or burstiness, of the jobs submitted by an individual user, which is well supported by the observation that 25% inter-arrival times were less than two seconds, and 50% inter-arrival times were less than five seconds. The job burstiness was generally caused by the fact that users usually repeated submitting the same jobs or jobs with similar attributes in a short time span. Due to the similarity of these distributions, a single statistical model can be applied to model the inter-arrival times of the workload. In this work, we apply a Weibull distribution to model the logarithm of the job inter-arrival times, with MLE to estimate the model parameters. The results are shown with the blue curves in Figure 7.

5 Job Execution Characteristics

In this section, the job execution characteristics are investigated. First, we analyze the characteristics of each individual job attribute in the Kraken workload, including actual job queuing time, actual job runtime, number of compute nodes used by a job, and total memory consumed by a job. Then, the correlations between these job attributes are studied. The results of the characteristics of each job attribute and the correlations between job attributes are drawn with scatter-plot matrices which is depicted in Figure 8. Each diagonal plot presents the univariate histogram of the logarithm of one job attribute, along with the fitted distribution. The plots above the diagonal, with different shapes and colors, present the queue types to which each data point belongs, which also indicate the geographical distributions of the jobs in the 2-dimensional attribute space. Although plots below the diagonal contain the same data points as the plots above the diagonal, a smooth curve is fitted to each plot using the locally weighted scatter plot smoothing [4] to intuitively present the correlations between the job attributes. It should be noted that, because incorrect jobs are always distracting in investigating the underlying distributions of the job attributes, we intentionally remove the incorrect jobs, and only use the correct jobs in the set $\mathcal{J}_c$, defined in (1), for the following analysis. Moreover, the characteristics of the jobs in the HPSS queue are not analyzed, because these jobs do not consume any compute cores and memory, and users generally do not care about the performance of such jobs.

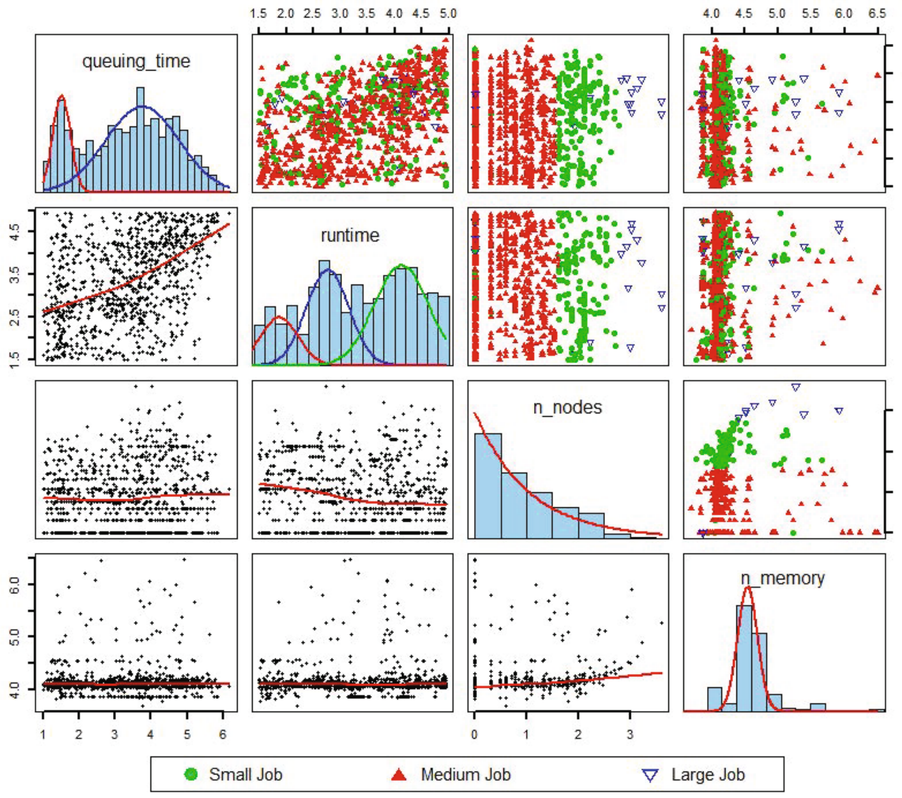


Fig. 8. Execution characteristics of actual queuing time, actual runtime, requested compute nodes, and consumed memory in the Kraken workload. All quantities in the figure are in a logarithmic scale based on 10.

