Eitan Frachtenberg
Uwe Schwiegelshohn (Eds.)

LNCS 5798

Job Scheduling Strategies for Parallel Processing

14th International Workshop, JSSPP 2009
Rome, Italy, May 2009
Revised Papers

Springer

Eitan Frachtenberg
Uwe Schwiegelshohn (Eds.)

Job Scheduling Strategies for Parallel Processing

14th International Workshop, JSSPP 2009
Rome, Italy, May 29, 2009
Revised Papers

Springer

Volume Editors

Eitan Frachtenberg
Microsoft
475 Brannan St., San Francisco, CA, 94107, USA
E-mail: etc_26@yahoo.com

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

Library of Congress Control Number: 2009936148

CR Subject Classification (1998): D.4.1, I.2.8, F.2.2, C.1.2, C.1.4

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

ISSN 0302-9743
ISBN-10 3-642-04632-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-04632-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12761739 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 14th workshop on Job Scheduling Strategies for Parallel Processing. The workshop was held in Rome, Italy, on May 29, 2009, in conjunction with the IEEE International Parallel Processing Symposium 2009.

This year 25 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 excellent, detailed reviews: Su-Hui Chiang, Walfredo Cirne, Allen Downey, Dror Feitelson, Alexander Fölling, Allan Gottlieb, Christian Grimme, Andrew Grimshaw, Moe Jette, Joachim Lepping, Raquel Lopes, Reagan Moore, Jose Moreira, Bill Nitzberg, Alexander Papaspyrou, Lars Schley, Mark Squillante, John Towns, Dan Tsafrir, Jon Weissman, and Philipp Wieder.

As a result of the review process 14 papers were accepted for oral presentation at the workshop. One additional paper is included in these proceedings after making substantial improvements based on the comments of the referees. The final versions of the papers in this volume have addressed the comments of the referees and partially reflect the discussions held during the workshop.

This year we observed an increasing trend towards heterogeneous and multicore architectures. The paper by Gong, Pierces, and Fox proposes an improved heuristic approach to workflow scheduling and shows its efficiency with the help of simulations. This workshop series used to exclude task scheduling. However, precedence constraints are starting to play an important role in grid jobs. Therefore, DAGs and workflows are becoming more important in the context of job scheduling. Fölling, Grimme, Lepping, and Papaspyrou show in their paper that grids can produce win-win situations for independent sites if these sites are willing to collaborate by exchanging some jobs. However contrary to job scheduling on classical high performance architectures, there are hardly any workload traces available for grid computing, making it difficult to evaluate new scheduling approaches with respect to practical application. This problem is addressed by Lingrand, Montagnat, Martyniak, and Colling who analyze the workload of the presently largest production grid EGEE.

There are still a number of open issues in classical job scheduling on parallel architectures. For instance, Guim, Rodero, and Corbalan present an improvement of backfilling. Birkenheuer, Brinkmann, and Karl suggest applying overbooking, an approach well known in other scheduling and allocation areas, to job scheduling. Workload modeling also remains important in job scheduling for classical and new parallel architectures as workload traces remain rare and are not always applicable without modifications. To improve the present models, Minh and Wolters present an approach to better include temporal locality. The problem of preventing faulty jobs from disrupting the schedule is addressed in the paper by Thebe, Bunde, and Leung who suggest using trial runs of restartable jobs.

Pascual, Navaridas, and Miguel-Alonso discuss allocation policies that better consider the topology of the parallel architecture. This issue may become important again in the context of very large parallel architectures. In situations in which performance is more important than efficiency, Sinnen, To, and Kaur propose to use task duplication in order to improve the speed-up of jobs. The paper by Wolf, Bansal, Hildrum, Parekh, Rajan, Wagle, and Wu presents approaches for scheduling and allocation of streaming applications. These applications are likely to become more important for parallel architectures as these architectures enter a broader market. Bobroff, Coppinger, Fong, Seelam, and Xu suggest an extension of the well-known LoadLeveler job scheduler to handle virtualization.

Job scheduling problems that are relevant in the context of multi-core architectures were the topic of the last session of the workshop. Sun, Cao, and Hsu suggest using resource augmentation to handle non-clairvoyant and malleable jobs. They evaluate their approach with both simulation and theoretical analysis. Zeng and Sodan show that resource utilization on multi-core architectures can be improved with the help of forming appropriate groups of jobs. This holds for time and space sharing. Then Sodan reports on first experiences with adaptive scheduling that adjusts the size of jobs according to the actual load situation. This is also done in the context of virtual machines. The final paper by Vrba, Espeland, Halvorsen, and Griwodz discusses the benefit of workload stealing for complex applications with respect to utilization and load balancing in multi-core architectures.

The proceedings of previous workshops are available from Springer as LNCS volumes 949, 1162, 1291, 1459, 1659, 1911, 2221, 2537, 2862, 3277, 3834, 4376, and 4942. Since 1998 these volumes have also been available online.

Finally, we would like to explicitly thank Joachim Lepping for his support in organizing the publication of this volume.

July 2009

Eitan Frachtenberg
Uwe Schwiegelshohn

Organization

Workshop Organizers

Eitan Frachtenberg	Microsoft
Uwe Schwiegelshohn	TU Dortmund University

Program Committee

Su-Hui Chiang	Portland State University
Walfredo Cirne	Google
Allen Downey	Olin College
Dror Feitelson	The Hebrew University
Allan Gottlieb	New York University
Andrew Grimshaw	University of Virginia
Moe Jette	Lawrence Livermore National Lab
Virginia Lo	University of Oregon
Jose Moreira	IBM T.J. Watson Research Center
Reagan Moore	San Diego Supercomputer Center
Bill Nitzberg	Altair Engineering
Mark Squillante	IBM T.J. Watson Research Center
Dan Tsafrir	IBM T.J. Watson Research Center
John Towns	NCSA
Jon Weissman	University of Minnesota
Ramin Yahyapour	TU Dortmund University

Table of Contents

Dynamic Resource-Critical Workflow Scheduling in Heterogeneous Environments 1
Yili Gong, Marlon E. Pierce, and Geoffrey C. Fox

Decentralized Grid Scheduling with Evolutionary Fuzzy Systems 16
Alexander Fölling, Christian Grimme, Joachim Lepping, and Alexander Papaspyrou

Analyzing the EGEE Production Grid Workload: Application to Jobs Submission Optimization 37
Diane Lingrand, Johan Montagnat, Janusz Martyniak, and David Colling

The Resource Usage Aware Backfilling 59
Francesc Guim, Ivan Rodero, and Julita Corbalan

The Gain of Overbooking 80
Georg Birkenheuer, André Brinkmann, and Holger Karl

Modeling Parallel System Workloads with Temporal Locality 101
Tran Ngoc Minh and Lex Wolters

Scheduling Restartable Jobs with Short Test Runs 116
Ojaswirajanya Thebe, David P. Bunde, and Vitus J. Leung

Effects of Topology-Aware Allocation Policies on Scheduling Performance 138
Jose Antonio Pascual, Javier Navaridas, and Jose Miguel-Alonso

Contention-Aware Scheduling with Task Duplication 157
Oliver Sinnen, Andrea To, and Manpreet Kaur

Job Admission and Resource Allocation in Distributed Streaming Systems 169
Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, and Kun-Lung Wu

Scalability Analysis of Job Scheduling Using Virtual Nodes 190
Norman Bobroff, Richard Coppinger, Liana Fong, Seetharami Seelam, and Jing Xu

Competitive Two-Level Adaptive Scheduling Using Resource Augmentation 207
Hongyang Sun, Yangjie Cao, and Wen-Jing Hsu

Job Scheduling with Lookahead Group Matchmaking for Time/Space Sharing on Multi-core Parallel Machines 232
Xijie Zeng and Angela C. Sodan

Adaptive Scheduling for QoS Virtual Machines under Different Resource Allocation – Performance Effects and Predictability 259
Angela C. Sodan

Limits of Work-Stealing Scheduling 280
Željko Vrba, Håvard Espeland, Pål Halvorsen, and Carsten Griwodz

Author Index 301

Dynamic Resource-Critical Workflow Scheduling in Heterogeneous Environments*

Yili Gong[1], Marlon E. Pierce[2], and Geoffrey C. Fox[3]

[1] Computer School, Wuhan University, Wuhan, HuBei, P.R. China 430079
yiligong@whu.edu.cn
[2] Community Grids Lab, Indiana University, Bloomington, IN 47404
mpierce@cs.indiana.edu
[3] Community Grids Lab, Department of Computer Science, School of Informatics, Indiana University, Bloomington, IN 47404
gcf@indiana.edu

Abstract. Effective workflow scheduling is a key but challenging issue in heterogeneous environments due to the heterogeneity and dynamism. Based on the observations that not all tasks can run on all resources and acquired data transferring and queuing for a resource can be concurrent, we propose a dynamic resource-critical workflow scheduling algorithm which take into consideration the environmental heterogeneity and dynamism. We evaluate its performance by simulations and show that it outperforms another selected widely used approach.

Keywords: Dynamic Scheduling, Resource-Critical Scheduling, Workflow, Heterogeneous Environments.

1 Introduction

Heterogeneous distributed systems are widely deployed for executing computation and/or data intensive parallel applications, especially scientific workflows [1]. A workflow is a set of ordered tasks that are linked by logic or data dependencies and a workflow management system is employed to define, manage and execute these workflow applications [12]. The efficient execution of workflows in this kind of environments requires an effective scheduling strategy which decides when and which resources the tasks in a workflow should be submitted to and run on.

The environment includes both heterogeneous resource and policy. The software installation and configuration on resources are different as well as their physical computing capabilities. On the other side, the administration policies, such as access control policies, are autonomous and diverse. The dynamism means that the resource status, e.g. load, waiting time in the queue, availability, etc., changes over time are often uncontrollable. Thus the environment requires

* This work is supported by 973 Program (Grant No. 2005CB321807) and the National Natural Science Foundation of China (Grants No. 60773058 and No. 60672051).

that the workflow scheduling take into consideration both heterogeneity and dynamism, which make the problem very unique and challenging.

Concerning heterogeneity, we find that in practice due to access control policy, software version incompatibility or special hardware requirement, it is common that some tasks can not run on certain resources. With this observation, the tasks which can run on every resource are more flexible for scheduling than the resource-critical ones which can only run on just a few resources. For a resource-critical task, considering the more resource-flexible tasks before and after it as a group when scheduling should be better than scheduling them individually. This is the key idea of our resource-critical algorithms.

In terms of the timing of scheduling, there are two categories of workflow scheduling approaches: static scheduling and dynamic scheduling. A static scheduling system makes a schedule before the workflow starts to run based on available resource and environment information; while a dynamic scheduling approach schedule a workflow realtime. The static approach is comparatively simpler and easier to implement. However, its performance heavily relies on the accuracy of the resource and environment information. Unfortunately it is difficult to precisely predict this information due to resource autonomy and free will user behavior. To make full advantage of the known and predicted information as well as to adapt to dynamics of environment, dynamic scheduling is introduced. After initially scheduling, the schedule can be re-assigned according to the hitherto workflow execution progress and resource status at runtime. Thus we use the resource-critical mapping algorithm as a base, but when resource status changes, we using the base algorithm to reschedule the unfinished part of a workflow.

With respect to the architecture of a scheduling system, it could be either centralized or distributed. In a centralized workflow scheduling system, all the scheduling is fulfilled by a central scheduler. While in a decentralized scheduling system, there are many distributed brokers. The cooperation among the brokers is a tough problem and makes the system complicated. Since generally speaking, the calculation overhead of a dynamic scheduling algorithm is far less than the execution cost of a workflow, we still prefer a centralized approach.

Analyzing the makespan of a workflow, it can be seen that it is composed of tasks' execution time, data transferring time and waiting time in resource queues. To reduce any of these three items is beyond the reach of a workflow management system, but it is possible that the data transferring time and the waiting time can be concurrent, i.e. a task can be inserted into a resource waiting queue though its required data are not transferred to the resource yet. As long as the data are available when the task can actually get to use the resource, it works. This is also a principal distinction between our work and other existing work.

In this paper, our main contributions include that we propose a dynamic resource-critical workflow scheduling approach and prove that it outperforms the other selected widely used approach by simulations.

The rest of the paper is organized as follows. The related work is discussed in Section II. In Section III, the proposed dynamic resource-critical workflow scheduling algorithm is described. We elaborate the design of experiments and evaluate the performance of our algorithm in Section IV. The conclusion is shown in Section V.

2 Related Work

Extensive work has been done in the field of workflow scheduling in distributed environments. The key differentiators of our work in this paper from the related work lies in that (1) we do not assume that a task can run on all resources, which greatly extends the meaning of heterogeneity; (2) we assume that the data transferring and the waiting time for resources can be concurrent.

HEFT(Heterogeneous Earlier Finish Time) [10] is one of the most popular static heuristic and proven to be superior to other alternatives. Thus we select it as a base algorithm for comparison. In [4], Yu et al. proposes a HEFT-based adaptive rescheduling algorithm, AHEFT. It assumes the accuracy of estimation, i.e. communication and computation cost is estimated accurate and task starts and finishes punctually as predicted. In contrast, our proposed algorithm, DRCS, does not assume this. On the other side, in the AHEFT algorithm, a task can not start without all required inputs available on the resource on which the task is to execute; while we take advantage of the fact that data transferring and waiting in a queue for a resource can be concurrent. In AHEFT, if a task has not finished by *clock*, it will be rescheduled; while in DRCS, the unfinished tasks will be rescheduled when the resource's waiting time changes. [9] is a HEFT-based algorithm for dynamically created DAG scheduling.

The authors of [7] present a decentralized workflow scheduling algorithm which utilizes a P2P coordination space to coordinate the scheduling among the workflow brokers. It is a static scheduling approach and focuses on the scheduling coordination.

In [6] a distributed dynamic scheduling is proposed and it needs to collect resource information from local resource monitor services. Since the calculation overhead of a scheduling algorithm is far less than the execution duration of a workflow and resource information is available from existed third party sercies, we still adopt a centralized approach to avoid additional resource information propagation and synchronization.

Besides using makespan as the single criteria, there are some work on multi-criteria workflow scheduling. [8] proposes a bi-criteria scheduling heuristic based on dynamic programming. [5] presents a bi-criteria approach to minimize the latency given a fixed number of failures or the other way round.

3 Dynamic Resource-Critical Workflow Scheduling Algorithm

In this section, we give the details of our dynamic resource-critical workflow scheduling algorithm.

3.1 Task Status

During the execution of a workflow, a task is in one of the five possible statuses: unmapped, mapped, submit, running and finished, shown in Figure 1.

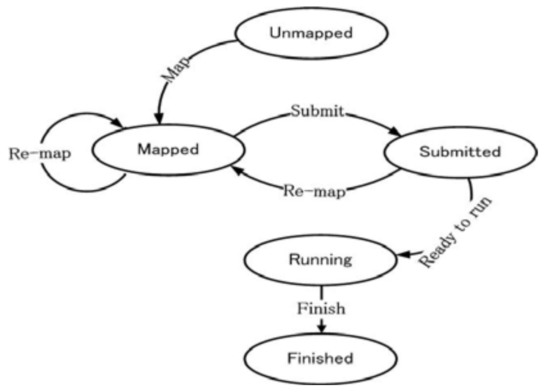


Fig. 1. A task's possible statuses and their transitions

- Unmapped: The task has not been mapped yet.
- Mapped: The task is assigned with a resource but has not been submitted.
- Submitted: The task has been submitted to the resource and is in the waiting queue.
- Running: The task is running.
- Finished: The task has finished and the result is ready for use or transfer.

If it is unmapped, mapped or submitted, a task is called in unfixed status or *unfixed* for short, and we consider it could be rescheduled; if it has started running or is finished, it should not.

3.2 Revised Resource-Critical Mapping

In [2], we proposed a Static Resource-Critical workflow Mapping heuristic, referred as SRCM here. Its key idea is that it is better to map neighboring resource-critical tasks as a group than to map them individually. In this paper, we adapt the static approach to dynamic scheduling.

Given a DAG (Directed Acyclic Graph) of a workflow application, $G = (V, E)$, $V = \{v_1, \ldots, v_N\}$ is the set of nodes in the DAG, i.e. tasks in the workflow, N is the total number of nodes. Hereafter, we use the two terms – node and task interchangeably. vol_{ij} denotes the volume of data generated by node i and required by node j, $i, j \in V$ and $ij \in E$.

Let the set of resources be $R = \{r_1, \ldots, r_M\}$ and M be the number of resources. c_{ij} is the computation cost of task i on resource j. If task i can not run on resource j, c_{ij} is infinity.

In a batch system, after submitted, a task typically has to wait for some time in a queue before actually get started. Due to the load on the resource, the waiting time varies with time. $w_{ij}(t)$ is the waiting time for task i on resource j at time t. Since in most heterogeneous environments, resources are shared among a lot of autonomous users, it's impossible to know the exact waiting time in future. So far we use QBETS [3] to predict the waiting times, represented as $w'_{ij}(t)$, which might be different from the actual waiting time of a task.

tr_{kl} is the transfer rate from resource k to l, $k, l \in R$. t_{ij}^{kl} is the communication cost between task i and j when i is executed on resource k and j on l, and $t_{ij}^{kl} = \frac{vol_{ij}}{tr_{kl}}$, $i, j \in V$, $k, l \in R$. When task i and j are executed on the same resource k, the communication cost is zero, i.e. $t_{ij}^{kk} = 0$.

Let $parent(v)$ be the parent(s) of task v and $child(v)$ be the child(ren) of task v, $v \in V$. These functions can be inferred from the DAG. We assume that the DAG has a single start node v_0 which has no parent, i.e. $parent(v_0) = \phi$ and a single end node v_{N-1} which has no child, i.e. $child(v_{N-1}) = \phi$; any of the other nodes has at least one parent and one child.

The main difference of the dynamic scheduling from the original static mapping is that the assignment of a task to a resource might be changed during the workflow execution, thus a variable, time t, is introduced. The function $map(v, t) : V \rightarrow R$ is the resource mapping of the task v at time t. When scheduling, the new mapping is only related to the last time scheduling result. t represents the current time and t' is the last scheduling time, correspondingly $map(v, t)$ is the current mapping and $map(v, t')$ is the last time mapping.

Let $EST(v, r, t)$ and $EFT(v, r, t)$ be the earliest start time and the earliest finish time of task v on resource r at time t respectively by estimation. $AST(v)$ is the actual start time and $AFT(v)$ is the actual finish time of task v.

To calculate the makespan of a workflow, we set $EST(v_0, r, 0) = 0$, $r \in R$, which means that the entry task v_0 can run on any satisfactory resource at time 0. For a task v, $EST(v, r, t)$ means calculated at time t, the earliest time at which all data that v requires have been transferred to resource r and v gets the right to run on r. Here, we make an important assumption that the waiting for the data and the resource be concurrent. Thus $EST(v, r, t)$ is defined as

$$EST(v, r, t) = \begin{cases} \max\left(drt(v, r, t), rat(v, r, t)\right), & v\ is\ unfixed, \\ AST(v), & otherwise. \end{cases}$$

Wherein $drt(v, r, t)$ is task v's data ready time and $rat(v, r, t)$ is its resource available time, which are described in detail later.

$EFT(v, r, t)$ is the earliest finish time of task v on resource r and

$$EFT(v, r, t) = \begin{cases} EST(v, r, t) + c_{vr}, & case\ 1, \\ AST(v) + c_{vr}, & case\ 2, \\ AFT(v), & case\ 3, \\ Infinity, & otherwise, \end{cases}$$

Wherein,
case 1: task v is unfixed;
case 2: task v is running and $map(v, t) = r$;
case 3: task v is finished and $map(v, t) = r$.

When a task finishes, its output will be transferred to its child(ren)'s assigned resource(s) immediately. Thus when calculating the data ready time for a parent-child pair, if the previously arranged data transfer is no longer valid (either or both of the mappings for the parent and the child change, we need arrange a new transfer. The earliest data ready time for data from parent u to child v on resource r at time t, $edrt(u, v, r, t)$, is as follows:

$$edrt(u, v, r, t) = \begin{cases} t + t_{uv}^{map(u,t),r}, & case\ 1, \\ EFT(u) + t_{uv}^{map(u,t),r}, & otherwise, \end{cases}$$

Wherein,
case 1: task u is finished and either $map(u, t') \neq map(u, t)$ or $map(v, t') \neq r$ or both.

The data ready time for all data that task v requires, $drt(v, r, t)$, is the maximum of the data ready times for all parents, i.e.

$$drt(v, r, t) = \max_{\forall u \in parent(v)} edrt(u, v, r, t).$$

The resource available time for task v on resource r at time t is the earliest time that v can get r and start to run. If a task has been submitted to its resource's waiting queue, as long as its data can arrive before it finishes waiting and gets the resource, the submission is valid. Otherwise, we need to resubmit the task at time t. Here, we assume that resources are FIFO batch systems and jobs submitted earlier should get resources no later than those submitted later.

$$rat(v, r, t) = \begin{cases} rat(v, r, t'), & case\ 1, \\ t + w'_{vr}(t), & otherwise, \end{cases}$$

Wherein,
case 1: v is submitted and $map(v, t') = r$ and $rat(v, r, t') > t$ and $drt(v, r, t) < rat(v, r, t')$.

The makespan, the overall execution time of the workflow, is the actual finish time of the end node, v_{N-1}, i.e. $AFT(v_{N-1})$.

In Algorithm 1, we show the revised resource-critical mapping (RRCM) algorithm. The key idea is to consider resource-critical jobs with their resource-flexible neighbors together as a group for mapping is better than mapping them individually.

Since a job may not run on all resources, we define $MR(v)$ as the *match ratio* of the number of resources on which the job v can run and the number of all resources, $v \in V$. By checking the computation cost array, it is easy to get $MR(v)$ by calculating the number of c_{vr} which is not equal to infinity, $r \in R$.

The algorithm has three steps: ranking, grouping and group scheduling.

Algorithm 1. The revised resource-critical mapping (RRCM) algorithm

```
1: // ranking
2: Set weights of nodes and edges with mean values.
3: Compute the rank of nodes by traversing the DAG upward, starting from the end
   node.
4: Sort the nodes in a non-ascending order of the rank values.
5: // grouping
6: $G_0 \leftarrow \phi; i \leftarrow 0$.
7: repeat
8:    Get a node $v$ in the order of nodes' rank values.
9:    if $v$'s mapping is unfixed and it is ungrouped then
10:      $G_i \leftarrow G_i + \{v\}$.
11:      for all $u$ such that $u$ is $v$'s descendants do
12:         if all ancestors of $u$ have been grouped, all nodes on the path from $v$ to $u$
            is in $G_i$ and $MR(u) \leq \alpha$ then
13:            $G_i \leftarrow G_i + \{u\}$.
14:         end if
15:      end for
16:      $i \leftarrow i + 1$; $G_i \leftarrow \phi$.
17:   end if
18: until there are no more nodes.
19: // mapping
20: for all group $G_i$, in ascending order of $i$. do
21:    Schedule the jobs in $G_i$.
22:    Choose the schedule with the smallest finish time.
23: end for
```

In the first step, each node and edge of the DAG is given the mean value of all its non-infinite values. The weight of a node is the mean of its computation cost on all matched resources. The weight of an edge is the mean of the maximum of the communication cost and the waiting time of all possible combinations of resources.

With the weights, upward ranking is computed and a rank value is given to each node. The rank value, $rank_i$, of a node i is recursively defined as follows: $rank_i = nw_i + \max_{\forall j \in children(i)} (ew_{ij} + rank_j)$, where nw_i is the weight of node i, and ew_{ij} is the weight of the edge connecting node i and j.

In the second step, nodes are grouped. First of all, nodes are sorted in the non-ascending order of their rank values. Tie-breaking is done randomly. Mark all fixed nodes as grouped. The first ungrouped node with the highest rank value is added to a group numbered 0. Check each of the node's children if its ancestors are grouped and its match ratio is below a certain valve α. If so, add the child node into the group, mark it as grouped and check its children further on. If no additional such node is found, make the next ungrouped node with the highest rank value as the first node of a new group, and so on. The outcome of this process is a set of ordered group, each of which consists of a node and its descendants on the path to which the match ratios of the nodes are all lower than the valve α.

In the third step, the node groups are mapped, where any algorithm for scheduling a DAG could be used. Since when scheduling a group, the mapping is probably incomplete, the makespan of the whole workflow is not a proper metric to value different assignments. Given a mapping, an end node is defined as a node which either has no children or whose children have not all yet been mapped. The *finish time* of a schedule is defined as the largest EFT of all end nodes in the group. Comparing two mappings, the one with the smaller finish time is preferred; if they have the same finish time, i.e. the same largest EFT, the one with the smaller second largest EFT is better; and so on. If all EFTs of the end nodes are the same, choose one of them randomly. So far, we adopt an enumerative algorithm to try all combinations of resources for a group and choose the one with the best EFTs of all end nodes.

The main difference between RRCM and SRCM lies in:

– Grouping nodes: on line 9, RRCM requires that if a node is fixed, it will not be grouped.
– EFT calculation: on line 21 and 22, RRCM's method to calculate EFT is different as described above.

3.3 Dynamic Resource-Critical Scheduling

In this section, we will introduce the dynamic resource-critical scheduling algorithm, which is based on RRCM. Specifically we use RRCM to schedule the unfinished workflow tasks, shown in Algorithm 2.

When a workflow is first submitted for execution, an initial resource schedule is generated. When some triggering events happen, such as the resource waiting time changing, the tasks would be rescheduled.

Algorithm 2. The dynamic resource-critical scheduling (DRCS) algorithm

```
1: $S \leftarrow \phi$
2: while ((($S == \phi$) OR (triggering event happens)) AND ($v_{N-1}$ is not finished)) do
3:     update the resource statuses
4:     update the task statuses
5:     call the revised resource-critical mapping (RRCM) algorithm
6:     update mapping() and schedule submit and/or data transfer events
7: end while
```

4 Experiments

In this section, we evaluate the performance of our dynamic resource-critical workflow scheduling algorithm. First, we introduce the experimental environment, followed by the metrics that we select. Then, we compare our DRCS with three other algorithms: AHEFT [4], HEFT [10] and SRCM [2].

4.1 Simulation Setup

1. *DAG Generator*
   We generate parameter sweep DAGs, whose structure is shown in [2]. Every DAG has one start node and one end node. Tasks on the same level in different branches have same resource requirements and similar execution time. We vary the branch number and the depth respectively from 4 to 12 and from 8 to 24, correspondingly the number of nodes varies from 34 to 290.
2. *Heterogeneity Model*
   The heterogeneity model we adopt is based on the loosely consistent heterogeneity model, also called the proportional computation cost model in [11]. Instead of generating the resource computing power randomly, we use the practical numbers from TeraGrid.

   The baseline execution time of a task is chosen by using a random uniform distribution over the interval [10, 100]. The computing cost of a task on a resource is a random between 95% and 105% of the quotient of its baseline time divided by the resource's computing power number.
3. *Match Ratio*
   This is a factor used in SRCM and DRCS introduced by the factor that some tasks can never run on certain kinds of resources. The match ratio for a task is the ratio of the matching and total resource numbers. The ratios are generated randomly among (0, 1] and a task can run on at least one resource.
4. *Communication Bandwidth*
   The communication bandwidth between any two resources is a random number between 5M/s and 300M/s, which are the bandwidth range we measured on TeraGrid.
5. *Communication-to-Computation-Ratio (CCR)*
   CCR of a workflow is defined as its average communication cost divided by its average computation cost for all resources. If a workflow's CCR is low, it would be considered as a computation intensive application; while if the CCR is high, it is data intensive.
6. *Waiting-to-Computation Ratio (WCR)*
   WCR is the ratio of the average resource waiting time to the workflow computation time.
7. *Match Ratio Threshold (MRT)*
   This value is used by SRCM and DRCS to decide what kind of nodes should be grouped together for mapping. If MRT is so small that no node's match ratio below it and every node is an individual group, the SRCM and DRCS will degenerate to HEFT and AHEFT respectively. If MRT is large, the group size grows, it is time-consuming to find the best solution for a big group. In our experiments, we set MRT between 0.1 to 0.5.
8. *Parameters for Dynamic Changing of Resources*
   We use two parameters to represent the changing of resources:
   – Resource Change Period (RCP) – the interval of the resource waiting time change;

– Resource Fluctuation Indicator (RFI) – the waiting time fluctuation percentage from the initial value.

4.2 Metrics

To compare the performance of the four algorithms, the main metric we use is average makespan difference ratio, which is based on two metrics: makespan and average makespan difference ratio.

1. *Makespan*
   Makespan is the complete time needed to finish a workflow under a certain workflow scheduling algorithm.
2. *Makespan Difference Ratio*
   We use the makespan of HEFT algorithm as a base, and the performance of other algorithms is compared with HEFT's. Thus the average makespan difference ratio of HEFT is always 0.
3. *Average Makespan Difference Ratio*
   For any given branch number and depth, we generate 200 DAGs with their own task computation costs, communication cost, resource matchings and resource bandwidths, each of which is called a case. With each combination of the branch number, depth, CCR and MRT, these four algorithms will run on the 200 cases.

   The average makespan difference ratio is the average of the makespan difference ratios for the 200 cases under the same environmental setting.

4.3 Results

In our simulation, we vary the factors introduced above to evaluate their influence on the four workflow scheduling approaches.

Except in the experiment 3, which deals with how the DAG shape of the parameter sweep applications affects the scheduling, the DAG branch number and depth are fixed at 8 and 16 respectively.

1. *Communication-Computation-Ratio (CCR)*
   To analyze the influence of CCR on the scheduling performance, we set $WCR = 1.0$, $RCP = 5000$, $RFI = 0.2$, and $MRT = 0.3$ for the two resource-critical algorithms. The makespans and the average makespan difference ratios under various CCR values are shown in Figure 2 and Figure 3 respectively. Since we set computation cost fixed, bigger CCR means bigger communication cost, thus for all the algorithms, the overall makespan gets longer.

   When CCR is small, the two static approaches, HEFT and SRCM, and the two dynamic approaches, AHEFT and DRCS, perform almost the same. As CCR grows, the performance of SRCM and DRCS get better and when CCR is over 3, the static approach SRCM even outperforms the dynamic approach AHEFT. This surpassing depends on the fact that most benefit of the resource-critical algorithms comes from the communication time saving.

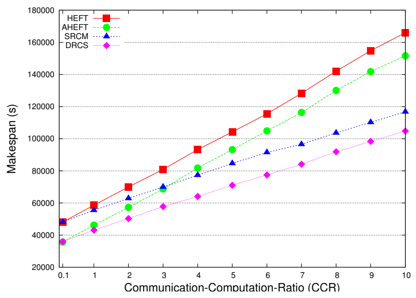


Fig. 2. Makespans under various CCRs

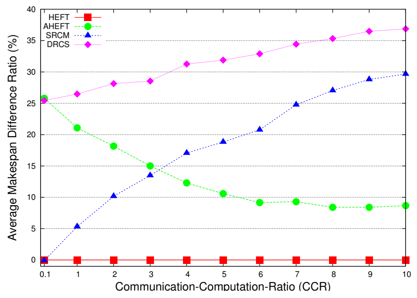


Fig. 3. Average makespan difference ratios under various CCRs

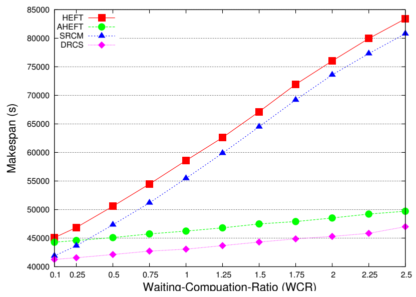


Fig. 4. Makespan under various WCRs

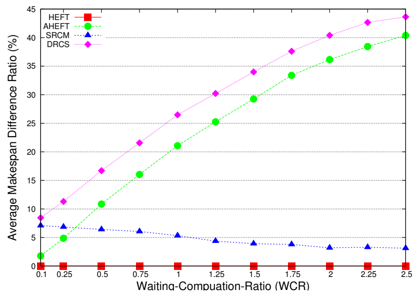


Fig. 5. Average makespan difference ratios under various WCRs

As the weight of the communication time in the makespan gets higher, the benefit gets bigger. Therefore, SRCM and DRCS are more suitable for the data intensive applications.

Figure 3 presents the improvement of AHEFT, SRCM and DRCS over HEFT, from which we can notice more clearly the tendency that AHEFT approaches HEFT and SRCM approaches DRCS. In further on simulation, when $CCR = 100$, the difference between HEFT and AHEFT is about 0.69% and the difference between SRCM and DRCS is about 1.19%. This is because as CCR increases, the dynamic scheduling algorithms have less opportunity to re-assign the tasks, since the cost of moving data gets bigger.

2. *Waiting-Computation-Ratio (WCR)*
Here $CCR = 1.0$, $RCP = 5000$, $RFI = 0.2$, and $MRT = 0.3$. Figure 4 and Figure 5 present the results. For all four algorithms, the WCR increasing causes the increasing of the waiting time cost, correspondingly the increasing of the makespan.

When WCR is small ($= 0.1$), the two resource-critical algorithms performs almost the same and better than HEFT and AHEFT. While as WCR grows, the two dynamic algorithms are much less affected than the static

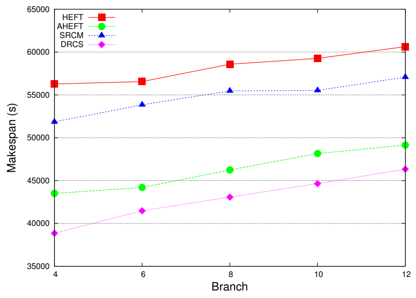


Fig. 6. Makespan under various branch numbers

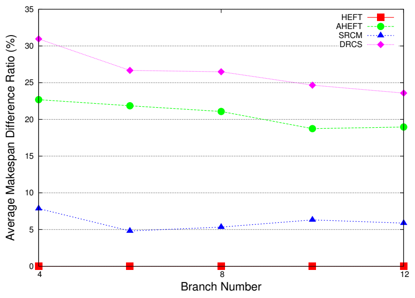


Fig. 7. Average makespan difference ratios under various branch numbers

ones. It shows that dynamic scheduling can adjust the schedule when the waiting time changes to shorten the overall execution time and the longer the waiting time, the more obvious the advantage. It can be seen that DRCS is always performs better than the other three, including AHEFT.

From Figure 5, it can been seen that the performance of HEFT and AHEFT tends to close to that of SRCM and DRCS respectively. When $WCR = 10$, the average makespan difference ratio of SRCM over HEFT is only 0.65%, and the difference between AHEFT and DRCS is 0.92%. This shows again that the benefit of SRCM and DRCS are from the communication cost reduction, once the waiting time gets longer, the weight of the communication cost decreases, thus the performance improvement decreases.

3. *DAG branch number and depth*
In this set of experiments, $CCR = 1.0$, $WCR = 1.0$, $RCP = 5000$, $RFI = 0.2$, $MRT = 0.3$. When the branch number varies, the depth is fixed at 16; while when the depth varies, the branch number is 8.

As the branch number varies from 4 to 12, the makespan of four algorithms increases (refer to Figure 6). This happens due to the reason that the branch number growth causes more tasks are ready to run at approximately the same time, since the capacity of resources is limited, some of the tasks have to wait longer to actually acquire the resources.

Figure 7 presents the performance improvement of the two dynamic algorithms decreases with the branch number. For instance, when the branch number is 4, the makespan difference ratios of AHEFT and DRCS are 22.69% and 30.96 respectively; while when the branch number is 12, the ratios are 18.96% and 23.59%.It shows that when the resource competition is fierce, there is little room for the dynamic approaches to reschedule the tasks to get better waiting time. In contrast, the difference ratio of SRCM over HEFT does not change much with the different branch numbers.

It is evident that the makespan increases approximately linearly as the depth varies from 8 to 24 (see Figure 8 and Figure 9), since more tasks should be executed sequentially. The deeper the depth, the bigger the improvement ratio of the two resource-critical algorithms than the corresponding HEFT or

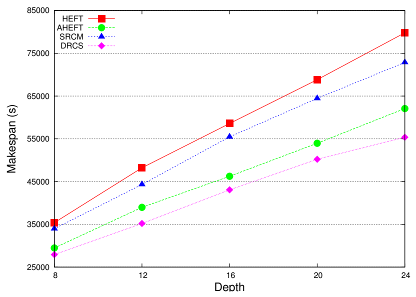


Fig. 8. Makespan under various depths

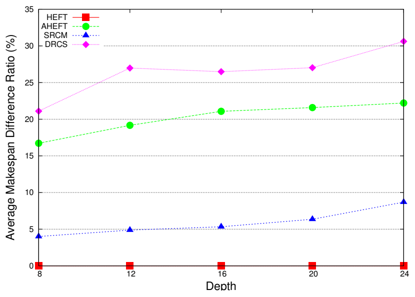


Fig. 9. Average makespan difference ratios under various depths

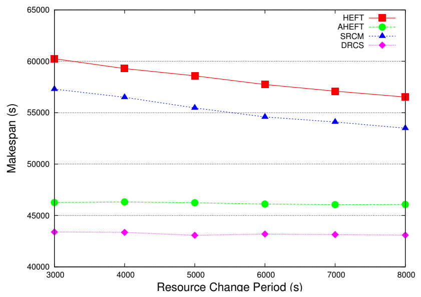


Fig. 10. Makespan under various resource change periods

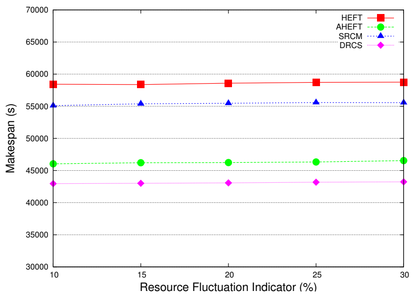


Fig. 11. Makespan under various resource fluctuation percentages

AHEFT algorithms. The improvement ratio of SRCM over HEFT increases from 4.00% to 8.69% and that of DRCS over AHEFT increases from 5.26% to 10.82%. This shows that deeper depth allows the resource-critical algorithms group more nodes together to achieve better schedule.

4. *Resource Change Period (RCP) and Resource Fluctuation Indicator (RFI)* To measure how the resource changing affect the algorithms, we introduce two factors: Resource Change Period and Resource Fluctuation Indicator, which depict when and by what degree resources change.

   In Figure 10, the setting is $CCR = 1.0$, $WCR = 1.0$, $RFI = 0.2$, and $MRT = 0.3$. We can see that the resource change period has no influence on the performance of the dynamic approaches. In contrast, as the period grows, the makespan of the static ones decreases. The static approaches decide the schedule of the workflow before it starts, and will not change during the its execution duration. Thus when the resources change, i.e. the waiting times change, the initial schedule will become unsuitable and the performance suffers. If the resource change period is long, it would change less times during the workflow execution and the suffering would be less,

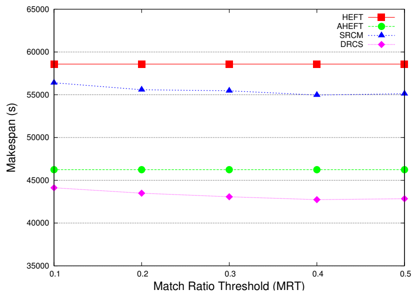


Fig. 12. Makespan under various match ratio thresholds

correspondingly the makespan improves. As a result, the dynamic scheduling methods are adapted to the dynamic resource environment. In Figure 11, $CCR = 1.0$, $WCR = 1.0$, $RCP = 5000$, and $MRT = 0.3$. It shows that the resource fluctuation percentage does not affect the performance of workflow scheduling much. This is because the resource status fluctuation makes some jobs finish earlier than predicted and some later, and the influence is balanced out.

5. *Match Ratio Threshold (MRT)*
Match ratio threshold is only used in the resource-critical algorithms. Here, we set $CCR = 1.0$, $WCR = 1.0$, $RCP = 5000$, and $RFI = 0.2$.
In Figure 12, as the MRT increases from 0.1 to 0.5, the makespan of SRCM and DRCS decreases from 56404.30s to 55122.44s and from 44122.46s to 42841.48s respectively. This is because with a bigger MRT, the algorithms could group more nodes together and try all the combinations to select the best out them.

5 Conclusion

In this paper we have presented DRCS, an efficient workflow scheduling approach for heterogeneous and dynamic systems based on the resource-critical algorithm. Aiming at heterogeneity, the algorithm combines the resource-critical tasks with their ancestors and/or descendants together and finds the best schedule for them as a group. For dynamism, it reschedules the unfinished tasks according to the current resource status. To evaluate the performance of DRCS, simulation studies were conducted to compare it with other competitors in the literature, HEFT, AHEFT and SRCM. It is shown that DRCS outperforms HEFT, AHEFT and SRCM in almost all environments in terms of makespan. Especially, the two resource-critical idea based algorithms, DRCS and SRCM are suited for data-intensive applications. The two dynamic scheduling algorithm, DRCS and AHEFT are superior in the long waiting time systems.

To further on adapt to the unreliable, dynamic and heterogeneous environment, we plan to investigate the effect of resource liability and task failure on the scheduling performance.

References

1. The QuakeSim Project Website, http://quakesim.jpl.nasa.gov/
2. Gong, Y., Pierce, M.E., Fox, G.C.: Matchmaking Scientific Workflows in Grid Environments. In: 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007), Cambridge, MA (November 2007)
3. 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)
4. Yu, Z., Shi, W.: An Adaptive Rescheduling Strategy for Grid Workflow Applications. In: 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS 2007), Long Beach, CA (March 2007)
5. Benoit, A., Hakem, M., Robert, Y.: Ault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms. In: 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008) Miami, FL (April 2008)
6. Dong, F., Akl, S.G.: Mobile Agent Based Workflow Rescheduling Approach for Grids. I. In: 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007), Cambridge, MA (November 2007)
7. Ranjan, R., Rahman, M., Buyya, R.: A Decentralized and Cooperative Workflow Scheduling Algorithm. In: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), Lyon, France (May 2008)
8. Wieczorek, M., Podlipnig, S., Prodan, R., Fahringer, T.: Bi-criteria Scheduling of Scientigc Workflows for the Grid. In: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), Lyon, France (May 2008)
9. Hunold, S., Rauber, T., Suter, F.: Scheduling Dynamic Workflows onto Clusters of Clusters Using Postponing. In: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), Lyon, France (May 2008)
10. Topcuouglu, H., Hariri, S., Wu, M.Y.: Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
11. Kwok, Y., Ahmad, I.: Dynamic Critical Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. IEEE Transactions on Parallel and Distributed Systems 7(5), 506–521 (1996)
12. Yu, J., Buyya, R.: Taxonomy of Workflow Management Systems for Grid Computing. Journal of Grid Computing 3(3-4), 171–200 (2005)

Decentralized Grid Scheduling with Evolutionary Fuzzy Systems

Alexander Fölling, Christian Grimme,
Joachim Lepping, and Alexander Papaspyrou

Robotics Research Institute, TU Dortmund University, 44221 Dortmund, Germany
{firstname.lastname}@udo.edu

Abstract. In this paper, we address the problem of finding workload exchange policies for decentralized Computational Grids using an Evolutionary Fuzzy System. To this end, we establish a non-invasive collaboration model on the Grid layer which requires minimal information about the participating High Performance and High Throughput Computing (HPC/HTC) centers and which leaves the local resource managers completely untouched. In this environment of fully autonomous sites, independent users are assumed to submit their jobs to the Grid middleware layer of their local site, which in turn decides on the delegation and execution either on the local system or on remote sites in a situation-dependent, adaptive way. We find for different scenarios that the exchange policies show good performance characteristics not only with respect to traditional metrics such as average weighted response time and utilization, but also in terms of robustness and stability in changing environments.

1 Introduction

Modern science more and more relies on experimental scientific discovery made with extensive simulations, and during the last decade, Grid Computing has become the key infrastructure in academia to support this development. The use of Grid Computing, however, is not anymore limited to HPC/HTC-centric communities such as High Energy Physics, Astronomy, or Climate Research, which have a certain tradition of using such infrastructures. Other sciences—e.g. Financial Services, Construction Engineering, and even arts and humanities—also start to adopt Grid Computing as a tool for e-Science, and show an ever-increasing demand for computing power and storage space.

While well-established approaches such as the EGEE environment [6] have relied on centralized middleware infrastructures for whole e-Science communities, other—mostly emerging—efforts have chosen a *Service Grid* approach with smaller, more community-tailored Grids. In the latter case, however, a strong demand for enabling collaboration and cooperation on the infrastructure layer between the different communities and Grids can be observed.

A major issue in such collaborations is the possibility of inter-community resource usage: Although most communities run their own data centers, working together in an ad-hoc manner by allowing alien workload to be run on community hardware is still a tedious task and usually requires resorting to 1980s-style command line interfaces and undesirable micro-management. This is mainly due to technical issues: Many e-Science infrastructures show a lack of standardization, and therefore, collaborations between the workload gateways (usually Grid schedulers or brokers) fail on a compatibility level. There are, however, also organizational issues: Each community Grid strives for delivering the highest possible Quality of Service to its *own* users and, as such, is only interested in participating in joint efforts if they are beneficial for all participants likewise.

This last aspect is an open research problem in the field of Grid scheduling, see Grimme et al. [7,8]: algorithms for the exchange of workload between different Grid communities—with respect to common performance metrics—have to perform at least as good as in the non-cooperative case. Otherwise, the motivation for participating in a HPC/HTC federation, vanishes quickly, since one of the participating user communities will suffer from the collaboration. Here, we can identify four important properties for such algorithms:

– *Support for environments with very strict information policies:* Although almost every Grid provides various kinds of information services, data regarding the machines themselves such as their current or overall utilization, average response times, or throughput is often kept confidential due to competition reasons.
– *Strict separation from local resource management systems (LRMS):* Machine owners usually have their own operational policies implemented on their systems and obviously are not willing to cease control over the machines they are obliged to fund.
– *Situation-dependent, adaptive decision-making:* The current state of the system is crucial when deciding on whether to accept or decline foreign workload, e.g. allowing for additional remote jobs if the local system is already highly loaded seems to be inappropriate.
– *Robustness and stability in changing environments:* Even with respect to future, still unknown (and usually unpredictable) job submissions, it is crucial that aspects such as complete site failures or even rogue participants are handled gracefully with respect to the own *and* overall performance.