5.1 Individual Attribute Characteristics

When analyzing the characteristics of each individual job attribute and plotting the scatter-plot matrices, the job submissions belonging to the queues of type *Capacity* and *Dedicated* are also intentionally removed, because these jobs only take over 0.21% of the entire job traces in the Kraken workload, which are always treated as outliers by the statistical models that are used to fit the individual attribute distributions.

We first analyze the characteristics of the job attribute *queuing time*, denoted as T_q, which is the actual waiting time of a job in a queue between job submission and execution. Different from the conclusions in previous research that a single distribution, such as the log-uniform distribution discussed in [10], can be used to present the queuing time distribution in a small system, for the workload of Kraken, a petascale supercomputer, any single distribution does not well fit to the job queuing time. In this study, the Gaussian mixture model (GMM), with two Gaussian component distributions, is applied to model the logarithm of the job queuing time, as illustrated in Figure 8-$(1,1)$. The parameters of the GMM model are estimated using MLE. The data of the job queuing time are well fitted by the GMM model with two components, which actually shows the characteristics of the scheduling queues. The first Gaussian component with a smaller mean is resulted from the backfilling policy of the scheduler, in which case the jobs requesting a small portion of compute resources can be backfilled and executed faster. The second Gaussian component represents the distribution of the queuing time of the non-backfilled jobs. On average, a job needs to wait in the queue for 34.5 minutes before starting execution.

Similarly, the distribution of the actual job runtime, denoted as T_r, cannot be simply modeled with a single distribution as well, such as the log-uniform [10], hypergamma [12], or Webull [1] distributions. In this study, we apply the GMM model to present the distribution of the logarithm of the actual job runtime, as shown in Figure 8-$(2,2)$. The GMM model contains three Gaussian components, which well fits the distribution of the logarithm of the job runtime. A possible explanation for the three Gaussian components is that the actual runtime of the jobs in each queue conforms to a logarithm-normal distribution, in which case the logarithm of the actual runtime of all jobs conforms to a mixture Gaussian distribution. The average job actual runtime is 36.8 minutes.

The characteristics of job size, denoted as N_n in number of compute nodes, for the Kraken workload are also investigated. On Kraken, there are 12 cores per node, and it is not possible to allocate part of a node. After applying logarithm on the number of nodes, the resulted distribution still has a short tail, as shown in Figure 8-$(3,3)$. This tailed distribution indicates that there are a great number of small jobs along with few large jobs are submitted to Kraken, which is consistent with the results shown in Figure 3. In more detail, there are 80.1% jobs submitted to the queue of type *Small*, and 26.5% jobs requesting only one node with twelve compute cores for learning HPC or debugging code. Consequently, we argue that, while the scheduler is collecting resources for larger jobs, backfilling policy is important for a petascale supercomputer to make jobs with short wall-clock limits and small core counts to start execution faster, in order to respond to the users with small job submissions quickly.

Memory usage is defined as the maximum amount of physical memory that is consumed by a job at some time point during its execution, which is recorded by the PBS on Kraken. Kraken is a distributed-memory system, with 16 GB memory for each compute node. The univariate histogram and the fitted distribution of the logarithm of the memory usage are depicted in Figure 8-$(4,4)$. In this study, a single Gaussian distribution is applied to fit the memory usage represented in a log scale. Although the Gaussian distribution fits most memory usage data very well, it cannot well model the heavy tail, which actually indicates that the entire workload is dominated by the computationally-intensive jobs with few memory-intensive jobs. A possible solution to address this problem is first to cluster the jobs into computationally-intensive or memory-intensive categories, then to model each job category separately. On Kraken, an average

Table 3. Correlation coefficients between job attributes

Correlation	$T_q - T_r$	$T_q - N_n$	$T_q - M_u$	$T_r - N_n$	$T_r - M_u$	$N_n - M_u$
Pearson	0.2439	0.0061	0.0210	−0.0481	−0.0177	0.0620
Log-Pearson	0.3946	0.0768	0.0286	−0.2092	0.0070	0.2501
Spearman	0.3951	0.0593	0.0367	−0.2293	−0.0716	0.4893

memory usage for the computationally-intensive jobs is around 12.3 GB, and the memory-intensive jobs usually consume over 80 GB memory.

5.2 Attribute Correlations

The correlations between job attributes are computed using all jobs in the queues of type *Small*, *Medium*, *Large*, *Capacity*, and *Dedicated* in the job set $\mathcal{J}_c$. Pearson product-moment correlation coefficient is the most commonly used measure of the strength of linear dependence between object attributes. Pearson correlation works well in the cases that the values of the attributes are roughly normally distributed. But Pearson correlation is very sensitive to the strong outliers, and works poorly on modeling correlations between data sampled from heavily tailed distributions. Thus, we also apply the Spearman's rank correlation coefficient to study the correlations between job attributes, which is a non-parametric correlation measure that is less sensitive to strong outliers. The correlations between job attributes are intuitively presented with the red smooth curves in the plots below the diagonal in Figure 8. We also applied the Pearson measure on both the raw job attributes and the logarithmic-scaled job attributes. Because Spearman measure is invariant to the logarithm operation, the Spearman correlation coefficients are only computed using the raw job attributes. The resulted coefficients are quantitatively listed in Table 3.

A strong positive correlation between job queuing time T_q and actual runtime T_r is observed from all three correlation measures, which can also be intuitively observed from the curve with a large positive slope in Figure 8-$(2, 1)$. Because the Spearman correlation and Pearson correlation have similar values when the data are roughly normally distributed with few outliers, the almost identical coefficients computed from the Log-Pearson measure and the Spearman measure indicate that the GMM model is a good model for fitting logarithm-scaled job queuing time and actual runtime. The second observation of job attribute correlations is that the job size N_n is positively correlated, in a non-linear manner, with the memory usage M_u, which can be observed from the curve with a positive slope in Figure 8-$(4, 3)$. A possible explanation for this observation is that Kraken is a distributed-memory system which always allocates 16 GB memory with each compute node allocation. Because each core can only access to the memory on the same node, the total allocated memory is always proportional to the number of allocated nodes. In this sense, a user tends to use more memory, when more compute cores are allocated with a larger amount of memory available. Third, there is a negative correlation between job size N_n and job actual runtime T_r, which can be easily observed from the curve with a negative slope in Figure 8-$(3, 2)$. This phenomenon indicates that, in general, using more compute resources reduces the wall-clock time of a job, which is actually the initial reason to use a supercomputer. Fourth, job queuing time T_q can be considered independent with job size and memory usage, as indicated by the coefficients that are close to zero. Similarly, job actual runtime T_r is statistically independent with job memory usage M_u. These independences indicate that job queuing time and memory usage are more random than other job attributes, which means it is harder to predict these two job attributes. The fitting curves with slopes close to zero graphically indicate the independences between the job attribute pairs, as shown in Figure 8-$(3, 1)$, 8-$(4, 1)$, and 8-$(4, 2)$. Finally, for the correlation measures, the Spearman measure generally performs better than the Pearson measure in the sense of detecting the non-linear correlations between job attribute pairs.

6 Conclusion

In this paper, the workload characteristics of a petascale academic supercomputer is comprehensively and systematically investigated, based on the twelve-month workload dataset collected from the world's most powerful academic supercomputer, with around 700 thousand jobs submitted by more than one thousand users from 25 research fields. The general characteristics, temporal characteristics, and execution characteristics of the workload traces are investigated and well represented with statistical models, with the objective of providing a realistic basis for scheduler design and comparison, as well as helping HPC centers to better understand the characteristics of workload and user community. For the highly-utilized petascale academic Kraken supercomputer, the most important observations and conclusions are reiterated as follows, according to the order they are discussed in the paper:

– Jobs are not uniformly distributed over research fields and users, and several users from a few research fields usually dominate the job submissions.
– The distributions of jobs over queues are very similar in different research fields. Thus, the same multinomial distribution can be applied, in all research fields, to model how jobs distribute over queues.
– Users with large job submissions generally have more HPC knowledge than users only with small job submissions. However, in general, the job runtime estimate is shown to be highly inaccurate.
– The Kraken workload does not show stationary patterns over a month or a year. But the workload exhibits obvious patterns in weekly or daily cycles.
– Due to the strong similarity between the patterns of overall job inter-arrival time and inter-arrival time of the jobs submitted by a user, a single statistical model is enough to represent the distributions of job inter-arrival time.
– Backfilling policy is important for a petascale supercomputer to enable small jobs to run effectively, which take over 80% job submissions in Kraken.