In the work at hand, we address these properties using a Fuzzy based approach for job exchange in Computational Grids, where the controller acts depending on the current system state. The states are modeled by Fuzzy sets which are represented by simple membership functions. Such Fuzzy System based scheduling techniques have been successfully applied to online scheduling problems before, see for example Franke et al. [4]. They outperform most static scheduling heuristic due to their ability to flexibly adapt decisions to changing environments. As they have proven to be a reliable concept to tackle challenging online scheduling problems, we decide to also apply them in the Grid context. In order to establish good rules for the Fuzzy System, we furthermore use evolutionary algorithms for finding parameterization of the Fuzzy membership functions. This approach is especially suitable because of the possibility to find a simple and efficient encoding of the whole controller. This combination of Fuzzy Systems and evolutionary algorithm is commonly denoted as *Evolutionary Fuzzy Systems*, see Cordón et al. [1]. We show that our approach, while respecting the aforementioned requirements for Grid scheduling algorithms, shows adequate performance characteristics in real setups.

The remainder of the paper is organized as follows: In Section 2, we establish the basis for understanding our model, algorithm, and optimization. We then introduce our system model in Section 3 and our Fuzzy Grid Scheduling approach in Section 4. After discussing tools for performance measurement in Section 5, we depict the evolutionary learning of rule sets in Section 6. Next, we evaluate our approach with respect to adaptiveness in a Grid federation in Section 7 and robustness in unknown environments in Section 8 and conclude our work Section 9.

2 Background

This section briefly introduces the basics of job scheduling on Massively Parallel Processing (MPP) systems, Evolutionary Fuzzy Systems, and evolutionary algorithms. These definitions and tools are applied throughout the paper to easily describe the used Grid architecture as well as the proposed approach for realizing job migration.

2.1 Job Scheduling for MPP Systems

The scheduling of MPP systems is an online problem as jobs are submitted over time and the precise processing times of those jobs is unknown in advance. Furthermore, information about future jobs are not available. We assume independent rigid parallel batch jobs for our analysis, which are dominant on most parallel computer systems. Those jobs are neither moldable nor malleable and require concurrent and exclusive access to the requested resources. Formally, each job j is characterized by its degree of parallelism m_j and its processing time p_j. Although many additional criteria are conceivable, see Feitelson et al. [3], we restrict ourselves to only those two required job properties.

During the execution phase, job j requires the concurrent and exclusive access to $m_j \leq m_k$ processing nodes with m_k being the total number of nodes on the MPP system at site k. The number of required processing nodes m_j is available at the release date r_j of job j and does not change during the execution. As the network does not favor any subset of the nodes and all nodes of a parallel computer system are either identical or very similar, we assume that a job j can be processed on any subset of m_j nodes of the system.

Further, most current real installations of parallel computers do not use preemption but let all jobs run to completion. The completion time of job j within the schedule S is denoted by $C_j(S)$.

2.2 Evolutionary Algorithms

Optimization algorithms that mimic the natural process of Darwinian evolution are widespread in computer science and often applied for parameter optimization when the fitness landscape of the optimization problem is unknown.

A specific type of these algorithms—Evolution Strategies [11]—operate on a population of μ individuals, where each individual represents a real-coded solution to the given optimization problem. These approaches apply variation operators like mutation (a random change in genome) and recombination (combining two or more parent individuals' genomes) to breed λ offspring individuals from those μ parental individuals, followed by a global selection process in which the individuals compete against each other to form the new μ parents for the next generation. The above described evolutionary loop is executed until a given termination criterion, like a fixed number of generations or a quality level within the objective space, is satisfied. Two versions of Evolution Strategies are distinguishable: In the (μ, λ)-strategy, the next parent generation is selected just from the offspring individuals while the $(\mu + \lambda)$-strategy selects the best individuals of both the parent and offspring generations. All other individuals are removed from the system and the next loop iteration starts.

2.3 Evolutionary Fuzzy Systems

Since their conceptualization in the early 1960s Fuzzy Systems have been widely and successfully applied to various areas like for example control systems or classification. Especially in control systems, they are particularly suited for the representation of problem specific knowledge, as imprecision or vague descriptions are common properties of expertise. Currently, many decision making methods (e.g. in the fields of resource management or robot behavior) solve problems in a heuristic fashion. They give advices for actions in certain—often fuzzy described—situations that have turned out to be profitable with respect to a given objective. Such a collection of situation-dependent expertise is called a *knowledge base*.

There are several advantages to represent a knowledge base by Fuzzy logic within a Fuzzy System: The interpolative nature of Fuzzy Systems has the ability to express partial and concurrent activations of behaviors and gradual transitions between them. Further, the behavior can be conveniently synthesized by a set of IF-THEN rules using linguistic terms to encode the expert knowledge. Finally, due to its approximate reasoning capabilities, Fuzzy logic produces controllers that are robust to uncertainty and imprecision. Especially, the latter property is of great importance for the problem addressed in this paper, as we aim to produce robust exchange mechanisms within changing environments.

However, one of the major drawbacks of classic Fuzzy Systems is their missing learning ability. They always require a existing knowledge base that has to be derived from experts knowledge which is often called training data. In many cases, that data is not available and the design of Fuzzy Systems is not possible at all. Also for the problem at hand we cannot revert to any kind of training data. Therefore, we employ an evolutionary learning process to automate the Fuzzy System design.

Evolutionary Fuzzy Systems are Fuzzy Systems derived and optimized by an evolutionary learning process. For these systems an evolutionary algorithm is employed to learn or tune different components. They are always applied, if neither expert knowledge nor training data is available or cannot be transformed directly into corresponding rules. Those algorithms do not require particular knowledge about the problem structure and can be applied to various systems.

3 System Model

The problem of job distribution between federated compute clusters has been continuously studied since the emergence of Grid computing in the beginning of the 1990s. Early approaches favor a hierarchical scheduling structure, where a central scheduler instance—often called Meta-Scheduler, Grid Scheduler, or Broker—delegates submitted jobs to subordinated partner sites [10]. The most profound problem of this scheduling structure is its bad fault-tolerance and lack of scalability.

With respect to the basic parameters of modern e-Infrastructures regarding organizational autonomy and equity, we therefore assume our Computational Grid as a loose cooperation between different HPC centers—further referred to as sites—and consider Massively Parallel Processing (MPP) systems as their basic entities. For every MPP entity we assume an own local user demand for computational resources which is reflected by the sites' originating workload. This includes the submission characteristics, but also the adaptation of the submitted jobs' resource demand to the local configuration. This scenario is based on the perception that, as a general rule, Grid environments are not build from scratch, but emerge from collaborations between different organizational domains, each of which already operating one or more MPP systems for internal purposes, in order to serve a prescribed, project-driven community of users.

More formally, a Computational Grid consists of $|K|$ independent sites. Each site $k \in K$ is modeled by m_k parallel processors which are identical such that a parallel job can be allocated on any subset of these machines. Splitting jobs over multiple sites (multi-site computation) is not allowed. Moreover, we assume that all sites only differ in the number of available processors, but not in their speed: As we focus on the job exchange algorithms, the differences in execution speeds can be neglected, see Schwiegelshohn et al. [12].

The workload management within the infrastructure is conducted by a two-tier middleware, see Figure 1, comprising a Local Resource Management System (LRMS) and a Grid Resource Management System (GRMS) on each site. While the LRMS takes care of assigning workload to resources for the local site only, the GRMS decides on the delegation of jobs from and to the site. Users submit their workload to the local site in the same manner as on classic LRMS systems; a small submission component intercepts those and forwards them to the local GRMS for further inspection.

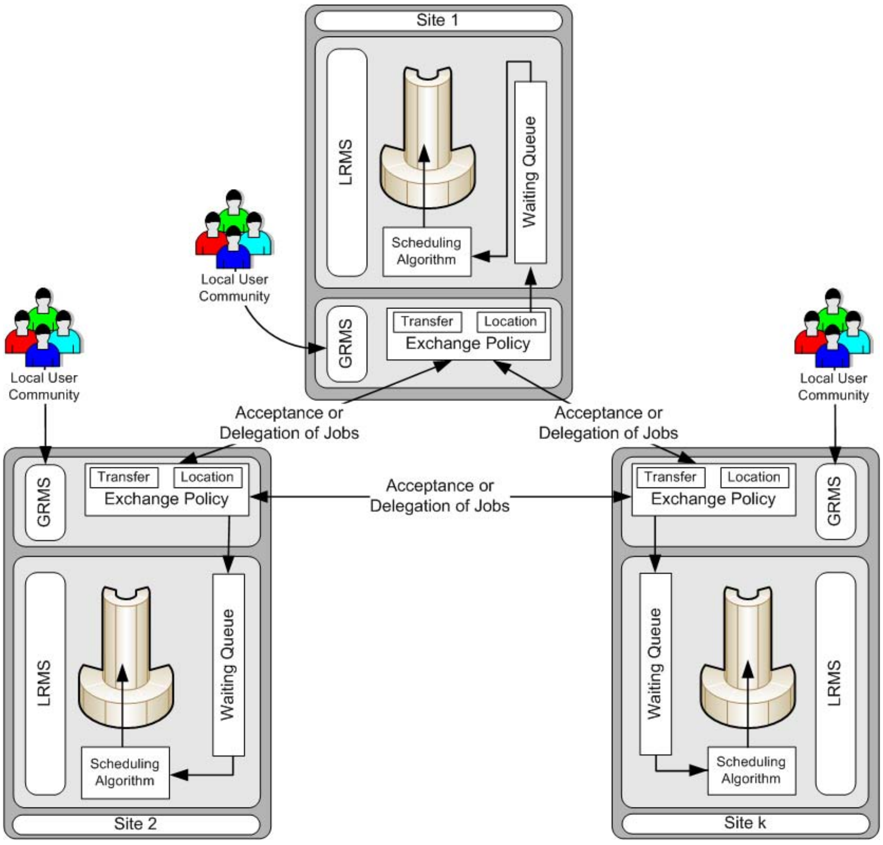


Fig. 1. Computational Grid scenario with independent sites in a federated environment

That is, jobs that are submitted to the local site scheduler may not be accepted for execution elsewhere because of their resource demand being oversized for some or all of the other sites. Ignoring the inter-site collaboration for a moment, we describe the local scheduling problem on MPP systems in the next paragraph.

3.1 LRMS Layer

The Local Resource Management System (LRMS) layer consists of a waiting queue and a scheduler. The waiting queue stores all locally submitted jobs while the scheduler executes a specific scheduling strategy in order to assign jobs from the waiting queue onto the available local resources. On MPP system layer, this approach allows the realization of priorities for jobs of different user groups. Usually, the scheduling strategies are formulated by the system provider to fulfill the users' needs. Although many special-purpose algorithms exists that are tailored for certain MPP system owner priorities, we use the basic and simple First-Come-First-Serve (FCFS) algorithm as an example on LRMS. The heuristic starts the first job of the waiting queue whenever enough idle resources are available. Despite the very low utilization that is produced in the worst case this heuristic works well in practice [13]. Please note that our job exchange methodology is not restricted to any kind of local scheduling algorithm but it serves for any arbitrary scheduling algorithm on the LRMS layer.

3.2 GRMS Layer

The Grid Scheduling Resource Management System (GRMS) extends every site by an additional layer on top of the LRMS, see Figure 1. The GRMS accepts locally submitted jobs on behalf of the underlying LRMS. The actual exchange behavior is realized exclusively by the GRMS and due to this strict layered architecture the LRMS is kept completely unmodified. Both removal of jobs from LRMS queues as well as any kind of intervention in the local scheduling process is prohibited. Furthermore, the GRMS is transparent to local users and the LRMS. From the users point of view, all submitted jobs are executed on the local site, whereas each LRMS considers every job as a locally submitted independent of its origin. Decisions about a job's delegation to another GRMS or local scheduling is made by a deployed exchange policy.

This exchange policy can be differentiated into two independent policies:

Location Policy
This policy becomes relevant if more than one exchange partner is available in the Grid. Thus, there exists more than one possibility to delegate a job to a remote Grid participant. For such scenarios, the location policy determines as a first step the sorted subset of possible delegation targets ①, see Figure 2.

Transfer Policy
After the location policy has been applied the transfer policy specifies whether a job should be delegated to a certain partner or not. For this purpose the policy is applied separately on each partner in an redetermined order. Every time the transfer policy is consulted it decides whether the job should be executed locally ② or delegated to the considered partner ③. In the first case, the job is sent to the remote LRMS ④ and in the other case the considered partner is requested for a job's acceptance. A request can be replied in two different ways:

1. The job is accepted by the remote partner ⑤. In this case, the job is delegated to this partner and no other further delegation attempts have to be made.
2. If the acceptance is declined the transfer policy is applied for another partner in the Grid ⑥. This iterative procedure is continued until all partners have been requested. If none of the Grid participants is willing to accept the job, the requesting site must execute it locally ⑦, ⑧, and ④.

Further, the transfer policy has to decide about jobs that are offered from remote sites and can choose between accepting or declining a job offer. In the former case, the job is immediately forwarded to the LRMS ⑨, while it is rejected in the latter case ⑩.

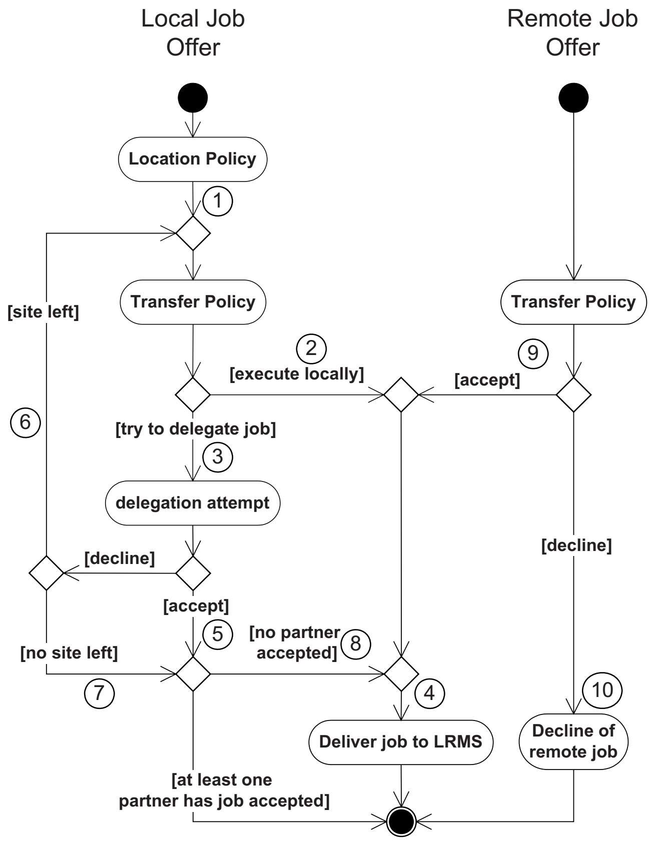


Fig. 2. Decision making concept at GRMS layer

4 Fuzzy System Based Grid Scheduling Approach

For the design of our GRMS decision policy we apply the method of Fuzzy inference proposed by Takagi, Sugeno and Kang, which is known as the Takagi-Sugeno-Kang (TSK) model in Fuzzy Systems literature [14].

Such a decision policy is founded on a set of rules. Each specific rule describes a system state in which decisions about the acceptance or refusal of jobs must be made. Thus, each system state is described by a set of features. From the different parts of the overall system various state describing features are conceivable. They might be related to the current state of the LRMS layer or to the currently job to decide. Please note that information about remote sites' systems states is assumed strictly classified.

Following the Fuzzy rule concepts, a rule consists of a feature describing conditional part and a consequence part that decides on the acceptance or decline of an offered job. The so composed rule base constitutes the core of the rule system that can therefore be considered as a controller. The current system is checked whenever a new job has been submitted to the local system or has been offered from remote sites. In all those cases the current system state might change and the controller output has to be changed if necessary. The controller concept is described in the next paragraph.

4.1 Fuzzy System for Decision Making

The general TSK model consists of N_r IF-THEN rules R_i such that

$$\begin{aligned} R_i \quad := \quad & \text{IF } x_1 \text{ is } g_i^{(1)} \text{ and } \dots \text{ and } x_{N_f} \text{ is } g_i^{(N_f)} \\ & \text{THEN } y_i = b_{i0} + b_{i1}x_1 + \dots + b_{iN_f}x_{N_f} \end{aligned} \tag{1}$$

where $x_1, x_2, \dots, x_{N_f}$ are input variables and elements of a vector $\boldsymbol{x}$, and y_i are local output variables. Further, $g_i^{(h)}$ is the h-th input Fuzzy set that describes the membership for a feature h. Thus, system state is described by a number of N_f features. The actual degree of membership is computed as function value of an input Fuzzy set which is characterized for example by a Gaussian Membership Function (GMF). The here used Fuzzy sets are explained in the next section. Furthermore, b_{ih} are real valued parameters that specify the local output variable y_i as a linear combination of the input variables $\boldsymbol{x}$. The overall output of the system $y_D(\boldsymbol{x})$ is computed by Equation 2.

$$y_D(\boldsymbol{x}) \quad = \quad \frac{\sum_{i=1}^{N_r} \phi_i(\boldsymbol{x}) y_i}{\sum_{i=1}^{N_r} \phi_i(\boldsymbol{x})} = \frac{\sum_{i=1}^{N_r} \phi_i(\boldsymbol{x})(b_{i0} + b_{i1}x_1 + \dots + b_{iN_f}x_{N_f})}{\sum_{i=1}^{N_r} \phi_i(\boldsymbol{x})} \tag{2}$$

where $\phi_i(\boldsymbol{x})$ is the *degree of membership* of rule R_i for a given input vector $\boldsymbol{x}$, which is defined as

$$\phi_i(\boldsymbol{x}) = g_i^{(1)}(x_1) \wedge g_i^{(2)}(x_2) \wedge \dots \wedge g_i^{(N_f)}(x_{N_f}) \tag{3}$$

Each rule's recommendation is weighted by its degree of membership with respect to the input vector $\boldsymbol{x}$. The corresponding output value of the TSK-System is then computed by the weighted average output recommendation over all rules. In the following, we explain how this very general model is adapted to the here addressed problem of decision making. The specific coding of rules and the output computation will be detailed in the following paragraphs.

4.2 Encoding of Rules

For a single rule R_i every feature h of all N_f features is modeled by a $(\gamma_i^{(h)}, \sigma_i^{(h)})$-Gaussian Membership Function (GMF)[1] with no normalization as shown in Equation 4.

[1] Different from the common notation we denote the mean of the GMF by γ to avoid conflicts with the parental population size of Evolution Strategies which is in this paper denoted by μ.

$$g_i^{(h)}(x) = \exp\left\{\frac{-(x-\gamma_i^{(h)})^2}{{\sigma_i^{(h)}}^2}\right\} \tag{4}$$

This function is completely described by defining the $\gamma_i^{(h)}$ and $\sigma_i^{(h)}$ values. The $\gamma_i^{(h)}$-value adjusts the center of the feature value, while $\sigma_i^{(h)}$ models the region of influence for this rule in the feature domain. In other words, for increasing $\sigma_i^{(h)}$ values the GMF becomes wider, while the peak value remains constant at 1. Using this property of a GMF we are able to steer the influence of a rule for a certain feature by $\sigma_i^{(h)}$.

Using this GMF as membership function a feature can be coded as a pair of real values $\gamma_i^{(h)}$ and $\sigma_i^{(h)}$ following the approach of Juang et al. [9].Using this feature description, a single rule's conditional part is composed as shown in Figure 3. For the consequence part, the general model in Equation 2, can be simplified as we have to deal with binary decisions only. Dependent on the current system state, the Fuzzy decision maker has to decide whether to accept an offered job or not. Thus, we represent the acceptance of a job by an output value of 1 and the corresponding refusal of a job by -1. With this binary decision concept, all weights except b_{i0} in Equation 2 are set to 0 and the TSK model output becomes $y_i = b_{i0}$. As we have to decide between the acceptance/decline of a job offer, the output values for a rule R_i can be chosen as

$$y_i = \begin{cases} 1, & \text{if job is accepted} \\ -1, & \text{otherwise} \end{cases} \tag{5}$$

This scheme allows the encoding of a single rule by a string of $2 \cdot N_f$ real-valued and one integer variable, see Figure 3. The whole rule base is encoded by concatenation of single rules. A whole rule base consisting of N_r rules is therefore

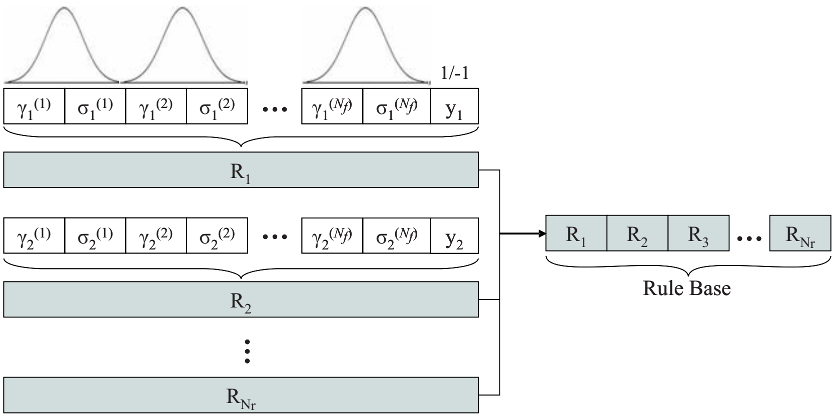

1/-1
γ1(1) σ1(1) γ1(2) σ1(2) ... γ1(Nf) σ1(Nf) y1
R1
γ2(1) σ2(1) γ2(2) σ2(2) ... γ2(Nf) σ2(Nf) y2
R2
RNr
R1 R2 R3 ... RNr
Rule Base

Fig. 3. Encoding pattern for single rules and construction concept for a whole rule base using concatenation

entirely described by a set of

$$l = N_r(2 \cdot N_f + 1) \tag{6}$$

parameters, see Equation 6. This encoding scheme is perfectly suited as individual representation within an evolutionary algorithm where individuals have the length l.

4.3 Computation of the Controller Decision

To determine the actual controller output for a set of input states $\boldsymbol{x}$ the superposition of all degrees of memberships for a single rule R_i is computed first. For each rule R_i a degree of membership $g_i^{(h)}(x_h)$ of the h-th of all N_f features is determined for all h. This value is computed as the function value of the h-th GMF for the given input feature value x_h. According to the general model, see Equation 3, the multiplicative superposition of all these values as "AND"-operation leads to an overall degree of membership $\phi_i(\boldsymbol{x})$ for rule R_i as shown in Equation 7.

$$\phi_i(\boldsymbol{x}) \quad = \quad \bigwedge_{h=1}^{N_f} g_i^{(h)}(x_h) = \prod_{h=1}^{N_f} \exp\left\{ -\frac{(x_h - \gamma_i^{(h)})^2}{\sigma_i^{(h)2}} \right\} \tag{7}$$

Further, the final controller output Y_D can be computed by considering the leading sign only, see Equation 8,

$$Y_D = \mathrm{sgn}(y_D(\boldsymbol{x})) \tag{8}$$

where a positive number again represents the acceptance of the job and a negative values the decline. Note that the value zero corresponds to a decline as well.

The TSK-model allows including an arbitrary number of features as controller input. Thus, it is possible to achieve a preferably accurate state description. However, this would increase the number of adjustable system parameters drastically as each feature requires an additional (γ, σ)-pair per rule. As the proposed Fuzzy system is going to be optimized with an evolutionary algorithm the number of system describing parameters must be kept as small as possible as every additional parameter increases the search space of the optimization problem and might deteriorate the solution quality. Thus, we restrict ourselves to only two features for the system state description and detail them in the next paragraph.

4.4 Feature Selection for System State Description

For the description of the current system state we rely on only $N_f = 2$ different features that will constitute the conditional part of a rule. We denote jobs that have been inserted into the waiting queue ν at site k as $j \in \nu_k$. In order to cover comprehensive system information with only a single feature we consider the Normalized Waiting Parallelism at site k (NWP_k) as the first feature, see Equation 9.

$$\text{NWP}_k = \frac{1}{m_k} \sum_{j \in \nu_k} m_j \tag{9}$$

This feature indicates how many processors are expected to be occupied by all submitted jobs (note that the number of requires processors m_j is known at release time) related to the maximum number of available processors m_k at site k. It reflects the efficiency of the currently running LRMS and measures the near future expected load of the machine.

The second features focuses on the actual job that has to be decided. The ratio of a job's resource requirements m_j and the maximum number of available resources m_k at the job's submission site k is expressed by the Normalized Job Parallelism (NJP), see Equation 10.

$$\text{NJP}_j = \frac{m_j}{m_k} \tag{10}$$

With those two selected features we approximate every possible system state.

4.5 Configuration of the Evolutionary Fuzzy System

Before we present the evaluation results the configuration of the Evolutionary Fuzzy System and the further evaluation circumstances are detailed. We generate our Evolutionary Fuzzy Systems with a fixed number of $N_r = 10$ rules. Previous studies of Franke et al. [5] revealed that rule bases consisting of five to ten rules yield good results. As we encode the whole rule base in one individual, we have to optimize a problem with $N_r \cdot (N_f \cdot 2 + 1) = 10 \cdot (2 \cdot 2 + 1) = 50$ parameters, see Equation 6.

For the tuning of the Fuzzy System we apply a $(\mu + \lambda)$-Evolution Strategy. During the run of 150 generation a continuous progress in fitness improvement is observable. As recommended by Schwefel [11], the ratio of $\mu/\lambda = 1/7$ should be used for Evolution Strategies. We created a parent population of $\mu = 13$ individuals which results in a children population of $\lambda = 91$ individuals. Hence, 91 individuals must be evaluated within each generation.

For the variation operators we used further the following configurations: The mutation is performed with an individual mutation step-size for each feature. As the two features vary in they possible value range by a ratio of 1:10, see Section 4.4, we used a mutation step-size of 0.01 for NWP and 0.1 for NJP respectively. This mutation is applied for the conditional part of the rule as they are real values. For the binary consequence part we mutate values by flips from -1 to 1 or vice versa. Further, we apply discrete recombination in each reproduction step.

The population is uniformly initialized within the ranges [0, 10] for the (γ, σ)-values of NWP and [0, 100] for NJP respectively. As the fitness evaluation of an individual is quite time consuming (from several minutes up to half an hour) we evaluated the whole population in parallel on a 200 node cluster with Pentium IV, 2.4Ghz machines.

5 Performance Evaluation

In order to evaluate the performance of our approach in the given scenario, we introduce the tools we use for assessing the optimized exchange policies against a realistic background. To this end, we define several well-known performance indicators for job scheduling in the context of Grid computing, both from the users' and the providers' point of view. Additionally, we discuss the workload traces we use as input data that are derived from real-world setups.

5.1 Average Weighted Response Time

This objective is computed for all jobs $j \in \tau_k$ that have been initially submitted to site k, see Equation 11. It is widely agreed that a short AWRT is the best way to describe that on average users do not wait long for their jobs to complete. Following Schwiegelshohn and Yahyapour [13], we use the resource consumption $(p_j \cdot m_j)$ of each job as weight. This ensures that neither splitting nor combination of jobs can influence the objective function in a beneficial way.

$$\mathrm{AWRT}_k = \frac{\sum\limits_{j \in \tau_k} p_j \cdot m_j \cdot (C_j(S) - r_j)}{\sum\limits_{j \in \tau_k} p_j \cdot m_j} \tag{11}$$

Note that this also respects the execution on remote sites and, as such, the completion time $C_j(S)$ refers to the site that executed job j.

5.2 Squashed Area and Utilization

The first two objectives are *Squashed Area* SA_k and *Utilization* U_k, both specific to a certain site k. They are measured from the start of the schedule S_k, that is $\min_{j \in \pi_k}\{C_j(S_k) - p_j\}$ as the earliest job start time, up to its makespan $C_{max,k} = \max_{j \in \pi_k}\{C_j(S_k)\}$, that is the latest job completion time and thus the schedule's length.

SA_k denotes the overall resource usage of all jobs that have been executed on site k, see Equation 12.

$$\mathrm{SA}_k = \sum_{j \in \pi_k} p_j \cdot m_j \tag{12}$$

U_k describes the ratio between overall resource usage and available resources after the completion of all jobs $j \in \pi_k$, see Equation 13.

$$\mathrm{U}_k = \frac{\mathrm{SA}_k}{m_k \cdot \left(C_{max,k} - \min\limits_{j \in \pi_k}\{C_j(S_k) - p_j\}\right)} \tag{13}$$

U_k describes the usage efficiency of the site's available machines. Therefore, it is often serving as a schedule quality metric from the site provider's point of view.

However, comparing single-site and multi-site utilization values is forbidden: since the calculation of U_k depends on $C_{max,k}$, valid comparisons are only admissible if $C_{max,k}$ is approximately equal between the single-site and multi-site scenario. Otherwise, high utilizations may indicate good usage efficiency, although the corresponding $C_{max,k}$ value is very small and shows that only few jobs have been computed locally while many have been delegated to other sites for remote execution.

As such, we additionally introduce the *Change of Squashed Area* $\Delta\mathrm{SA}_k$, which provides a makespan-independent view on the utilization's alteration, see Equation 14.

$$\Delta\mathrm{SA}_k = \frac{\mathrm{SA}_k}{\sum\limits_{j \in \tau_k} p_j \cdot m_j} \tag{14}$$

From the system provider's point of view this objective reflects the real change of the utilization when jobs are shared between site compared to the local execution.

5.3 Input Data

The Parallel Workloads Archive[2] provides job submission and execution traces recorded on real-world MPP system sites.Relevant details of the examined cleaned traces are given in Table 1.

Table 1. Workload characteristics of the used input data, including AWRT in seconds, U in %, and C_{max} in seconds for single site execution with FCFS

Identifier	#Jobs	m_k	AWRT	U	C_{max}
KTH-5	11780	100	488387.49	64.84	13765377
KTH-6	16699	100	99236.27	68.52	16420782
CTC-5	35360	430	57897.77	63.74	13009718
CTC-6	41839	430	59118.15	67.05	16346403
SDSC05-5	28184	1664	56925.10	45.94	13078215
SDSC05-6	46719	1664	77463.52	70.97	16419455
SDSC00-6	16316	128	413957.04	73.38	17002360

Naturally, the total number of available processors differs in workloads which makes it possible to model unequally sized site configurations. Further, the original workloads record time periods of different length. In order to be able to combine different workloads in a multi-site simulations and to have a validation set available, we shortened and separated the workloads to set of five and six month respectively. To reflect different site configurations (e.g. small machine and large machine) we combine only workloads that represent a long record period to obtain meaningful results. Therefore, we created shortened versions of the KTH, CTC, and SDSC05 workloads[3]. In the remainder of this work, the

[2] http://www.cs.huji.ac.il/labs/parallel/workload/

[3] http://www.it.irf.uni-dortmund.de/~lepping/traces/

five month sequence will serve as training sequences for the Evolutionary Fuzzy Systems. The six month sequences are then used for application tests. In this context, the SDSC00-6 trace will only be used to investigate the behavior of the trained system when an previously unknown partner participates in the system. Thus, we created no five month training sequence of the SDSC00 workload.

Further, we do not shift the traces regarding their originating timezones. We restrict our study to workloads which are all submitted within the same timezone. Therefore, the known diurnal rhythm of job submission is similar for all sites in our scenario and time shifts cannot be availed to improve scheduling. In a global grid the different timezones even benefit the job scheduling as idle machines at night can be used by jobs from peak loaded sites at noon, see Ernemann at al. [2]. As we cannot benefit from timezone shifts the presented results might be even better in a global grid.

We simulated the workload on their original machines with a LRMS that applied FCFS, see Section 3.1 for the local scheduling. The results for the above described performance metrics as well as other relevant job characteristics are listed in Table 1. We will refer to this non-cooperative case for the matter of comparison in the rest of this paper.

6 Learning a Basic Rule Set

So far, we presented our Fuzzy controller concept at the GRMS layer, explained how the complex decision making process can be adjusted by a set of parameters, and discussed metrics and test data for the system's performance evaluation. Now, we will introduce the learning process of rule sets for the Fuzzy system, detail the evolution-driven optimization procedure, and present corresponding results.

Starting with no rule set at all, we need to bootstrap the system: A first, basic set of rules has to be learned. Although it is generally necessary to create rule sets for both location and transfer policy, see Section 3.2, we start with a pair-wise training approach in order to reduce other partners' influences as much as possible. To this end, we limit job exchange to a single partner only—thus needing no location policy at all—and concentrate on the optimization of the transfer policy. Furthermore, we evolve only one site at a time while applying a static transfer policy on the other site. This policy realizes an *Accept When Fit (AWF)* behavior, which accepts all jobs offered to the GRMS layer for local execution if they do not require more than the currently free resources. Otherwise, the job is offered to the other participants.

The motivation for using a static transfer policy for the training partner and refraining from developing both sites together lies in the application of evolutionary optimization methods: A simultaneous training of two rule bases would lead to a mutual adaption of both training partners. This however, would result in an environment subject to continuous change, making an evolutionary-guided adaptation very difficult: A rule base that leads to good results during one generation might fail completely in the next generation if the partner site changes its behavior completely, too. The aspired robustness in changing environments will be achieved by additional refinements of the concept in the next sections.

6.1 Results for Training Sequences

The training results are listed in Table 2; gray-shaded lines indicate the evolved site while the other lines indicate the static site as described above. As expected, the optimization leads to significant improvements of the AWRT in all examined setups.

Table 2. Results for the pair-wise rule base training. The gray shaded rows indicated the optimized rule base.

Setup	Site	$AWRT_k$	U_k	$\Delta AWRT_k$	ΔU_k	ΔSA_k	$\Delta C_{max,k}$
I	KTH-5	66100.77 sec.	35.33%	86.47%	-45.51%	-48.56%	5.60%
	CTC-5	63534.29 sec.	71.40%	-9.74%	12.01%	12.15%	-0.14%
II	KTH-5	62884.33 sec.	73.00%	87.12%	12.58%	6.40%	5.49%
	CTC-5	54745.53 sec.	62.70%	5.44%	-1.64%	-1.60%	-0.05%
III	KTH-5	59409.60 sec.	45.15%	87.84%	-30.36%	-34.04%	5.28%
	SDSC05-5	58799.15 sec.	47.34%	-3.29%	3.06%	3.04%	0.01%
IV	KTH-5	70561.62 sec.	53.39%	85.55%	-17.66%	-21.75%	4.97%
	SDSC05-5	54773.39 sec.	46.85%	3.78%	1.98%	1.94%	0.02%
V	CTC-5	45997.73 sec.	58.55%	20.55%	-8.14%	-8.15%	0.01%
	SDSC05-5	57013.44 sec.	47.27%	-0.16%	2.90%	2.91%	-0.02%
VI	CTC-5	57069.59 sec.	63.30%	1.43%	-0.69%	-0.51%	-0.20%
	SDSC05-5	49916.01 sec.	46.04%	12.31%	0.22%	0.18%	0.02%

This results in larger AWRT for the partner site that does not adapt its behavior. For instance in Setup I, the AWRT improves by 86.47% compared to FCFS, see Table 1, while the AWRT for the CTC worsens for almost 10%. Note that this corresponds to a strong shift of work as the Squashed Area (ΔSA) is 48.56% lower for the KTH and approximately 12% higher on the CTC site. However, when the focus is changed, see Setup II, and CTC is optimized we achieve also improvements of 5.44% for AWRT and slight load relief for the CTC site. Besides that, the AWRT is still significantly improved in Setup II although we do not focus on the KTH. This is due to the worse performance in the non-cooperative case and indicates that Grid computing is for this site very advantageous.

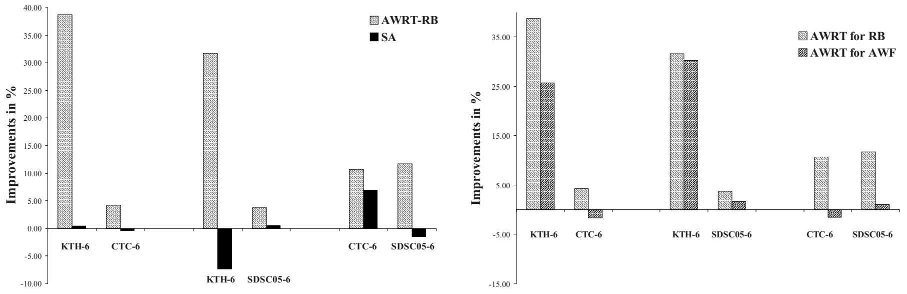
Furthermore, in Setup III and IV the small KTH interacts with the very large SDSC05 compute center and naturally the KTH benefits from more available resources. It is remarkable that also the SDSC05 can improve its AWRT for more than 3%, see Setup IV. At the same time, the Squashed Area is slightly increased which indicated that an improvement in AWRT is not necessarily caused by smaller utilization.

When the CTC interacts with a large compute center, see Setup V, the CTC also strongly benefits as its AWRT is decreased by more than 20%. Likewise, the SDSC05 can benefits from the cooperation with a medium size compute installation like the CTC, see Setup VI.

6.2 Robustness of Trained Rule Sets

To test the robustness of the pair-wise learned rule bases, we apply them to the 6 month workloads within the same setups. To this end, only two site Grids are considered and every partner applies its egoistically learned rule base. Note that AWF is not used in these scenarios anymore.

In Figure 4(a), the changes in AWRT and SA are depicted when both partners apply their learned rule bases to previously unknown job submissions.



(a) Application to non-trained data sets. (b) RB approach compared to AWF.

Fig. 4. AWRT and SA improvements for the optimized rule sets

Obviously, the Evolutionary Fuzzy Systems still decrease the AWRT significantly in all cases. This indicates a high robustness with respect to submission changes.

Further, we show in Figure 4(b) the AWRT improvements in comparison to the AWF transfer policy. Although AWF performs good for KTH and leads to slight improvements for SDSC05 it completely fails for the CTC workload trace. However, the rule based transfer policy outperforms AWF in all cases and leads even to shorter AWRT values for the SDSC05 together with the much smaller KTH.

7 Coping with More Than One Partner

After setting up the basic rule sets for job exchange in a controlled environment with a single partner, we now focus on the applicability of the rule bases in a Grid scenario with more participants. To this end, KTH, CTC, and SDSC05 are combined and the unknown submissions from the remaining six month of the traces are used. This time, however, a location policy needs to be applied in order to prioritize the options of delivering jobs to another participant.

7.1 An AWRT-Based Location Policy

In order to create a prioritization of the available potential delegation targets we follow a two-step approach. As first step, we generate the subset of sites that in total provide enough machines to execute the job. That is, we sort out all sites with $m_k < m_j \ \forall \ k \in K$.

As second step the generated subset is sorted according to their former achievement with respect to a delegation source. Good achievements can be measured by short AWRT of jobs on the corresponding sites. For the here proposed location policy, a site calculates the AWRT metric with respect to every exchange partner. To this end, only jobs are considered that have been delivered to the corresponding partner. The AWRT indicates how long the delegation source had to wait for the completion of its delivered jobs in the past. This metric is based on the assumption that a short AWRT for delivered jobs in the past is expected to yield also short AWRT values for future delegated job.

7.2 Results for Multiple Partners

The results in Figure 5(a) clearly indicate that the AWRT is still significantly improved for all sites while the utilization decreases for the small partner. However, although the CTC and SDSC05 are slightly more utilized it does again improve their objective values.

8 Coping with Alien Partners

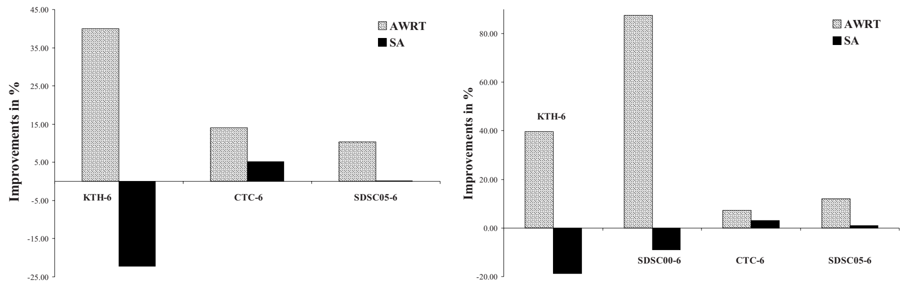
Until now, our learning approach was suited to generate a pool of rule bases for partners that are known in advance. This, however, requires knowledge about the submitted workload in order to tune the transfer policies. With respect to the robustness requirement, we therefore extend our approach to being able to perform well in an environment with previously unknown Grid participants. This requires the automatic adjusting of transfer behavior to partners that were not part of a training scenario.

8.1 Selection of Rule Base

As mentioned in Section 3.2, the rule based transfer policy is applied to each partner site separately. If a new partner arises a transfer policy has to be selected from the pool of all learned transfer policies. To identify the best suitable transfer policy we assume a correlation between delegation targets' maximum amount of available resources and their transfer behavior. We conjecture that the behavior within the grid mainly depends on a site's resource number. Thus, we categorize the various trained rule bases by the machines sizes they belong to. Among the whole trained pool of transfer policies the best fitting one, with respect to the number of maximum available resources, is selected to make the decision for a submitted job.

8.2 Results for Alien Partners

Finally, we investigate the performance of the rule base selection concept and add the SDSC00 as a site with $m_k = 128$ processors to the Grid. Following the rule base selection concept, every site uses the KTH learned rule base for the interaction with SDSC00 as it has the greatest similarity with respect to the machine size. The SDSC00 site, in turn, uses AWF for exchange purpose.



(a) Three-site Grid scenario (b) Four-site Grid scenario

Fig. 5. AWRT and SA improvements for the optimized rules in a three-site Grid scenario on non-trained data sets (a) and in a four-site Grid scenario with SDSC00 as unknown participant (b)

In Figure 5(b) the results for interaction with three other partners are depicted and, again, we observe strong AWRT improvements. Similarly to the KTH, also SDSC00 shows a poor performance for exclusive single site execution. Therefore, there is a high potential to improve the AWRT. However, it is important to see that not only this partner can improve its AWRT but also other participants are able to improve their AWRT for at least 10%.

Summarizing, the learned Evolutionary Fuzzy Systems realize a beneficial job exchange for several cooperative computing environments. The examined Grid sizes range from two to four sites and include unknown job submissions as well as previously unknown Grid participant. In all cases, the AWRT can be significantly decreased which results in improvements of about 10%-20% for large sites and 40%-80% for larger sites. It has been shown, that the job exchange policies show a strong robustness with respect to both new sort of job submissions and environmental changes.

9 Conclusion and Future Work

We presented an Evolutionary Fuzzy System approach to finding non-invasive, situation-adaptive, and robust algorithms for workload distribution in decentralized Computational Grids. Such environments assume full autonomy of the participating HPC/HTC centers and strict confidentiality of dynamic system information and demand Grid middlewares that do not interfere with the running LRMS.

In our model, we introduced a decoupled GRMS layer on top of the available systems, which decides upon execution on the local system or delegation to a remote site for user-submitted jobs in an online, non-clairvoyant manner. The decision mechanism is established by using a Fuzzy controller system with flexible rule sets that are optimized using evolutionary computation, using a pair-wise training approach and performance metric-based rule base selection.

The presented system shows that—using real-world data—it is possible to establish job exchange policies which lead to significantly improved performance for all user communities in terms of response time and utilization. We further find that our approach behaves robustly with respect to fluctuations in the workload pattern and shows situational adaptiveness even under circumstances of unknown submission characteristics. Overall, we think that the derived controllers provide a stable basis for workload distribution and interchange in Computational Grids, and may qualify as a promising technology for future Service Grid-based e-Science infrastructures.

References

1. Cordón, O., Herrera, F., Hoffmann, F., Magdalena, L.: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases. In: Genetic Fuzzy Systems. Advances in Fuzzy Systems - Applications and Theory, vol. 19. World Scientific, Singapore (2001)
2. Ernemann, C., Hamscher, V., Yahyapour, R.: Benefits of global grid computing for job scheduling. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID 2004), pp. 374–379. IEEE Computer Society, Los Alamitos (2004)
3. Feitelson, D.G., Nitzberg, B.: Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 337–360. Springer, Heidelberg (1995)
4. Franke, C., Hoffmann, F., Lepping, J., Schwiegelshohn, U.: Development of Scheduling Strategies with Genetic Fuzzy Systems. Applied Soft Computing 8(1), 706–721 (2008)
5. Franke, C., Lepping, J., Schwiegelshohn, U.: Genetic Fuzzy Systems applied to Online Job Scheduling. In: Proceedings of the 2007 IEEE International Conference on Fuzzy Systems, London, June 2007, pp. 1573–1578. IEEE Press, Los Alamitos (2007)
6. Gagliardi, F., Jones, B., Grey, F., Begin, M.-E., Heikkurinen, M.: Building an infrastructure for scientific grid computing: status and goals of the egee project. Philosophical transactions. Series A, Mathematical, physical, and engineering sciences 363(1833), 1729–1742 (2005)
7. Grimme, C., Lepping, J., Papaspyrou, A.: Prospects of Collaboration between Compute Providers by means of Job Interchange. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 132–151. Springer, Heidelberg (2008)
8. Grimme, C., Lepping, J., Papaspyrou, A.: Discovering Performance Bounds for Grid Scheduling by using Evolutionary Multiobjective Optimization. In: Keijzer, M., et al. (eds.) Prococeedings of the Genetic and Evolutionary Computation Conference (GECCO 2008), Atlanta, Georgia, USA, July 2008, pp. 1491–1498. ACM Press, New York (2008)
9. Juang, C.-F., Lin, J.-Y., Lin, C.-T.: Genetic Reinforcement Learning through Symbiotic Evolution for Fuzzy Controller Design. IEEE Transactions on System, Man and Cybernetics 30(2), 290–302 (2000)

10. Marinescu, D.C., Boloni, L., Hao, R., Jun, K.K.: An alternative model for scheduling on a computational grid. In: Proceedings of ISCIS 1998, the Thirteenth International Symposium on Computer and Information Sciences, Antalya, pp. 473–480. IOP Press, Amsterdam (1998)
11. Schwefel, H.-P.: Evolution and Optimum Seeking. John Wiley & Sons, New York (1995)
12. Schwiegelshohn, U., Tchernykh, A., Yahyapour, R.: Online scheduling in grids. In: 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008). IEEE Press, Los Alamitos (2008)
13. Schwiegelshohn, U., Yahyapour, R.: Fairness in parallel job scheduling. Journal of Scheduling 3(5), 297–320 (2000)
14. Takagi, T., Sugeno, M.: Fuzzy Identification of Systems and Its Applications to Modeling and Control. IEEE Transactions on Systems, Man, and Cybernetics, SMC 15(1), 116–132 (1985)

Analyzing the EGEE Production Grid Workload: Application to Jobs Submission Optimization

Diane Lingrand[1], Johan Montagnat[1], Janusz Martyniak[2], and David Colling[2]

[1] University of Nice - Sophia Antipolis / CNRS - France
{lingrand,johan}@i3s.unice.fr
http://www.i3s.unice.fr/~lingrand/
[2] Imperial College London, The Blackett Lab - UK
{janusz.martyniak,d.colling}@imperial.ac.uk

Abstract. Grids reliability remains an order of magnitude below clusters on production infrastructures. This work is aims at improving grid application performances by improving the job submission system. A stochastic model, capturing the behavior of a complex grid workload management system is proposed. To instantiate the model, detailed statistics are extracted from dense grid activity traces. The model is exploited in a simple job resubmission strategy. It provides quantitative inputs to improve job submission performance and it enables quantifying the impact of faults and outliers on grid operations.