– The distributions of job queuing time and actual runtime on a multi-queue petascale supercomputer cannot be modeled with a single distribution. Gaussian mixture models perform well on modeling the log-scaled attributes.
– Job actual runtime is positively correlated with job queuing time, and negatively correlated with job size. Job memory usage has positive correlation with job size. Job queuing time can be considered independent with job size and memory usage. Job actual runtime and memory usage are also statistically independent.

References

1. Chiang, S.-H., Vernon, M.K.: Characteristics of a Large Shared Memory Production Workload. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 159–187. Springer, Heidelberg (2001)
2. Christodoulopoulos, K., Gkamas, V., Varvarigos, E.: Statistical analysis and modeling of jobs in a grid environment. Journal of Grid Computing 6, 77–101 (2008)
3. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: IEEE International Workshop on Workload Characterization, pp. 140–148 (2001)
4. Cleveland, W.S.: Robust locally weighted regression and smoothing scatterplots. Journal of the American Statistical Association 74(368), 829–836 (1979)
5. Denneulin, Y., Romagnoli, E., Trystram, D.: A synthetic workload generator for cluster computing. In: International Parallel and Distributed Processing Symposium, p. 243 (April 2004)
6. Feitelson, D.G.: Workload Modeling for Performance Evaluation. In: Calzarossa, M.C., Tucci, S. (eds.) Performance 2002. LNCS, vol. 2459, pp. 114–141. Springer, Heidelberg (2002)
7. Li, H.: Workload dynamics on clusters and grids. The Journal of Supercomputing 47, 1–20 (2009)
8. Li, H., Groep, D., Wolters, L.: Workload Characteristics of a Multi-cluster Supercomputer. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 176–193. Springer, Heidelberg (2005)
9. Li, H., Muskulus, M.: Analysis and modeling of job arrivals in a production grid. SIGMETRICS Perform. Eval. Rev. 34, 59–70 (2007)
10. Li, H., Wolters, L., Groep, D.: Workload characteristics of the das-2 supercomputer (June 2004)
11. 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.) JSSPP 1998. LNCS, vol. 1459, pp. 25–46. Springer, Heidelberg (1998)
12. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63, 2003 (2001)
13. Medernach, E.: Workload Analysis of a Cluster in a Grid Environment. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 36–61. Springer, Heidelberg (2005)
14. Minh, T.N., Wolters, L.: Modeling Parallel System Workloads with Temporal Locality. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 101–115. Springer, Heidelberg (2009)

15. National Institute for Computational Sciences. Running jobs on Kraken, http://www.nics.tennessee.edu/node/16 (accessed November 11, 2011)
16. Rosenblatt, M.: Remarks on Some Nonparametric Estimates of a Density Function. The Annals of Mathematical Statistics 27(3), 832–837 (1956)
17. Song, B., Ernemann, C., Yahyapour, R.: Modelling of parameters in supercomputer workloads. In: International Conference on Architecture of Computing Systems, pp. 400–409 (2004)
18. Song, B., Ernemann, C., Yahyapour, R.: User group-based workload analysis and modelling. In: IEEE International Symposium on Cluster Computing and the Grid, vol. 2, pp. 953–961 (May 2005)
19. Top500. Application area share for 06/2011, http://www.top500.org/list/2011/11/100 (accessed November 11, 2011)
20. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling User Runtime Estimates. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 1–35. Springer, Heidelberg (2005)
21. Wolter, N., McCracken, M., Snavely, A., Hochstein, L., Nakamura, T., Basili, V.: What's working in HPC: Investigating HPC user behavior and productivity. CT-Watch Quarterly 2(4A) (2006)
22. xRAC, http://www.teragridforum.org/mediawiki/index.php?title=XRAC

Author Index