1 Introduction

In response to the growing consumption of computing resources and the need for global interoperability in many scientific disciplines, inter-continental production grid infrastructures have been deployed over recent years. Grids are understood here as the federation of many regular computing units distributed world-wide, taking advantage of high-bandwidth Internet connectivity. Productions grids are systems exploiting dedicated resources administrated and operated 24/7, as opposed to desktop grids that federate more volatile individual resources. The production systems operated today (*e.g.* EGEE[1], OSG[2], NAREGI[3]...) have emerged as a global extension of institutional clusters. They federate computing centers which operate pools of resources almost autonomously. The grid middleware is designed to sit on top of heterogeneous, existing local infrastructures (typically, pools of computing units interconnected through a LAN and shared through batch systems) and to adapt to different operation policies.

These complex systems have passed feasibility tests and are exploited as the backbone of many research and industrial projects today. They provide users with an unprecedented scale environment for harnessing heavy computation tasks and building large collaborations. Their exploitation has led to new distributed computational models. However, they also introduce a range of new problems directly

[1] Enabling Grids for E-sciencE, http://www.eu-egee.org
[2] Open Science Grid, http://www.opensciencegrid.org
[3] NAREGI, http://www.naregi.org

related to their scale and complex software stacks: high variability of data transfer and computation, heterogeneity of resources, many opportunity for faults, hardware failures, difficulty with bug tracking, etc. The workload management system of grid infrastructures is probably one of the most critical and most studied service provided. Despite the tremendous efforts invested in guaranteeing reliable and performing workload managers, the current records demonstrate that grid reliability remains an order of magnitude lower than clusters reliability, and performances may be disappointing when compared to the promise of virtually unlimited resources aggregation. As a consequence, grid users are directly exposed to system limitations. They adopt empirical application level strategies to cope with the problems most commonly encountered.

Production grid infrastructures remain to a large extent complex systems with behavior that is little understood and for which "optimization" strategies are often empirically designed. The reason for this cannot be attributed to the youth of grid systems alone. The complexity of software stacks, the split of resources over different administrative domains and the distribution at a very large scale makes it particularly difficult to model and comprehend grid operations. Structured investigation techniques are needed to analyze grids behavior and optimize grid performances. Considering the grid workload management systems in particular, users are often in charge of manually resubmitting jobs that failed. They need assistance to adopt smart resubmission strategies that improve performance according to global criteria.

1.1 Objectives and Organization

In this paper we analyze the operation of the EGEE production grid infrastructure and more particularly its Workload Management System (WMS) in order to assist users in performing jobs submission reliably and improving application performance. Experience shows that EGEE users are facing a significant ratio of faults when using the WMS and their applications' performance is impacted by very variable latencies. Each job submitted to the grid may succeed, fail, or become an outlier (*i.e.* get lost due to some system fault). The execution time of successful jobs is impacted by the system latency. Faulty jobs and outliers are similarly introducing variable delays before the error is detected and the jobs can be resubmitted. From the user point of view, the overall waiting time, including all necessary resubmissions, should be minimized. Ad-hoc fault detection and resubmission strategies are typically implemented on a per-application basis. Determining the optimal grace delay before resubmission is difficult though, due to the absence of notification of outliers and the impact of faults. The objective of this study is to provide quantitative input and optimal resubmission timing.

Previous works have demonstrated that statistics collection on the live grid system and derived probabilistic models could help in optimizing grid performance according to user-oriented and system-oriented criterions [1,2]. However, the statistics utilized so far were collected through invasive probing of the grid infrastructure, thus leading to rather sparse and incomplete data retrieval, difficult to update although grid workload is non-stationarity. This work describes a more structured approach leveraging on the international effort to set up a *Grid Observatory*[4] which tackles the problems related to grid operation traces collection in order to provide accurate, dense and relevant statistics for modeling and optimizing the infrastructure.

In the remainder, the EGEE grid architecture, and more specifically its Workload Management System, is introduced. The Grid Observatory implementation, based on grid service log files analysis and merging, is then described. The data extracted and its exploitation for deriving a novel probabilistic model of the grid job latencies is presented. Finally, a simple job resubmission strategy is optimized, based on the probabilistic model proposed.

1.2 Related Work

Frachtenberg and Schwiegelshohn [3] have pointed that in case of failure, rescheduling is needed in order to reduce submission cost. They also pointed out that very few real production grid workload traces and models are available. A few local pieces of work have been done however, such as on the Auvergne regional part of EGEE by Medernach [4]. An initiative of data publication and organization is the Grid Workloads Archive [5] which proposes a workload data exchange format and associated analysis tools in order to share real workload data from different grid environments.

Early work such as [6] have set up a methodology of statistical workload modeling from real data with the characteristics observed on Grids: heavy tailed distribution and rare events. More recent works have proposed to model different parameters such as job inter-arrival time, job delays, job size and their correlation on different platforms: the EGEE grid [7,8] or the Dutch DAS-2 multi-cluster environment [9] for different periods of time from 1 month to 1 year.

Real workload models are mandatory to test new algorithms at different stages of jobs life-cycle such as submission (client side) or scheduling (middleware side). Authors of [8] have used their data to compare two user-level schedulers algorithms.

Workload models are also used for platform analysis and comparison. For example, authors of [7] have compared their results on the EGEE Grid with a real local cluster and an ideal cluster.

Finally, workload models will also enable more realistic simulations when used in Grid simulator such as SimGrid [10].

2 EGEE Grid Infrastructure

EGEE is an unprecedented large scale federation of computing centers, each operating internal clusters in batch mode. EGEE today accounts for more than 80,000 CPU cores distributed in greater than 250 computing centers of various sizes. With more than 9,000 users authorized to access the infrastructure and

[4] Grid Observatory, http://www.grid-observatory.org

more than 200,000 computing tasks handled daily, EGEE experiences very variable load conditions and strong latencies in user requests processing, mostly due to the middleware latency and the batch queuing time of requests.

EGEE operates the gLite middleware[5]. gLite is a collection of interoperating services that cover all functionality provided, including grid-wide security, information collection, data management, workload management, logging and bookkeeping, etc. A typical gLite deployment involves many hosts distributed over and communicating through the WAN. The main services provided by gLite are: the security foundational layer (based on GLOBUS Toolkit 2), the Information System collecting status information on the platform hierarchically, the Data Management System providing a unified view of files distributed over many sites, and the Workload Management System (WMS) in charge of dispatching and monitoring computing tasks. Each of these systems is a compound, distributed architecture in its own right.

EGEE is a multi-sciences grid and EGEE users and resource are grouped into Virtual Organizations (VOs) which define both communities of users sharing a common goal and an authorization delineation of the resources accessible to each user group.

2.1 EGEE Workload Management System

The EGEE WMS is seen from the user as a two-level batch system: the User Interface (client) connects to a Workload Manager System (WMS). The WMS is interfaced to the grid Information System to obtain indications on the grid sites status and workload conditions. It queues user requests and dispatches them to one of the sites connected. The sites receive grid jobs through a gateway known as Computing Element (CE). Jobs are then handled through the sites' local batch systems. To comprehend the complexity of the system, a more complete view of the WMS architecture, extracted from the WMS user guide [11] is depicted in figure 1.

When submitting a job, the client User Interface connects to the core Workload Manager through a WMProxy Web Service interface on the Network Server. The Workload Manager queries the Resource Broker and its Information SuperMarket (repository of resource information) to determine the target site that will handle the computation task, taking into account the job specific requirements. It then finalizes the job submission through the Job Adapter and delegates the job processing to CondorC. The job evolution will be monitored by the Log Monitor (LM) which intercepts interesting events (affecting the job state machine) from the CondorC log file. Finally, the Logging and Bookkeeping service (LB) logs job events information and keeps a state machine view of the job life cycle. The user can later on query the LB to receive information on her job evolution.

For load balancing and system scalability, the EGEE infrastructure operates around a hundred of similar WMS. However, if those WMSs share the same population of connected CEs, they are not interconnected and they do not perform

[5] gLite middleware, `http://www.glite.org`

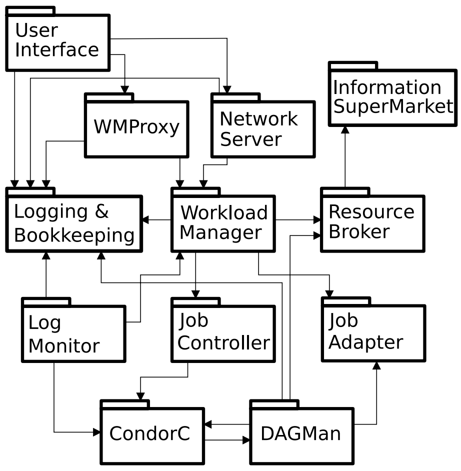


Fig. 1. gLite Workload Management System architecture; source: WMS user guide

internal load balancing. It is up to the clients to select their WMS at submission time. The client User Interface implements a simple round-robin WMS selection policy to assist users in their job submission process.

In the remainder we are particularly interested in the impact of the grid middleware on the job execution time, *i.e.* the *latency* induced by the middleware operation, that is not related to the job execution itself. This latency is a measure of the middleware overhead. In case of faults (scheduling problems, middleware faults...) this latency will arbitrarily increase and the job needs to be considered lost after a long waiting time enough to prevent application blocking.

2.2 Job's Life Cycle

The job's life cycle is internally controlled through a state machine displayed in figure 2 (source: WMS user guide).

The normal states assigned to a job are underlined in boxes with thick borders (they correspond to the case of a job completed successfully):

SUBMITTED the job was received by the WMS and the submission event is logged in the LB
WAIT the job was accepted by WM, waiting to match a CE
READY the job is sent to its execution CE
SCHEDULED the job is queued in the CE batch manager
RUNNING the job executes on a worker node of the target site
DONE (ok) the job completed successfully.

Other states may also be encountered:

ABORT in any state, the middleware can abort the operation. An additional status reason is usually returned.

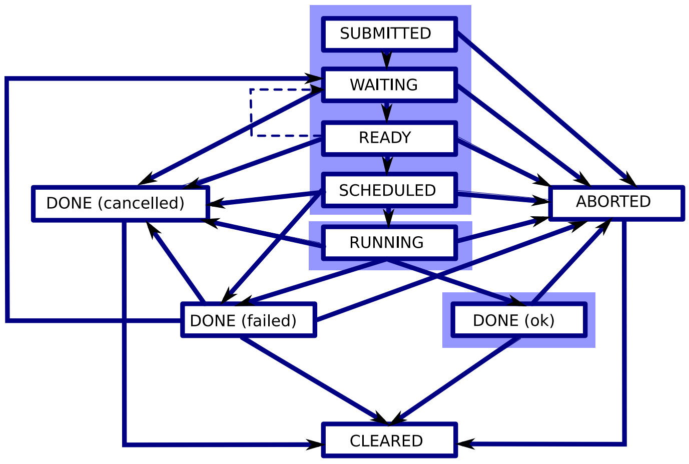


Fig. 2. Jobs life cycle state machine; source: WMS user guide

DONE (failed) some errors may prevent correct job completion. An additional status reason is usually returned.

DONE (cancelled) in any state, the submission user can cancel her job.

CLEARED after outputs of a completed job have been retrieved by the user, the job is cleared.

2.3 Grid Observatory

The basis of our work on the WMS behavior modeling is the collection of statistical information on job evolution on the live grid infrastructure. The relevant information for performance modeling is the duration of jobs, including fine details on the intermediate times spent between transitions of the state diagram. This information collection step is difficult by itself. In a previous work [12], we collected such information through poll jobs submission on the infrastructure and monitoring of the polls life-cycle. Although this strategy is easy to implement (all that is needed is a user interface connected to the infrastructure), it is both restrictive (the polls are specific short duration jobs, the jobs are limited to the resources accessible to the specific user performing submission) and has limited accuracy (only a limited number of polls can be simultaneously submitted to avoid disturbing normal operation, the traces are collected by periodic polling and the period selected impacts the accuracy of results).

A more satisfying approach is to collect traces from regular jobs submitted on the grid infrastructure during normal grid operation, thus assembling a complete and accurate corpus of data. However, there are much more difficulties in implementing this approach than would be expected, including:

– traces are recorded by different inter-dependent services (WM, CondorC, LM, LB...) that are tracing partly redundant and partly complementary information;
– traces are collected on many different sites (operating different WMSs) administrated independently: agreement to collect the data has to be negotiated with the (many) different site administrators;
– different versions of the middleware services co-exist on the infrastructure and traces are produced by slightly varying sources (including changes in states, labels, spell fixing in messages returned, etc);
– traces are recorded on different computers which clocks are not always well synchronized (although NTP *should* be installed on every grid hosts);
– traces collected are incomplete as parts of them can be lost (log files loss, disk crashes, etc) and all job states are not always recorded (middleware latency and faults cause some transition losses);
– as it will appear in the rest of this paper, the traces recorded often do not match precisely the information documented in the existing guides (state name changes, etc).

The most accurate source of traces available on the EGEE grid today is the Real Time Monitor[6] (RTM) implemented at the Imperial College London for the need of real time grid activity monitoring and visualization. The RTM gathers information from EGEE sites hosting Logging and Bookkeeping (LB) services. Conforming to our college policy, information is cached locally at a dedicated server at Imperial College London and made available for clients to use in near real time.

The system consists of 3 main components: the RTM server, enquirer and an apache Web Server which is queried by clients. The RTM server queries the LB servers at fixed time intervals, collecting job related information and storing this in a local database. Job data stored in the RTM database is read by the enquirer every minute and converted to an XML format which is stored on the Web Server. This decouples the RTM server database from potentially many clients which could bottleneck the database.

The RTM also provides job summary files for every job as text files ("Raw Data"). These data are analysed off-line and fixed record length tuples are created on daily basis, one file per LB server. These files are used for the analysis presented in this paper.

The systematic collection of grid traces for studying grid systems has been recognized as a key issue and significant effort has been recently invested in setting up a *Grid Observatory*[7] which aims at collecting information and easing access to it through a portal. The grid observatory has long term objectives of curating and harmonizing the data. It currently provides access to first data corpuses collected in Paris regional area (GRIF) and by the RTM.

[6] Real Time Monitor, `http://gridportal.hep.ph.ic.ac.uk/rtm`
[7] EGEE Grid Observatory, `http://www.grid-observatory.org`

3 Statistical Data

The data considered in this study are RTM traces of the EGEE grid activity during the period from September 2005 to June 2007. 33,419,946 job entries were collected, each of them representing a complete job run. Among the information recorded in an entry can be found: the job ID, the resources used (UI, RB, CE, WN), the VO used, the job specific requirements, the job life cycle concatenated field and a complementary "final reason" text detailing the reason for the final state reached. Different epoch times are given, allowing to measure the duration of each step of the job life cycle:

epoch_regjob_ui: registration of a job on a User Interface
epoch_accepted_ns: job accepted by the network server
epoch_matched_wm: job matched to a target CE
epoch_transfer_jc: job accepted and being transfered to the CE
epoch_accepted_lm: job accepted by the CE
epoch_running_lm: job started running (logged by the LM)
epoch_done_lm: job completed (successfully or not)
epoch_running_lrms: job started running (logged by the LRMS)
epoch_done_lrms: job completed (successfully or not)

The last two couples of epoch data can be redundant: one is given by the LM while the other is given by the local resource management system (LRMS) or batch system. The LM data is less accurate than the LRMS, but the LRMS data does not exist for all CEs.

The `life cycle` field holds information on the different states the job has encountered during its life cycle (see figure 2). It is composed of the concatenation of the different state names, considering some minor variations in names (e.g. RAN corresponds to a past RUNNING state; REGISTERED corresponds to a job registered on the UI it has been SUBMITTED to). In the data considered in this paper, 77 different values of the `life cycle` field have occurred with different frequency. They correspond to different situations: job successfully terminated and data retrieved (REGISTERED_DONE_RAN_CLEARED), job aborted (REGISTERED_ABORT), etc. The top 5 life cycle values with their frequencies are given in table 1.

To give more information on the reason for the final state of a job (especially in case of error), the `final_reason` field provides a user readable message. Unfortunately, the set of possible values is larger, due to the diversity of cases that may occur but also to the different versions of middlewares, sometimes displaying

Table 1. Top 5 life cycle values with their frequencies

REGISTERED_DONE_RAN_CLEARED	41.3 %
REGISTERED_ABORT	29.4 %
REGISTERED_DONE_RAN	22.8 %
REGISTERED_DONE	2.7 %
REGISTERED_ABORT_RAN	0.9 %

Table 2. Abbreviations for some type values

RDRC	REGISTERED-DONE-RAN-CLEARED
RDR	REGISTERED-DONE-RAN
RA	REGISTERED-ABORT
RAC	REGISTERED-ABORT-CLEARED
RD	REGISTERED-DONE
UA	UNDEFINED-ABORT
Una	UNDEFINED-na
RE	REGISTERED-ENQUEUED
RnaR	REGISTERED-na-RAN
UDR	UNDEFINED-DONE-RAN
UDRC	UNDEFINED-DONE-RAN-CLEARED
RRR	REGISTERED-RUNNING-RAN
RT	REGISTERED-TRANSFER

different messages for the same reason. Combined with the `life cycle` field, we counted 315 different cases. Some `final_reason` fields were shortened to exclude non relevant specific information such as particular file name or site name appearing in the message. For instance, the text `"cannot retrieve previous matches for https://lcgrb01.gridpp.rl.ac.uk:9000/XCKb1dsA3fXbzsY7Q"` was replaced by `"cannot retrieve previous matches for"`.

Before exploiting the data, some curation was needed for proper interpretation. Namely: data sources were selected when redundant information was available (LM and LRMS traces redundancy); specific text `final_reason` fields were truncated; and rare events were neglected in order to reduce the number of cases to analyze (an experimental justification if given in paragraph 5.3). As a result, table 3 details the 37 most frequent cases, representing 99.4% of the total data. This selection is a compromise between data completeness and number of cases to analyze. The last column of table 3 proposes a classification of the cases into 3 classes that are detailed below.

3.1 Successful Jobs

The first class corresponds to jobs that have started running and either terminated successfully or were canceled by the user. We consider that these jobs were possibly successful job even if the intervention of the user changed the final status or if some produced files were not retrieved nor used. For these jobs, we denote by R the job latency, *i.e.* the time between the epoch of registration on the UI and the epoch where the job starts running. Due to some clock synchronization problems it may happen that a latency value R is negative: such entries have been excluded from the study. Of course, such problem may also alter some positive values. However, these events are rare and the synchronisation difference are small compared to the values considered.

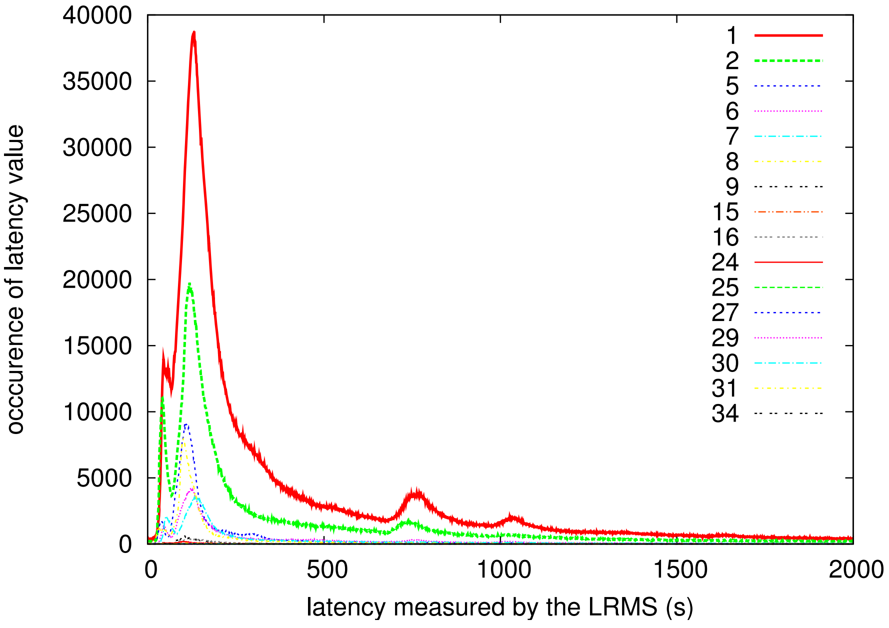
It also happens that either LM or LRMS traces are available. Since LRMS values are more accurate, we decided to keep only data where LRMS values were

Table 3. The 37 most frequent cases of type and final reason field values are totalizing 99.4% of the total data. Type values have been abbreviated for readability of the table, using table 2. The last column distinguishes correctly running jobs with latency (R with number of data entries remaining after cleaning), failed jobs (F) and outliers.

case	type and final_reason	occurrences	%	class
1	RDRC Job terminated successfully	11,563,331	34.6%	R (9,999,928)
2	RDR Job terminated successfully	5,639,638	16.9%	R (5,035,776)
3	RA Job RetryCount (0) hit	3,838,380	11.5%	outlier
4	RA Cannot plan: BrokerHelper: no compa	3,422,319	10.2%	F
5	RDRC -	1,299,235	3.89%	R (1,138,324)
6	RDRC There were some warnings: some file	1,004,800	3.01%	R (884,932)
7	RDR There were some warnings: some file	911,500	2.73%	R (813,405)
8	RDR -	877,229	2.62%	R (750,473)
9	RD Aborted by user	863,094	2.58%	R (875,89)
10	RA Job RetryCount (3) hit	582,152	1.74%	outlier
11	RA -	557,055	1.67%	F
12	RA Job proxy is expired.	495,519	1.48%	F
13	RA cannot retrieve previous matches fo	322,839	0.97%	F
14	RAR Job proxy is expired.	267,890	0.80%	F
15	Una -	235,458	0.70%	R (10,632)
16	RDR Aborted by user	188,421	0.56%	R (15,3479)
17	RA Job RetryCount (1) hit	165,231	0.49%	outlier
18	UA Error during proxy renewal registra	149,095	0.45%	F
19	RA Unable to receive	115,867	0.35%	F
20	RE -	109,089	0.33%	F
21	RA Cannot plan: BrokerHelper: Problems	89,553	0.27%	F
22	UA Unable to receive	70,215	0.21%	F
23	RA Job RetryCount (2) hit	63,595	0.19%	outlier
24	RnaR -	56,044	0.17%	R (53,055)
25	RD -	45,400	0.14%	R (2,091)
26	RAC cannot retrieve previous matches fo	35,887	0.11%	F
27	UDR Job terminated successfully	31,722	0.09%	R (236)
28	RAR -	26,268	0.08%	F
29	RDRC There were some warnings: some outp	25,868	0.08%	R (20,341)
30	RDR There were some warnings: some outp	23,178	0.07%	R (18,315)
31	RRR -	22,983	0.07%	R (19561)
32	RA Submission to condor failed.	22341	0.07%	F
33	RA Job RetryCount (5) hit	22,260	0.07%	outlier
34	RT Job successfully submitted to Globu	18,972	0.06%	R (3,768)
35	RT unavailable	18,065	0.05%	F
36	RA Job RetryCount (7) hit	17,328	0.05%	outlier
37	RA hit job shallow retry count (0)	16,863	0.05%	outlier

available. The number of remaining data is given for each case in table 3 inside the parenthesis after the *R* symbol. This class is composed of 18,991,905 entries.

Figure 3 displays the distribution of latency values for all successful cases from table 3. We observe that all profiles are similar although the frequencies differ,



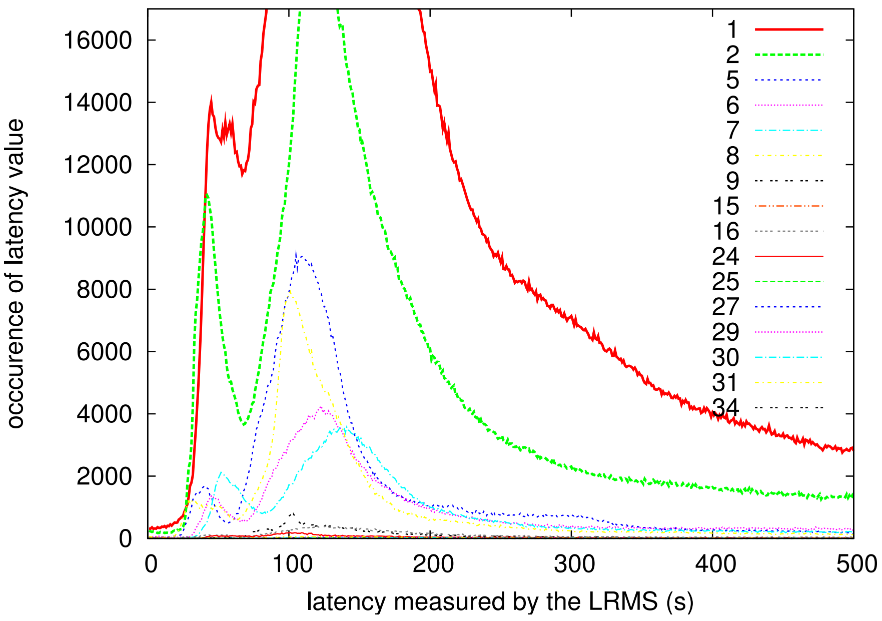


Fig. 3. Occurrences of latency values for different cases (see table 3) of successful jobs. The figure below gives more details for low values. The first two cases (1 and 2) are plotted thicker for an easier reading.

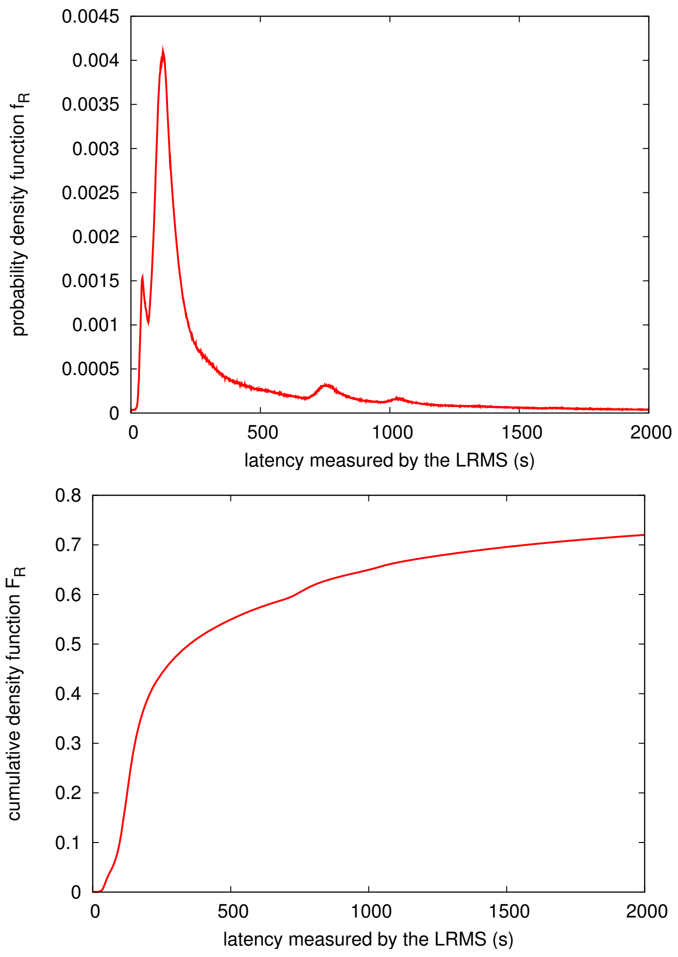


Fig. 4. cdf (top) and pdf (bottom) of the latency in all the cases displayed in figure 3

and the first class represents most of the data. Figure 4 displays the probability density function (pdf) of the latency on top and its cumulative density function (cdf) on bottom. These laws are known to be heavy tailed [13] meaning that the tail is not exponentially bounded (see figure 5):

$$\forall \lambda > 0, \lim_{t \to \infty} e^{\lambda t}(1 - F_R(t)) = +\infty$$

3.2 Failed Jobs

The second class corresponds to jobs that have failed for different reasons, leading to abortion by the WMS (no compatible resources, proxy error, BrokerHelper problem, CondorC submission failure...). Most jobs are aborted after a delay,

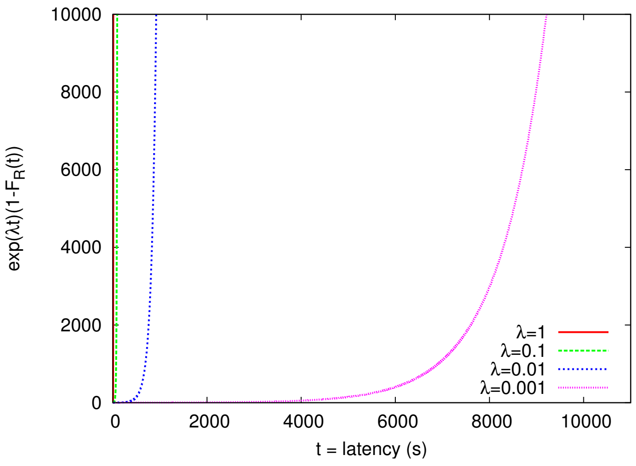


Fig. 5. Product $e^{\lambda t}(1 - F_R(t))$ for different values of λ. This illustrates that the distribution of latency is heavy tailed.

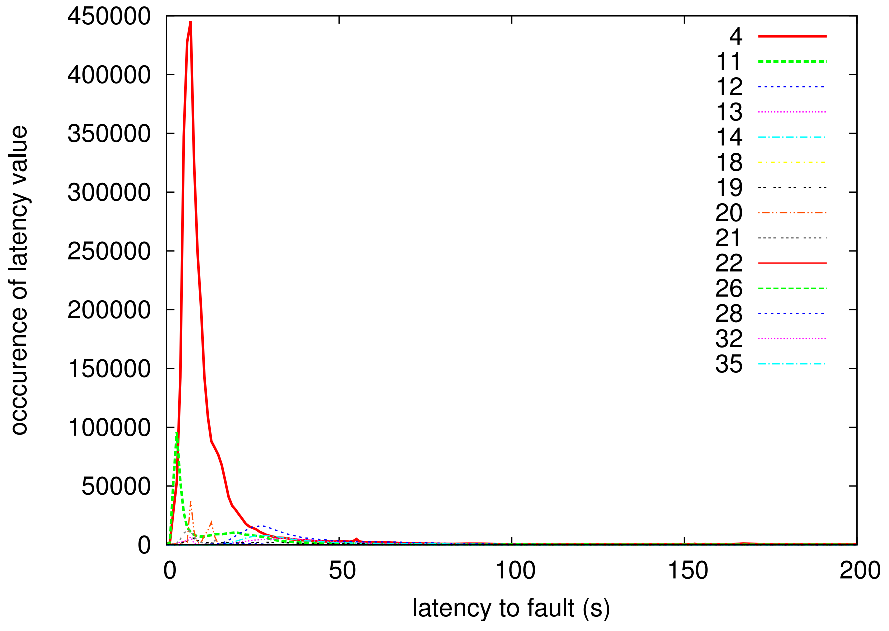
denoted by the variable F, computed from the epoch of job registration until the done state epoch corresponding to the abortion instant. The delay F is one of the subjects of this study. Similarly to the previous class, some synchronization clock problems led to exclude some data. Moreover, the terminal "done" status may not be reached in some cases, as for example 18 and 22. We have decided to assume that the fault was immediately reported to the system in these cases. This class is finally composed of 5,607,329 entries.

The different fault latency profiles (F) for the different cases of table 3 labelled as faults are displayed in figure 6. Contrarily to the study of successful jobs, we observe that the profiles of the curves corresponding to each case conducting to fault are quite different. The corresponding pdf and cdf of F are plotted in figure 7. They corresponds approximately to the profile of case number 4 even if they have been computed on all failed jobs: case number 4 is predominant (10.2% of entries compared to second larger, case number 11 with 1.67% of entries).

3.3 Outliers

Jobs with type "REGISTERED-ABORT" and final reason "Job RetryCount (any number) hit" are jobs that have failed at least once at a site and been submitted to other sites until the user defined maximum number of retries is reached at which point the WMS gives up on the jobs. The WMS is aware of such failures either because it is notified of the job failure or because the job times out.

The final reason for a large part of these jobs is known after a very long delay (few 100000s seconds) when compared to other failed jobs. They correspond to jobs that never return due to some middleware failure or network interruption (jobs may have been sent to a CE that has been disconnected or crashed and the



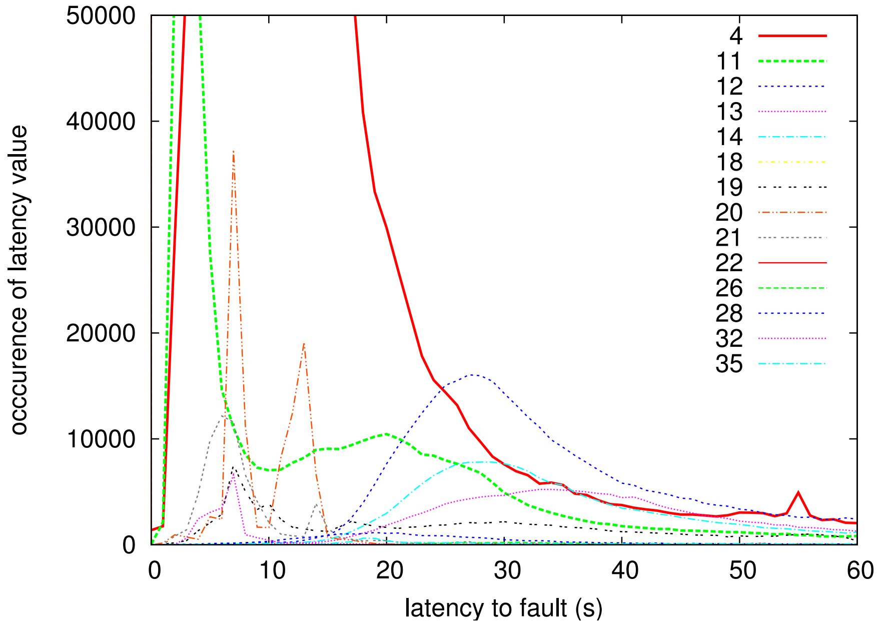


Fig. 6. Occurrences of latency to fault values for different cases (see table 3) and detail for low values. The first two cases (4 and 11) are plotted thicker for an easier reading.

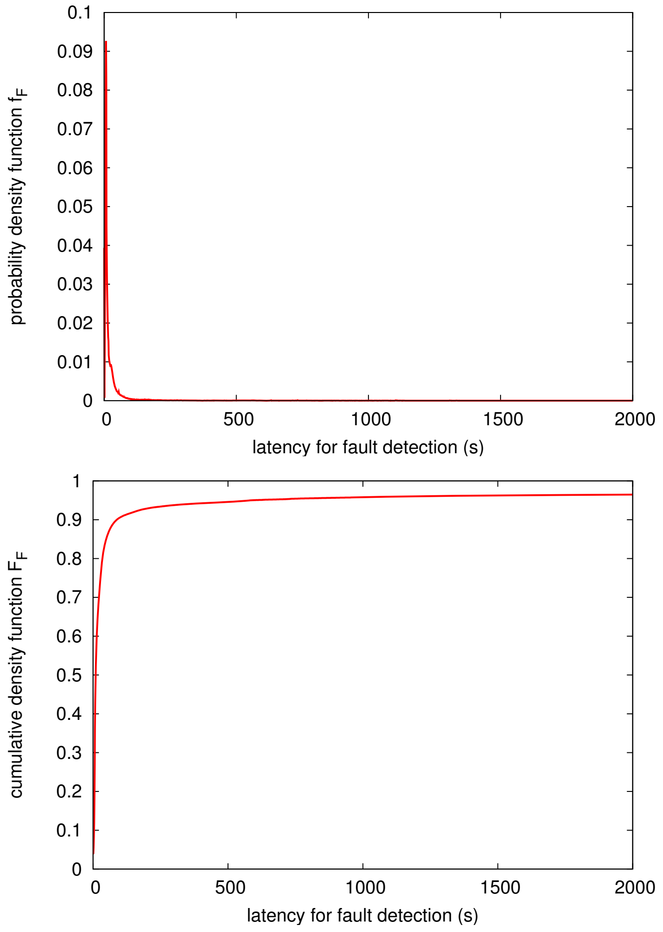


Fig. 7. pdf (top) and cdf (bottom) of the latency for fault detection in all the cases examined in figure 6

LB will never receive notification of the completion). They are usually detected using a timeout value by the WMS. This last class of jobs, labelled as outliers, contains 4,705,809 entries.

3.4 Summary

We denote as ρ the ratio of outliers and ϕ the ratio of faulty jobs. In the complete data set considered, we measure:

$$\begin{array}{lr} \text{outliers} : & \rho = 16.1\% \\ \text{faults} : & \phi = 19.1\% \\ \text{successful} : & 1 - \rho - \phi = 64.8\% \end{array}$$

When comparing the distribution of F to the one of R, we observe that, even if faults are not always known immediately, they are usually identified in a shorter time than the latency impacting most successful jobs. We will now study the impact of the delay before faults detection on the total latency of a job.

4 Resubmission after Fault

4.1 Probabilistic Modeling

In the remainder, a capital letter X traditionally denotes a random variable with pdf f_X and cdf F_X. Let R denote the latency of a successful job and F denote the failure detection time. Assuming that faulty jobs are resubmitted without delay, let L denote the job latency taking into account the necessary resubmissions. L depends on the distribution of the jobs failure time. With ρ the ratio of outliers and ϕ the ratio of failed jobs, the probability, for a job to succeed is $(1-\rho-\phi)$.

A job encounters a latency $L < t$, t being fixed, if it is not an outlier and either:

- the job does not fail (probability $(1-\rho-\phi)$) and its latency $R < t$ (probability $P(R < t) = F_R(t)$); or
- the job fails at $t_0 < t$ (probability $\phi f_F(t_0)$) and the job resubmitted encounters a latency $L < (t - t_0)$

The cumulative distribution of L is thus defined recursively by:

$$F_L(t) = (1-\rho-\phi)F_R(t) + \phi \int_0^t f_F(t_0).F_L(t-t_0)dt_0$$

where the distributions of R and F are for instance numerically estimated from the statistical data set described in the previous section. However, in this equation, the cdf F_L appears both in left and right sides. Moreover, its value at time t does appear in both terms.

In order to compute the cdf F_L, we discretize this equation with some considerations:

- No successful job has a null latency: $F_R(0) = 0$
- We introduce the second as the discretization step for the variable t. Indeed, in practice we know that we cannot have a higher precision than the second for our measurements. The discretization step is chosen accordingly.
- Some jobs are immediately known to fail (for example if the fault occurs on the client side). We thus consider $F_F(0) \neq 0$

Since no job has a null latency, this is also the case with resubmitted jobs: $F_L(0) = 0$. Supposing now $t > 1$, we get:

$$F_L(t) = (1-\rho-\phi)F_R(t) + \phi \sum_{t_0=0}^{t-1} f_F(t_0)F_L(t-t_0)$$

This equation is resolved differently in the cases $t = 1$ and $t > 1$. For $t = 1$, it simplifies to:

$$F_L(1) = (1-\rho-\phi)F_R(1) + \phi f_F(0)F_L(1) \Rightarrow F_L(1) = \frac{1-\rho-\phi}{1-\phi f_F(0)}F_R(1)$$

For $t > 1$, we can write:

$$F_L(t) = (1-\rho-\phi)F_R(t) + \phi f_F(0)F_L(t) + \phi\sum_{t_0=1}^{t-1} f_F(t_0)F_L(t-t_0)$$

leading to:

$$F_L(t) = \frac{1}{1-\phi f_F(0)}\left[(1-\rho-\phi)F_R(t) + \phi\sum_{t_0=1}^{t-1} f_F(t_0)F_L(t-t_0)\right]$$

On the right side of this equation, the terms in F_L are in the form $F_L(u)$ with $u \in [1\ ;\ (t-1)]$. $F_L(t)$ can therefore be computed recursively. The complete formula is given by equation 1:

$$\boxed{\begin{aligned} F_L(0) &= 0 \\ F_L(1) &= \frac{1-\rho-\phi}{1-\phi f_F(0)}F_R(1) \\ F_L(t>1) &= \frac{1}{1-\phi f_F(0)}\left[(1-\rho-\phi)F_R(t) + \phi\sum_{u=1}^{t-1} f_F(t-u)F_L(u)\right] \end{aligned}} \qquad (1)$$

4.2 Exploitation of the Grid Traces

Figure 8 displays the cdfs of several variables. F_R and F_F have been estimated from the grid traces data. Equation 1 enables to compute F_L, the cdf of successful jobs including resubmission in case of failures. We clearly observe the impact of failures in this latency L when compared to R. F_L's curve is lower: the probability of achieving a given latency when faults occur is thus lower. In order to see more precisely the impact of failures, we also plotted $(1-\rho)F_R$ which corresponds to the outliers and the successful jobs, ignoring failed jobs. This last curve is slightly above F_L: while L displays a probability of 50% for jobs to have a latency lower than 761 seconds, it reduces to 719 seconds when ignoring failures (or the difference of probability is 1% for the same latency value).

Having established the distribution properties of L, we will now focus on the exploitation of the data for implementing a realistic resubmission strategy that aims at reducing the latency experienced by users.

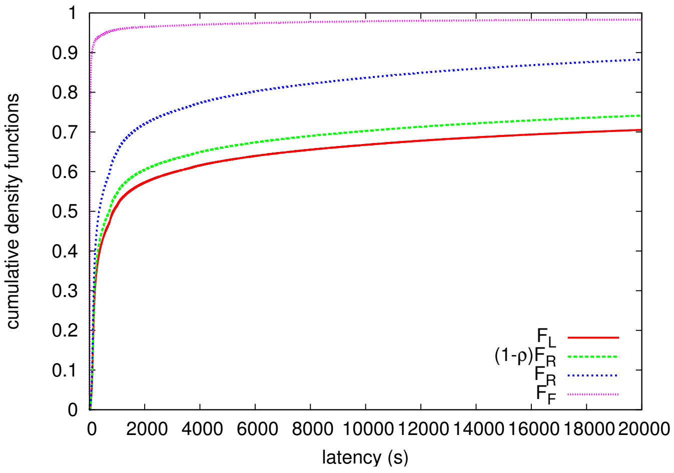


Fig. 8. cdfs: latency of fault detection (F_F), latency of successful jobs (F_R), latency of successful jobs using resubmission in case of failures (F_L). For comparison: $\tilde{F_R} = (1-\rho)F_R$.

5 Resubmission Strategy

5.1 Modeling

As seen in the previous section, the probability for a job to terminate before a given instant t is given by $F_L(t)$. We consider the resubmission strategy developed in [13] where a job is canceled and resubmitted if its latency is higher than a given timeout value t_∞ which value needs to be optimized. The work presented in [13] was based on probe jobs that neglected faults (they were excluded from the data) but some jobs did not return and were labelled as outliers. We denote $\tilde{F_R}(t)$ the probability for a job to face a latency lower than t. When neglecting faults, $\tilde{F_R}$ is related to the distribution of latency F_R and the ratio of outliers:

$$\tilde{F_R}(t) = (1-\rho)F_R(t)$$

We denote J the total latency including resubmissions after waiting periods of t_∞. From [13], we can express the expected total latency E_J, considering resubmissions at t_∞ as:

$$E_J(t_\infty) = \frac{1}{\tilde{F_R}(t_\infty)} \int_0^{t_\infty} (1-\tilde{F_R}(u))du \tag{2}$$

Thanks to the more complete workload data studied in this paper, we can refine the model by taking the latency for fault detections into account. We thus consider the following resubmission strategy: jobs for which the latency L, including resubmissions due to failures, is greater than a timeout value t_∞ are canceled and resubmitted. Observing that $F_L(t)$ corresponds to the probability for a job to succeed with a latency lower than t, we can replace, in equation 2, $\tilde{F_R}$ by F_L:

$$\boxed{E_J(t_\infty) = \frac{1}{F_L(t_\infty)} \int_0^{t_\infty} (1 - F_L(u))du} \quad (3)$$

Minimizing this equation leads to the estimation of the optimal timeout t_∞ value.

5.2 Experiments: Taking into Account Faults in the Model

The profile of the expectation of the total latency, including all resubmissions and computed from equation 3 is plotted in figure 9. The curve reaches a minimum value $E_J = 584s$ for an optimal timeout value $t_\infty = 195s$.

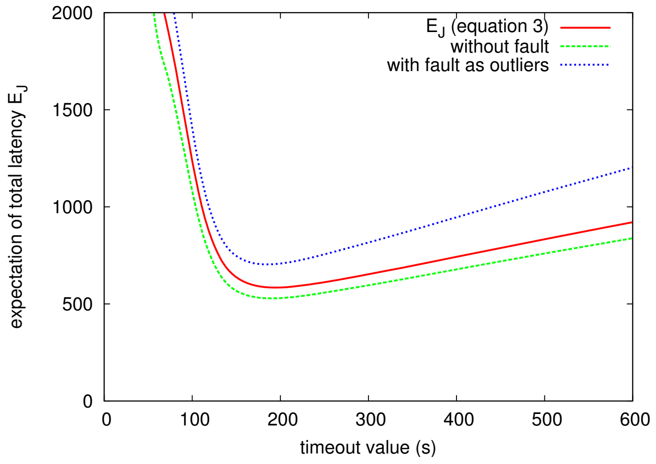


Fig. 9. Expectation of total latency with respect to timeout value t_∞. The first curve is obtained from equation 3. The result is compared with the case ignoring failures and the case where failures are accounted as outliers.

Two others profiles are plotted for comparison. The first one is the case ignoring the failures and corresponding to equation 2 with $\tilde{F_R} = (1 - \rho)F_R$. In that case, the minimum is reached at $t_\infty = 191s$, leading to $E_J = 529s$ which is under-evaluated.

The second comparison is performed with the assumption that failures can be considered as outliers, thus leading to a total of 35% of outliers. In this case, E_J reaches a minimum at $t_\infty = 185s$, which is underestimated and conducts to minimal value $E_J = 704s$, highly overestimated.

This experiment shows that taking into account a model of latency for faults detection has an influence on the parameters for this particular resubmission strategy.

5.3 Experiments: Reducing the Number of Cases

In section 3, we have retained the 37 most frequent cases, displayed in table 3. Here, results obtained with different number of most frequent cases are compared, in order to measure the relevance of reducing the number of cases to be taken into account. Figure 10 presents the variation of E_J with respect to the timeout value t_∞ for different numbers of most frequent cases. The optimal values of t_∞ leading to minimal E_J values are given in table 4. We observe that reducing the number of cases from 37 to 30 does not impact the results of the resubmission strategy, showing that not taking care of all possible cases (315 cases) does not impact the final result, since we are considering the most frequent ones. However, reducing the number to 20 or less cases impacts the final result. In table 4, results concerning the model including faults and the previous model without including faults are displayed. These results shows that reducing the number of cases to less than 20 cases impacts more than not considering the faults in the model.

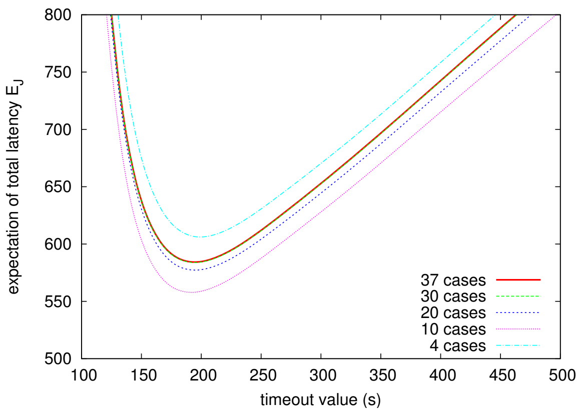


Fig. 10. Variations of the expected total latency (E_J) including resubmissions with respect to the timeout value, for different number of cases from table 3. We observe no visual difference between 37 and 30 cases. For less cases, we observe variations of E_J.

Table 4. Influence of the number of most frequent cases taken into account in the model on the estimation of optimal timeout value (t_∞) and minimal expectation of total latency including resubmission (E_J). Comparison of the results in two cases: with or without faults included in the model.

nb. of	with faults (F_L)		without faults ($\tilde{F_R}$)	
cases	opt. t_∞	min. E_J	opt. t_∞	min. E_J
37	195s	584s	191s	529s
30	194s	584s	191s	529s
20	195s	577s	191s	524s
10	192s	558s	189s	530s
4	199s	606s	197s	570s

6 Conclusions and Perspectives

Probabilistic models of the grid jobs latency enable us to capture the complex behavior of grid workload management systems. The model proposed in this paper relies on statistic collection of job execution traces in order to estimate the cdf of several parameters stochastically modeled. As compared to previous work, the model as been enriched to take into account normal operations, outliers and faults, which frequency is high on grids and therefore significantly impacts job execution time. The model is exploited to optimize a simple job resubmission strategy that aims at optimizing applications performance using objective information. The more jobs a grid application is composed with, the more it will be sensitive to such fault recovery procedures. In the future, more elaborate submission strategies commonly implemented on grids, such as multiple submission of a same task, will be considered.

This paper also emphasizes on the practical difficulties encountered when collecting and then exploiting traces on a large scale, heterogeneous production grid infrastructure. The set up of a Grid Observatory with well established procedures for traces collection, harmonization and curation is critical for the success of such grid behavioral analysis. It will allow to focus on modeling and experimentation without having to consider heavy-weight technical problems in the context of each new study. In addition, the Grid Observatory ensures dense data collection for accurate estimations without disturbing the normal grid operation.

The preliminary work detailed in this paper exploits a consistent but archived set of traces for *a posteriori* analysis. In the future, the Grid Observatory is expected to provide live information for tackling the non-stationarity of the grid workload manager and enabling relevant estimate of the grid running conditions.

Acknowledgments

The datasets used in this work have been provided by the Grid Observatory and the ICL RTM. The Grid Observatory is part of the EGEE-III EU project INFSO-RI-222667.

The authors warmly thank Cécile Germain-Renaud (LAL-LRI Orsay) for founding the Grid Observatory, Stephen McGough (Imperial College London) who is participating to this working group and Gidon Moont, the original RTM developer.

Special thanks are dedicated to Tristan Glatard who presented the paper at the JSSPP workshop at the time Iris was born.

This work is partly funded by French National Agency for Research, NeuroLOG project, under contract number ANR-06-TLOG-024.

References

1. Glatard, T., Montagnat, J., Pennec, X.: A probabilistic model to analyse workflow performance on production grids. In: Priol, T., Lefevre, L., Buyya, R. (eds.) IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), Lyon, France, pp. 510–517. IEEE, Los Alamitos (2008)

2. Lingrand, D., Montagnat, J., Glatard, T.: Modeling user submission strategies on production grids. In: International Symposium on High Performance Distributed Computing (HPDC 2009), München, Germany, June 2009, pp. 121–129 (2009)
3. 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)
4. 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)
5. Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., Epema, D.: The Grid Workloads Archive. Future Generation Computer Systems 24(7), 672–686 (2008)
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. Christodoulopoulos, K., Gkamas, V., Varvarigos, E.A.: Statistical Analysis and Modeling of Jobs in a Grid Environment. Journal of Grid Computing (JGC) 6(1), 77–101 (2008)
8. Germain, C., Loomis, C., Mościcki, J.T., Texier, R.: Scheduling for Responsive Grids. Journal of Grid Computing (JGC) 6(1), 15–27 (2008)
9. Li, H., Groep, D., Walters, 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)
10. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation (UKSim), Cambridge, UK, April 2008, pp. 126–131 (2008)
11. Pacini, F.: WMS User's Guide. Technical Report EGEE-JRA1-TEC-572489, EGEE (May 2006)
12. Lingrand, D., Montagnat, J., Glatard, T.: Estimating the execution context for refining submission strategies on production grids. In: Assessing Models of Networks and Distributed Computing Platforms (ASSESS / ModernBio) (CCgrid 2008), Lyon, pp. 753–758. IEEE, Los Alamitos (2008)
13. Glatard, T., Montagnat, J., Pennec, X.: Optimizing jobs timeouts on clusters and production grids. In: International Symposium on Cluster Computing and the Grid (CCGrid 2007), Rio de Janeiro, pp. 100–107. IEEE, Los Alamitos (2007)

The Resource Usage Aware Backfilling

Francesc Guim, Ivan Rodero, and Julita Corbalan[⋆]

Computer Architecture Department, Technical University of Catalonia (UPC), Spain

Abstract. Job scheduling policies for HPC centers have been extensively studied in the last few years, especially backfilling based policies. Almost all of these studies have been done using simulation tools. All the existent simulators use the runtime (either estimated or real) provided in the workload as a basis of their simulations. In our previous work we analyzed the impact on system performance of considering the resource sharing (memory bandwidth) of running jobs including a new resource model in the Alvio simulator. Based on this studies we proposed the *LessConsume* and *LessConsume Threshold* resource selection policies. Both are oriented to reduce the saturation of the shared resources thus increasing the performance of the system. The results showed how both resource allocation policies shown how the performance of the system can be improved by considering where the jobs are finally allocated.

Using the *LessConsume Threshold* Resource Selection Policy, we propose a new backfilling strategy : the *Resource Usage Aware Backfilling* job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different *Threshold* configurations. This backfilling variant that considers how the shared resources are used by the scheduled jobs. Rather than backfilling the first job that can moved to the run queue based on the job arrival time or job size, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalized runtime caused by the resource sharing saturation.

In the paper we demostrate how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. We show how it can achieve a close response time performance that the shortest job first Backfilling with First Fit (oriented to improve the start time for the allocated jobs) providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime.

1 Introduction

Several works focused on analyzing job scheduling policies have been presented in the last decades. The goal was to evaluate the performance of these policies with specific workloads in HPC centers. A special effort has been devoted to evaluating backfilling-based ([4][22]) policies because they have demonstrated an ability to reach the best performance results (i.e: [12] or [21]). Almost all of these studies have been done using simulation tools. To the best of our knowledge, all the existent simulators use the

[⋆] This paper has been supported by the Spanish Ministry of Science and Education under contract TIN200760625C0201 and by the IBM/BSC MareIncognito project under the grant BES-2005-7919.

runtime (either estimated or real) provided in the workload as a basis of their simulations. However, the runtime of a job depends on runtime issues such as the specific resource selection policy used or the resource jobs requirements.

In [15] we evaluated the impact of considering the penalty introduced in the job runtime due to resource sharing (such as the memory bandwidth) in system performance metrics, such as the average bounded slowdown or the average wait time, in the backfilling policies in cluster architectures. To achieve this, we developed a job scheduler simulator (Alvio simulator) that, in addition to traditional features, implements a job runtime model and resource model that try to estimate the penalty introduced in the job runtime when sharing resources. In our previous work and we only considered in the model the penalty introduced when sharing the memory bandwidth of a computational node. Results showed a clear impact of system performance metrics such as the average bounded slowdown or the average wait time. Furthermore, other interesting collateral effects such as a significant increment in the number of killed jobs appeared. Moreover the impact on these performance metrics was not only quantitative.

Using the conclusions reached in our preliminary work, in [16] we described two new resource selection policies that are designed to minimize the saturation of shared resources. The first one, the *LessConsume* attempts to minimize the job runtime penalty that an allocated job will experience. It is based on the utilization status of shared resources in the current scheduling outcome and the job resource requirements. The second once, the *LessConsume Threshold*, finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. This resource selection policy was designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job. Both resource allocation policies showed how the performance of the system can be improved by considering where the jobs are finally allocated. They showed a very important improvement in the percentage of penalized runtimes of jobs due to resource sharing saturation, and more importantly, in the number of killed jobs. Both have reduced by four or even six times the number of killed jobs versus the traditional resource selection policies.

In this paper we propose a new backfilling strategy: the *Resource Usage Aware Backfilling* job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different *Threshold* configurations. In brief, this backfilling variant is based on the Shortest-Backfilled First backfilling variant. Rather than backfilling the first job that can be moved to the run queue, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalty factors. However, it also takes into consideration the expected response time of the jobs that it evaluates during the backfilling process. The presented paper uses the model described in our previous work where the memory usage is considered.

The rest of the paper is organized as follows: section 2 presents the related work; section 3 briefly introduces the resource and runtime models that we proposed; next, the *LessConsume* Threshold resource selection policies are described; the RUA-Backfilling is presented in section 5; in section 6 we present an evaluation of different scheduling configurations; and finally, in section 9 we present the conclusions of this work.

2 Related Work

Authors like Feitelson, Schwiegelshohn, Calzarossa, Downey or Tsafrir have modeled logs collected from large scale parallel production systems. They have provided inputs for the evaluation of different system behavior. Such studies have been fundamental since they have allowed an understanding how the HPC centers users behave and how the resources of such centers are being used. Feitelson has presented several works concerning this topic, among others, he has published papers on log analysis for specific centers [10], general job and workload modeling [8][11][9], and, together with Tsafrir, papers on detecting workload anomalies and flurries [25]. Calzarossa has also contributed with several workload modellization surveys [1][2]. Workload models for moldable jobs have been described in works of authors like Cirne et al. in [5][6], by Sevcik in [18] or by Downey in [7]. These studies have been considered in the design of new scheduling strategies.

From the early nineties, local scheduling architectures and policies have been one of the main goals of research in the area of high performance computing. Backfilling policies have been deployed in the major HPC centers. A backfilling scheduling policy is an optimization of the simplest scheduling algorithm: First-Come-First-Serve (FCFS). It starts jobs that have arrived later than the job at the head of the wait queue if the estimated start time of this job is not delayed. Typically, this is called a reservation for the first job. This backfilling is the most basic backfilling policy proposed by Lifka et al. in [20] and it is called EASY-Backfilling. Many variants of this first proposal have been described in several papers. The differences between each of them can be identified as follows:

– The order in which the jobs are backfilled from the wait queue: in the EASY variant the jobs are backfilled in arrival order, other variants have proposed backfilling the jobs in shortest job first order (Shortest-Job-Backfilled-First [23][22]). More sophisticated approaches propose dynamic backfilling priorities based on the current wait time of the job and the job size (LXWF-Backfilling [4]).
– The order in which the jobs are moved to the head of the wait queue, i.e.: which job is moved to the reservation. Similar to backfilling priorities, in the literature many papers have proposed pushing the job to the reservation in FCFS priority order or using the LXWF-Backfilling order.
– The number of reservations that the scheduler has to respect when backfilling jobs. The EASY variant is the most aggressive backfilling since the number of reservations is 1. As a result, in some situations the start time for the jobs that are queued behind the head job may experience delays due to the backfilled jobs. More conservative approaches propose that none of the queued jobs are delayed for a backfilling job. However, in practice, this last kind of variant is not usually used in real systems.

General descriptions of the most frequently used backfilling variants and parallel scheduling policies can be found in the report that Feitelson et al. provide in [12]. Moreover, a deeper description of the conservative backfilling algorithm can be found in [21], where the authors present a characterization and explain how the priorities can be used to select the appropriate job to be scheduled.

Backfilling [20] policies have been the main goal of study in recent years. As with research in workload modeling, authors like Frachtenberg have provided the community with many works regarding this topic. In [12] general descriptions of the most commonly used backfilling variants and parallel scheduling policies are presented. Moreover, a deeper description of the conservative backfilling algorithm can be found in [21], where the authors present policy characterizations and how the priorities can be used when choosing the appropriate job to be scheduled. Other works are [13] and [4].

More complex approaches have been also been proposed by other researchers. For instance, in [17] the authors propose maintaining multiple job queues which separate jobs according to their estimated run time, and using a backfilling aggressive based policy. The objective is to reduce the slowdown by reducing the probability that short job is queued behind a long job. Another example is the optimization presented by Shmueli et al. in [19] which attempts to to maximize the utilization using dynamic programming to find the best packing possible given the system status.

3 The Runtime Model

In this section we provide a brief characterization for the runtime model that we designed for evaluate the resource sharing in the Alvio simulator. The main goal of this simulator is to model the different scheduling entities and computing resrouces that are included in the current HPC architectures. Despite the simulator allows to simulate distributed systems, in the work presented in this paper only one HPC center is considered. The accesses to the HPC resources are controlled by two different software components: the Job Scheduler and the Local Resources Manager. The figure 1 provides a general overview of the different elements that are involved in a HPC system and their relations. As can be observed, these computational resources are composed by a set of physical resources (the processors, the memory, the I/O system etc.) that are managed by the local scheduler and the local resource manager (LRM). The local scheduler has the responsibility of scheduling the jobs that the users submit and the local resource manager has the responsibility to control the access to the physical resources.

In [15] we present a detailed description of the model and its evaluation.

3.1 The Job Scheduling Policy

The job scheduling policy uses as input a set of job queues and the information provided by the local resource manager (LRM) that implements a Resource Selection Policy (RSP). It is responsible to decide which of the jobs that are actually waiting to be executed have to be allocated to the free resources. To do this, considering the amount of free resources it selects the jobs that can run and it requires to the LRM to allocate the job processes.

3.2 The Resource Selection Policy

The Resource Selection Policy, given a set of free processors and a job α with a set of requirements, decides to which processors the job will be allocated. To carry out this

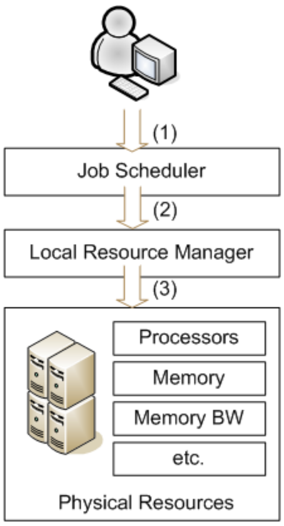


Fig. 1. The local scheduler internals

selection, the RSP uses the Reservation Table (RT, see figure 2). The RT represents the status of the system at a given moment and is linked to the architecture. The reservation table is a bi dimensional table where the X axes represent the time and the Y axes represent the different processors and nodes of the architecture. It has the running jobs allocated to the different processors during the time. One allocation is composed of a set of buckets[1] that indicate that a given job α is using the processors $\{p_0, .., p_k\}$ from *start time* until *end time*.

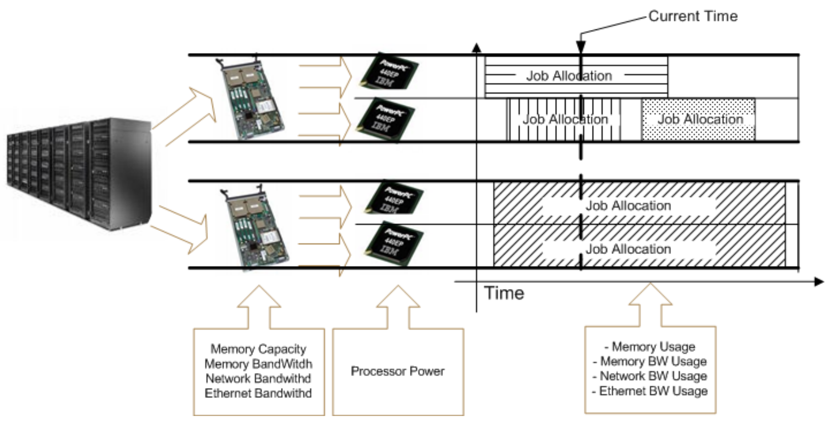


Fig. 2. The reservation table

[1] The $b_{(i,t_{i_0},t_{i_1})}$ bucket is defined as the interval of time $[t_x, t_y]$ associated to the processor p_i.

An allocation is defined by: $allocation\{\alpha\} = \{[t_0,t_1], P = \{p_{\{g,n_h\}},..p_{\{s,n_t\}}\}\}$ and indicates that the job α is allocated to the processors P from the time t_0 until t_1. The allocations of the same processors must satisfy that they are not overlapped during the time.

Figure 3 provides an example of a possible snapshot of the reservation table at the point of time t_1. Currently, there are three jobs running in three different job allocations:
$a_1 = \{[t_0,t_2], \{p_{\{1,node_1\}}, p_{\{2,node_1\}}, p_{\{3,node_1\}}\}\}$
$a_2 = \{[t_1,t_3], \{p_{\{4,node_1\}}, p_{\{5,node_1\}}, p_{\{1,node_2\}}, p_{\{2,node_2\}}\}\}$
$a_3 = \{[t_1,t_4], \{p_{\{5,node_2\}}\}\}$

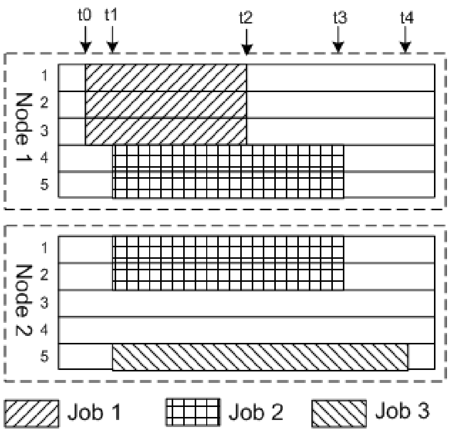


Fig. 3. Reservation Table Snapshot

3.3 Modeling the Conflicts

The model that we have presented in the previous subsection has some properties that allow us to simulate the behavior of a computational center with more details. Different resource selection policies can be modeled. Thanks to the Reservation Table, it knows at each moment which processors are used and which are free.

Using the resource requirements for all the allocated jobs, the resource usage for the different resources available on the system is computed. Thus, using the Reservation Table, we are able to compute, at any point of time, the amount of resources that are being requested in each node.

In this extended model, when a job α is allocated during the interval of time $[t_x,t_y]$ to the reservation table to the processors $p_1,..,p_k$ that belong to the nodes $n_1,..,n_j$, we check if any of the resources that belong to each node is saturated during any part of the interval. In the affirmative case a runtime penalty will be added to the jobs that belong to the saturated subintervals. To model these properties we defined the Shared Windows and the penalty function associated to it.

The Shared Windows. A *Shared Window* is an interval of time $[t_x,t_y]$ associated to the node n where all the processors of the node satisfy the condition that: either no process is allocated to the processor, or the given interval is fully included in a process that is running in the processor.

The penalty function. This function is used to compute the penalty that is associated with all the jobs included to a given Shared Window due to resources saturation. The input parameters for the function are:

– The interval associated to the Shared Window $[t_x, t_y]$.
– The jobs associated to the Shared Window$\{\alpha_0, .., \alpha_n\}$
– The node n associated to the Shared Window with its physical resources capacity.

The function used in this model is defined as [2]:

$$\forall res \in resources(n) \rightarrow demand_{res} = \sum_{\alpha}^{\{\alpha_0,..,\alpha_n\}} r_{\alpha,res} \tag{1}$$

$$Penalty = \sum_{resources(n)}^{res} \left(\frac{\max\left(demand_{res}, capacity_{res}\right)}{capacity_{res}} - 1\right) \tag{2}$$

$$PenlizedTime = (t_y - t_x) * Penalty \tag{3}$$

First for each resource in the node the resource usage for all the jobs is computed. Second, the penalty for each resource consumption is computed. This is a linear function that depends on the saturation of the used resource. Thus if the amount of required resource is lower than the capacity the penalty will be zero, otherwise the penalty added is proportional to the fraction of demand and availability. Finally, the penalized time is computed by multiplying the length of the Shared Window and the penalty. This penalized time is the amount of time that will be added to all the jobs that belong to the Window corresponding to this interval of time. This model has been designed for the memory bandwidth shared resource and can be applicable to shared resources that behave similar. However, for other typology of shared resources, such as the network bandwidth, this model is not applicable. Our current work is focused on modeling the penalty model for the rest of shared resources of the HPC local scenarios that can impact in the performance of the system.

For compute the penalized time that is finally associated to all the jobs that are running: first, the shared windows for all the nodes and the penalized times associated with each of them are computed; second the penalties of each job associated with each node are computed adding the penalties associated with all the windows where the job runtime is included; and finally, the final penalty associated to the job is the maximum penalty that the job has in the different nodes where it is allocated.

4 The LessConsume Resource Selection Policies

Using the model that we have presented in the previous section we designed two new Resource Selection Policies. First, the *LessConsume* that attempts to minimize the job runtime penalty that an allocated job will experience. Based on the utilization status of the shared resources in current scheduling outcome and job resource requirements, the

[2] Note that all the penalty, resources, resource demands and capacities shown in the formula refer to the node n and the interval of time $[t_x, t_y]$. Thereby, they are not specified in the formula.

LessConsume policy allocates each job process to the free allocations in which the job is expected to experience the lowest penalties. Second, we designed the *LessConsume Threshold* selection policy which finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. It is a variant of the *LessConsume* policy and was designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job.

The core algorithm of the *LessConsume* selection policy is similar to the *First Fit* resource selection policy. This last one selects the first $\alpha_{\{CPUS,p\}}$ where the job can be allocated. However, in contrast to this previous algorithm, the *LessConsume* policy, once the base allocation is found, the algorithm computes the penalties associated with the different processes that would be allocated in the reservation. Thereafter it attempts to improve the allocation by replacing the selected buckets (used for create this initial allocation) that would have higher penalties with buckets that can be also selected, but that have not been evaluated. The *LessConsume* algorithm will iterate until the rest of the buckets have been evaluated or the penalty factor associated to the job is 1 (no penalty).[3]

In some situations this policy not only minimizes the penalized factor of the allocated jobs, but it also provides the same start times as the first fit allocation policy, which in practice provides the earliest possible allocation start time. However, in many situations the allocation policy of the lower penalty factor provides a start time that is substantially later than that achieved by a *First Fit* allocation. To avoid circumstances where the minimization of the penalty factor results in delays in the start time of scheduled jobs, we have designed the *LessConsume Threshold*. This is a parametrized selection policy which determines the maximum allowed penalty factor allocated to any given job.

In contrast to this first selection policy, the *LessConsume Threshold* policy allows the scheduler or to the administrator to specify the maximum desired penalty factor that the scheduler accepts for a given job. Thus, it is able to carry out the scheduling decisions taking into account the resource sharing saturation and it is able to verify how the job response time is affected by different allocations of the job.

The main differences between the two policies is that the second one will stop the process of evaluating all selected buckets when the penalty of the job is lower than the provided *Threshold*. Thus, in some situations this resource selection policy will return an allocation that has a higher penalty that the once that would have returned the *LessConsume* policy, however with a earlier start time. This policy provides the trade off to the scheduler to balance the benefits of delaying the job start time an obtaining a lower threshold, or advancing it and having a higher penalty.

5 The RUA-Backfilling

The *LessConsume Threshold* resource selection policy has been mainly designed to be deployed in two different scenarios. In the first case, the administrator of the local

[3] The penalty factor is computed:

$$PenaltyFactor_{\alpha} = \frac{\alpha_{\{RunTime,rt\}} + \alpha_{\{PenalizedRunTime,prt\}}}{\alpha_{\{RunTime,rt\}}}.$$

scenario specifies in the configuration files the penalty factor of a given allocated job. This factor could be empirically determined by an analytical study of the performance of the system. In the second, more plausible, scenario, the local scheduling policy is aware of how this parameterized RSP behaves and how it can be used by different factors. In this second case the scheduling policy can take advantage of this parameter to decide whether a job should start in the current time or whether it could achieve performance benefits by delaying its start time. In this last scenario the response time of a job can be improved in two different ways:

- Reducing the final runtime of the job by minimizing the penalty factor associated to the job.
- Reducing the wait time of the job by minimizing the start time of the job.

The Resource Usage Aware Backfilling Scheduling (RUA-Backfilling) policy takes into account both considerations when inspecting the wait queue for backfilling the jobs or finding the allocations for the reservations. In brief, this backfilling variant is based on the Shortest-Backfilled First backfilling variant. Rather than backfilling the first job that can moved to the run queue, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalty factors. However, it also takes into consideration the expected response time of the jobs that it evaluates during the backfilling process.

The different parameters of the RUA-Backfilling are:

1. The number of ***reservation*** (number of the jobs in the queue whose estimated start time can not be delayed) is *1*.
2. The different ***Thresholds*** that will be used to calculate the appropriate allocation for the job that is moved from the reservation. In the evaluation the thresholds used by the policy are $RUA_{thresh} = \{1.15; 1.20; 1.25; 1.5\}$.
3. The jobs are moved from the wait queue to the reservation using the First Come First Serve priority. This priority assures that the jobs submitted to the system will not suffer starvation.
4. The backfilling queue is ordered using a dynamic criteria that is computed each time that the backfilling processes is required. It is described below.

When a job α has to be moved to the reservation the following algorithm is applied:

1. For each *Threshold* in the RUA_{thresh} specified in the configuration of the policy (in the presented evaluation $\{1.15; 1.20; 1.25; 1.5\}$):
   (a) The allocation based on the *LessConsume Threshold* resource selection policy with a parameter of $PenaltyFactor = Threshold$ is requested from the local resource manager.
   (b) The slowdown for the job is computed based on the start/wait times, the penalized runtime of the job in the returned allocation, the current wait time of the job and its requested runtime.
2. The allocation with less slowdown is selected to allocate the job. The local scheduler contacts the local resource manager to allocate the job in the given allocation.

Given the jobs that are queued in the backfilling queue, the backfilling algorithm behaves as follows:

1. In the first step, for each job α in the backfilling queue its allocation is computed based on the algorithm introduced in the previous paragraph. If the start time for the returned allocation is the current time the job is added to the backfilling queue where the allocations are ordered by the penalized factor associated to the allocation and secondly by its length. Note that each job has only one assigned allocation.
2. In the second step, the backfilling queue has all the jobs that could be backfilled in the current time stamp ordered in terms of the associated penalty. The queue is evaluated and the first job that can be backfilled is allocated to the reservation table using allocation computed in the previous step. Note that the allocation will be exactly the same as the one computed in the first step.
   (a) If no job can be backfilled the process of backfilling is terminated.
   (b) Otherwise, steps 1 and 2 will be iterated again.

The key concept of this backfilling variant is to find out the allocation that provides the best slowdown for the job that is moved to the reservation, and to backfill the jobs in the manner that the saturation of the shared resources is minimized. The second goal will reduce the number of killed jobs due to resource sharing saturation.

Note that in this algorithm the allocations are computed using the estimated runtime that is provided by the user. In the version of the policy evaluated in this scenario we have supposed that when a job is allocated with a penalty factor of $\alpha_{penalty}$, the estimated runtime is updated according this penalty. Based on our studies in prediction systems in backfilling policies [14], in our future versions of the RUA we plan to use the predicted runtime in the *LessConsume Threshold* and keep the original user requested runtime.

6 Experiments

In this section we characterize the different experiments that we defined in order to validate the performance of the different scheduling strategies that we propose.

6.1 Workloads

For the experiments we used the cleaned [24] versions of the workloads SDSC Blue Horizon (SDSC-BLUE) and Cornell Theory Center (CTC) SP2. For the evaluation experiments explained in the following section, we used the 10000 jobs of each workload plus 10000 jobs that were used in order to warm-up the system and achieve a steady state. Based on these workload trace files, we generated three variations for each one with different memory bandwidth pressure:

– HIGH: 80% of jobs have high memory bandwidth demand, 10% with medium demand and 10% of low demand.
– MED: 50% of jobs have high memory bandwidth demand, 10% with medium demand and 40% of low demand.
– LOW: 10% of jobs have high memory bandwidth demand, 10% with medium demand and 80% of low demand.

6.2 Architecture

For each of the workloads used in the experiments we defined architecture with nodes of four processors, 6000 MB/Second of memory bandwidth, 256 MB/Second of Network bandwidth and 16 GB of memory. In addition to the SWF [3] traces with the job definitions we extended the standard workload format to specify the resource requirements for each of the jobs. Currently, for each job we can specify the average memory bandwidth required (other attributes can be specified but are not considered in this work). Based on our experience and the architecture configuration described above, as a first approach we defined that a *low memory bandwidth demand* consumes 500 MB/Second per process; a *medium* memory bandwidth demand consumes 1000 MB/Second per process; and that a *high memory bandwidth demand* consumes 2000 MB/Second per process. These memory requirements were selected based on the typology of jobs that were running in our centers.

7 Scenarios

In the experiments we evaluate the impact of the RUA-Backfilling in the system. To do this, we compare its performance against the Shortest Job Backfilled First policy under the *LessConsume* resource selection policies. For the analysis of the RUA-Backfilling job scheduling policy the following configurations were evaluated:

1. The Shortest Job Backfilled First scheduling policy using:
   – The **LessConsume** resource selection policy.
   – The **LessConsume Threshold** resource selection policy with four different factors (*1*, *1,15*, *1,25* and *1,5*).
   – The First-Fit resource selection policy.
2. The RUA-Backfilling policy.

All the simulations have used the the job runtime model with resource sharing model introduced in the first part of this paper. In the rest of the section we analyze the different configurations that we have introduced: first, we provide a discussion concerning the differences between using the *LessConsume* and *LessConsume Thresolds* policies in the SJBF Backfilling variant. Next, we compare the performance of the SJBF Backfilling with the First-Fit and LessConsume resources selection policies against the results obtained using the RUA-Backfilling.

8 Evaluation

In this section we present the evaluation of the RUA-Backfilling job scheduling policy. However, in order to provide a characteriztation of the *LessConsume* resource selection policies, first we present their performance analysis. This analysis is used later on in the discussion of the RUA-Backfilling.

8.1 The LessConsume and LessConsume Threshold

Tables 1 and 2 present the 95_{th} percentile and average of the bounded slowdown for the CTC and SDSC centers for each of the three workloads for the *First Fit* (FF), *LessConsume* and *LessConsume Threshold* resource selection policy. The last one was evaluated with three different factors: *1*, *1,15*, *1,25* and *1,5*. In both centers the *LessConsume*policy performed better than the *LessConsume Threshold* with a factor of *1*. One could expected that the *LessConsume* should be equivalent to use the *LessConsume Threshold* with a threshold of *1*. However, note that this affirmation would be incorrect. This is caused due to the *LessConsume* policy evaluates all the buckets in a subset of all the possible allocations. The goal of this policy is to optimize the *First Fit* allocation but without carry out a deeper search of other possibilities. However, the *LessConsume Threshold* may look further in the future in the case that the penalty is higher than the provided threshold. Thereby, this last one is expected to provide higher wait time values. On the other hand, as we had expected, the bounded slowdown decreases while increasing the factor of the *LessConsume Threshold* policy. In general, the ratio of increment of using a factor of *1* and a factor of *1,5* is around a 20% in all the centers and workloads.

The performance of these two resource policies, compared to the performance of the *First Fit* policy, shows that *LessConsume* policies give an small increment in the bounded slowdown. For instance, in the CTC high memory pressure workload the 95_{th} percentile of the bounded slowdown has increased from 4,2 in the *First Fit* to 5,94 in the *LessConsume* policy, or to 7,92 and 5,23 in the *LessConsume Threshold* with thresholds of *1* and *1,5* respectively.

Table 1. Bounded-Slowdown - 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	4,2	5,94	7,92	6,12	5,32	5,23
	Med	2,8	3,55	4,22	3,82	3,65	3,52
	Low	2,2	3,12	3,62	3,82	3,45	3,52
SDSC	High	99,3	110,21	128,08	115,28	109,51	106,23
	Med	55,4	68,06	74,32	72,83	71,37	68,52
	Low	37,8	45,37	57,27	52,86	42,28	42,28

Table 2. Bounded-Slowdown - Average

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	8,2	10,44	18,02	12,38	11,32	13,54
	Med	5,3	6,65	7,52	8,05	6,85	7,75
	Low	3,2	5,62	5,92	5,82	8,84	8,56
SDSC	High	22,56	24,41	27,37	25,54	24,26	23,53
	Med	11,32	12,51	14,76	14,46	14,17	13,6
	Low	7,54	7,8	9,08	9,5	8,47	8,27

Tables 3 and 4 show the 95_{th} and average of the wait time for the CTC and SDSC centers for each of the three workloads for the *First Fit*, *LessConsume* and *LessConsume Threshold* resource selection policy. This performance variable shows similar pattern to the bounded slowdown. The *LessConsume* policy shows a better performance result that using the *LessConsume Threshold* with a factor of *1*.

The 95_{th} percenage of penalized runtime is presented in the table 5 and the average is shown in the table table 6. The penalized runtime clearly increases by incrementing the threshold. For instance, the 95_{th} Percentile of the percentage increases from 8,31 in the SDSC and the high memory pressure workload with a factor of *1* until 11,64 with a factor of *1,5*. The *LessConsume*, different from to the two previously described variables, shows similar values to the *LessConsume Threshold* with a factor of *1,5*. This percentage of penalized runtime was reduced with respect to the *First Fit* when using all the different factors in both centers.

Table 3. Wait Time - 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	10286	12588	17945	15612	14555	10188
	Med	8962	9565	13391	13123	9186	9094
	Low	4898	5034	6544	7198	5235	5759
SDSC	High	55667	63293	70964	69632	59978	41779
	Med	44346	45164	58713	59300	47440	45616
	Low	32730	35092	38265	37499	33374	33785

Table 4. Wait Time - average

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	20082	24576	35035	30480	28416	19890
	Med	13443	14347	20086	19684	13779	13641
	Low	7124	7322	9518	10469	7614	8376
SDSC	High	12647	14379	16122	15819	13626	9491
	Med	9061	9228	11996	12116	9693	9320
	Low	3931	4214	4595	4503	4008	4057

Table 5. Percentage of Penalized Runtime - 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	8,8	8,01	7,69	7,87	7,91	8,1
	Med	4,8	3,81	3,01	3,52	4,06	3,90
	Low	0,92	0,78	0,51	0,72	0,62	0,80
SDSC	High	11,8	11,33	8,31	10,37	11,58	11,64
	Med	6,7	6,01	4,70	4,85	5,64	5,96
	Low	1,4	1,03	0,75	0,81	0,94	1,19

Table 6. Percentage of Penalized Runtime - average

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	7,8	6,81	6,98	7,17	7,39	7,34
	Med	3,8	2,54	2,93	3,02	3,1	3,20
	Low	0,72	0,7	0,21	0,42	0,33	0,64
SDSC	High	15,1	10,32	7,41	11,73	10,85	12,32
	Med	10,2	7,2	4,70	4,85	5,64	5,96
	Low	5,2	4,53	2,56	3,11	4,58	4,23

Table 7. Number of Killed Jobs 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	428	120	57	70	87	97
	Med	247	101	76	77	102	99
	Low	64	45	36	38	58	52
SDSC	High	475	105	87	130	127	130
	Med	255	89	76	79	103	145
	Low	51	34	22	27	33	41

The number of killed jobs is the performance variable that showed most improvement in all the memory pressure workloads. The number of killed jobs is qualitatively reduced with the *LessConsume Threshold* with a factor of *1*: for example with the high memory pressure workload and the CTC center, the number of killed jobs was reduced from 428 with the *First Fit* to 70. The other threshold factors also showed clear improvements; the number was halved. As to the *LessConsume* policy, the number of killed jobs was reduced by a factor of 4 compared to the *First Fit* and the high and medium memory pressure workloads of both centers.

The *LessConsume* policy shows how the percentage of penalized runtime and number of killed jobs can be reduced in comparison to the *First Fit*, by using this policy with EASY backfilling. In traditional scheduling architectures this RSP can be used rather than traditional policies, without any modifications in local scheduling policies. Furthermore, the *LessConsume* threshold shows how, with different thresholds, performance results can also be improved. Higher penalty factors result in better performance of the system. However, in this situation the number of killed jobs and the percentage of penalized runtime is increased. The *LessConsume* policy shows similar performance results as the *LessConsume Threshold* with factors of *1,25* and *1,5*.

8.2 The Thresholds Trade Offs

Figures 4, 5, 6 and 7 present the performance of the *LessConsume* policies (using bounded slowdown) against the percentage of penalized runtime of the jobs and the number of killed jobs. The goal of these figures is to show the chance that the *LessConsume Threshold* and *LessConsume* policies have to improve the performance of the system while achieving an acceptable level of performance. As can be observed in

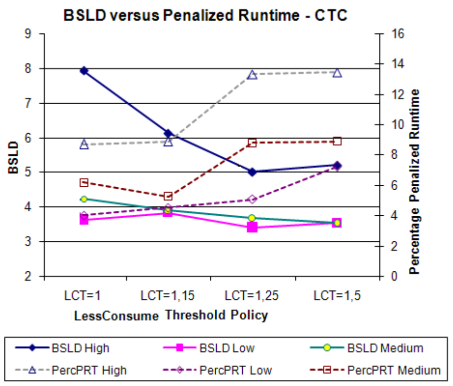


Fig. 4. BSLD versus Percentage of Penalized Runtime - CTC Center

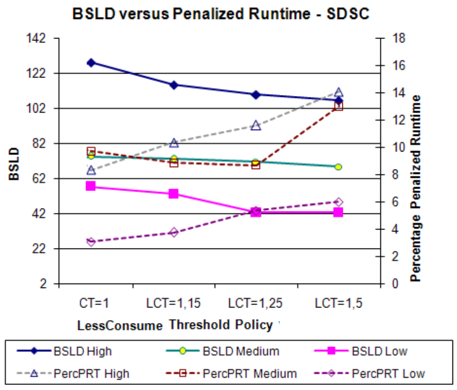


Fig. 5. BSLD versus Percentage of Penalized Runtime - SDSC Center

figures 6 and 4 a good balance is achieved in the CTC center using the threshold of *1,15* where both the number of killed Jobs and the percentage of penalized runtime converge are in acceptable values. In the case of the SDSC center, this point of convergence is not as evident as the CTC center. Considering the tendency of the bounded slowdown, it seems that the *LessConsume Threshold* with a factor of *1.15* is an appropriate configuration for this center, due to the fact that the penalized runtime and the number of killed jobs presents the lowest values, and the bounded slowdown shows values that are very close to the factors of *1,15* and *1,25*. However, the configuration of the *LessConsume Threshold* with a factor of *1,15* also shows acceptable values.

8.3 The RUA-Backfilling

In the previous subsections we have present the performance that the LessConsume resource selection policies achieve when they are used together with the SJBF-Backfilling

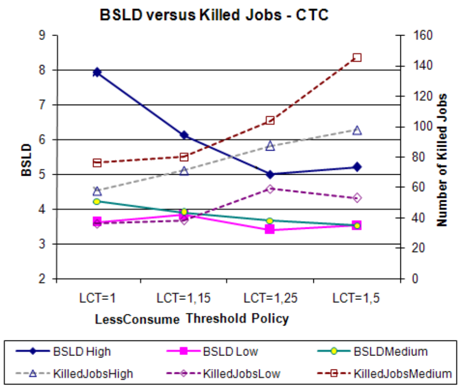


Fig. 6. BSLD versus Killed Jobs - CTC Center

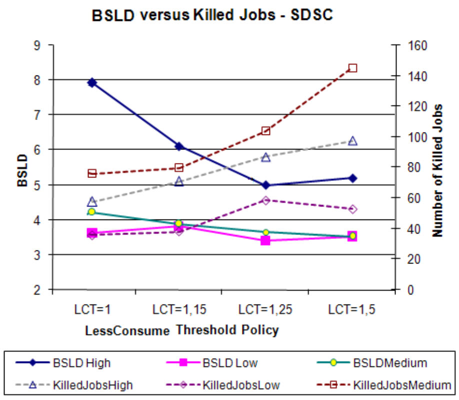


Fig. 7. BSLD versus Killed Jobs - SDSC Center

variant. We have observed that the *LessConsume Threshold* resource selection policy can provide good results when the used threshold is between *1.15* and *1.25* depending on the workload. Using this results in the first RUA-Backfilling version shown in this paper we decided to use threshold values presented in the section 5. In this section we present the benefits of the usage of a backfilling variant that interacts with the local resource manager against the traditional approaches.

The figures 9 and 8 present the performance that the RUA-Backfilling scheduling policy has achieved with respect the Shortest-Job-Backfilled First Backfilling (SJBF-Backfilling) with the *First Fit* and *LessConsume* Resource Selection Policies. The results shows also how each of the policies behaved with the three different memory pressure workloads for the SDSC and CTC workloads. The figure shows the 95_{th} Percentile of the BSLD, the Wait time and the Percentage of Penalized Runtime that the jobs have experimented and the number of killed jobs that three scheduling strategies have achieved in the simulations.

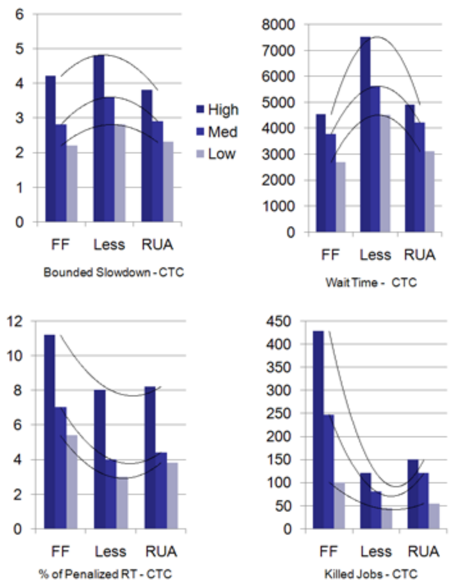


Fig. 8. RUA Performance Variables for the CTC Workload

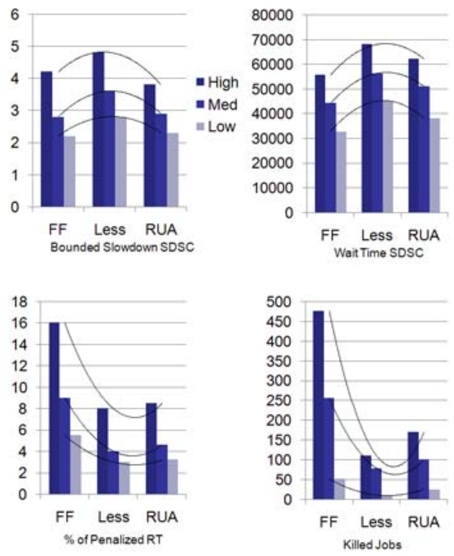


Fig. 9. RUA Performance Variables for SDSC Workload

The bounded slowdown shows how in both workloads the RUA-Backfilling achieves slightly worst performance that the SJBF-Backfilling with the first fit selection policy. In both cases the difference between the BSLD is less than a 5%. For instance, the 95_{th} Percentile of the BSLD with the SJBF-Backfilling in the SDSC workload with high memory pressure is 100 and with the RUA-Backfilling is around 110. Note that we could expect that this last one should achieve smaller BSLD that the once obtained by the SJBF-Backfilling with FF due to it takes into account the resource usage. However, in the RUA-Backfilling the number of jobs that are used for compute the BSLD (number of finished jobs) is substantially bigger that the once used in the other (400 less in the SJBF). Respect the SJBF-Backfilling with the *LessConsume* resource selection policy, the RUA-Backfilling shows in both workload better bounded slowdowns.

The wait time shows similar patterns that the Bounded Slowdown. However, the CTC workload shows higher differences between the SJBF-Backfilling and the other two strategies. For example, while the 95_{th} Percentile of wait time for the RUA-Backfilling and the SJBF-Backfilling with FF remains around 4000 and 5000 seconds in the high pressure scenario, the SJBF-Backfilling with *LessConsume* presents 95_{th} Percentile of the wait time around 7000. This, may indicate that the RUA Backfilling is more stable than using the *LessConsume* with a non resource usage aware scheduling strategy.

Finally, the number of killed jobs and the 95_{th} Percentile of percentage of penalized runtime show a qualitative improvement respect the SJBF-Backfilling with *First Fit*. For example, the RUA-Backfilling shows a reduction of a 500% in the number of killed jobs in the high memory pressure scenario of the SDSC workload and a reduction of 300% in the CTC scenario also with the high memory pressure scenario. Although the percentage of penalized run time shows an improvement in both center using the RUA-Backfilling, a higher improvement is shown in the SDSC center. For example, in this last case the percentage of penalized runtime is reduced a 50% in the workload with a medium memory pressure.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with *LessConsume* scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

9 Conclusions

In this paper we have shown how the performance of the system can be improved by considering resource sharing usage and job resource requirements in the new RUA Backfilling variant. In this proposal the local scheduler cooperates with the local resource manager in order to find out the allocation that minimizes the job runtime penalty due to the saturation of the resource sharing. This is a backfilling variant scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different configurations of the *LessConsume Threshold* resource selection policy that we proposed in our previous work. In the first part of the paper we have introduced the key concepts of our previous works that are used in the RUA-Backfilling algorithm. First the runtime model used in our simulator, and second, the *Find LessConsume* and LessConsume Threshold resource selection policies.

In this paper we evaluate the effect of considering the memory bandwidth usage in the different scheduling strategies under several workloads. Two different workloads from the Standard Workload Archive have been used in the experiments (the SDSC Blue Horizon (SDSC-BLUE) and the Cornell Theory Center). For each of them we have generated three different scenarios: with high (HIGH), medium (MED), and low (LOW) percentage of jobs with high memory demand. We have evaluated the impact of using the *LessConsume* and *LessConsume Threshold* with the Shortest Job Backfilled first and the RUA-Backfilling presented in this paper. These synthetic workloads have been used as a first approach to evaluate the potential of the proposed techniques.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with *LessConsume* scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

Concerning the penalty model used in our system, our future work will consider how other shared resources may impact in the performance of the system. Clearly, the penalty function that has been presented in our model, has to be extended for consider penalties that other typologies of resource may show. For instance, the network bandwidth shows patterns in the job execution that are not considered in the penalty function. On the other hand, our future research will evaluate the impact of having inaccurate estimations in the job resource sharing requirements. Related to this, we will work in the usage of prediction techniques in order to estimate the resource requirements of the submitted jobs.

The RUA-Backfilling that we have presented in this paper uses a set of pre-configured *Threshold* for finding out the job allocations. In our research we have stated that the workloads can show very different load patterns during the time. Thus, depending of the epoch, the system may experiment better performance using different *Threshold* values. Considering this phenomena we will extend the RUA in order to dynamically determine which factors should be used in each scheduling moment. The first step will be studying the correlation of the system performance against the load of the system and *Threshold* configuration. Afterward, we will use this information for extend the current RUA-Policy policy.

References

1. Calzarossa, M., Haring, G., Kotsis, G., Merlo, A., Tessera, D.: A hierarchical approach to workload characterization for parallel systems. In: Hertzberger, B., Serazzi, G. (eds.) HPCN-Europe 1995. LNCS, vol. 919, pp. 102–109. Springer, Heidelberg (1995)
2. Calzarossa, M., Massari, L., Tessera, D.: Workload characterization issues and methodologies. In: Reiser, M., Haring, G., Lindemann, C. (eds.) Dagstuhl Seminar 1997. LNCS, vol. 1769, pp. 459–482. Springer, Heidelberg (2000)
3. 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, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 66–89. Springer, Heidelberg (1999)
4. Chiang, S.-H., Arpaci-Dusseau, A.C., 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)
5. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: 4th Ann. Workshop Workload Characterization (2001)
6. Cirne, W., Berman, F.: A model for moldable supercomputer jobs. In: 15th Intl. Parallel and Distributed Processing Symp. (2001)
7. Downey, A.B.: A parallel workload model and its implications for processor allocation. In: 6th Intl. Symp. High Performance Distributed Comput. (August 1997)
8. Feitelson, D.G.: Packing schemes for gang scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 89–110. Springer, Heidelberg (1996)
9. 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)
10. Feitelson, D.G., Nitzberg, B.: Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 337–360. Springer, Heidelberg (1995)
11. Feitelson, D.G., Rudolph, L.: Metrics and benchmarking for parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 1–24. Springer, Heidelberg (1998)
12. 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)
13. Feitelson, D.G., Weil, A.: Utilization and predictability in scheduling the ibm sp2 with backfilling. In: Proceedings of the 12th. International Parallel Processing Symposium, pp. 542–546 (1998)
14. Guim, F., Corbalan, J.: Prediction f based models for evaluating backfilling scheduling policies. In: The 8th International Conference on Parallel and Distributed Computing, Applications and Technologies (2007)
15. Guim, F., Corbalan, J., Labarta, J.: Modeling the impact of resource sharing in backfilling policies using the alvio simulator. In: 15th Annual Meeting of the IEEE / ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (2007)
16. Guim, F., Corbalan, J., Labarta, J.: Resource sharing usage aware resource selection policies for backfilling strategies. In: The 2008 High Performance Computing and Simulation Conference (2008)
17. Lawson, B.G., Smirni, E.: Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 72–87. Springer, Heidelberg (2002)

18. Sevcik, K.C.: Application scheduling and processor allocation in multiprogrammed parallel processing systems. Performance Evaluation, 107–140 (1994)
19. Shmueli, E., Feitelson, D.G.: Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 228–251. Springer, Heidelberg (2003)
20. Skovira, J., Chan, W., Zhou, H., Lifka, D.A.: 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)
21. Talby, D., Feitelson, D.: Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In: Parallel Processing Symposium, pp. 513–517 (1999)
22. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using runtime predictions rather than user estimates. Technical Report 2005-5, School of Computer Science and Engineering, The Hebrew University of Jerusalem (2005)
23. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE TPDS (2006)
24. Tsafrir, D., Feitelson, D.G.: Workload flurries. Technical report, School of Computer Science and Engineering and The Hebrew University of Jerusalem (2003)
25. Tsafrir, D., Feitelson, D.G.: Instability in parallel job scheduling simulation: the role of workload flurries. In: 20th Intl. Parallel and Distributed Processing Symp. (2006)

The Gain of Overbooking

Georg Birkenheuer, André Brinkmann, and Holger Karl

Paderborn Center for Parallel Computing (PC^2)
University of Paderborn, Germany
{birke,brinkmann,holger.karl}@uni-paderborn.de

Abstract. This paper analyzes the effect of overbooking for scheduling systems in a commercial environment. In this scenario each job is associated with a release time and a finishing deadline as well as a fee for a successful execution and a penalty for violating the deadline. The core idea is to exploit overestimations of required job execution times, providing an opportunity to aggressively schedule additional jobs. The proposed probabilistic scheduler is based on histories of job execution times, device failure rates, and penalties for SLA service violations. This paper includes a theoretical background and a mathematical model of the overbooking approach and a simulative evaluation with a synthetic workload on a single-processor system.

1 Introduction

The commercial use of grid and cloud infrastructures is steadily growing. Current developments in automatic negotiation of service level agreements (SLAs) and the provision of quality of service (QoS) with fault tolerance mechanisms will further increase its commercial acceptance and adoption [1]. This acceptance allows providers to think about new business cases to foster their competitiveness in the emerging global service economy.

Grid or Cloud contracts will be based on the negotiation of SLAs between the service providers and their customers. During SLA negotiation, the customers reserve the amount of required computing and storage resources for a given time period. If the computation takes longer than expected, jobs are commonly killed at the end of the SLA-defined lifetime. Therefore, users are cautious not to lose their jobs and tend to overestimate their job's execution time and reserve resources accordingly. In practice, this leads to underutilized resources, as jobs finish often much earlier than expected.

Overbooking is used in many commercial fields, where more people buy resources than actually use them. To improve profit, airlines have e.g. become used to sell seat reservations more than once [2,3]. The number of reservations that will not be used is estimated based on prior experiences and used in the planning processes. This estimated number is only correct with a specific probability. Consequently, if more passengers appear than expected and not enough seats are available in the aircraft, the airline has to pay a penalty to its customers.

Obviously, the objective of overbooking is to improve the expected profit. Instead of selling each seat once, profit can be increased by selling them several times. This opportunity has to be compared with the risk implied by overbooking, i.e. the compensation for the buyer if no seat is available combined with the probability of that event. The best estimation of risk and opportunity will provide the best profit.

In this paper, we propose different *overbooking* strategies for grid, cloud or HPC infrastructure providers to increase their profit and competitiveness. These strategies differ in many aspects from traditional overbooking strategies for aircraft seats or hotel beds. On the one side, the number of concurrent users of a compute resource is smaller than the number of passengers of an airplane, making it harder to predict expected behavior. On the other side, computing jobs can be started nearly anytime, while a plane only takes off once for each flight.

Conservative scheduling strategies, which do not use overbooking, do not accept a job, if the maximum estimated job duration is even slightly longer than any gap in the current schedule. Applying overbooking, the scheduler can assess the risk to place the job in a gap that is smaller than the estimated execution time. For such an *overbooked job*, the probability of failure is no longer only dependent on machine failure rates (as in conservative scheduling), but it also depends on the likelihood that the real execution time of the job is longer than the gap length.

The proposed strategies are based on an analytical model for overbooking that uses the convolution of the probability density functions of the runtime estimates of the jobs to calculate the probability of failure (PoF) for a SLA. When the calculated risk is acceptably small in comparison to the opportunity, the service provider can accept the SLA.

The experimental evaluation of the strategies is based on real job traces, which show the benefits as well as associated risks of overbooking, especially if the customer base is diverse or unknown to the service provider. The job traces are derived from a one year record of job estimates and actual runtimes on a 400 processor cluster at the Paderborn Center for Parallel Computing (PC^2). To focus on the influence of overbooking, we have reduced the dimension of the job traces from a parallel machine to a single resource. However, we plan to generalize the described approach to overbooking in parallel machines.

The paper is organized as follows: In the next section we discuss the technical foundations of our work, followed by a description of our model and strategies for risk-aware overbooking in Section 3. In Section 4, we evaluate the risk and opportunity of overbooking mechanisms for Grid providers.

2 Related Work

This chapter summarizes relevant related work. Firstly, it discusses scheduling approaches in distributed environments, followed by related work on overbooking.

2.1 Scheduling

Most scheduling strategies for cluster systems are based on a first-come first-serve (FCFS) approach that schedules jobs based on their arrival times. FCFS guarantees fairness, but leads to a poor system utilization as it might create gaps in the schedule. The gaps can occur, because each job description does not only contain execution time information, but also information about its earliest starting time / release time. Start times in the future are common for interactive jobs or for jobs which are part of a workflow. Interactive jobs are monitored and adopted by the users and the job runtimes have therefore to be known in advance by the user and have to fit to his working times and schedules. Workflow jobs can in principle run at arbitrary times, but dependencies between sub jobs enforce that the start time of a sub job is after the deadline of the preceding one. Gaps can therefore arise, if a new job arrives with a starting time after the end time of the last job in the current schedule. As standard FCFS schedules jobs strictly according to their arrival times, resulting gaps will remain idle and waste resources.

To increase system utilization and throughput in this scenario *Backfilling* has been introduced [4]. Backfilling schedules a new job not necessarily at the end of the plan, but is able to fill gaps if a new job fits in. The additional requirement for the ability to use backfilling instead of simply FCFS is an estimation about the runtime of each job. This allows to determine if the job fits in a gap in the schedule.

The *EASY* (Extensible Argonne Scheduling sYstem) backfilling approach [4] can be used to improve system utilization. Within EASY backfilling putting a job in a gap is acceptable if the first job in the queue is not delayed. This preserves starvation and leads to an increased utilization of the underling system. However, EASY backfilling has to be used with caution in systems guaranteeing QoS aspects, since jobs in the queue might be delayed.

Therefore, [5] introduced the *conservative* backfilling approach which only uses free gaps if no previously accepted job will be delayed. Thus, conservative backfilling still preserves fairness. Additionally, it is possible to plan a job, this means to determine the latest start time for every job. The latest start time is the time a job starts when all its predecessors use their complete estimated runtime.

Simulations show that both backfilling strategies help to increase overall system utilization [6] and reduce the slowdown and waiting time of the scheduling system. The work also shows that the effect of both described backfilling approaches is limited due to the inaccurate user estimation concerning the runtime of their jobs.

Several papers analyzes the effect of bad runtime estimations to the scheduling performance. The interesting effect is that bad estimations can lead to a better performance [7]. However, references [8] approach to improve the scheduling results by adding a fixed factor to the user estimated runtimes shows no advantage while using real job traces. Therefore, effort has been taken to develop methods to cope with the bad runtime estimations and have more accurate estimations. Reference [9,10,11] tried to automatically predict the application runtimes based on the history of similar jobs. Tsafier et al. present a backfilling approach which uses system-generated runtime predictions instead of given user runtime estimations [12]. The paper presents a scheduling algorithm similar to the EASY approach (called EASY++) that uses system-generated execution time predictions and shows an improved scheduling performance for the jobs' waiting times. The approaches show that automatically runtime prediction can improve backfilling strategies, but the usability of the automatically runtime predicting approaches lack on wrong decisions according the runtime.

The approaches found in literature are not directly applicable to our work. Aim of the shown algorithms is to improve the system utilization as whole and decrease the slowdown of the single jobs. They assume to run in a queuing based scheduling system which tries best effort and does not deal with strict deadlines, thus SLA are not usable. Our work instead assumes a planning based scheduling scenario with strict SLAs and tries to improve a providers profit by overbooking ressources.

2.2 Overbooking

Overbooking is widely used and analyzed in the context of hotels [13] or airline reservation systems [3,2]. However, overbooking of grid or cloud resources significantly differs from those fields of applications. The grid does not require fixed starting times for a resource, while e.g. a seat in an airplane cannot be occupied after the aircraft has taken off. As a consequence, results and observations from overbooking in the classical application fields cannot be reused for grid computing.

2.3 Use of Overbooking for Planning and Scheduling

Overbooking for web and internet service platforms is presented in [14]. It is assumed that different web applications are running concurrently on a limited set of nodes. The overbooking approach is based on firstly deriving an accurate estimate of application resource needs by profiling applications on dedicated nodes, and then by using these profiles to guide the placement of application components onto shared nodes. By overbooking cluster resources in a controlled fashion, the approach is able to provide performance guarantees to applications even when overbooked. The difference between this and our approach is that nodes are typically exclusively assigned. Therefore, it is at least difficult to share resources between different applications, while it is possible to use execution time length overestimations, which are not applicable for web hosting.

Overbooking for high-performance computing (HPC), cloud, and grid computing has been proposed in [15] or [16]. However, the references only mention the possibility of overbooking, but do not propose solutions or strategies. In the grid context, overbooking has been integrated in a three-layered negotiation protocol [17]. The approach includes the restriction that overbooking is only used for multiple reservations for workflow sub-jobs, which were queried by the negotiation protocol for optimal workflow planning. Chen et al. [18] use time sharing mechanisms to provide high resources utilization for average system and application loads. At high load, they use priority based queues to ensure responsiveness of the applications. Sulisto et. al [19] try to compensate no shows of jobs with the use of revenue management and overbooking. However they do not deal with the fact that jobs can start later and run shorter than estimated.

Nissimov and Feitelson [20] introduce a probabilistic backfilling approach where user runtime estimations and a probabilistic assumption about the real finish of the job allows to use a gap smaller as the estimated execution time. A job is allowed to be backfilled if the probability that the backfilling postpones the start of the rst job in the queue is less than a given trashold. If the job runs longer than the gap size the following jobs are delayed. In scope of assessing the success of putting a job in a gap the probabilistic backfilling approach is similar to our overbooking scenario. The difference in the concepts are that the acceptance test of [20] is applied to an already scheduled job with aim to reduce its slowdown while in our approach each job has a deadline and the acceptance test is applied at arrival time to determine if the job could be executed before the deadline.

We propose to combine backfilling with overbooking concerning the acceptance test in a commercial scenario with focus on increasing profit of a grid or could provider. This instrument should increase system utilization and should not affect already planned jobs. To successfully use overbooking strategies, we have to be able to calculate the risk of violating SLAs and therefore we have to know the distribution of job execution time overestimations. The probability of success (PoS) for overbooking can then be calculated based on the likelihood that the job finishes in the given gap and the chance that the resource lives at the beginning of the planned execution and survives the job.

3 Overbooking Model and Algorithms

This section depicts the details of the applied overbooking model and our algorithmic approach, which is presented in Section 3.1. As we want to improve the profit by overbooking certain time slots, we need to estimate the PoF and PoS, which is (1- PoF), for each overbooked schedule. The accuracy of the PoF depends on the quality of the predicted job execution times (see Section 3.2).

Overbooking means to put a job in a gap in a schedule that is smaller than the job's maximum execution time. In fact, this job may actually try to use more time than the available size of the gap, leading either to a loss of this job or to a postponement of the following job. Both cases might lead to a penalty for the service provider. We assume a system with a single resource that has a failure rate λ and a repair rate μ, which are distributed according to a Poisson

Table 1. Job scheduling information

Variable	Content	Comment
r	release time	earliest start time
ω	estimated execution time	given by the user
ddl	deadline	given by the user
s	start time	planned start time
f	finish time of the job	planned job end

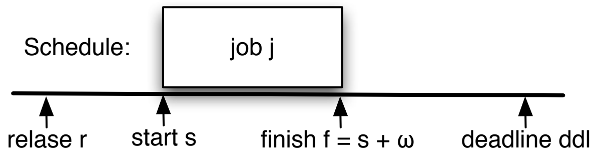


Fig. 1. Example of job information in a schedule

distribution. A job j has an earliest release time r, an estimated execution time ω, and a deadline *ddl* (see also Fig. 1). When the job is placed, the start time s is either its release time or the finish time of the previous job. The finish time f is important if the scheduling strategy follows conservative backfilling, where the job should not delay following jobs. Therefore, the job will be killed at $f = s_{\text{next}}$.

3.1 Overbooking Algorithm

This paragraph briefly defines possible scheduling strategies for backfilling with overbooking. Generally, the scheduler holds a list of all jobs in the schedule. For each new job j_{new} arriving in the system, the scheduler computes the PoS for the execution of this job in every space free in the schedule where the job might be executed. For the concrete implementation of the scheduling algorithm, several decision strategies could be applied.

- A conservative approach could be chosen, where the job is placed in the gap with the highest PoS.
- A best fit approach uses the gap providing the highest profit, while still ensuring an acceptable PoF.
- A first fit approach, where the job is placed in the first gap with an acceptable PoS.

If the job is not placeable within the schedule, it can be planned as last job, if it is still executable before the user given deadline of the job. The calculation of the PoS is defined in Section 3.3 and decent PoF threshold values are evaluated in Section 4. In this paper we will further investigate an overbooking strategy based on first fit.

3.2 Job Execution Time Estimations

Evaluations of job execution time estimations ω and their corresponding real execution times x show that the job duration is typically overestimated by a factor of two to three [21]. A closer look on job traces shows that the distribution of the actual to estimated execution times seems to be uniform and has two peaks a the beginning and end of the spectrum forming a bathtub [6]. The first peak can be explained by jobs missing their input data and test jobs. The second peak includes 15% to nearly 30% of the jobs, which *underestimate* their execution time and which are killed after the negotiated execution time. The theoretical

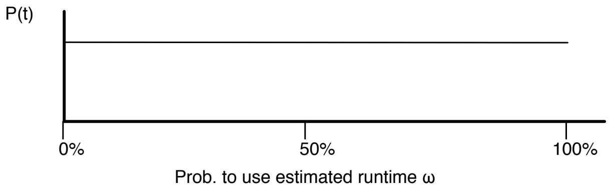


Fig. 2. The probability density function for a single job

examples inside this paper assume the execution time distribution of the jobs to be uniform (see Fig. 2). However, it is important to notice that all results inside this paper do not depend on the shape of the execution time distributions. The simulations shown in Section 4 therefore also use other distributions that are derived from real job run traces.

For the calculation of the probability that the job 2 ends at time t, it is necessary to calculate the expected joint probability density function for the execution time distribution for job 2 and its predecessor job 1, $P_1(t)$ and $P_2(t)$. If jobs are scheduled following to each other, three different cases can be distinguished:

Jobs do not overlap. The jobs are not interfering if $r_2 > f_1$. In this case, the original probability distribution remains unchanged for each job.

Jobs are overlapping and have the same release time. The expected joint execution time distribution of two jobs $P_1(t)$ and $P_2(t)$ with the same release time can be calculated by a convolution of the execution time distributions of the two jobs.

The convolution leads to a distribution as shown in Fig. 3, if $\omega_1 < \omega_2$ and it leads to a simple triangle, if $\omega_1 = \omega_2$. If three or more jobs with the same release time r are convolved, the resulting distribution converges against a Gaussian distribution.

Jobs can overlap. The next job j_2 has a release time r_2 with $r_1 < r_2 < r_1 + \omega_1$. Here the probability distribution of job 2 has only to be convoluted with $P_1(t)$ for

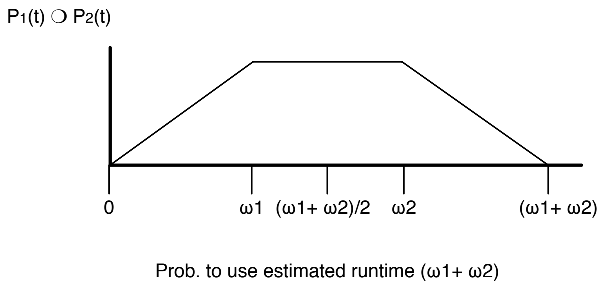


Fig. 3. The probability density function for two jobs with same release time

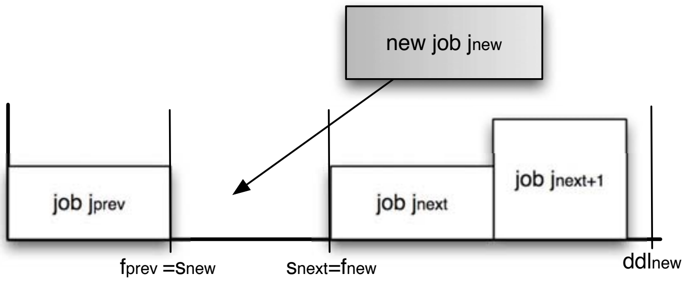


Fig. 4. The job can only be accepted using overbooking

$t > r_2$. Furthermore, the probability p to end job 1 before r_2 has to be multiplied with the distribution of the original distribution of the job 2 and added to the convolution.

$$P_2^{new} = p \cdot P_2 + P_2 \circ P_1(t > r_2)$$

In most cases, the convolution has to be calculated numerically, as no (reasonable) closed formula exists.

3.3 PoS for Overbooking

The probability to successfully complete an overbooked job depends on the probability of a machine failure and the probability of the new job to finish in time. To finish in time means that the job has an execution time that fits into a gap between j_{prev} and j_{next}.

For the calculations we will define a job with a tuple $[r, \omega, ddl]$.

PoS(j_{new}). The probability PoS(j_{new}) depends on the probability $P_{\text{available}}(s)$ that the resource is operational at start time s, the probability $P_{\text{executable}}(j_{\text{new}})$ that the job is able to end with its given maximum execution time, and $P_{\text{success}}(j_{\text{new}})$ which is given by the machine failure rate λ and the job's execution time.

$$\text{PoS}(j_{\text{new}}) = P_{\text{available}}(s) \cdot P_{\text{executable}}(j_{\text{new}}) \cdot P_{\text{success}}(j_{\text{new}})$$

$P_{\text{available}}(s)$. The probability that the resource is operational at the start time is

$$P_{\text{available}}(s) = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + \frac{1}{\mu}} = \frac{1}{1 + \frac{\lambda}{\mu}}$$

where MTTF is the mean time to failure $\frac{1}{\lambda}$ and MTTR is the mean time to repair $\frac{1}{\mu}$.

***$P_{\text{executable}}(j_{\text{new}})$*.** The calculation of the probability to execute successful $P_{\text{executable}}(j_{\text{new}})$ is described in in detail in Section 3.2. If the job j_{new} has no predecessor it is scheduled at its release time and $P_{\text{executable}}(j_{\text{new}})$ is given by the execution time distribution and the maximal execution time x of the job. For a uniform distribution it holds that $P_{\text{executable}}(j_{\text{new}}) = \frac{x}{\omega}$. If $x = \omega$ then $P_{\text{executable}}(j_{\text{new}}) = 1$.

If the job j_{new} has a direct predecessor j_{prev} the convolution of the execution time distribution has always to be computed with the previous job's distribution. The reason for calculating the distribution with the previous job results from the fact, that due to overbooking the job j_{new} has no defined start time any more. j_{new} starts at the end of its predecessor j_{prev}. As the distribution of j_{prev} already includes the distributions from all possibly influencing previously planned jobs the convolution has only to be done with j_{prev}. $P_{\text{executable}}(j_{\text{new}}) = 1$ if the job has its full estimated execution time ω available and less if the job is overbooked and $s + \omega < ddl$.

***$P_{\text{success}}(j_{\text{new}})$*.** The probability that the job's resources survive the execution time is given by their failure probability which is determined by the machine failure rate λ and the job's execution time x. $P_{\text{success}}(j_{\text{new}}) = e^{-\lambda \cdot x}$

Decision Making. During SLA negotiation a simple equation can decide whether it is beneficial to accept an SLA with overbooking or not:

If$(\text{PoS}_{SLA} \cdot \text{Charge}_{SLA} > \text{PoF}_{SLA} \cdot \text{Penalty}_{SLA})$ accept the SLA.
else reject the SLA.

Nevertheless, in the following we will use an overbooking strategy that bases its decision only on a threshold for the probability of failure PoF to investigate the influence of different, more or less aggressive strategies. If the ratio between charges and penalties is constant, then it is possible to derive the results for the presented decision making strategies by simply setting the threshold to the learned best ratio.

4 Evaluation

This section evaluates the possible additional profit that a provider can earn with overbooking. Generally, the expected profit for a job $E[\text{profit}_{job}]$ for overbooking is:

$$E[\text{profit}_{job}] = \text{Charge}_{job} \cdot \text{PoS}_{job} - \text{Penalty}_{job} \cdot \text{PoF}_{job}$$

As this section will show, the additional profit strongly depends on the quality of the prediction of the execution times.

4.1 Simulation Model

Several parameters influence the simulation results:

Simulation Resources. Actually, only a single resource is considered. We plan to extend the simulation to be able to cope with $n \in \mathrm{N}$ resources.

Simulation Metrics. For simplification, the simulation considers the relation of charge to penalty as equal, of for each booked time unit. However, Section 4.8 shows that the relation of charges to penalties does not affect PoF values for maximum profit.

Job Creation Model

– The average job length is exponentially distributed with a mean of 72 time units.
– The earliest release time of the jobs also follows an exponential distribution with a mean of 30 time units which is added to the release time of the previously created job.
– After the creation process for the jobs, their release times are monotonously increasing in the job number. But an increasing order of release times would simply favor FCFS and would not be realistic according to our scenario. Therefore, the order is afterwards randomly permuted to create a more realistic release time distribution.
– The deadline of each job is set to its release time plus five times the estimated execution time ($ddl = r + 5 * \omega$).

The mean of the release time distribution compared to the mean of the job length directly describes possible load of the system. If the mean of the release times is bigger as the mean job length more jobs are arriving as feasible in the scheduling system. The chosen simulation parameters enforce that more jobs are submitted than the system could successfully execute. Each simulation ends after the deadline of the last accepted job.

Resources Stability

– The resource's MTTF is 14505 time units or $\lambda = 0.00165$.
– The resource's MTTR is 12 time units or $\mu = 0.0833$.

These assumptions are taken for most of the shown simulation. We will additionally show in Section 4.8 that different MTTF values only lead to an offset in the profit curve for the overbooking strategies.

Simulations Usage. For each test run, the incoming jobs, job length, and release times as well as the up and downtime of the resource have been randomly chosen. Based on this input data, the three strategies have been applied and the results of the strategies have been evaluated and aggregated to the following figures. For each test point, we have performed 10000 runs with 100 incoming jobs per run. Therefore, every result is based on 600000 schedules.

Execution Time Distribution. The required execution time of a job is calculated based on the applied execution time distribution. We have evaluated four different distribution schemes. The first one is the simple uniform distribution. The second one is a bathtub distribution, which seems most realistic if applied to a huge number of jobs and users [6]. This bathtub curve could also be derived from the traces of our local cluster system. The next execution time distributions are derived from traces of two dedicated users of our cluster system. They should depict the fact that given enough information (traces from jobs) of a single user, better overbooking results could be achieved.

Simulations result. We calculate a threshold P_{max} that will provide the maximum PoF acceptable by the scheduler for different situations. To evaluate the best threshold for overbooking, we have implemented FCFS and conservative backfilling to be able to compare the overbooking strategies with standard implementations. The overbooking strategy of accepting jobs is based on the PoF given by the convolution of the execution time distribution with the distribution of the previous jobs. A job is placed in the first gap where the calculated PoF is lower than $P_{\max}$.

4.2 Uniform Execution Time Distribution

The evaluation starts with the assumption that the execution time distribution follows an uniform distribution. The simulation results for each test run contain the charge and the penalty of each strategy. The profit of a test run is its charge minus the penalty. Penalties for FCFS and backfilling without overbooking can only occur, if the machine fails during the execution of a job (see Figure 5).

Depicted results shown in the figure 5 are

– y-axis *charge* and *penalty* for each strategy
– x-axis maximal PoF $P_{\mathbf{max}}$ accepted

In the Figure 6 and following figures showing simulation results are

– y-axis *profit.*
– x-axis maximal PoF $P_{\mathbf{max}}$ accepted

The simulation starts with a maximum acceptable PoF for a job of 0.05 and ends with 1. The x-axis has, besides fluctuations of the randomized measurement parameters, no influence on FCFS and backfilling without overbooking. Markers around the values in the figures show the 95 % confidence intervals.

For a uniform execution time distribution, the FCFS strategy fills on average 1100 time units per schedule, while the backfilling strategy has a mean planned execution time of 2200 time units per schedule. The overbooking strategy is able to fill on average 2570 time units, depending on the accepted PoF $P_{\max}$ and varying from 2440 to 2630 time units per schedule. It is also shown in Figure 5 that the penalties for overbooking start lower than for conservative backfilling. This is caused by the fact that conservative backfilling is mostly unable to restart a job after a resource outage, because the remaining time slot is not long enough. In contrast, overbooking still has the opportunity to fill the new, smaller gap in the schedule.

Due to space limitations, we will only present the accumulated profit in the following. The profits with overbooking are, until a $P_{\max} = 0.8$, better than

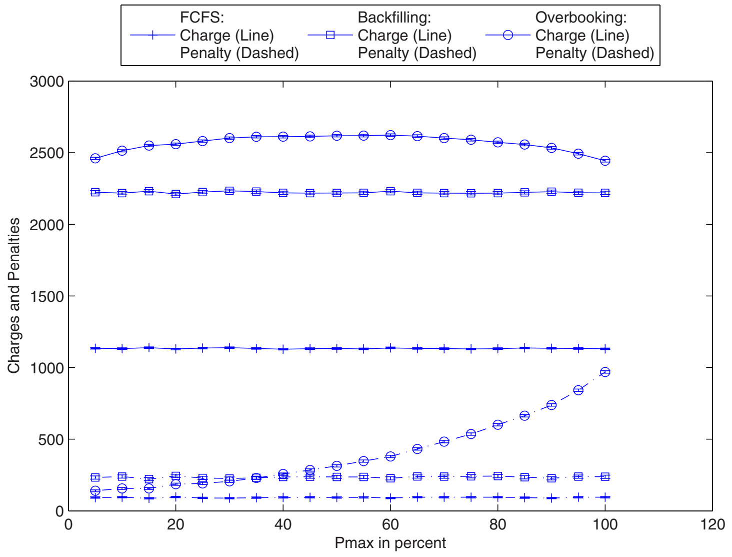


Fig. 5. Charge and penalties with different scheduling strategies

with simple backfilling (see Figure 6). Overbooking strongly depends on $P_{\max}$. The profit is increasing at the beginning due to additionally accepted jobs and is shrinking at the end again due to the increasing amount of violated SLAs caused by too high accepted values of $P_{\max}$. For these assumptions, $P_{\max} = 0.3$ should be chosen to maximize profit, increasing the profit by 20 % compared to a conservative backfilling strategy.

4.3 Bathtub Distribution

The following simulations are based on a job execution time analysis of the year 2007 for the Arminius HPC cluster system at the Paderborn Center for Parallel Computing (PC^2). Within the analyzed 23286 jobs, 6109 jobs used less than 1 % of the booked execution time, while 3553 jobs used 100 %. For the simulation, we have assumed that 26 % of the jobs have zero execution time, 15 % use 100 % percent of their execution time and 59 % of the jobs are uniformly distributed in between.

Figure 8 shows the profit similar to Section 4.2. FCFS andgad conservative backfilling do not dependent on the quality of the runtime estimations and we will not discuss their behavior again. The overbooking strategy fills on average 2490 time units for each schedule, varying from 2330 to 2610 time units for different $P_{\max}$. It is shown in Figure 8 that the maximum profit after subtracting the penalties is achieved for $P_{\max} = 0.6$. Starting with $P_{\max} = 0.9$, overbooking induces negative effects. The profit is increased compared to a conservative backfilling strategy by 19 % for $P_{\max} = 0.6$.

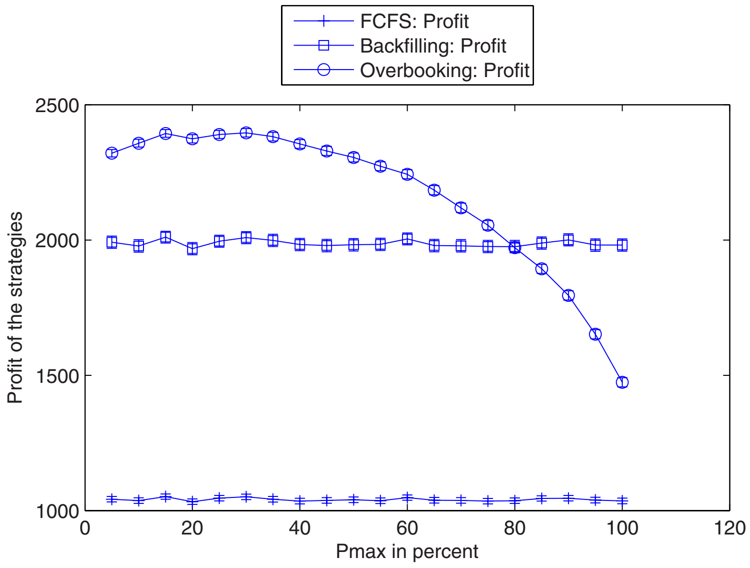


Fig. 6. The accumulated profit for uniform execution time distributions

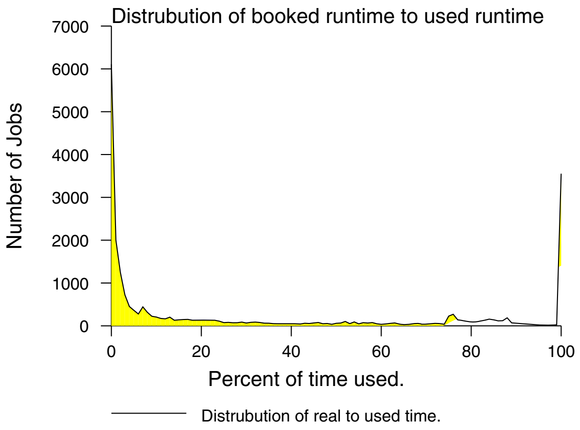


Fig. 7. Execution time distribution on a real cluster system for the year 2007

4.4 Simulations with Precise Execution Time Estimations

This section analyses overbooking approaches for customers with very precise execution time estimations. The simulation uses input data of a user which nearly always uses 88.2 % of the reserved execution time. There are some jobs which have been killed due to missing input data and some which have underestimated their required execution time (see Figure 9).

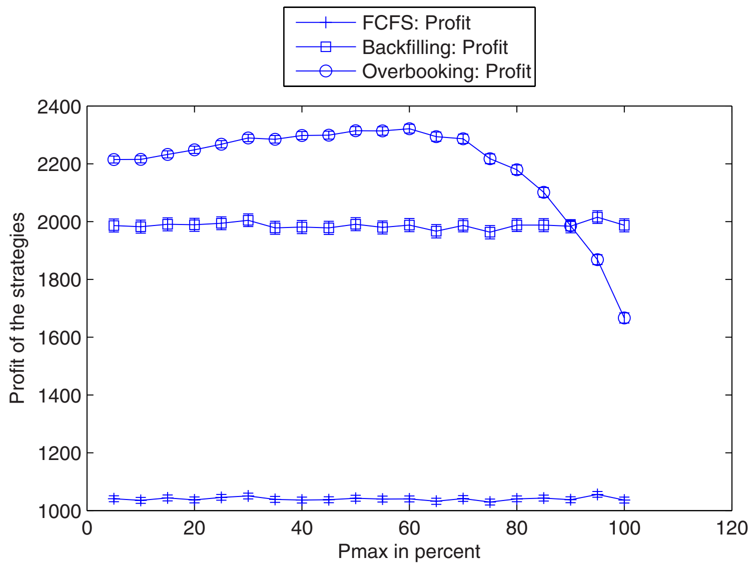


Fig. 8. Profit with different scheduling strategies for a bathtub distribution

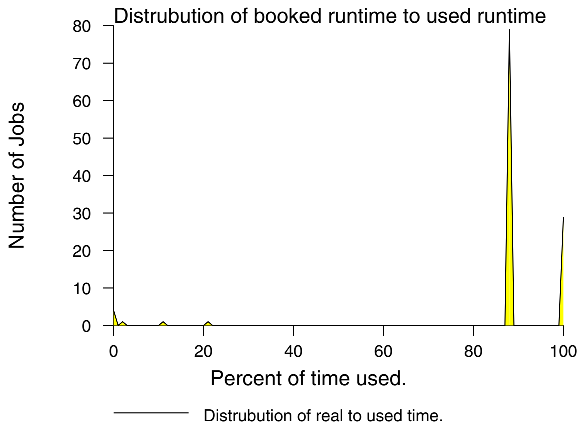


Fig. 9. Execution time distribution with a peak at 88 % for a single user

The results of this simulation are shown in Figure 10. The overbooking strategy fills on average 2370 time units, varying from 2320 to 2390 time units per schedule. At $P_{\max} = 0.35$ the maximum profit is available. Then, until $P_{\max} = 0.95$, the additional profit is nearly stable. Starting from $P_{\max}$ of 0.95, overbooking jobs for this customer has a negative effect. The profit is increased compared to a conservative backfilling strategy by 13 % for $P_{\max} = 0.35$.

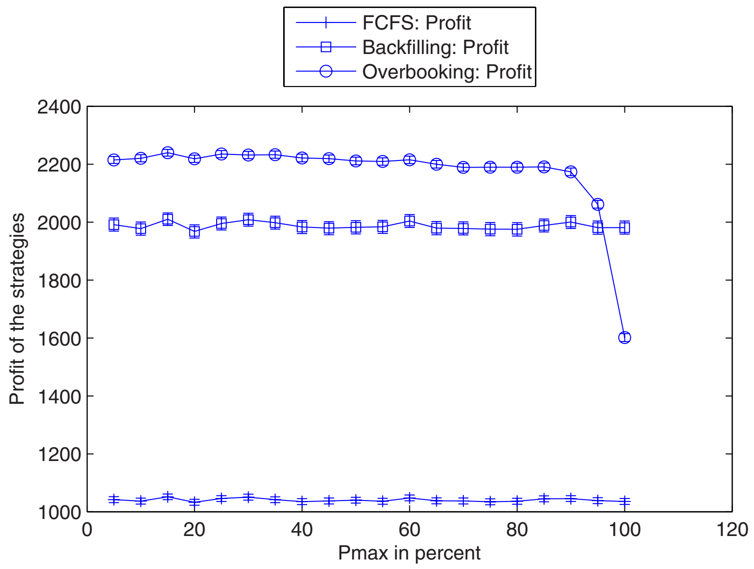


Fig. 10. Profit with different scheduling strategies for precise user estimations

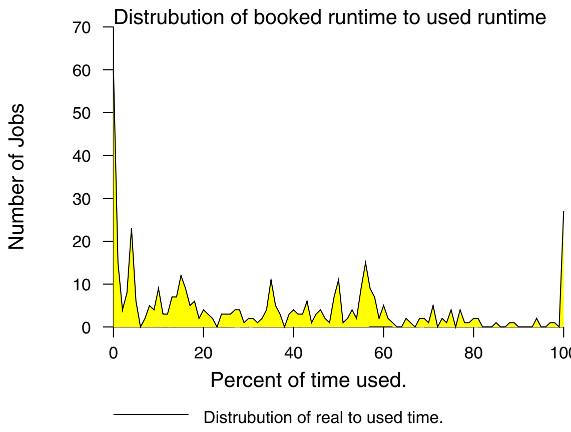


Fig. 11. Job execution time distributions with a near bathtub characteristic

4.5 Execution Time Distributions with Near-Bathtub Behavior

The following simulations are based on a user profile, where the user rarely uses more than 60 % of the reserved execution time. There is a peak in the distribution at zero and at 100 % and a high probability that the user consumes between 1 % to 60 %. This interval includes 85 % of all jobs and the interval from 60 % to 99 % only contains 5 % of the jobs (see Figure 11). The overbooking strategy fills on average 2660 time units, varying from 2340 to 2780 time units per schedule. The maximum profit is achieved for $P_{\max} = 0.4$. Then, until $P_{\max} = 0.9$, the

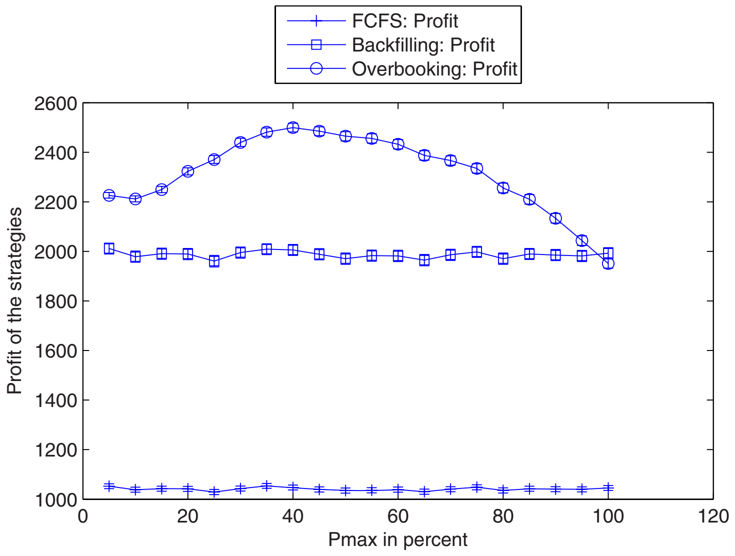


Fig. 12. Simulation results for user with most of the jobs gathered between 1 % to 60 % of the execution time

additional profit is nearly stable. Starting from $P_{\max} = 0.97$, overbooking jobs of this customer has a negative effect. The profit is increased compared to a conservative backfilling strategy by 22 % for $P_{\max} = 0.4$.

4.6 Uniform vs. Bathtub

In the previous sections, the calculation for the overbooking strategies have been based on the same runtime distributions as the simulated jobs. We will investigate in the following sections the influence of imprecise runtime estimations on the quality of the scheduling. Figure 13 shows the case in which the overbooking strategy assumes a uniform job length distribution, while the simulated jobs behave according to bathtub distribution from section 4.3.

The overbooking strategy fills on average 2570 time slots, varying from 2450 to 2610 time units per schedule. The maximum profit can be achieved for $P_{\max} = 0.15$. Starting from $P_{\max} = 0.8$, overbooking jobs of this customer has a negative effect. The profit is increased compared to a conservative backfilling strategy by 19 % for $P_{\max} = 0.15$. For these comparable probability density functions, overbooking is still able to produce very good results.

4.7 Uniform vs. Peak

Now, the simulation evaluates a very different user behavior compared to the runtime prediction. The simulation still assumes a uniform distribution, while the simulated jobs are created according to the peak distribution from Section 4.4.

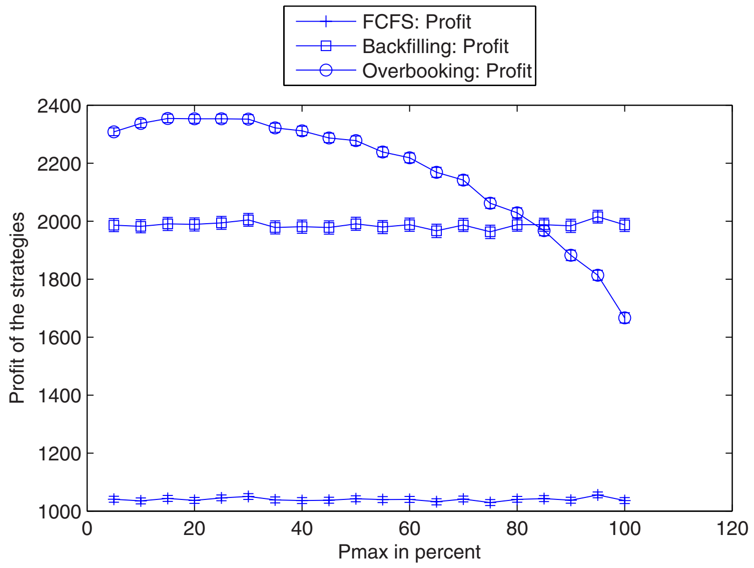


Fig. 13. Here a bathtub execution time distribution occurs while the scheduler assumes a uniform distribution

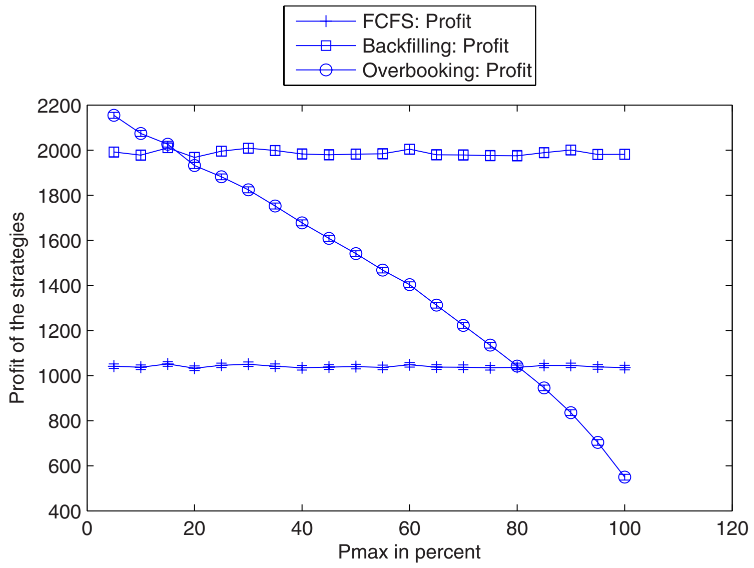


Fig. 14. Here a peak execution time distribution occurs while the scheduler assumes a uniform distribution

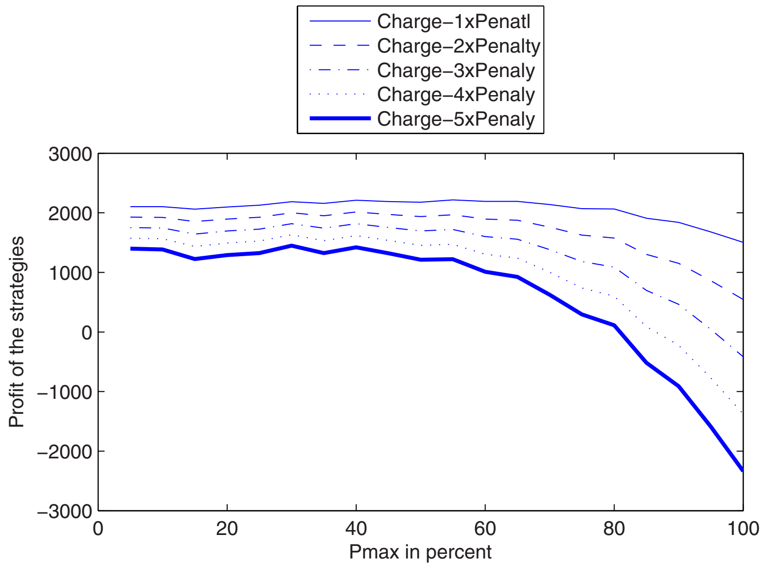


Fig. 15. The resulting profit with different penalty factors

The overbooking strategy fills on average 2210 time units per schedule, varying from 1980 to 2380 time units per schedule. At $P_{\max} = 0.05$ the maximum profit is available there the profit is increased compared to a conservative backfilling strategy by 9 %. Already starting from $P_{\max} = 0.15$, overbooking jobs of this customer has a negative effect and starting from $P_{\max} = 0.8$ the profit is worser than with FCFS.

It is clear that in this case overbooking has a very bad impact on the schedule, as the user does a very exact assessment and the overbooking assumes a uniform distribution. The result shows that for users which are able to accurately predict their job runtimes, overbooking has to be applied very carefully.

4.8 Dependency on Penalty and MTTF

The previous simulation results assumed equal profit and penalty for each time unit. Figure 14 contains results for different ratios of profit and penalty, starting from factor one and going up to a factor of five. As input we have chosen the results of the measurement of Section 4.3. It is clear that an increased penalty decreases the achievable profit. However, the shape of the curves is not affected.

Figure 16 shows additional simulations concerning resource stability. They were performed to evaluate the impact of the machine failure rates on the predictions for overbooking. The result shows that the value of $P_{\max}$ is nearly independent of the machine outage characteristics.

The evaluation shows that the impact of overbooking in Grid, Cloud, or HPC environments is dependent on the accuracy of the underlying assumptions of the

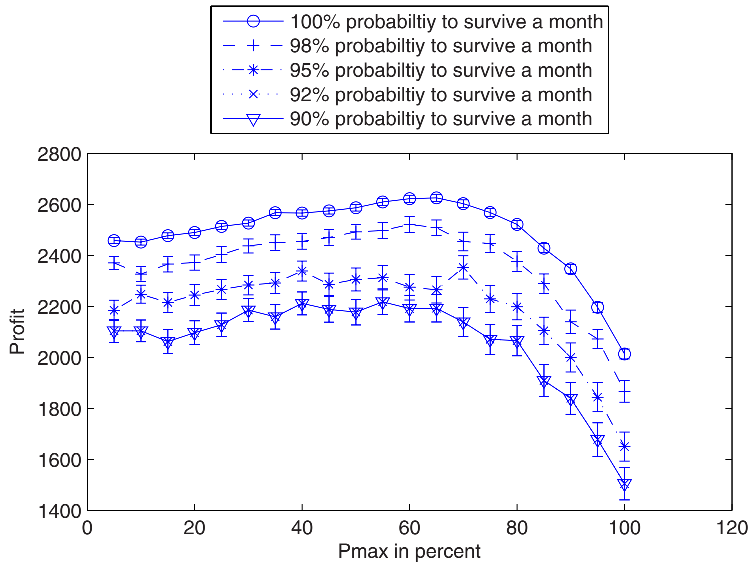


Fig. 16. The resulting profit with different penalty factors

relation of booked to real used runtime. With the given data of a real cluster system and assuming SLA negotiations, it is possible to increase the profit of a cluster system by 20 %. Furthermore, the simulations show that assuming accurate assumptions for user's runtime estimations the profit of a cluster system can be further increased. On the other hand, incorrect assumptions can have a negative impact on the profit.

5 Conclusion and Future Work

This paper has motivated the need for overbooking in Grid, Cloud or HPC environments and outlined the limitations of current scheduling algorithms. Thereafter, the idea of using overbooking to increase the ability to accept more SLAs has been shown. As overbooking increases the risk of SLA violations, mechanisms for determining whether or not it is worthy to use overbooking have been shown followed by an evaluation of the impact from the proposed methods on the ability to successfully accept additional SLAs. Therefore, a threshold P_{max} has been defined with which a provider can maximize the profit for a overbooking strategy. The evaluation shows that the additional profit depends on the accuracy of the underlying runtime estimations and can be given real runtime distributions around 20 % of additional utilisation.

An interesting point of future work will be to define overbooking strategies which will be able to combine multiple resources and that can be used for parallel jobs. It might also be interesting to determine if there are user and application specific distributions which would allow to increase the quality of the risk estimations for overbooking. With this knowledge, the quality of the estimations could be further increased.

Acknowledgment

The authors would like to thank the EU for partially supporting this work within the 6th Framework Programme under contract IST-031772 "Advanced Risk Assessment and Management for Trustable Grids" (AssessGrid).

References

1. Battre, D., Hovestadt, M., Kao, O., Keller, A., Voss, K.: Increasing fault tolerance by introducing virtual execution environments. In: Proceedings of the 1. GI/ITG KuVS Fachgespräch Virtualisierung (2007)
2. Rothstein, M.: Or and the airline overbooking problem. Operations Research 33(2), 237–248 (1985)
3. Subramanian, J., Stidham, S., Lautenbacher Jr., C.J.: Airline yield management with overbooking, cancellations, and no-shows. Transportation Science 33(2), 147–167 (1999)
4. Feitelson, D., Jette, M.: Improved utilization and responsiveness with gang scheduling. In: Proceedings of the Job Scheduling Strategies for Parallel Processing: IPPS 1997 Workshop, Geneva, Switzerland (April 5, 1997)
5. Feitelson, D., Weil, A.: Utilization and predictability in scheduling the ibm sp2 with backfilling. In: Proceedings of the 12th International Parallel Processing Symposium (January 1998)
6. Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp 2 with backfilling. IEEE Transactions on Parallel and Distributed Systems 12(6), 529–543 (2001)
7. Zotkin, D., Keleher, P.: Job-length estimation and performance in backfilling schedulers. In: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing HPDC, (January 1999)
8. Tsafrir, D., Feitelson, D.: The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In: Proceedings of the IEEE International Symposium on Workload Characterization (2006)
9. Gibbons, R.: A historical application profiler for use by parallel schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291. Springer, Heidelberg (1997)
10. Smith, W., Foster, I., Taylor, V.: Predicting application run times using historical information. In: Proceedings of the Job Scheduling Strategies for Parallel Processing JSSPP (January 1998)
11. Tsafrir, D., Etsion, Y., Feitelson, D.: 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)
12. Tsafrir, D., Etsion, Y., Feitelson, D.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Transactions on Parallel and Distributed Systems (TPDS), 789–803 (2007)

13. Liberman, V., Yechiali, U.: On the hotel overbooking problem-an inventory system with stochastic cancellations. Management Science 24(11), 1117–1126 (1978)
14. Urgaonkar, B., Shenoy, P.J., Roscoe, T.: Resource overbooking and application profiling in shared hosting platforms. In: Proceedings of the 5th Symposium on Operating System Design and Implementation, OSDI (2002)
15. Andrieux, A., Berry, D., Garibaldi, J., Jarvis, S., MacLaren, J., Ouelhadj, D., Snelling, D.: Open issues in grid scheduling. UK e-Science Report UKeS-2004-03 (April 2004)
16. 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)
17. Siddiqui, M., Villazón, A., Fahringer, T.: Grid allocation and reservation - grid capacity planning with negotiation-based advance reservation for optimized qos. In: Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, p. 103 (2006)
18. Chen, M., Wu, Y., Yang, G., Liu, X.: Efficiently rationing resources for grid and p2p computing. In: Jin, H., Gao, G.R., Xu, Z., Chen, H. (eds.) NPC 2004. LNCS, vol. 3222, pp. 133–136. Springer, Heidelberg (2004)
19. Sulistio, A., Kim, K.H., Buyya, R.: Managing cancellations and no-shows of reservations with overbooking to increase resource revenue. In: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), pp. 267–276 (2008)
20. Nissimov, A., Feitelson, D.G.: Probabilistic backfilling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 102–115. Springer, Heidelberg (2008)
21. Streit, A.: Self-tuning Job Scheduling Strategies for the Resource Management of HPC Systems and Computational Grids. PhD thesis, University of Paderborn (2003)

Modeling Parallel System Workloads with Temporal Locality

Tran Ngoc Minh and Lex Wolters

Leiden Institute of Advanced Computer Science
Leiden University, 2333 CA, Leiden, The Netherlands
minhtn@liacs.nl, llexx@liacs.nl

Abstract. In parallel systems, similar jobs tend to arrive within bursty periods. This fact leads to the existence of the locality phenomenon, a persistent similarity between nearby jobs, in real parallel computer workloads. This important phenomenon deserves to be taken into account and used as a characteristic of any workload model. Regrettably, this property has received little if any attention of researchers and synthetic workloads used for performance evaluation to date often do not have locality. With respect to this research trend, Feitelson has suggested a general repetition approach to model locality in synthetic workloads [6]. Using this approach, Li et al. recently introduced a new method for modeling temporal locality in workload attributes such as run time and memory [14]. However, with the assumption that each job in the synthetic workload requires a single processor, the parallelism has not been taken into account in their study. In this paper, we propose a new model for parallel computer workloads based on their result. In our research, we firstly improve their model to control locality of a run time process better and then model the parallelism. The key idea for modeling the parallelism is to control the cross-correlation between the run time and the number of processors. Experimental results show that not only the cross-correlation is controlled well by our model, but also the marginal distribution can be fitted nicely. Furthermore, the locality feature is also obtained in our model.

1 Introduction

Parallel systems from supercomputers to clusters and grids have become more and more popular for solving many problems not only in scientific computing but also in industry. To enable effective resource allocation on these systems, numerous schedulers have been built such as Maui [5] and KOALA [15]. The quality of schedulers depends on their algorithms and has a considerable impact on the overall performance of parallel systems. Hence, the evaluation of scheduling algorithms is an essential part to build a scheduler. The accuracy of the evaluation highly relies upon workloads applied, where real workloads (called traces) are usually chosen because they reflect the system reality. However, there are several reasons showing that workload models have a number of advantages over traces [24]. Workload modeling creates a general model which can be used to generate synthetic workloads.

Arrival time, run time and parallelism (the number of processors) are three important workload attributes necessary to be modeled to apply for studies on performance evaluation. While the arrival time attribute can be modeled individually, two remaining attributes are more difficult and require to be modeled at the same time because it is proven that there exists a cross-correlation between the run time and the parallelism [13,24]. In [11,23], a multifractal wavelet model is introduced to control the fractal behaviour and the temporal correlation of arrival rate processes. In [24], a combined model is suggested where the inter-arrival times fit a hyper-Gamma distribution and the job arrival rates match the daily cycle. Models for run time and parallelism are also proposed recently based on fitting the marginal distribution [24] or Markov chains to control the cross-correlation between these two attributes [1]. However, it can be seen that although the phenomenon of locality - a persistent similarity between nearby jobs - has been recognized to strongly exist in real parallel computer workloads [6], this important characteristic is not taken into account in the studies mentioned.

With respect to this research trend, Feitelson [6] has proposed a general repetition approach to model locality in synthetic workloads. Using this approach, Li et al. [14] recently introduced a new two-stage method for modeling the run time attribute with a temporal locality feature. The first stage consists of applying the model called *mixture of Gaussians*, whose parameters are estimated via the Model-Based Clustering (MBC) framework [3]. The second stage includes a Localized Sampling (LS) algorithm [14] for generating temporal locality in the data series. However, with the assumption that each job in the synthetic workload requires a single processor, the parallelism has not been taken into account in their model (MBC-LS). Furthermore, we also found that MBC-LS does not control the locality very well to fit the locality of real data. In this paper, we propose a new model for parallel computer workloads based on MBC-LS. In our research, we firstly improve MBC-LS to control the temporal locality feature of a run time process so that the locality of real data is matched better. Then, the parallelism is modeled with the key idea that the cross-correlation between two workload attributes -run time and parallelism- fits the real data as accurately as possible. Experimental results show that not only the cross-correlation is controlled well by our model, but also the marginal distribution can be fitted nicely. Moreover, the temporal locality feature is also obtained by improving and applying MBC-LS in our model.

The rest of this paper is organized as follows. Section 2 describes workload data used in our experiments. MBC-LS and our improvements are presented in section 3. We continue in section 4 to model the parallelism as well as to control the cross-correlation between the run time and the parallelism. Experimental results are shown in section 5. Finally, we conclude in section 6 our study and discuss some future research.

2 Workload Data

Table 1 describes details of the traces used in our study. KTH is from the IBM SP2 machine installed at the Swedish Royal Institute of Technology in Stockholm and is scheduled using the EASY backfilling scheduler [4]. LANL is from the Connection Machine CM-5 installed at Los Alamos National Lab and is scheduled using DJM [8]. SDSC is collected from the San Diego Supercomputer Center Intel Paragon machine whose scheduling is based on NQS [16]. This trace is available under two separate one-year traces, namely SDSC95 and SDSC96. Though the first four traces in Table 1 are rather old, we select them in our experiments for comparison with recent studies [1,24].

Table 1. Traces used in the experiments

Trace	Period	Number of processors	Number of jobs
KTH	09/1996-08/1997	100	28480
LANL	01/1996-09/1996	1024	37517
SDSC95	12/1994-12/1995	416	53885
SDSC96	12/1995-12/1996	416	32032
LLNLATLAS	02/2007-05/2007	9216	23911
GRID5000	07/2006-08/2006	3216	42171

Newly collected traces are also considered in our study, including LLNLATLAS and GRID5000. At the cluster level, LLNLATLAS, a large Linux cluster called Atlas installed at the Lawrence Livermore National Lab and scheduled by MOAB [18], is selected. At the grid level, GRID5000 [10], consisting of 9 sites with a total of 15 clusters geographically distributed in France, is chosen. GRID5000 uses OARGrid as grid resource broker and OAR as batch scheduler for its local clusters [19]. Note that this grid trace includes both jobs submitted via the grid resource broker and jobs directly submitted to the clusters. All traces and detailed information are available on [21], except for GRID5000 which are collected from [9].

3 Modeling Job Run Time with Locality

We first show in this section the reason why we need to take into account temporal locality when modeling the run time attribute. Then, we present briefly a two-stage approach [14] to model job run time with the locality feature. The first stage consists of the *mixture of Gaussians* model, whose parameters are estimated via Model-Based Clustering (MBC) framework. The second stage includes a Localized Sampling (LS) algorithm for generating temporal locality in the data series. In addition, we also describe our improvement on this approach to control locality better.

3.1 Why Locality?

In the effort of improving the performance of parallel systems, several researches on predicting job run time using historical data to provide schedulers with better information have been done [12,22,26]. These studies are based on the belief that the recent past is indicative of the near future. It is a generalization of the idea of locality. Furthermore, we observe that the real workload data is far from independently and identically distributed, instead, similar jobs tends to arrive within bursty periods. Therefore, locality should be taken into account when modeling parallel system workloads to help studies on predicting more accurately.

3.2 Model-Based Clustering

Model-Based Clustering (MBC) [3] is a methodological framework that underlies a powerful approach not just to data clustering but also to discriminant analysis and multivariate density estimation. Instead of looking for a single probability density function for the distribution of the data, the main idea of MBC is that it considers the data as generated by a mixture of normal (Gaussian) probability density functions, where each function represents a different cluster[1]. The selection of the number of clusters is based on the Bayesian information criterion. Gaussian parameters for these clusters are calculated by combining agglomerative hierarchical clustering and the expectation-maximization algorithm for maximum likelihood. MBC is implemented in the MCLUST software and available on [17].

3.3 Localized Sampling

The localized sampling algorithm presented in this section is used to model the locality feature of a job run time process. The concept and phenomenon of temporal locality [20] has been recognized and recently studied to model parallel workloads. To this end, a locality of sampling algorithm has been introduced by Feitelson [6] based on the observation that the lengths of repetitions of equivalent jobs empirically follow a Zipf-like (power law) distribution [7]. This algorithm can be summarized by the following steps:

1. Sample a variate X from the probability distribution of the data.
2. Sample a variate R from the fitted Zipf distribution of repetitions in the data.
3. Repeat the X variate R times to distort the distribution locally.
4. Return step 1 until the desired number of samples has been generated.

Though the locality can be obtained well, this algorithm still has a number of limitations. Firstly, it is not easy to know exactly the probability density function

[1] The term "cluster" stems from the concept of "data clustering". Data clustering is the classification of similar objects into different groups, or more precisely, the partitioning of a data set into subsets (clusters), so that the data in each subset (ideally) share some common trait. (definition from Wikipedia).

of the real data. Hence, sampling for the variate X in step 1 is very difficult and almost infeasible. Secondly, repetition of a single value in step 3 is a simple treatment and more sophisticated techniques are needed for better stochastic approximation. To this end, a localized sampling algorithm has been proposed by Li et al. in [14] to overcome these limitations. They found that not only the repetitions of real data but also the repetitions of cluster labels empirically follow a power law. The classification/cluster labels can be obtained via MBC presented in section 3.2. The idea of their algorithm is that the cluster label series instead of the real data is taken into account to serve as the input for the 4-step procedure above. In this way, the first limitation is solved because the probability distribution of cluster labels is known in advance via MBC. Futhermore, the second limitation is also eliminated because each cluster label represents a cluster of values instead of a single value. Each cluster label in the generated series is converted into a specific value in the final synthetic run time process by sampling the distribution of the corresponding cluster, which is also known in advance via MBC.

Although using MBC to classify data series and applying these classifications to the 4-step procedure above is a good idea, we found that a new limitation occurs with this approach. That is only repetitions of cluster labels follow the power law, but repetitions of the final synthetic run time process do not fit the Zipf-like distribution as the real data any more. It is because the authors use a cluster of values instead of a single value to overcome the second limitation in Feitelson's algorithm. Therefore, we propose a solution to solve this problem. When we repeat a cluster label R times, we also equivalently sample the distribution of that cluster R times to produce R specific values in the final synthetic run time process. Our idea is that instead of sampling the distribution of a cluster R times, we will produce a single value r times with $r < R$ and then sample the distribution of the cluster $R - r$ times. So how to obtain the value r? Observing all the times where a cluster label appears repeatedly in the cluster label series, we recognize that there are periods where no single value is produced repeatedly. Based on this observation, we calculate a probability p to indicate the ability that there is a single value produced repeatedly at a repetition time of a cluster label. As such, r can be calculated by sampling from the fitted Zipf distribution of repetitions in the real data with a probability p. Otherwise, with a probability $1 - p$, r is assigned to be equal to 0.

Combining the idea of Feitelson, the idea of Li et al. and our idea, we summarize the following steps to model the locality feature for a run time process:

1. Run the model-based clustering procedure in section 3.2, where the real run time process $Data_i, i = 1 \rightarrow n$ serves as an input, to obtain *mixture of Gaussians* parameters $(\mu_k, \sum_k; p_k), k = 1 \rightarrow G$ and classifications $L_i \in \{1, \cdots, G\}, i = 1 \rightarrow n$; where G is the number of clusters, μ_k, $\sum_k$ and p_k are mean, variance and probability of cluster k, respectively.
2. Count the lengths of repetitions in L_i and fit them to a Zipf distribution Z_L. For example, if $L_i = \{2, 2, 2, 3, 1, 1, 4, 5, 5, 5, 5\}$, we have a series of lengths of repetitions as $\{3, 1, 2, 1, 4\}$ and fit this series to a Zipf distribution.

3. Count the lengths of repetitions in $Data_i$ and fit them to a Zipf distribution Z_D.
4. Calculate the probability p for the occurence of the repetition of a single value within each repetition in classifications L_i.
5. Generate a series of cluster labels C according to the cluster probability p_k.
6. Set the window size W. Form a series C_σ by applying the cluster permutation procedure[2] . This step is used to control the autocorrelation in the synthetic data and completely independent on the locality. The autocorrelation increases when W is large and in the simplest case, $C_\sigma = C$ when $W = 1$. This step can be bypassed without any impact on the locality by simply setting $W = 1$ if users do not want to control the autocorrelation.
7. Select a cluster label c from C_σ sequentially.
8. Sample a variate R from the fitted Zipf distribution Z_L.
9. Sample a variate $Prob$ from the uniform distribution over the range $[0, 1]$.
10. If $Prob \le p$, sample a variate r from the fitted Zipf distribution Z_D, else assign $r = 0$. Note that the sampling work is done by a loop until we obtain $r < R$.
11. Sample the Gaussian distribution $f_c(\mu_c, \sum_c)$ to obtain a single value and repeat this value r times.
12. Sample the Gaussian distribution $f_c(\mu_c, \sum_c)$ $R - r$ times.
13. Return step 7 until the desired number of samples has been generated.

4 Modeling Parallelism and Control the Cross-Correlation

For most parallel systems, parallelism is another vital workload attribute beside run time. Furthermore, the cross-correlation between it and the run time is also very important. In [25], Lo et al. demonstrated how different degrees of this cross-correlation might lead to discrepant conclusions about the evaluation of scheduling performance. Therefore, we should take into account this cross-correlation when modeling parallel system workloads.

As indicated in [14], despite the fact that the *mixture of Gaussians* model is a good choice for fitting the marginal distribution, it is not suitable for some attributes with discrete values such as parallelism. Hence, we propose in this section a new three-stage approach to model the parallelism as well as control the cross-correlation between it and the run time. Firstly, the parallelism process is classified into a number of classes. However, different from the run time process with continuous values, the parallelism process with discrete values can not be classified via MBC in section 3.2. Rather than, we create a new method for the classification of the parallelism process. Secondly, we control the cross-correlation between the run time and the parallelism by creating and using a transition table. Thirdly, we convert class labels into specific values based on the sample probability.

[2] Hereby we give an example of the cluster permutation procedure, for details see [14]. If we have a series of cluster labels generated in step 5 $C = \{1, 2, 1, 3, 2, 2, 3, 2, 4, 1, 4\}$ and $W = 4$, we deduce $C_\sigma = \{1, 1, 2, 3, 2, 2, 2, 3, 4, 4, 1\}$.

4.1 Classify the Parallelism

Our approach to classify the parallelism is presented in detail in Algorithm 1. We start by grouping jobs that require the same number of processors and count the number of jobs in each group. Then, each group is assigned a label which is an integer calculated by rounding the logarithm of the number of jobs in that group to the base 2 and adding 1. Jobs belong to a group are also classified with its label. As such, groups that have approximately equal quantities of jobs will be assigned the same label. For example, if there are 250 jobs requesting 4 cpus and 300 jobs requesting 10 cpus, all of them will be classified as 9. Note that this classification approach can only be applied on a series with discrete values such as parallelism.

Algorithm 1. Classify the parallelism process. The operator $length(\cdot)$ indicates the length of a series and the operator $round(X)$ rounds X to the nearest integer.

```
Input: A parallelism process $P_i, i = 1 \rightarrow n$.
Output: A classification process $C_i, i = 1 \rightarrow n$ where $C_i$ indicates the class to which
$P_i$ belongs.

Assign $maxcpus = max(\{P_i, i = 1 \rightarrow n\})$;
for $j = 1$ to $maxcpus$ do
    Calculate the number of occurences of $j$ in $\{P_i\}$: $count_j = length(\{x = j, x \in \{P_i\}\})$;
    if $count_j \neq 0$ then
        $count_j = round(log_2(count_j)) + 1$;
    end if
end for
for $i = 1$ to $n$ do
    $C_i = count_{P_i}$;
end for
```

4.2 Control the Cross-Correlation

We use Algorithm 2 to control the cross-correlation between the run time and the parallelism. Firstly, we calculate the transition conditional probability table $Pr(c, l)$, where c and l are labels of the parallelism and the run time, respectively. $Pr(c, l)$ indicates the probability for a job to have the parallelism label c with the condition that the label for its run time is known in advance as l. $Pr(c, l)$ of a job is calculated by the ratio between the probability $P(c, l)$ for that job to have the parallelism label c and the run time label l at the same time and the probability $P(l)$ for that job to have the run time label l. Secondly, we form a series of parallelism labels based on the transition probability table $Pr(c, l)$. Each parallelism label corresponds to a cluster label in the series CW. We obtain CW by using C_σ and repeating each cluster label in C_σ R times. Reminding that in the algorithm presented in section 3.3, C_σ is formed in step 6 by applying the cluster permutation procedure. Each cluster label from C_σ is selected and

Algorithm 2. Create a series of parallelism labels

```
Input: Classifications of the run time process $L_i, i = 1 \rightarrow n$ obtained via MBC
in section 3.2, classifications of the parallelism process $C_i, i = 1 \rightarrow n$ obtained via
Algorithm 1 and the series of cluster labels $CW$.
Output: A series of parallelism labels $CL$.

Assign $maxruntimelabel = max(\{L_i, i = 1 \rightarrow n\})$;
Assign $maxcpulabel = max(\{C_i, i = 1 \rightarrow n\})$;
for $l = 1$ to $maxruntimelabel$ do
    $P(l) = \frac{length(\{x=l, x \in L_i\})}{n}$;
end for
for $c = 1$ to $maxcpulabel$ do
    for $l = 1$ to $maxruntimelabel$ do
        $P(c,l) = \frac{length(\{i \in [1,n]: C_i = c, L_i = l\})}{n}$, where $i$ represents a job;
        $Pr(c,l) = \frac{P(c,l)}{P(l)}$;
    end for
end for
for each $cw_j$ in $CW$ do
    Select an integer $x \in [1, maxcpulabel]$ according to the transition conditional
    probability table $Pr(c,l)$ with $l = cw_j$ and assign $cl_j = x$ to form a series of
    parallelism labels $CL$;
end for
```

repeated R times in step 7 and step 8, where R is sampled from the fitted Zipf distribution. For example, if we have $C_\sigma = \{1, 3, 2\}$ and the values of R for these labels are $2, 1, 4$ respectively, we obtain $CW = \{1, 1, 3, 2, 2, 2, 2\}$.

4.3 Generate Specific Values

This stage receives a series of run time labels CW and a series of parallelism labels CL as its inputs. The way to obtain CW is presented in section 4.2. CL can be achieved via Algorithm 2. Combining CW and CL, we have a series of labels for parallel jobs (cw_i, cl_i), where $cw_i \in CW$ and $cl_i \in CL$. The specific value for the run time of job i is generated by sampling the distribution of the cluster with label cw_i. The specific value for the number of processors of job i with label cl_i is generated by the following steps:

1. Determine all jobs in the real data that have labels of (cw_i, cl_i) based on classifications of the run time process $L_i, i = 1 \rightarrow n$ obtained via MBC in section 3.2 and classifications of the parallelism process $C_i, i = 1 \rightarrow n$ obtained via Algorithm 1. We can know exactly and call the number of processors of these jobs $\{processors\}$.
2. Consider $\{processors\}$ as a sample space, the specific value we seek is selected in $\{processors\}$ according to the uniform probability of this space.

5 Experimental Results

Details of the traces used in our experiments are described in section 2. All these traces are applied on our model to generate synthetic workloads. The quality of these synthetic workloads is evaluated by comparing with the real data. Furthermore, we compare our model with the model of Song et al. [1] and the model of Lublin/Feitelson [24]. They are recent models for parallel system workloads.

Evaluation metrics used in our experiments include the marginal distribution, the cross-correlation between the run time and the parallelism, and the *squashed area* discussed in section 5.2. Note that we do not evaluate the locality feature of the run time process since it is assured by the Zipf distribution of repetitions. Instead, we only evaluate our improvement by comparing with MBC-LS model [14].

5.1 Locality of the Run Time Process

Figure 1 shows experimental results in evaluating our improvement. Reminding that our purpose is to fit the locality of the real data better. In order to compare our model with MBC-LS, we count the lengths of repetitions in both synthetic and real run time processes and draw the log-log histograms of these lengths. It can be seen that in all cases, our improved model fits the real data better than MBC-LS. Nevertheless, the model does not match the real data very well. It is because the probability p we calculate in step 4 of the algorithm presented in section 3.3 is not a perfect value. Another reason is that we only allow one sequence of r repeated values each time R values are generated. Of course, we can improve this matching by increasing p and allow more than one sequence but it depends on the traces. In our experiment, we found that this method indeed helps to match some traces better but also causes the situations of overfitting for other traces. Therefore, we decide to select the current method to avoid overfitting. However, more research is still left to improve the locality matching of the model compared with the real data.

5.2 The Metric Squashed Area

The *squashed area* (SA) metric is proposed by Song et al. [1]. It is the total resource consumptions of all jobs

$$SA = \sum_{j \in Jobs} req\_processor_j \times run\_time_j . \tag{1}$$

Furthermore, the difference of *squashed area* is calculated by

$$d\_SA = \frac{synthetic\_SA - original\_SA}{original\_SA} . \tag{2}$$

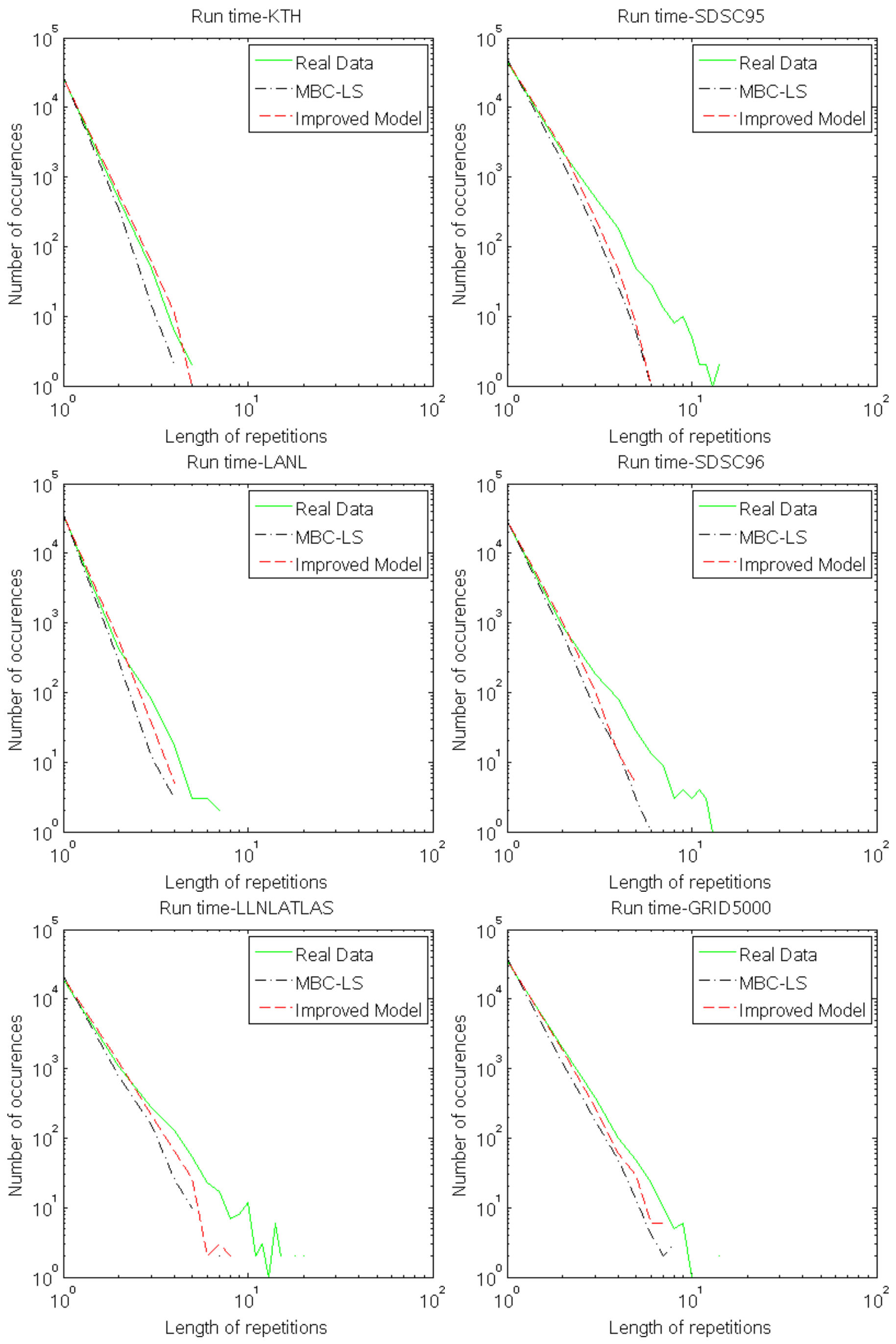


Fig. 1. Log-log histograms for the lengths of repetitions in the traces and synthetic run time processes

Table 2. The difference of *squashed area*. Results for Song et al.'s model are collected from [1].

Trace	Our model	Song et al.
KTH	0.39%	15%
LANL	2.21%	-1%
SDSC95	-0.04%	-3%
SDSC96	-3.12%	8%

In [2], Ernemann et al. demonstrated that the *squash area* has significant impacts on scheduling performance. It can be concluded from Equation (2) that if d_SA is closer to 0, the result is better (i.e. the model matches well the real traces). The results in Table 2 show that our model is better than the model of Song et al. since in most cases our differences of *squashed area* are smaller.

5.3 The Metric Cross-Correlation

One of the most difficult problems in modeling parallel workloads is how to control the cross-correlation between the run time and the parallelism as accurately as in the real data. The cross-correlation is measured by calculating the correlation coefficient between the run time and the parallelism. It can be seen from Table 3 that our model controls the cross-correlation well since our results are closer to the real data than those of the other models. As understanding from our experiments, the cross-correlation is controlled well thanks to the combination of Algorithm 2 and the way we generate specific values for parallelism labels as described in section 4.3.

Table 3. The cross-correlation between the run time and the parallelism. Results for the models of Song et al. and Lublin/Feitelson are collected from [1].

Trace	Real data	Our model	Song et al.	Lublin/Feitelson
KTH	0.011	0.015	0.005	0.005
LANL	0.172	0.192	0.226	0.29
SDSC95	0.277	0.233	0.140	0.105
SDSC96	0.371	0.332	0.155	0.116
LLNLATLAS	0.034	0.033	-	-
GRID5000	0.006	0.009	-	-

5.4 Marginal Distribution

Another important result from our model is that the marginal distribution is fitted very well. Figure 2 and Figure 3 show how well the cumulative density function (CDF) of the run time and the parallelism is fitted in our model. For the run time with continuous values, the marginal distribution is determined by the *mixture of Gaussians* model (see section 3.2). For the parallelism with discrete values, our experiment proves that the marginal distribution is fitted well by Algorithm 1 in section 4.1.

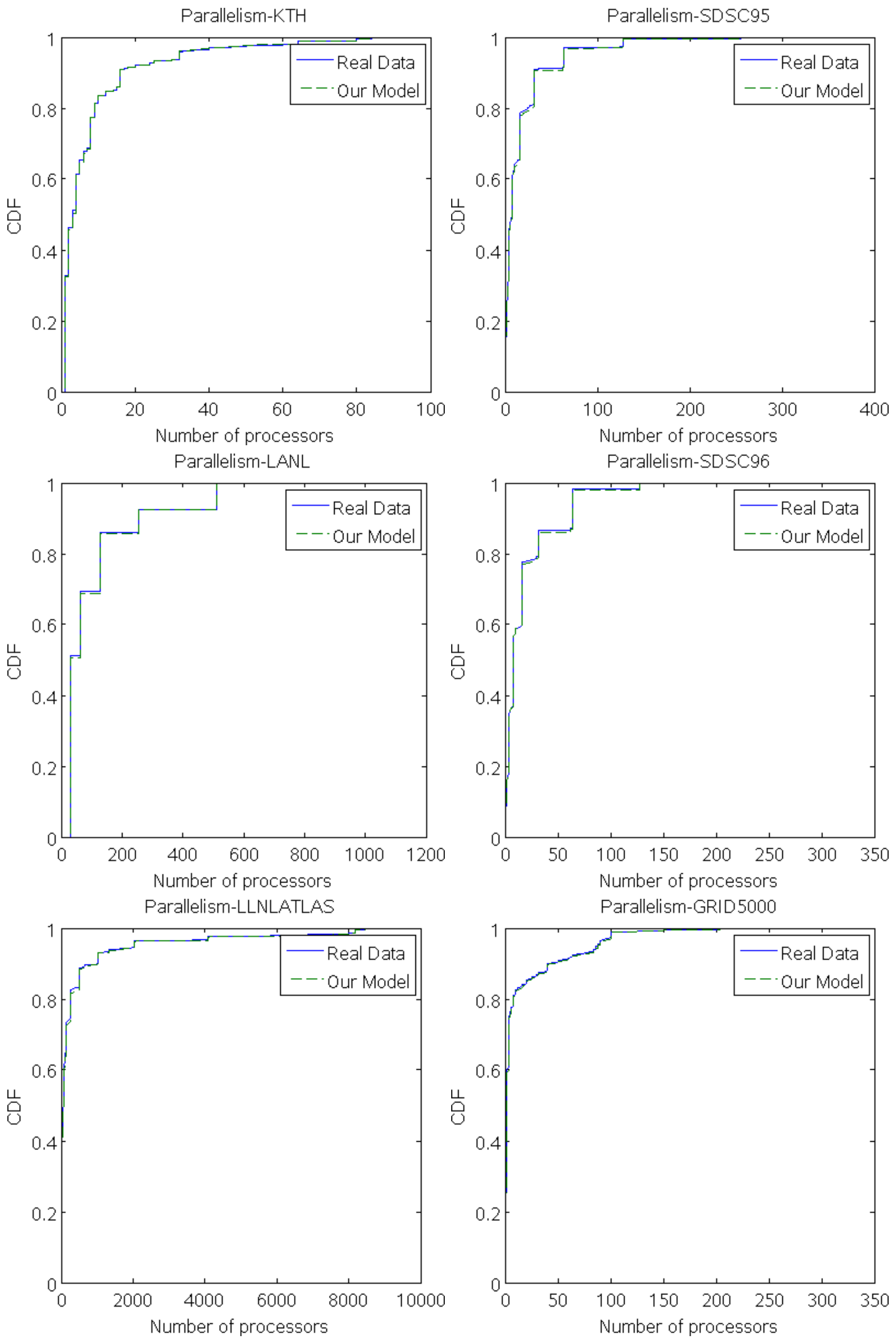


Fig. 2. Fitted marginal distribution of the parallelism

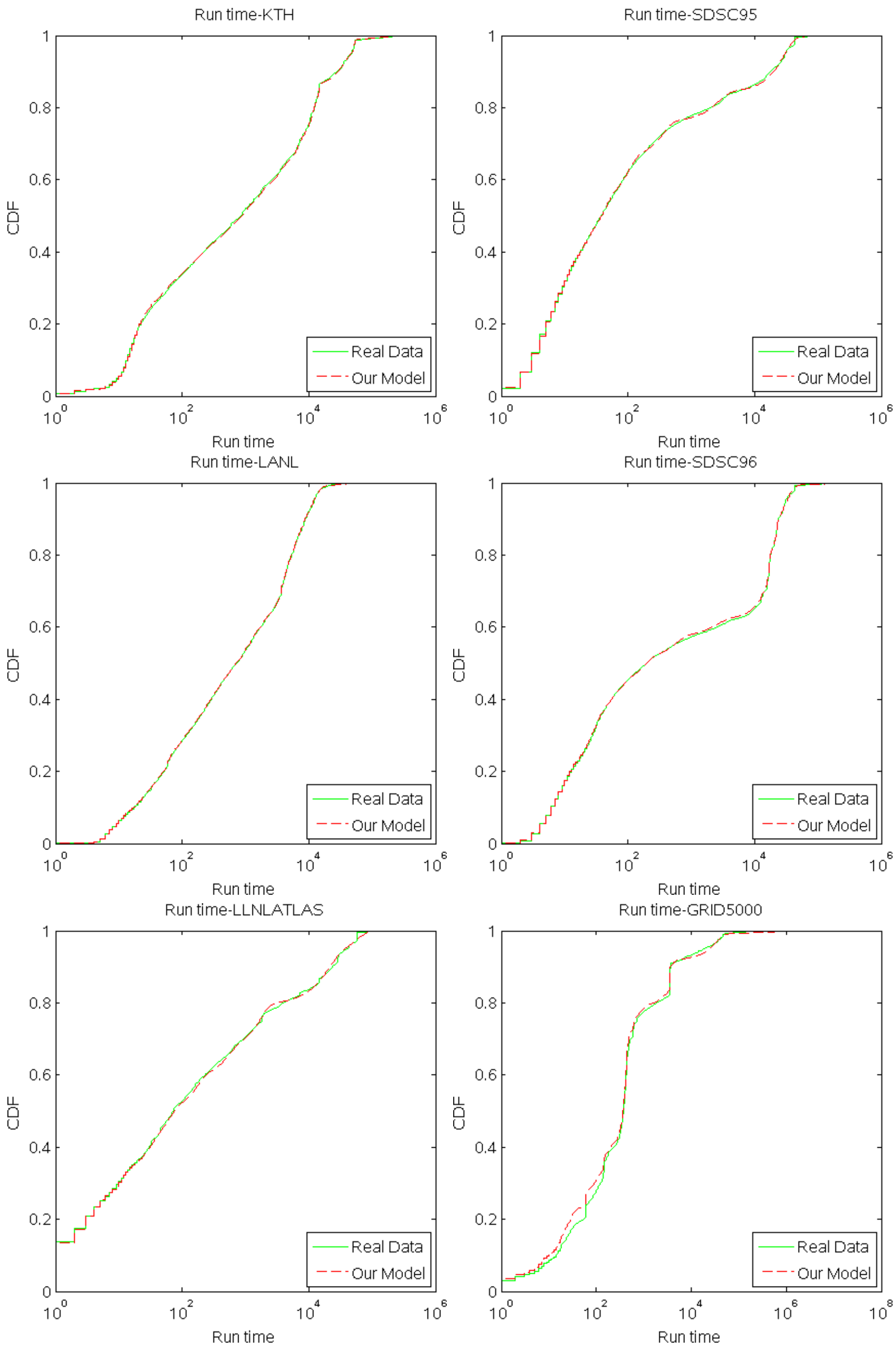


Fig. 3. Fitted marginal distribution of the run time

6 Conclusions and Future Work

When modeling parallel system workloads, researchers should take care of the locality and the cross-correlation between the parallelism and the run time. The locality feature is necessary for studies on predicting job run time, based on the belief that the recent past is indicative of the near future. The cross-correlation was demonstrated in [25] to have significant impacts on the evaluation of scheduling performance.

With respect to the locality, Li et al. [14] and Feitelson [6] recently introduced approaches to produce locality in the synthetic run time process. We also discussed some limitations of their methods and suggested a solution to overcome these limitations. Our solution indeed fits locality of the real data better than Li et al. 's model (see Figure 1) but not very well. The reason was already discussed in section 5.1 and more effort to improve this result is left for future.

For the cross-correlation, experimental results (see Table 2 and Table 3) showed that our model can control the cross-correlation between the run time and the parallelism more accurately, compared with recent models for parallel system workloads [1,24].

In addition, another important result from our model is that the marginal distributions of the synthetic run time and the synthetic parallelism fit the real data very well (see Figure 2 and Figure 3).

From our results, we believe that modeling parallel system workloads based on classifying data is a good approach. In future work, we continue to use this idea to model other workload attributes such as user estimated run time and requested memory in order to form a full synthetic workload with adequate necessary attributes including real run time, user estimated run time, requested memory and parallelism.

References

1. Song, B., Ernemann, C., Yahyapour, R.: Parallel computer workload modeling with markov chains. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 47–62. Springer, Heidelberg (2005)
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. Fraley, C., Raftery, A.E.: Model-Based Clustering, Discriminant Analysis, and Density Estimation. Journal of the American Statistical Association 97, 611–631 (2002)
4. Lifka, D.A.: 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. Jackson, D.B., Snell, Q., Clement, M.J.: Core algorithms of the maui scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
6. Feitelson, D.G.: Locality of Sampling and Diversity in Parallel System Workloads. In: Proceedings of 21st ACM International Conference on Supercomputing. ACM Press, USA (2007)

7. Feitelson, D.G.: Workload Modeling for Computer Systems Performance Evaluation. Book Draft, Version 0.18 (2008)
8. Distributed Job Manager, http://bradley.csail.mit.edu/cm5docs/manuals/cm5/doc/djm/
9. Grid Workloads Archive, http://gwa.ewi.t-udelft.nl/
10. Grid5000, http://www.grid5000.org/
11. Li, H.: Long Range Dependent Job Arrival Process and Its Implications in Grid Environments. In: Proceedings of MetroGrid Workshop, 1st International Conference on Networks for Grid Applications. ACM Press, France (2007)
12. Li, H., Groep, D., Wolters, L.: An Evaluation of Learning and Heuristic Techniques for Application Run Time Predictions. In: Proceedings of 11th Annual Conference of the Advance School for Computing and Imaging (ASCI), Netherlands (2005)
13. 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)
14. Li, H., Muskulus, M., Wolters, L.: Modeling Correlated Workloads by Combining Model Based Clustering and a Localized Sampling Algorithm. In: Proceedings of 21st ACM International Conference on Supercomputing. ACM Press, USA (2007)
15. Mohamed, H., Epema, D.: The Design and Implementation of the KOALA Co-Allocating Grid Scheduler. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 640–650. Springer, Heidelberg (2005)
16. Wan, M., Moore, R., Kremenek, G., Steube, K.: A Batch Scheduler for the Intel Paragon 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)
17. MCLUST, http://www.stat.washington.edu/mclust/
18. Moab Workload Manager, http://www.clusterresources.com/pages/products/mo-ab-cluster-suite/workloadmanager.php
19. OAR, http://oar.imag.fr/
20. Denning, P.J.: The Locality Principle. Communications of ACM 48, 19–24 (2005)
21. Parallel Workloads Archive, http://www.cs.h-uji.ac.il/labs/parallel/workload/
22. Gibbons, R.: A Historical Application Profiler for Use by Parallel Schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 58–77. Springer, Heidelberg (1997)
23. Riedi, R.H., Crouse, M.S., Ribeiro, V.J., Baraniuk, R.G.: A Multifractal Wavelet Model with Application to Network Traffic. Journal of IEEE Transactions on Information Theory 45(4), 992–1018 (1999)
24. Lublin, U., Feitelson, D.G.: The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing 63 (2003)
25. 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.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 25–46. Springer, Heidelberg (1998)
26. 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)

Scheduling Restartable Jobs with Short Test Runs

Ojaswirajanya Thebe[1], David P. Bunde[1], and Vitus J. Leung[2]

[1] Knox College
{othebe,dbunde}@knox.edu
[2] Sandia National Laboratories
vjleung@sandia.gov

Abstract. In this paper, we examine the concept of giving every job a trial run before committing it to run until completion. Trial runs allow immediate job failures to be detected shortly after job submission and benefit short jobs by letting them run and finish early. This occurs without inflicting a significant penalty on longer jobs, whose average and maximum waiting time are actually improved in some cases. The strategy does not require preemption and instead uses the ability to kill and restart a job from the beginning, which it does at most once for each job. While others have proposed similar strategies, our algorithm is distinguished by its determination to give each job a fixed-length trial run as soon as possible. Our study is also more focused, including a detailed description of the algorithm and an examination of the effect of varying the length of a trial run.

1 Introduction

It is widely known that user estimates of job runtimes are highly inaccurate (e.g. [12],[14]). Typically the worst overestimates are explained by pointing to programs that fail early in their execution. For example, Perković and Keleher [16] say "The presence of large runtime overestimations indicates the presence of applications still in development, and therefore, have high probability to die prematurely either because of bugs or because they run in a new environment". At the same time, job queues on large machines can be long, potentially preventing these failures from being discovered for quite some time. Waiting for an hour only to discover that your program died from an immediate segmentation fault increases the frustration already inherent in debugging tasks.

Furthermore, job failures turn out to be surprisingly common. Figure 1 reports the number and percentage of jobs that fail in traces from the Parallel Workloads Archive [7]. Many of the traces contain significant numbers of jobs that fail.

Based on the frequency of job failures and the frustration of waiting to discover them, we believe it is important to design schedulers so that they attempt to detect jobs that quickly fail as soon after submission as possible. Some of the failures are likely to be hardware problems, the detection of which cannot be improved by changes to the scheduler. When a job fails because of a programming

Trace	Num. Jobs	Num. failed	% failed
DAS2-fs0-2003-1.swf	219,618	2,643	1.2
SDSC-Par-1995-2.1-cln.swf	53,970	906	1.7
DAS2-fs3-2003-1.swf	66,112	1,143	1.7
DAS2-fs4-2003-1.swf	32,953	602	1.8
SDSC-Par-1996-2.1-cln.swf	32,135	814	2.5
DAS2-fs2-2003-1.swf	65,382	1,994	3.1
DAS2-fs1-2003-1.swf	39,356	1,554	4.0
LPC-EGEE-2004-1.2-cln.swf	220,695	10,490	4.8
LLNL-Thunder-2007-1.1-cln.swf	118,791	7,933	6.7
CTC-SP2-1995-1.swf	70,918	6,972	9.8
LANL-CM5-1994-3.1-cln.swf	122,060	20,368	16.7
LANL-O2K-1999-1.swf	116,996	23,670	20.2
CTC-SP2-1996-2.1-cln.swf	77,222	16,669	21.6
LLNL-Atlas-2006-1.1-cln.swf	38,194	10,250	26.8
KTH-SP2-1996-2.swf	28,489	7,948	27.9
LLNL-uBGL-2006-1.swf	19,405	6,835	35.2
SHARCNET-2005-1.swf	1,194,184	1,003,277	84.0

Fig. 1. Failing jobs by trace. Only traces with at least one failing job are presented. There were 2 other traces that reported all jobs succeeding, 2 that reported all jobs having "unknown" exit status, and 4 that reported various mixtures of succeeding, canceled, or unknown exit status. Also note that the number of jobs varies from the value reported in the Parallel Workloads Archive [7], sometimes greatly. We exclude jobs with unknown exit status and those that were canceled without running.

error or something wrong in the runtime environment, however, this failure can be detected by starting the job soon after its submission. Since failing jobs are only identified after they fail, this requires that all jobs be started soon after submission. If the system supports preemption, it is possible to do exactly this; as soon as a job arrives, preempt other jobs to give it sufficient processors to run for a brief period of time, after which the new job is itself preempted and the previous jobs resumed. In this way, any job failure occurring at the beginning of the job would be detected nearly immediately. If the period is brief enough, the previously-running jobs are not greatly inconvenienced. Thus, we are left with an engineering tradeoff to choose the length of a job's initial run, with longer runs finding more failures and shorter runs minimizing disruption to already-running jobs.

Unfortunately, preemption is difficult to implement in a large multiprocessor system because preempting a job requires saving its state on each of its processors and also catching all "in flight" messages traveling between them. Because of these difficulties, many multiprocessor systems do not support preemption. Instead, our algorithms use *restarts*, in which a job can be stopped and restarted, but does so from the beginning of its execution, effectively losing its progress from the first run. Restarts are less powerful than preemption and should be simpler to implement; it is not necessary to save any state, but merely to kill the job and ignore any of its messages. It is still technically challenging to restart jobs that perform side effects (e.g. file I/O), but we believe it is easier for systems to implement restarts than preemption. In exchange for being easier to implement, restarts impose greater cost on jobs on which they are used; all work previously done on that job is lost.

Now we can give the outline of our scheduling idea. As above, we attempt to start every job soon after its submission. We call the first time a job is started its *trial run*, which we only allow to continue for a bounded period of time. Jobs that do not fail (or complete) within this time are killed to be restarted later. When a job is restarted is determined by a *base scheduler* such as First-Come First-Served (FCFS) or EASY [13]. We call the combination of trial runs and the base scheduler a *timed-run* scheduler, which can be viewed as the base scheduler operating within a framework that manages trial runs. Our intent is for the timed-run scheduler to behave similarly to its base scheduler except for identifying failing jobs more quickly. In particular, once the base scheduler decides to start a job, that job is never restarted; our algorithm only kills jobs at the end of their trial run when relatively little work is lost by doing so. We say a job is *committed* when it has been started by the base scheduler. Exactly when a job should be committed proved to be a more subtle decision than we originally thought; we discuss this decision later in the paper.

As a side effect of giving jobs trial runs, the timed-run scheduler also benefits jobs that successfully finish within their trial run. We use the term *short jobs* to denote jobs that complete or fail during their trial run and *long jobs* to denote the others. Allowing short jobs to cut in front of longer jobs generally improves the system's average response time, though at some cost in fairness. For a short trial run length, we believe that the effect on long jobs is minimal in exchange for the benefits provided to short jobs, especially jobs that fail immediately after they start.

We show that this strategy can greatly reduce the time to detect problems in short failing jobs, the jobs on which users will be most frustrated to wait. The benefits of our strategy extend to all short jobs, which form a significant fraction of many workloads. The improvement is achieved with a non-preemptive strategy that restarts each job at most once. It is generally realized without significantly penalizing long jobs and even improves their average and maximum response time in some cases.

These results are based on event-based simulations using traces from the Parallel Workloads Archive [7]. We assume that the system being evaluated uses pure space-sharing to run rigid jobs.

We note that others have proposed similar strategies in the past. What distinguishes our algorithm is its focus on giving each job a fixed-length trial run as soon as possible. We also give a more thorough evaluation of trial runs in isolation, giving a detailed description of the algorithm and an examination of the effect of varying the length of a trial run.

The rest of the paper is organized as follows. In Section 2 we fully specify the timed-run scheduling strategy. Then, in Section 3 we evaluate this strategy. We discuss related work in Section 4. We conclude with a discussion of future work in Section 5.

2 Timed-Run Scheduling

Now, we are ready to formally define the timed-run algorithm. It maintains a list of jobs awaiting a trial-run in addition to whatever data structures are required for the base scheduling algorithm. Newly-arrived jobs are added to the end of this list as well as to the base scheduler's data structures. Whenever a job arrives or processors are freed due to a job completion or termination, the timed-run scheduler traverses this list looking for jobs to start. Any jobs encountered during this traversal that can start are removed from the list and started on their trial run. Only if no jobs can start trial runs is the base scheduler allowed to start jobs.

Our goals when designing this algorithm were to give jobs their trial runs as early as possible while impacting the base scheduler as little as possible. The prioritization of trial runs is reflected in our choice to look for jobs in the trial run list before consulting the base scheduler. Because the jobs are considered for trial runs in order of their arrival, we slightly favor earlier-arriving jobs and provide some measure of fairness. The jobs are not forced to receive trial runs in the order they arrive, however, to facilitate giving as many jobs as possible their trial runs soon after they arrive. We also allow the base scheduler to run jobs even when there are still jobs waiting for trial runs (provided none of them can start) to minimize the impact on the base scheduler. This decision and allowing trial runs to occur out of order both penalize large jobs, but we felt this discrimination was justified to avoid draining the machine just for a trial run of a large job. We consider it the base scheduler's responsibility to make such weighty decisions. In addition, we felt that failures of small jobs were more "justified" since users should test large programs on a smaller scale before running them on many processors.

The other obvious decision to make when implementing the timed-run scheduler is the duration of trial runs. We initially chose 90 seconds as the trial run length because this was the value given by Mu'alem and Feitelson [14] in their discussion of failing jobs. Another value mentioned in the literature is 1 minute, which Chiang and Vernon [5] observed was sufficient to complete 12–33% of jobs requesting over an hour and 11–42% of jobs requesting over 10 hours in a trace from NCSA's Origin 2000. They did not discuss the cause of these dramatic overestimates, but it seems likely that job failures played a role. Lawson and Smirni [11] suggest 180 seconds, which they observed to exclude most jobs that crashed. We discuss the effect of varying the trial run length in Section 3.2.

2.1 Optimizations and Complications

We decided on the aspects of timed-run scheduling described above without much difficulty. While implementing it and examining the schedules produced by our initial prototype, however, we discovered a number of complications. We now describe these and the policy decisions we made to resolve them.

Jobs wait until finishing their trial runs before committing. The first complication we discovered applies even to very small input instances. What should the scheduler do when the machine is idle and a single job arrives? As described above, the algorithm will select this job for a trial run and then schedule it again by following the base scheduling algorithm. Obviously, the job should not be started twice, but it seems premature to commit it to run to completion simply because it was the first job to arrive after an idle period. Nor is this necessarily a rare case since the same situation occurs if a job starts a trial run and then is selected by the base scheduler. We resolved this by not allowing a job to commit during its trial run. During this time, the base scheduler acts as if the job cannot fit on the machine.

To improve performance when a job's trial run and its selection by the base scheduler occur together, we implemented a fairly obvious optimization: when a job completes its trial run, if the timed-run scheduler will decide to start the same job for its committed run, we simply continue that job rather than stopping and restarting it. This optimization complicates the scheduler's logic somewhat, but clearly improves the schedule since it avoids wasting the time already spent on the trial run.

Jobs continue trial runs until replaced. After implementing the above, we noticed a related optimization. Consider the following job instance, scheduled on a 100-processor machine with 90-second trial runs and a First-Come First-Served (FCFS) base scheduler:

Job	Arrival time	Number processors	Runtime
A	0	80	300
B	100	100	40
C	110	20	> 90

This instance is scheduled as shown in Fig. 2. Notice that job A continues after its trial run because nothing else has arrived when it finishes the trial run. Job C does not get to continue, however, because FCFS wants to run job B first. Terminating job C at time 200 is not strictly necessary, however, since job B cannot start until time 300. Instead, we allow jobs that complete their trial runs to continue running until the scheduler has another use for their processors, either for a different job's trial run or for a committed run. For the example above, this means that job C is allowed to continue until time 300. If it has length between 90 and 190, this allows it to complete. Even if job C requires more time than this, nothing is lost since the processors it uses would have been idle otherwise. Note that extensions are granted even when a job's estimated running time indicates that it will not complete because the estimate may be inaccurate.

Avoiding restarts and extending trial runs are both achieved using lazy job termination. When a job finishes its trial run, it is added to a collection of jobs

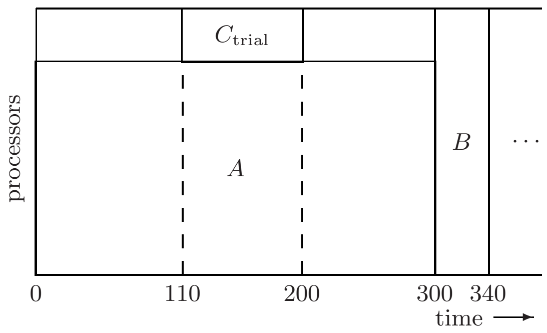


Fig. 2. Short jobs continue running until replaced

that can be terminated if needed. The scheduler makes its decisions as if all jobs in this collection had been terminated. If the scheduler decides to start a job that requires some of their processors, jobs from the collection are terminated as needed, beginning with the one whose trial run ended longest ago.

Long jobs must wait for their turn in the base scheduler. The next complication we encountered required a more difficult policy decision. Consider the following instance, again scheduled on a 100-processor machine with 90-second trial runs and FCFS scheduling:

Job	Arrival time	Number processors	Runtime
A	0	70	90
B	5	70	60
C	10	50	200
D	20	20	140
E	25	30	40

Two possible schedules are shown in Fig. 3. The difference is in when job D is committed. At time 150, job D has completed its trial run. The other job in the system is job C, which has not had a trial run, but should run first according to the base scheduler (FCFS).

Our first inclination was to start job D immediately in this situation since it seems wasteful to idle processors while waiting for a job that has already started (albeit only for a trial run). Our eventual conclusion, however, was to delay job D until its predecessor gets committed. The reason for this decision is to allow for the possibility that another job arrives during the trial run of job C. If we committed job D and a newly-arrived job prevents job C from committing, then the timed-run scheduler would have committed jobs out of the order given by the base scheduler, violating our intention to make the timed-run scheduler an augmentation of the base scheduler rather than its replacement. Note that delaying when jobs are committed in this way could harm our performance. An alternate solution would be to start job D, but kill it if job C ends up not committing. This would be similar to the speculative backfilling of Perković and Keleher [16].

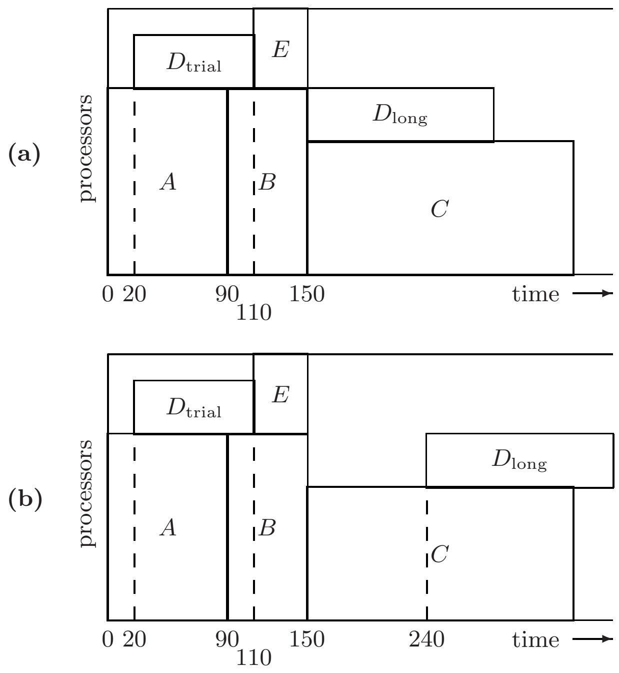


Fig. 3. Two possible ways to schedule the long run of job waiting for a job starting its trial run. In (a), job D starts as soon as job C begins its trial run. In (b), it waits for job C to be committed.

Dealing with job reservations. The final complication we encountered while implementing the timed-run strategy is how to combine it with base schedulers where jobs are given reservations. In keeping with our goal to give each job a trial run shortly after it arrives, our algorithm favors trial runs over committing jobs in the order given by the base scheduler. This means reservations may be violated since newly-arrived jobs can (briefly) grab processors at any time. However, we do recognize that reservations are desirable from a user perspective since they promote fairness and make the system more predictable for users. Thus, we wished to achieve a compromise by preserving the spirit of reservation-wielding base schedulers while violating the specific reservations.

For the EASY scheduler, there is a relatively straightforward way to achieve this compromise. We simply disabled the error checking that reports when a guarantee is violated. This works because our implementation of EASY (following [14]) does not build an entire schedule. Rather, it stores the jobs in arrival order, the currently-running jobs with their estimated completion times, and the first job's guaranteed time. To make a scheduling decision, it traverses the list of waiting jobs and starts any job that can be run without violating the guarantee.

It is much less clear how to use timed-run scheduling with algorithms that provide guarantees to more than one job such as Conservative backfilling. One solution is to rebuild the estimated schedule whenever trial runs cause it to break, but this could greatly slow down the scheduler. Another idea is to give initial guarantees with some slack to allow for trial runs by later jobs, but this seems to violate the spirit of Conservative backfilling. We believe more research is warranted on this question.

3 Experimental Results

To evaluate the timed-run strategy, we used an event-driven simulator. Events were generated for job arrivals, job completions, and at the end of trial runs. The data for our simulations were obtained from the online Parallel Workloads Archive [7]. All traces were in the standard workload format, from which we read the job arrival time, processors requested, actual running time, and user-estimated runtime (when available). For actual runtime, we used field 4 ("run time") if it was available and field 6 ("Average CPU time used") if it was not. We also used the status field (number 11) to identify failing jobs, but only as a post-processing step. Cleaned versions of the traces were used when available; the full filenames for the used traces are given in Fig. 1. We excluded the SHARCNET trace from our simulations because of its extraordinarily-high failure rate.

3.1 Ninety Second Trial Runs

We compared FCFS and EASY schedulers to their timed-run counterparts using average and maximum waiting time. We used waiting time since it is in line with our goal to minimize the absolute time before detecting a failure. It also lessens the emphasis on small jobs relative to slowdown or bounded slowdown. Note that the waiting times we record for a job under the timed-run scheduler is until that job starts the run that finishes, NOT the wait until the job gets a trial run. Put another way, the waiting time of a job is its completion time minus its arrival time minus its actual running time.

Our initial simulations used a trial-run length of 90 seconds. Figs. 4 and 5 show the percent improvement in average and maximum response time achieved by switching from a normal scheduler to a timed-run scheduler. From the results, the timed-run scheduler generally performs as expected, decreasing average waiting time in nearly all cases. The exceptions are all in the DAS2 family of traces. These traces, from a group of clusters used for distributed computing research, have quite low utilization (all less than 20%) so they are not representative of typical production workloads. Quite a few of the improvements in the other traces are significant, particularly with the FCFS base scheduler.

Also importantly, the improvement in average waiting time does not occur at the expense of increased maximum waiting time. Instead, maximum waiting time is largely unchanged, with the worst result an increase of less than 4%. On several of the traces, using timed-run scheduling with FCFS actually improves it by a significant margin.

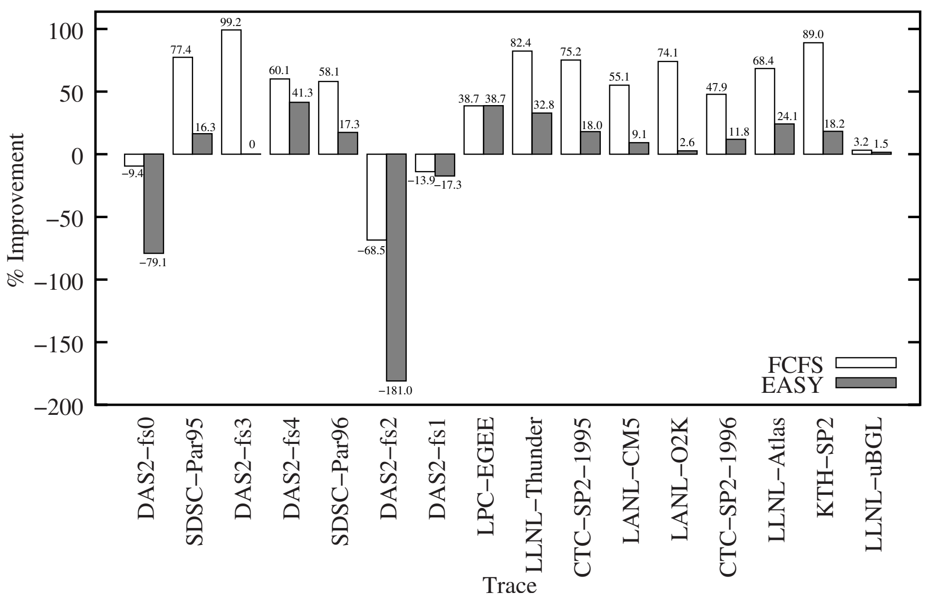


Fig. 4. Improvement in average waiting time of all jobs from using timed-run scheduler

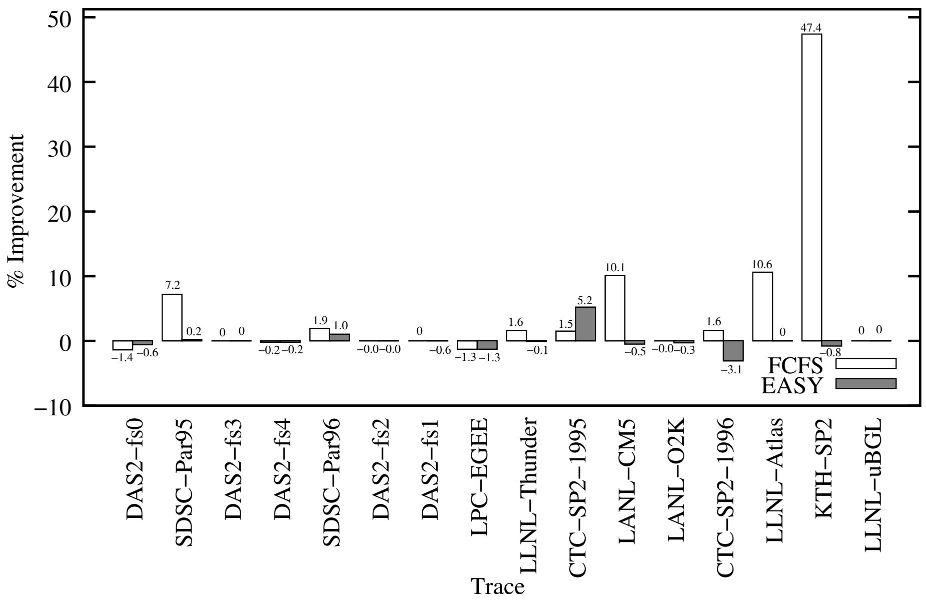


Fig. 5. Improvement in maximum waiting time of all jobs from using timed-run scheduler

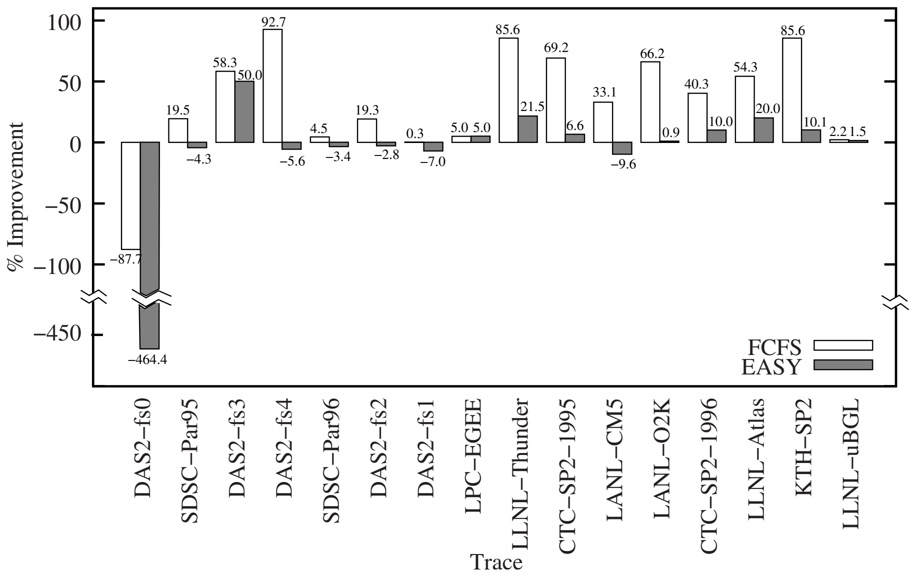


Fig. 6. Improvement in average waiting time of failing jobs from using timed-run scheduler

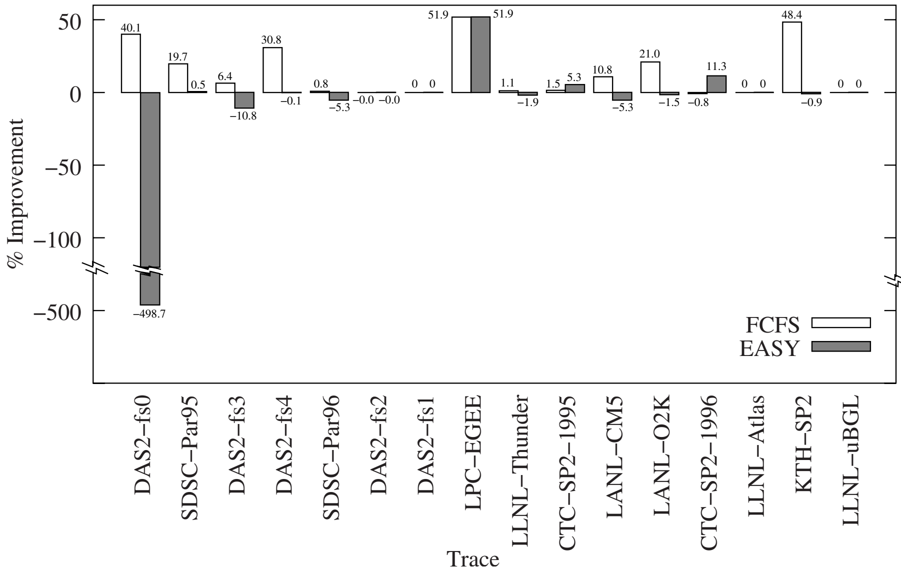


Fig. 7. Improvement in maximum waiting time of failing jobs from using timed-run scheduler

The percent improvements were better for FCFS than EASY. This is unsurprising since trial runs can act as an ad hoc version of backfilling. When working with FCFS, there are many opportunities for jobs to move up and the result is a significantly better schedule. In addition, the EASY base scheduler does a better job keeping the processors busy and so offers less room for improvement. In fact, for average waiting time, regular EASY outperformed FCFS with timed-run scheduling in almost all cases.

Our main motivation was to promptly identify failing jobs so their users could be notified soon after the jobs have been submitted. Figures 6 and 7 plot the improvement in average and maximum waiting time for failing jobs as we switch from regular scheduling and timed-run scheduling. Surprisingly, the results are not as good for failing jobs as they were for all jobs. The percentage improvements for average waiting time are generally smaller for FCFS and they essentially disappear for EASY. The other patterns are still there, though; FCFS is improved much more than EASY, the DAS2 traces contributed negative outliers to the percent improvement in average waiting time, and the affect on maximum waiting time of adding trial runs ranges is minimal with some improvements and a couple of good values.

We explain the relative lack of benefit for failing jobs with the observation that failing jobs are not necessarily short jobs. Although failing jobs ending prematurely is consistently one of the explanations given for the poor quality of user estimates, it turns out that job failures do not cause the short jobs in these traces. Figure 8 gives the percent of all jobs and the percent of failed jobs that are short in each trace. For all but 3 of the 16 traces, short jobs make up a smaller percentage of failing jobs than they represent of the trace as a whole. Only in LLNL-uBGL of these three is the difference large. However, there seems to be no relationship between the results in Figs. 4–5 and Figs. 6–7 and the percentage of failed jobs that are short. This could be due to the fact that the total number of failed jobs that are short is small compared to the total number of jobs that are short.

Figures 9 and 10 show the average and maximum waiting times for short jobs, respectively. Providing jobs with trial runs does result in short jobs waiting for considerably less time before running. Figures 11 and 12 shows the average and maximum waiting times for failed short jobs. The results were similar for average waiting time, but considerably improved for maximum waiting time. Here again, there is no relationship between the results in Figs. 9–10 and Figs. 11–12 and the percentage of all and failed jobs that are short because the numbers of short failed jobs are much smaller than the numbers of short jobs and a meaningful comparison cannot be made between the two.

3.2 Varying Trial-Run Length

We also investigated the effects of varying the length of the trial-run. An ideal length would balance catching failing jobs and increasing responsiveness by letting short jobs finish during their trial-run against making jobs wait too long while trial-runs occur. Figure 13 shows the average waiting time of short jobs as the length of the trial run varies, using the KTH-SP2 trace and FCFS as the

short jobs as...	% of jobs	% of failed jobs
DAS2-fs0	61.5 (134,991 jobs)	50.4 (1,331 jobs)
SDSC-Par95	60.9 (32,845 jobs)	0.1 (1 job)
DAS2-fs3	76.1 (50,321 jobs)	74.6 (853 jobs)
DAS2-fs4	42.9 (14,129 jobs)	48.0 (289 jobs)
SDSC-Par96	44.4 (14,268 jobs)	0.5 (4 jobs)
DAS2-fs2	64.5 (42,191 jobs)	10.4 (207 jobs)
DAS2-fs1	65.6 (25,803 jobs)	33.1 (514 jobs)
LPC-EGEE	69.9 (154,221 jobs)	9.7 (1,013 jobs)
LLNL-Thunder	59.1 (70,246 jobs)	65.2 (5,176 jobs)
CTC-SP2-1995	27.4 (19,404 jobs)	12.6 (880 jobs)
LANL-CM5	30.2 (36,910 jobs)	8.1 (1,650 jobs)
LANL-O2K	30.9 (36,132 jobs)	20.6 (4,877 jobs)
CTC-SP2-1996	21.6 (16,699 jobs)	15.0 (2,507 jobs)
LLNL-Atlas	51.9 (19,809 jobs)	47.5 (4,872 jobs)
KTH-SP2	32.9 (9,375 jobs)	28.5 (2,267 jobs)
LLNL-uBGL	56.7 (11,008 jobs)	92.3 (6,306 jobs)

Fig. 8. Short (< 90 sec) jobs by trace

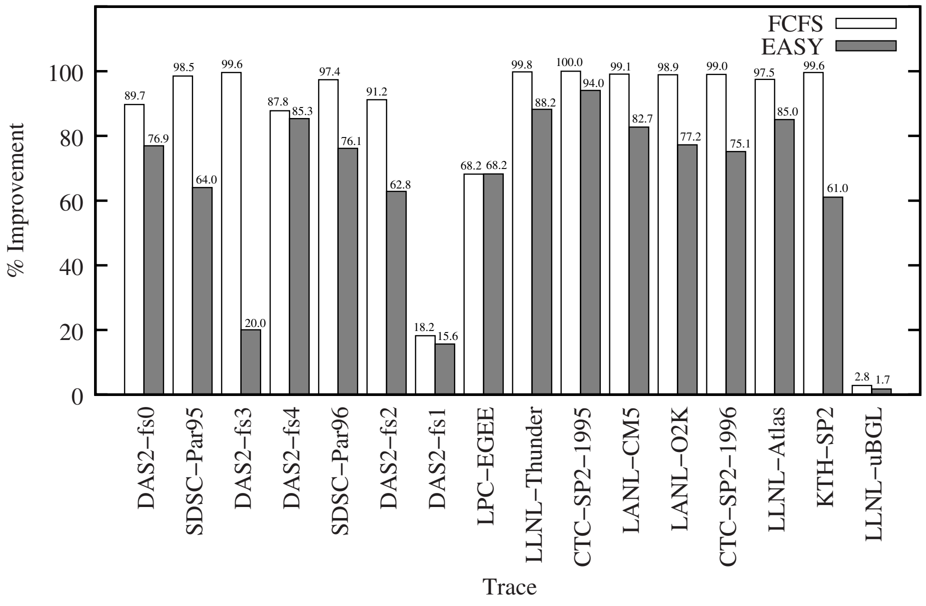


Fig. 9. Improvement in average waiting time of short (< 90 sec) jobs from using timed-run scheduler

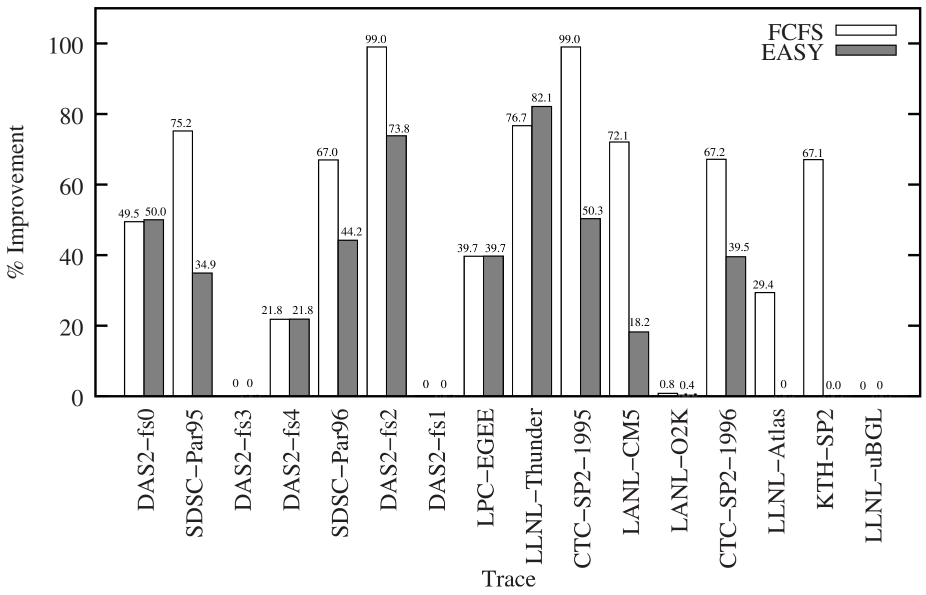


Fig. 10. Improvement in maximum waiting time of short (< 90 sec) jobs from using timed-run scheduler

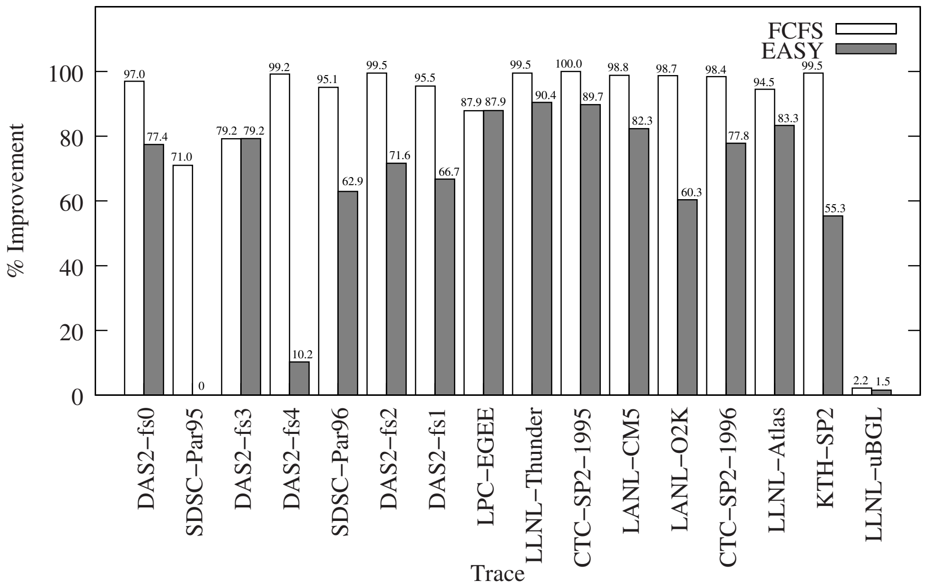


Fig. 11. Improvement in average waiting time of failing short (< 90 sec) jobs from using timed-run scheduler

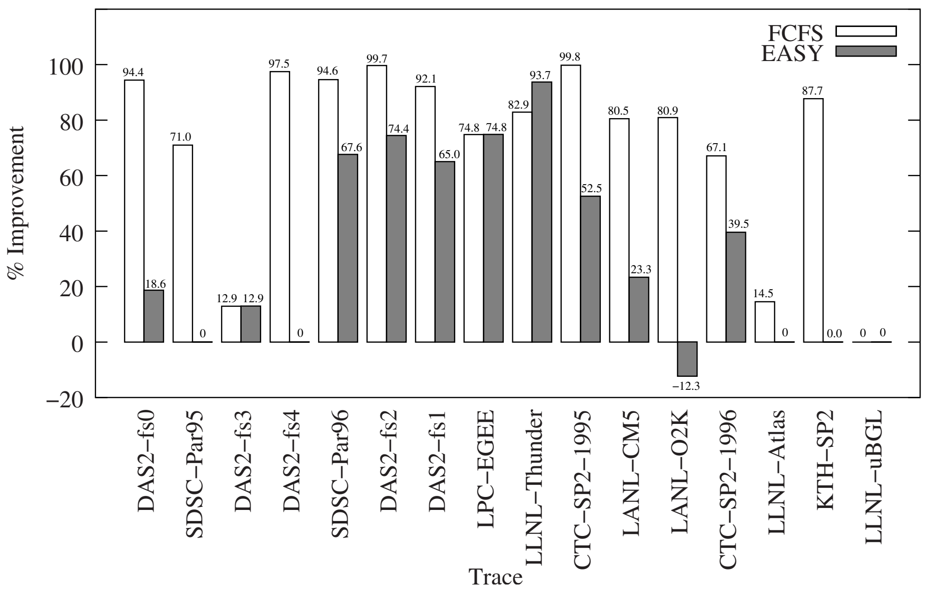


Fig. 12. Improvement in maximum waiting time of failing short (< 90 sec) jobs from using timed-run scheduler

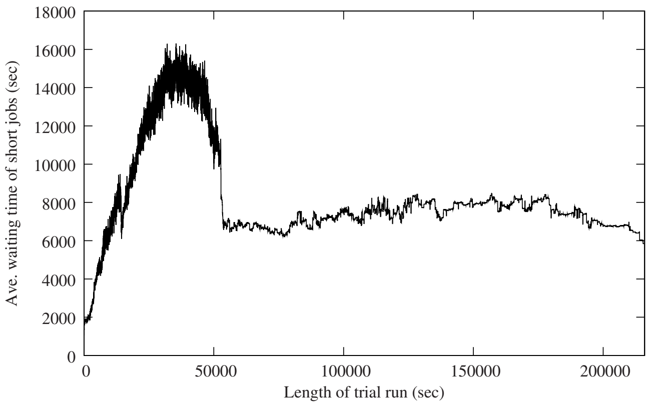


Fig. 13. Average waiting time for short jobs in KTH-SP2 trace with varying trial run lengths and FCFS scheduling

base scheduler. Note that a "short" job is one shorter than the trial run length so the jobs considered varies with the trial run length. We focus on short jobs to see how long potentially-identifiable failed jobs would have to wait. From the figure, we can see that relatively short trial runs will give us the fastest identification of short jobs. The same holds true when we switch to the EASY scheduler, which

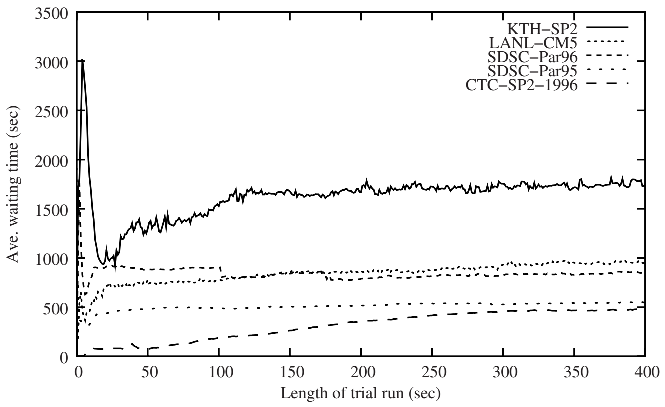


Fig. 14. Average waiting time for short jobs with varying trial run lengths (FCFS)

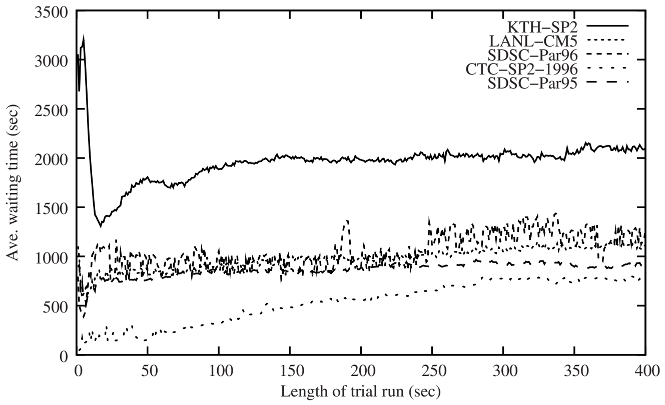


Fig. 15. Average waiting time for short jobs with varying trial run lengths (EASY)

gives a plot that is not visibly different from Fig. 13. The analogous plots for the other traces examined in this section also exhibit long climbs starting at low values of the trial run length.

Because of this observation, we focus on trial run lengths between 0 (no trial-runs) and 400 seconds for the rest of this section. (The climb depicted in Fig. 13 has already begun by 400 seconds.) The results of our experiments are presented in Figs. 14–19. Figures 14 and 15 show the average waiting time for short jobs. This figure provides an idea of how much time jobs failing within their trial run need to wait. Quick trial runs go through all available jobs faster, and so short job waiting times are lower since they get to finish quickly. However, the data have a distinct spike for extremely short trial-run lengths. This is because there are only a few jobs having extremely low runtimes, and when they do appear in

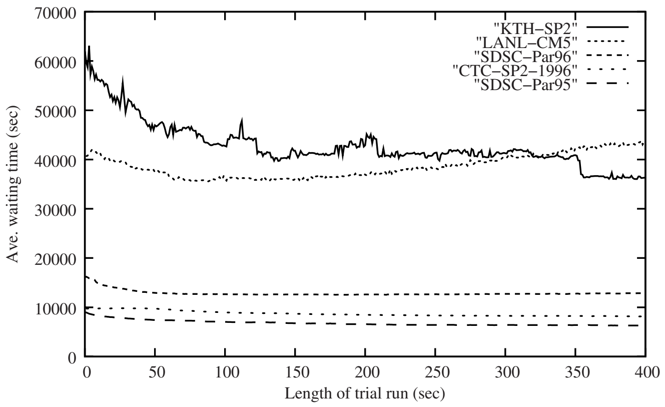


Fig. 16. Average waiting time for all jobs with varying trial run lengths (FCFS)

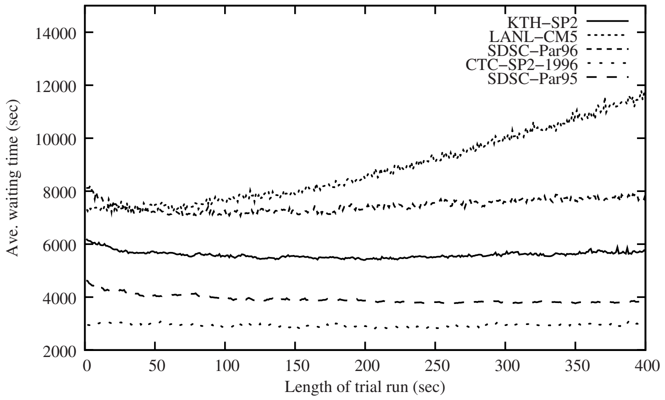


Fig. 17. Average waiting time for all jobs with varying trial run lengths (EASY)

the system, they need to wait for long jobs to finish and free processors before they get a chance to run. The average waiting time decreases after the spike since there are now more short jobs and they do not all need to wait for long jobs to finish. The waiting time increases after that because we are adding more overhead time for each job to have a trial run.

Figures 16 and 17 show the overall average waiting time for all jobs over increasing trial-run lengths. The two behaviors we see are gradually decreasing and gradually increasing average waiting time. For the most part we see a gradual decrease in average waiting time as the trial time is increased. This is probably due to an increasing number of jobs becoming short and having their waiting times dramatically reduced. In the one trace (LANL-CM5) where this does not happen, the waiting time increases probably due to the overhead time for each job to have a trial run.

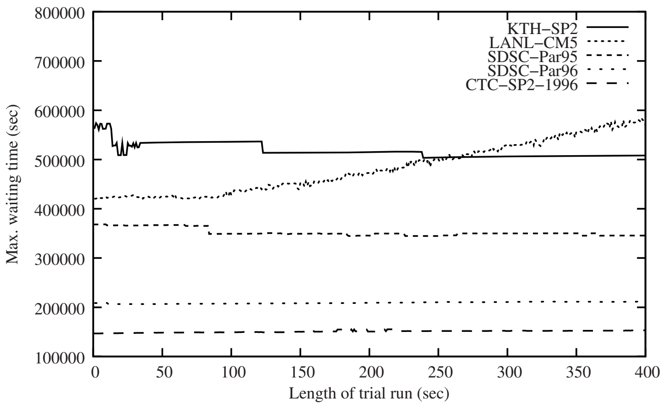


Fig. 18. Maximum waiting time for all jobs (FCFS)

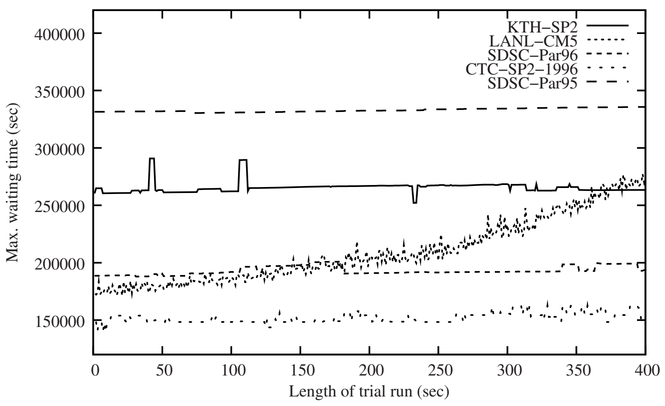


Fig. 19. Maximum waiting time for all jobs (EASY)

Long jobs have a higher maximum waiting times than short jobs. Figures 18 and 19, which show the maximum waiting time for all jobs, are also the same graphs as the maximum waiting time for long jobs. The two behaviors we see are the steps downward and the gradually increasing maximum waiting time. The reasons for both are simple. For the flat-line in the steps, that maximum waiting time is due to the same long job waiting in the base scheduler. However, when the trial runs are long enough, that job gets to run as a short job and does not need to wait in the base scheduler. So the maximum waiting time drops.

The gradual increase in the maximum waiting time is due to the extra overhead time from running each trial for a longer duration. This also allows more jobs to enter the system which increases the chances of jobs getting pushed further back.

From Figs. 14–19, we see that both the sixty seconds of Chiang and Vernon [5] and the ninety seconds of Mu'alem and Feitelson [14] give a reasonable balance between finding failures and minimizing wasted time during trial runs. The value of 180 seconds suggested by Lawson and Smirni [11] is also reasonable, but perhaps a bit too high, particularly for the LANL-CM5 trace.

4 Background and Related Work

Several other researchers have devised schedulers around the observation that many short jobs are submitted with greatly inflated estimates. The most similar idea appears in a system described by Perković and Keleher [16]. Their scheduler uses a number of different techniques, but among them are "speculative test runs" and "speculative backfilling". Speculative test runs are a version of our trial runs; jobs with long estimated running time (over 3 hours) are allowed a brief run on the machine (5–15 minutes) in the hopes of finishing early. This differs from what we do in that we give a trial run to all jobs whereas Perković and Keleher [16] give speculative test runs only to some fraction of the jobs and do so primarily as part of a larger speculative scheduling phase.

The other part of Perković and Keleher's speculative scheduling phase is speculative backfilling: starting a job in a "hole" that occurs in the schedule even when that job will not be able to complete unless its running time is overestimated. At the end of the hole, the speculatively backfilled job is killed if it has not already finished. In this way, only processors that would have been idle anyway are used in the speculation. This is similar to what our scheduler does when it continues to run jobs whose trial runs have expired, but we do not purposely start jobs speculatively after their single trial run (though we could). Again, this differs from our scheduler because we give trial runs to all jobs. The other main difference between our work and that of Perković and Keleher [16] is in our tighter focus; due to the number of ideas presented in their paper, they describe the idea only briefly and do not analyze the effect of this optimization alone or the effect of varying the length of a speculative execution.

Snell et al. [19] explored an idea similar to speculative backfilling. They allowed jobs to backfill even when there was not enough time in the schedule for them to complete, killing running jobs as needed to honor reservations. They considered a number of criteria for selecting the jobs to kill, finding that it was best to either kill the job with the most (estimated) time remaining or the job that was most recently started. These strategies improved system performance, but by relatively small amounts, apparently because of the work lost when jobs were killed. (Unlike in our strategy, they might kill a job that had already been processed for a considerable period of time.)

Lawson and Smirni [11] and Chiang et al. [4] take the idea of speculative execution and apply it to all jobs meeting some criteria rather than using it opportunistically. Lawson and Smirni [11] give each job whose estimated running time exceeds 1,000 seconds a 180-second trial run. Their algorithm is based on work by Lawson et al. [10], placing jobs into separate queues based on their running time. Each queue is serviced by part of the system so that short jobs do not wait for long jobs but no job can starve. The base algorithm assumes that job durations were estimated accurately, but Lawson and Smirni [11] found that similar ideas work when the speculative runs are used to detect the worst overestimates. Chiang et al. [4] similarly divided the system into two parts, each servicing a separate queue. Since they were concerned with systems where the users do not provide runtime estimates, the first queue is for jobs waiting to receive a trial run and the second plays the role of our base scheduler. The main difference between their work and ours is that they assume jobs can be assigned to different numbers of processors. Neither of these papers considered changing the length of the speculative run.

A manual version of timed-run scheduling was proposed by Chiang et al. [3]. They were concerned with the accuracy of user estimates and felt that users would be able to more accurately predict the runtime of many jobs by first making a "test run" on a smaller version of the problem or with slightly different input parameters. They examined the effect of test runs equaling 10% of the estimated run time but not more than 1 hour and then users submitting the real job with reasonably accurate estimates. They showed that such a scheme leads to performance improvements despite the overhead of the test runs. As with our algorithm, their test runs have the effect of identifying and finishing short jobs quickly. Our system differs in that it makes trial runs automatically rather than assuming users make them manually. The runs themselves are less time-consuming in our scheduler, but also do not provide improved estimates.

Others who have considered similar ideas to timed-run scheduling have done so in the context of systems that support preemption, which is much more flexible than the job restarts that we allow. Most related is work by Chiang and Vernon [5]; they consider backfilling with "immediate service", which attempts to give each newly-arriving job a one-minute run before putting it in the queue. It does this by preempting the currently-running jobs with lowest current slowdown among jobs that have not been preempted in 10 minutes. They showed that this strategy significantly improves average slowdown while having minimal effect on 95th percentile waiting time. This is similar to our results with the FCFS base scheduler, but preemption allows them to improve even on a scheduler with backfilling. Schwiegeishohn and Yahyapour [17] show how to improve FCFS by allowing preemption only to start jobs requiring a large number of processors. For other strategies utilizing preemption, see Kettimuthu et al. [9] and its references.

Although technically dissimilar, our overall goal is analogous to that of Shmueli and Feitelson [18], who argue that user productivity is a better metric than waiting time or slowdown. Their scheduler attempts to prioritize jobs whose submitter is likely to still be waiting for the result. Thus, jobs that can be finished shortly after submission are more critical than either long jobs or jobs that have already waited for a significant period of time. This is done so that users are able to continue working rather than needing to switch to another task and incurring a human "context switch" when they refocus on the task requiring the supercomputer system.

Intuitively, the success of our algorithm in quickly identifying failing or unexpectedly short jobs should have a similar benefit. Evaluating this would require modeling the user (as Shmueli and Feitelson [18] do) so that system performance affects job arrival rather than relying on the trace-based simulations we present here.

Our results can also be considered related to the work considering the effect of user estimates on scheduling performance. Many authors (e.g. [14]) have noted that user estimates tend to be dramatically high. One factor behind this is the very short jobs (including quick failures) we target with timed-run scheduling. In addition, users tend to reuse estimates once they find something that works and also strongly prefer "round" numbers (e.g. 5 minutes, 1 hour, etc). Accounts initially differed on whether overestimates improved [14,23] or hurt [3] performance. These observations were eventually reconciled with the finding that overestimates initially help but that extreme overestimates eventually hurt performance [20,22]. Regardless of this, it seems reasonable that good estimates could be useful since they provide more information to the scheduler. This has led to work trying to get users to improve their estimates [12] as well as work to have a system generate its own estimates [6,8,15,21].

5 Discussion

Our results in Section 3 show that timed-run scheduling can more quickly alert users about jobs that fail and benefit short jobs in general. In some cases, it has even been shown to improve average and maximum waiting times for the entire trace. We feel that this approach is promising and deserves further investigation.

The most obvious open problem is to adapt this strategy to other backfilling schemes. As mentioned in Section 2.1, it is not clear how to preserve the benefits of reservations when the reservations themselves may be violated if trial runs are granted to new jobs. One solution would be to rebuild the estimated schedule whenever trial runs cause reservations to be violated. Another solution would be to give initial guarantees with some slack to allow for trial runs by later arriving jobs. Each of these solutions has its own drawbacks, and we believe more research is necessary to address reservations.

It would also be beneficial to consider the affect of overhead for killing jobs. In our experiments, it was optimistically assumed that processors from a killed job could be instantly reassigned to another job. We do not believe that including realistic amounts of overhead will significantly change the results, but this would need to be verified before timed-run scheduling could be adopted.

Additionally, experiments with some of our policy decisions in Section 2.1 could further improve the performance of timed-run scheduling. Specifically, the decision that long jobs must wait for their turn in the base scheduler could be replaced by a policy similar to the speculative backfilling of Perković and Keleher [16].

It would also be interesting to evaluate the performance of our algorithm on user models such as those of Shmueli and Feitelson [18] to see if our quick completion of short jobs improves user satisfaction and productivity.

Acknowledgments. O. Thebe and D.P. Bunde were partially supported by contract 763836 from Sandia National Laboratories. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract No. DE-AC04-94AL85000. We thank the anonymous referees for many helpful comments. We also thank all those who contributed traces to the Parallel Workloads Archive. Specifically, we thank the following people for the traces we used: Dan Dwyer for the CTC-SP2 traces, Hui Li, David Groep, and Lex Wolters for the DAS2 traces, Lars Malinowsky for the KTH-SP2 trace, Curt Canada for the LANL-CM5 trace, Fabrizio Petrini for the LANL-O2K trace, Moe Jette for the LLNL-Atlas, LLNL-Thunder and LLNL-uBGL traces, Emmanuel Medernach for the LPC-EGEE trace, Reagan Moore and Allen Downey for the SDSC traces, and John Morton and Clayton Chrusch for the SHARCNET trace.

References

1. Feitelson, D.G., Rudolph, L. (eds.): IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291. Springer, Heidelberg (1997)
2. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.): JSSPP 2002. LNCS, vol. 2537. Springer, Heidelberg (2002)
3. Chiang, S.-H., Arpaci-Dusseau, A., Vernon, M.K.: The impact of more accurate requested runtimes on production job scheduling performance. In: Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing, [2], pp. 103–127
4. Chiang, S.-H., Mansharamani, R., Vernon, M.: Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In: Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, pp. 33–44 (1994)
5. Chiang, S.-H., Vernon, M.K.: Production job scheduling for parallel shared memory systems. In: Proc. 15th IEEE Intern. Parallel and Distributed Processing Symp. (2001)
6. Downey, A.B.: Using queue time predictions for processor allocation. In: Proc. 3rd Workshop on Job Scheduling Strategies for Parallel Processing [2], pp. 35–57
7. Feitelson, D.: The parallel workloads archive, `http://www.cs.huji.ac.il/labs/parallel/workload/index.html`
8. Gibbons, R.: A historical application profiler for use by parallel schedulers. In: Proc. 3rd Workshop on Job Scheduling Strategies for Parallel Processing [1]
9. Kettimuthu, R., Subramani, V., Srinivasan, S., Gopalsamy, T., Panda, D.K., Sadayappan, P.: Selective preemption strategies for parallel job scheduling. Intern. J. of High Performance Computing and Networking 3(2/3), 122–152 (2005)
10. Lawson, B., Smirni, E., Puiu, D.: Self-adapting backfilling scheduling for parallel systems. In: Proc. 31st Intern. Conf. Parallel Processing, pp. 583–592 (2002)
11. Lawson, B.G., Smirni, E.: Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In: Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing [2]
12. Lee, C.B., 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)

13. 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)
14. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel and Distributed Syst. 12(6), 529–543 (2001)
15. Nissimov, A., Feitelson, D.G.: Probabilistic backfilling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 102–115. Springer, Heidelberg (2008)
16. Perković, D., Keleher, P.J.: Randomization, speculation, and adaptation in batch schedulers. In: Proc. 2000 ACM/IEEE Conf. on Supercomputing (2000)
17. Schwiegelshohn, U., Yahyapour, R.: Improving first-come-first-serve job scheduling by gang scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS, vol. 1459, pp. 180–198. Springer, Heidelberg (1998)
18. Shmueli, E., Feitelson, D.G.: On simulation and design of parallel-systems schedulers: Are we doing the right thing? IEEE Trans. Parallel and Distributed Systems (to appear)
19. Snell, Q.O., Clement, M.J., Jackson, D.B.: Preemption based backfill. In: Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing [2], pp. 24–37
20. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Proc. Intern. Conf. on Parallel Processing Workshops, pp. 514–522 (2002)
21. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. on Parallel and Distributed Systems 18(6), 789–803 (2007)
22. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: Solving the mystery of why increased inaccuracy help. In: Proc. IEEE Intern. Symp. on Workload Characterization, pp. 131–141 (2006)
23. Zotkin, D., Keleher, P.J.: Job-length estimation and performance in backfilling schedulers. In: Proc. 8th IEEE International Symposium on High Performance Distributed Computing, pp. 236–243 (1999)

Effects of Topology-Aware Allocation Policies on Scheduling Performance

Jose Antonio Pascual, Javier Navaridas, and Jose Miguel-Alonso

The University of the Basque Country, San Sebastian 20018, Spain
{joseantonio.pascual,javier.navaridas,j.miguel}@ehu.es

Abstract. This paper studies the influence that job placement may have on scheduling performance, in the context of massively parallel computing systems. A simulation-based performance study is carried out, using workloads extracted from real systems logs. The starting point is a parallel system built around a k-ary n-tree network and using well-known scheduling algorithms (FCFS and backfilling). We incorporate an allocation policy that tries to assign to each job a contiguous network partition, in order to improve communication performance. This policy results in severe scheduling inefficiency due to increased system fragmentation. A relaxed version of it, which we call quasi-contiguous allocation, reduces this adverse effect. Experiments show that, in those cases where the exploitation of communication locality results in an effective reduction of application execution time, the achieved gains more than compensate the scheduling inefficiency, therefore resulting in better overall performance.

1 Introduction

Supercomputer centres are usually designed to provide computational resources to multiple users running a wide variety of applications. Users send jobs to a scheduling queue, where they wait until the resources required by the job are available. These jobs may vary from large parallel programs that need many processors, to small sequential programs. The scheduler manages system resources, taking into consideration different policies that may restrict the use in terms of maximum number of processors or maximum execution time. Other restrictions may be implemented such as user or group priorities, quotas, etc.

Generally, site performance is measured in terms of the utilization of the system and the slowdown suffered by jobs while waiting in the queue until the required resources become available. Consequently, a variety of scheduling policies [1] and allocation algorithms [2] [3] [4] have been developed aiming to minimize both the number of nodes that remain idle and the job waiting times. Scheduling policies are in charge to decide the order in which jobs are launched. Scheduling decisions may be based on different variables, such as job size, user priority or system status. Allocation algorithms map jobs onto available resources (typically, processors). Locality-aware policies select resources taking into account network characteristics, such as its topology or the distance between processors.

The most commonly used scheduling policies are FCFS (First-Come First-Serve) and FCFS + backfilling, sometimes with variations. The FCFS discipline imposes a strict order in the execution of jobs. These are arranged by their arrival time and order violations are not permitted, even when resources to execute the first job are not available but there are enough free resources to execute some other (or others) jobs in the queue. The main drawback of this policy is that it produces severe system fragmentation because some processors can remain idle during a long period of time due to the sequentially ordered execution of jobs. Idle processors could be used more efficiently running less-demanding jobs, thus achieving a performance improvement.

With the goal of minimizing the effect of this strictly sequential execution order, several strategies have been developed [1], backfilling being the most widely used due to its easy implementation and proven benefits. This policy is a variant of FCFS, based on the idea of advancing jobs through the queue. If some queued jobs require a smaller amount of processors than the one at the head, we can execute them until the resources required by the job at the head become available. This way, utilization of resources is improved because both network fragmentation and job waiting times decrease. The reader should note that, throughout this paper, we will often use the word *network* to refer to the complete parallel system.

Network fragmentation caused by scheduling algorithms is known as external fragmentation [5]. But a different kind of fragmentation appears in topologies like meshes or tori when the partitions reserved to jobs are organized as sub-meshes or sub-tori; for example, to allocate a job composed by 4x3 processes, some algorithms search for square sub-meshes, 4x4 being the smallest size that can be used to run the job. In this case, four processors reserved for the job will never be used. This effect is named internal fragmentation [5]. Some job allocation algorithms try to minimize this effect. However, this work *does not* consider this effect, because each parallel job will be assigned to the exact number of required nodes.

Neither FCFS nor backfilling are allocation algorithms, as they do not take into account the placement of job processes onto network nodes. In a parallel system, application processes (running on network nodes) communicate interchanging messages. 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. If we have several parallel jobs running in the same network, each of them randomly placed along the network, communication locality inside each job will not be exploited; and what is more, messages from different applications will compete for network resources, greatly increasing network contention. An effective exploitation of locality results in smaller communication overheads, which reflects in lower running times. Note that searching for this locality is expensive in terms of scheduling time, because jobs cannot be scheduled until contiguous resources are available (and allocated), so that network fragmentation increases. In order to avoid this effect, we propose the utilization of quasi-contiguous allocation schemes in which some restrictions of the purely-contiguous policy are relaxed, allowing the non-contiguous allocation of part of the required network nodes.

This way network occupancy can be increased, at the cost of some penalty in terms of application run times.

A trade-off has to be found between the gains attainable via exploitation of locality and the negative effects of increasing fragmentation. This is precisely the focus of this paper. We study only the placement in k-ary n-tree topologies [6], but the tools and methodology presented here will be extended to other topologies such as meshes or tori. Our final goal is to demonstrate that the introduction of locality-aware policies in the schedulers may provide important performance improvements in systems with multiple users and different applications.

The rest of the paper is organized as follows. In Section 2 we discuss some previous work on scheduling and allocation policies, describing in Section 3 those used in this paper. The simulation environment and the workloads used for the experiments are described in Section 4. Section 5 analyze a few preliminary experiments that provide evidence of the pros and cons of consecutive allocation schemes. These experiments are further elaborated in Section 6, that focuses on the search of a trade-off between application speedup and scheduling slowdown. Section 7 closes the paper with some conclusions and future lines of research.

2 Related Work

Extensive research has been conducted in the area of parallel job scheduling. Most works were focused on the search of new scheduling policies that minimize job waiting times, and on allocation algorithms that minimize network fragmentation. In [1] authors analyzed a large variety of scheduling strategies; however, none of them took into account virtual topologies of applications (the logical way of arranging processes to exploit communication locality) or network topology.

To our knowledge, only [5] described a performance study of parallel applications taking into account locality-aware allocation schemes. The starting point of this job was the fact that, in schedulers optimized for certain network topologies (they focused on meshes and tori), allocation was always done in terms of sub-meshes (or sub-tori). This policy optimized communication in terms of locality and non-interference, but caused severe fragmentation, both internal and external. The authors did not use scheduling with backfilling, a technique that would partly reduce this undesirable effect. However, they tested a collection of allocation strategies that sacrifice contiguity in order to increase occupancy. They claimed that the effect on application performance attributable to the partial loss of contiguity was low, and more than compensated by the overall improvement in system utilization.

A more recent paper [7] evaluated the positive impact that locality-aware allocations have on applications performance, but focused on three particular applications, running on supercomputers connected by 3-D interconnection networks.

Part of our experiments corroborates the conclusions of the cited papers. However, our work differs from them in several important aspects. Previous research work shows that, depending on the communication pattern of the application, contiguous allocation provides remarkable performance improvements [8]. Therefore, we do not make extensive use of non-contiguity to increase system utilization; instead, we incorporate backfilling scheduling policy into the scheduler. Additionally, we focus on k-ary n-trees, instead of meshes or tori.

A review of schedulers in use in current supercomputers, such as Maui, Sun Grid Engine, and PBS Pro, shows that they do not implement contiguous allocation strategies. Some of them provide methods for the system administrator to develop their own strategies but, in practice, this is rarely done. To our knowledge, the only two current schedulers that maintain the locality are the one used by the BlueGene family supercomputers [9] and SLURM. The BlueGene scheduler puts tasks from the same application in one or more midplanes of 8x8x8 nodes which decreases network contention and allows locality exploitation. SLURM performs always a best-fit algorithm building first a Hilbert curve through the nodes on the Sun Constellation and Cray XT systems in order to keep locality as higher as possible. In contrast, the scheduling strategy used by the default scheduler (PBS Pro) on Cray XT3/XT4 systems (also a custom-made 3D tori) simply gets the first available compute processors [10].

3 Scheduling and Placement Policies

We used simulation to carry out an analysis of the impact that contiguous and quasi-contiguous allocation strategies have on scheduling performance. Our simulator implements two different scheduling policies (FCFS with and without backfilling), as well as three allocation algorithms (non-contiguous, contiguous, and quasi-contiguous) implemented for k-ary n-trees. The workloads used to feed the simulations have been obtained from actual supercomputers and are publicly available at the Parallel Workload Archive [11].

The details of the scheduling algorithms used in the experiments are as follows:

1. **First Come First Serve (FCFS):** In this policy, jobs are strictly processed in arrival order and executed as soon as there are enough available resources. The scheduling process is stopped until this condition is reached, even if there are enough free resources that could be allocated to other waiting jobs.
2. **Backfilling (BF):** This strategy permits the advance of jobs, even when they are not at the head of the queue, in such a way that system utilization increases, but without delaying the execution of the jobs that arrived first. The mechanism works as follows. A reservation for the first job in the queue is done, if enough resources are not currently available; the reservation time is computed taking into account the estimated termination time of currently running jobs. Other waiting jobs demanding fewer resources may be allowed to run while the first one is waiting. When the time of the reservation is reached, the waiting job has to run; if at that point resources are not available, some running, advanced jobs must be killed, because otherwise the reservation would be violated. This way, the starvation of the first job is avoided. Reservations are computed using a parameter called User

Estimated Runtime, which represents a user-provided estimation of the job execution time [12]. In some cases the scheduling system itself may provide this value, based on estimations made over the historical system logs [13].

Other scheduling methods have been proposed in the literature, such as SJF (Shortest Jobs First [1]) which selects the jobs to be executed by their size instead of their arrival time, and several variations of backfilling (see [1]). However, the most commonly used algorithm in production systems is the EASY backfilling [1], also known as aggressive backfilling. EASY performs reservations only over the first job in the queue. This is the policy used in this study.

Regarding the allocation algorithms, the following are included in the study:

1. **Non-contiguous:** This policy performs a search of free nodes making a sequential search over them, ignoring the locality. This is the most used technique in commercial systems, like the Cray XT3/XT4 systems, that simply gets the first available compute processors [10]. This scheme provides a flat vision of the network, ignoring its topological characteristics and the virtual topologies of scheduled applications [4]. Note that in the long run it behaves as a random allocation of resources.
2. **Contiguous:** In this scheme job processes are allocated to nodes maintaining them as close as possible. To minimize the distance between processes (nodes) in a k-ary n-tree, we have defined the concept of level of a job. This level is related to the number of stages in the tree (n), and the number of ports per switch (k up and k down) [6]. Stage 1 corresponds to switches at the bottom of the tree, *i.e.*, those directly connected to compute nodes. Small jobs of less than k nodes can be allocated to a collection of nodes attached to the same stage-1 switch, without requiring communication involving switches in upper stages of the tree. These are level-1 jobs. However, jobs larger than k will require the utilization of switches at stages 2, 3, etc. In general, up to k^i nodes can be allocated using stage-i switches.
3. **Quasi-contiguous:** This algorithm is a relaxed version of the previous one. It searches nodes that are contiguously allocated but, if the required number of free nodes is not found at the job level, it searches for the remaining nodes using switches *one* level above; contiguity is partly kept. The threshold of required-but-not-found free nodes that triggers the search on a higher level is a parameter provided to the algorithm, and the value providing best results is highly dependent on the size and type of the jobs that are executed in the systems. This parameter, which we call *qct* (quasi-contiguity threshold) is actually a percentage of the job size representing the number of tasks of that job allowed to be allocated using one extra level of the tree. Using this equation

$$max_{j \in J} = \left\lceil \frac{qct}{100} \times size_j \right\rceil \quad . \tag{1}$$

the algorithm computes $max_{j \in J}$, the maximum number of tasks of the job j allowed to be allocated using switches at the next level.

The utilization of additional stages of the tree may increase network contention, so we try to keep it under control by reducing the number of messages traversing high-level switches. To do so, we maintain the maximum possible number of nodes under switches belonging to the same level; actually, in favorable conditions this algorithm behaves exactly like the purely contiguous one. However, as some tasks can be assigned to non-contiguous portions of the network, external fragmentation is reduced. The *qct* threshold will maintain the number of quasi-contiguously allocated tasks limited, in order to reduce the interference created by the messages of different applications.

The contiguous algorithm starts computing the level to which the job belongs, and the size of this level (*level_ size*, the number of compute nodes below a single switch located at that level, which is the maximum size of a job that can be contiguously allocated below that level). After this preliminary step, the search of free nodes is performed, in groups of *level_ size* nodes following a first fit allocation scheme, because this way all the allocated nodes would be contiguous, that is, connected by the same switch or switches at the required level. If the complete tree is traversed but the necessary number of nodes has not been found, the job cannot be allocated. For example, in a 4-ary 3-tree topology, if we need

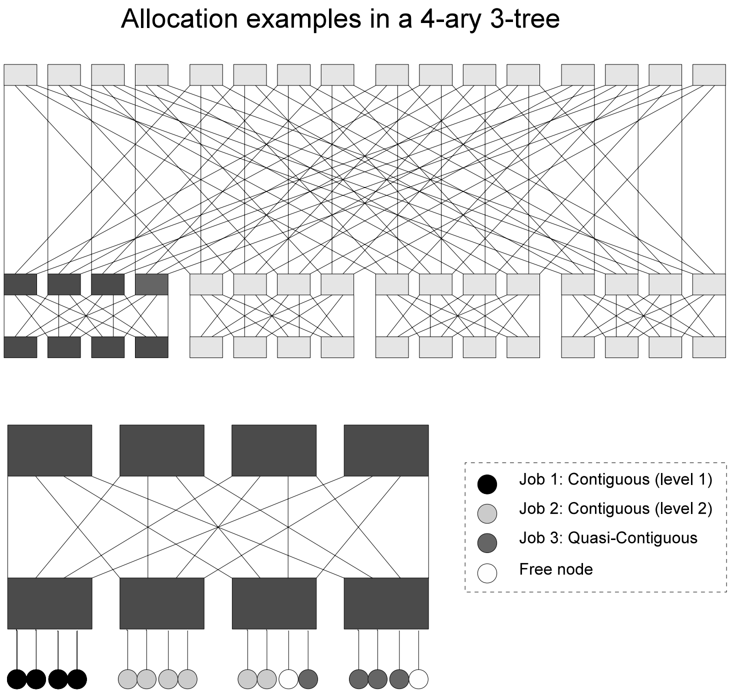

Allocation examples in a 4-ary 3-tree
Job 1: Contiguous (level 1)
Job 2: Contiguous (level 2)
Job 3: Quasi-Contiguous
Free node

Fig. 1. Top: a 4-ary 3-tree; compute nodes are not represented for the sake of clarity. Bottom: a section of the network, with some examples of allocated jobs.

to allocate a 4-node job, we have to find a completely empty stage-1 switch. For a 6-node job (level-2) we need to find 6 free nodes that are connected using only stage-1 and stage-2 switches.

The quasi-contiguous algorithm requires two steps. Firstly, it performs a search for contiguous partitions as we stated before. If not found, because there are not enough free nodes at the job level, and the percentage of non-allocated tasks is below the *qct* threshold, the search continues in the level above. For example, in a 4-ary 3-tree topology, if we need to allocate a 4-node job, we start searching for completely empty stage-1 switches but, if none is available, another search is performed using stage-2 switches.

In Figure 1 we represent some simple allocation examples in a 4-ary 3-tree topology. We can observe how Job 1, of size 4, can be allocated into a single stage-1 switch; this is a contiguous allocation. The level of Job 2, of size 6, is 2; this means that it is allocated to two stage-1 switches that directly connected via switches at stage 2. Therefore, allocation of Job 2 is also contiguous. Job 3 is quasi-contiguously allocated because it should be a level-1 job (size is 4) but it requires the utilization of stage-2 switches.

4 Description of the Workloads

As we stated before, in this work we evaluate the performance of schedulers using logs of workloads extracted from real systems that are available from the PWA (Parallel Workload Archive, [11]). These logs have information about the system as described in the SWF format (Standard Workload Format) [14]. In this study we used the following fields:

1. **Arrival Time:** The timestamp at which a job arrives to the system queue. Logs are sorted by this field.
2. **Execution Time:** The interval of time that the job was running in the system. In order to simulate the improvement of performance due to the exploitation of communication locality, we scale this field by applying a speed-up factor.
3. **Processors:** Number of processors required by the job.
4. **User Estimated Runtime:** This information is used only by the backfilling scheduling policy and represents a user estimation of the job execution time.
5. **Status:** This field represents the status of a job. Jobs can fail, or be cancelled by the user or by the system, before or after they started the execution. Some studies do not include in the simulations those jobs that were not successfully completed (due to failure or cancellation), but we consider important all the jobs because they stayed in the queues, delaying the execution of other jobs.

In our experiments, all times were measured in minutes. We only used workloads that provide User Estimated Runtime information, because of the need of this parameter to perform a backfilling scheduling policy.

In [15], the authors suggested a metric to measure the *load* managed by the scheduler. Selecting workloads with different values of this metric allows us to check our proposals on different scenarios. The *load* is computed as follows:

$$load = \left(\frac{\sum_{j \in J} size_j \times runtime_j}{P \times (T_{end} - T_{start})} \right) \quad . \tag{2}$$

where P is the number of processors, J is the set of jobs between T_{start} and T_{end}, T_{end} is the last termination time and T_{start} is the last arrival time of the first 1% of the jobs. This 1% of firstly arrived jobs and the jobs that terminate after the last arrival are removed, in order to reduce warm up and cool down effects.

From the workloads available at the PWA, we have selected these three:

1. **HPC2N (High Performance Computing Center North).** This is a system located in Sweden, composed by 240 compute nodes and using the Maui scheduler. The workload log contains information of 527,371 jobs. Load: 0.62.
2. **LLNL Thunder (Lawrence Livermore National Laboratory).** This is a Linux cluster composed by 4008 processors in which the nodes are connected by a Quadrics network. The scheduler used in this system is Slurm. The log is composed by 128,662 job records. Load: 0.76.
3. **SDSC BLUE(San Diego Supercomputer Center).** This system is an IBM SP located in San Diego, with 1152 processors. The scheduler in use is Catalina, developed at SDSC, and performs backfilling. The log contains information of 243,314 jobs. Load: 0.86.

We simulated these workloads in k-ary n-trees adapted to each system sizes. For the first workload we have simulated a 4-ary 4-tree with 256 nodes. For the other two we have used a 4-ary 6-tree with 4096 nodes. The number of nodes of the topologies does not match with the nodes of the workloads, so we have considered that the extra processors are not installed and they are ignored in the simulation.

5 Costs and Benefits of Contiguous Allocation Policies

Parallel applications performance depends on many factors, such as the communication pattern, distance between the application tasks, network contention, etc. The first one is an application-dependent characteristic, but the others are affected by the way the application is allocated.

A contiguous allocation strategy reduces the distance between the application tasks, to accelerate the interchange of messages and to reduce network utilization. An important, additional effect is that interference with other running applications is also reduced. This interference, that causes contention for network resources, may result in severe performance drops. Therefore, the contiguous allocation of a job improves the overall performance of the system, not only of that job.

In [8], the authors evaluate the possible benefits of contiguity for a collection of parallel applications. These benefits are highly dependent on the communication patterns of the applications. However, as we will show, the search of contiguity can be very expensive in terms of scheduling time. The execution of jobs may be delayed for a long time, until the required resources are available, the external fragmentation increases and the overall system utilization suffers. To minimize these negative effects we have introduced the concept of quasi-contiguity, a relaxed version of the contiguous allocation scheme which is expected to be less harmful in terms of scheduling time, while providing the same (or nearly the same) benefits in terms of application acceleration.

In order to validate the benefits of a contiguous and quasi-contiguous allocation policy, we have carried out several simulations using the INSEE simulator [16]. This tool does not simulate a scheduling algorithm, just the execution of a message-passing application on a multicomputer connected via an interconnection network. To feed this simulator we need traces of the messages interchanged by the communicating tasks. We have obtained these traces using a selection of the well-known NAS Parallel Benchmarks (NPB [17]). INSEE performs a detailed simulation of the interchange of the messages through the network, considering network characteristics (topology, routing algorithm) and application behavior (causality among messages). The output is a prediction of the time that would be required to process all the messages in the application, in the right order, and including causal relationships. Therefore, it only measures the communication costs, assuming infinite-speed CPUs. When using actual machines, a good portion of the time (ideally, most of the time) would be devoted

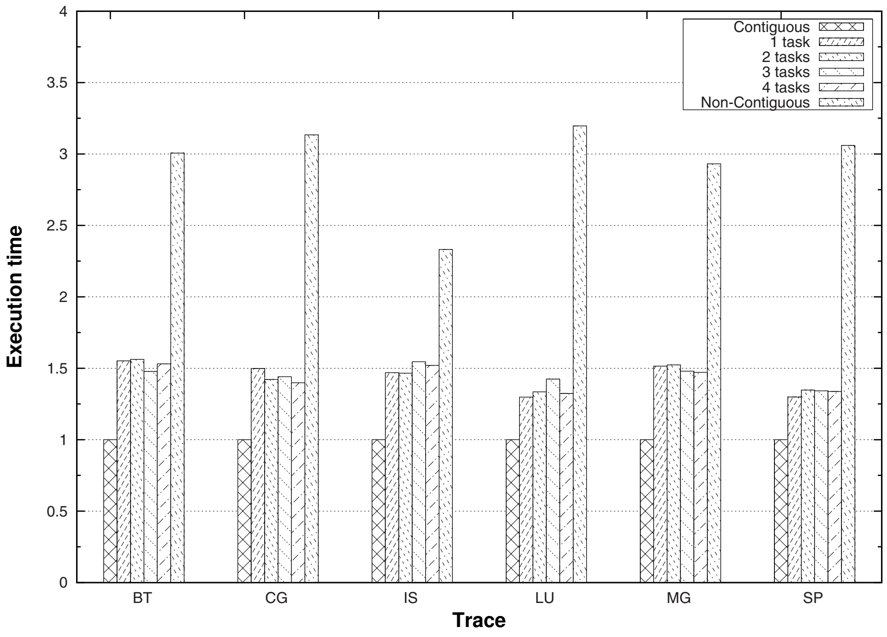


Fig. 2. Execution time for different allocation policies simulating the traces of some NAS Parallel Benchmarks in a 4-ary 4-tree topology. Values are normalized, so that 1 represents the contiguous allocation.

to CPU processing, and the impact of accelerated communications in overall execution time would be smaller.

The simulated topology is a 4-ary 4-tree, with 256 nodes. Instead of one application, we simulate the simultaneous execution of sixteen instances (jobs) of the same application (actually, trace), each one using sixteen nodes. The sixteen jobs have been allocated onto the network using three strategies:

1. **Contiguous:** Each job is allocated onto four level-2 switches, so the communications between tasks of the same job never need links or switches at level 3.
2. **Quasi-Contiguous:** In this strategy, we allow a partial non-contiguous allocation of the job tasks. The four experiments performed allow the non contiguous allocation of 1, 2, 3 or 4 tasks of each job, respectively.
3. **Non-Contiguous:** Tasks of each job are distributed along all the switches at level 4 (the maximum level of this tree). This means that intra-job communications do use level-4 switches, and also that messages of different jobs compete for network resources.

Figure 2 shows the execution time of each application using each strategy normalized to the time required by the contiguous placement. The benefits of contiguous allocation strategies are clear: non-contiguously allocated applications

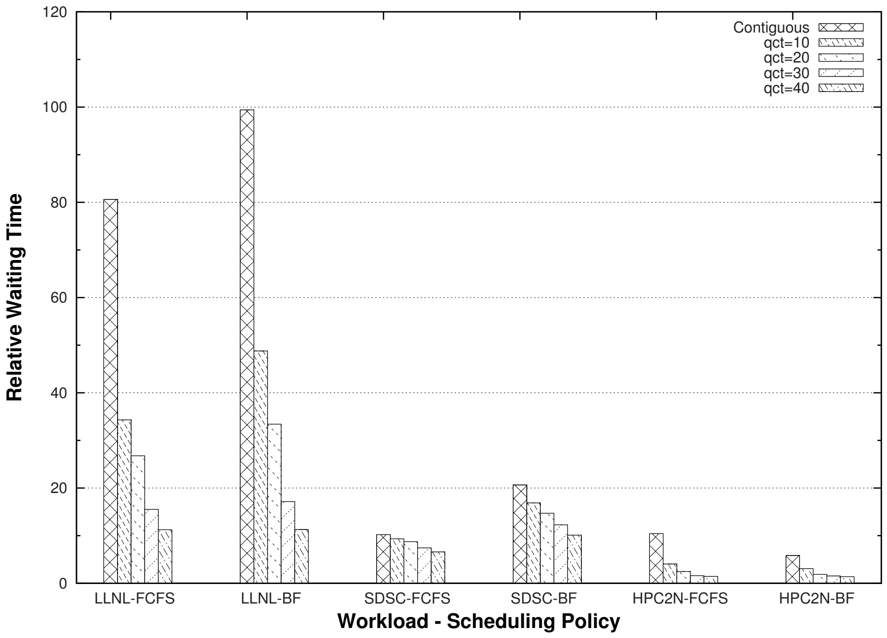


Fig. 3. Cost of contiguous and quasi-contiguous allocation, in terms of waiting time. A value of 1 would represent the average job waiting time for the non-contiguous allocation with the same scheduling policy.

run between 2 and 3 times slower. Regarding the quasi-contiguous allocation, we can appreciate that performance is always good, being only 30-50% higher to that obtained with purely contiguous allocation. These results confirm our expectations: a good allocation strategy can substantially reduce the execution time of a set of applications sharing a parallel computer as stated in [8].

Now we will asses the real cost of contiguity on scheduling. Using the scheduling simulator with the selected workloads (those from the PWA), we measure application waiting time for FCFS and backfilling scheduling algorithms, for purely contiguous allocation and quasi-contiguous allocation for four values of *qct*: 10, 20, 30 and 40%. Results are plotted in Figure 3. Note that values are relative to those obtained with the same workload and scheduling using non-contiguous allocation. Results are devastating: waiting times can be up to 100 times worse if contiguity is a requirement. Values are better for quasi-contiguity, but still bad. However, note that we *did not* take into consideration the acceleration that jobs experience due to better allocation. We will explore this issue in the next section. It is remarkable the difference between the LLNL workload waiting times and the other workloads waiting times, due to the presence of big size jobs (some of them of 1024 nodes). Finding contiguous partitions of this size is quite difficult, which results in longer waiting times for them and for the jobs that follow.

6 Tradding Off Costs and Benefits of Contiguous Allocation

In this section we carry out a collection of experiments to thoroughly evaluate the effect that contiguous allocation may have on scheduling performance. In these experiments we consider that contiguous allocation is able to accelerate the execution of parallel jobs. However, the actual values of attainable speed-ups are not available to us – they strongly depend on the communication characteristics of the applications, something that requires an exhaustive knowledge of each and all the applications included in the workload logs. We do not have that knowledge. For this reason, we introduce speed-up as a *parameter* of the simulation. With this setup we are able to know to what extent a certain level of application speed-up compensates the performance drop introduced by a restrictive allocation policy. This parameter is applied only to the parallel applications of the workload remaining the sequential jobs with the same runtime.

We have studied several combinations of scheduling and allocation policies. We evaluate them in terms of these two measurements:

1. **Job waiting time.** The time jobs spent in the queue.
2. **Job total time.** All the time spent in the system, which includes the time waiting at the queue and the execution time.

As stated before, when using contiguous and quasi-contiguous allocation, a speed-up factor has been applied to reduce the execution time. Note again that applying a speed-up factor to a running time improves not only the application finish time, but also reduces the time spent by the jobs using system resources; and therefore, the scheduling performance is increased too. In the simulations we used the workloads from the PWA described in Section 4.

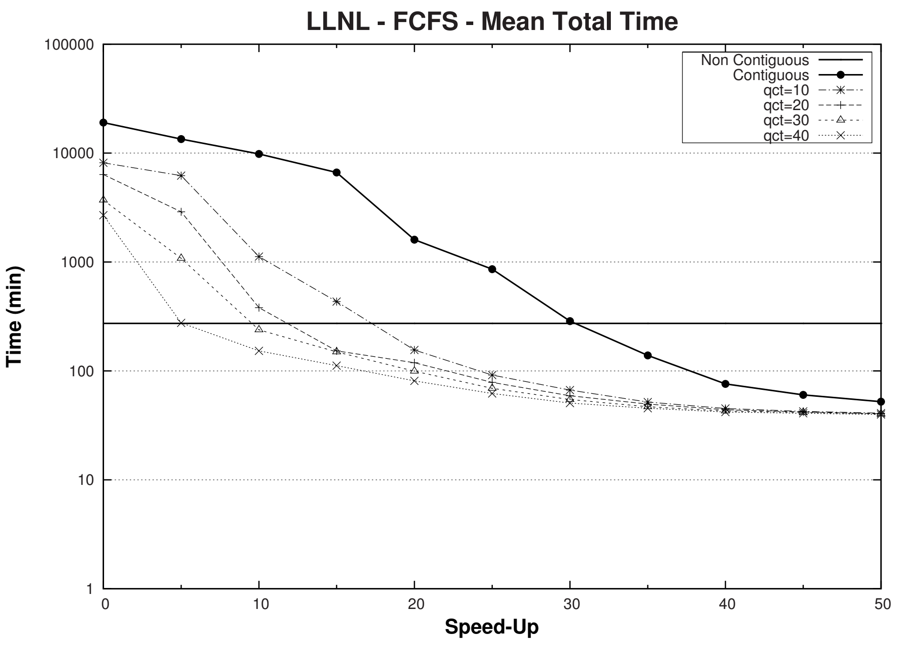
The quasi-contiguous strategy has been evaluated with four values of *qct*. Results are depicted in Figures 4, 5, 6 and 7. Note that, as the range of values is very wide, we used a logarithmic scale in the Y axis of all figures. We represent the averages of total time (waiting plus running) and, in some cases, waiting time alone. In each graph we can see six lines, one per allocation policy. Tested speed-up factors range from 0% to 50%. When this factor is 0% it means that, although the scheduler seeks contiguity, using it does not accelerate program execution. In all other cases we accelerate the execution times reported in the logs using the indicated speed-up factors (a value of 10% means that the execution requires 10% less time to be executed with that allocation scheme). Obviously, we cannot assume any acceleration with non-contiguous allocation, and for this reason the corresponding line is flat.

Let us now pay attention to Figure 4, where the LLNL workload is studied in detail. In all scheduling-allocation combinations, results with speed-up=0 are as appalling as described in the previous section. However, when this value increases (that is, when applications really run faster when allocated contiguous resources) the picture changes. At speed-up values between 5% - 30% the contiguous and quasi-contiguous approaches show their potential. It is clear that the quasi-contiguous strategies prove beneficial at lower speed-ups than the purely contiguous. Also, note that if the scheduler uses backfilling, global system efficiency is higher (the workload is processed faster), and the thresholds at which contiguity is advantageous are lower.

Figure 5 shows the results of the same experiments, but from a different perspective. Only waiting times are shown. A direct comparison with the previous figure help us to determine which part of the total time is spent in the queue, and which part is running time. For the cases with small speed-ups, most of the time is waiting time. When applying a speed-up factor, running time is accordingly reduced, but waiting time is also reduced.

In Figures 6 and 7 we have summarized results for workloads HPC2N and SDSC. To be succinct, and given that the qualitative analysis performed with LLNL is still valid, we only show results of total times for the FCFS and backfilling. For the SDSC workload, the threshold at which contiguous and non-contiguous allocation starts being beneficial falls between 15% and 25% (higher than that of LLNL). Similar, although slightly lower, values required by HPC2N are between 10% and 25%.

In all figures, we can see the benefits of using the quasi-contiguous policy. The scheduler performs better and, as described in the previous section, the expected speed-ups would be only slightly lower that those attainable with contiguous allocation. We have to remark that the implementation of this strategy tries always to find first a contiguous allocation, and only uses non-contiguous nodes as the last alternative. Therefore, if we estimate that we can obtain a certain



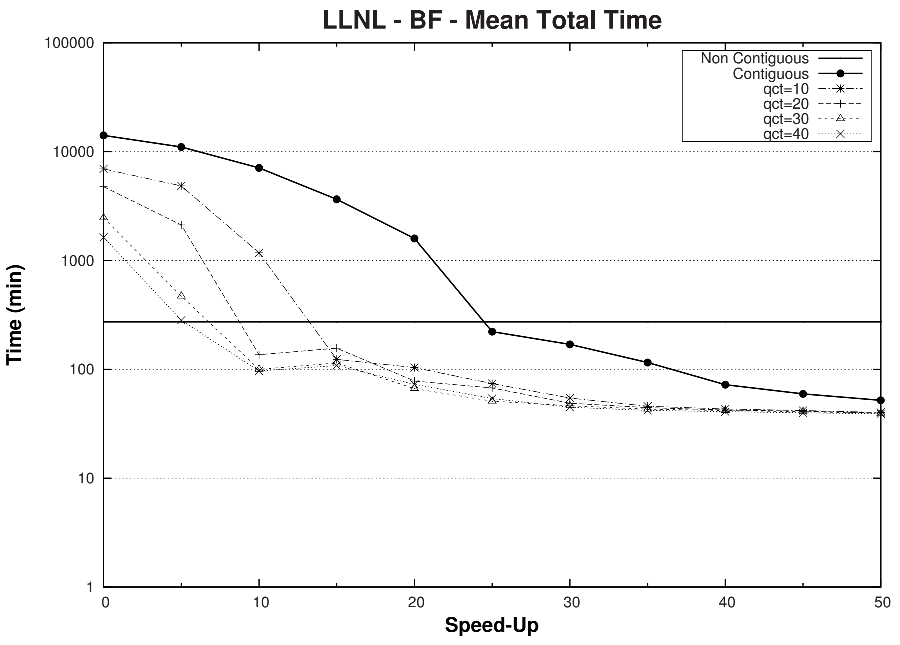
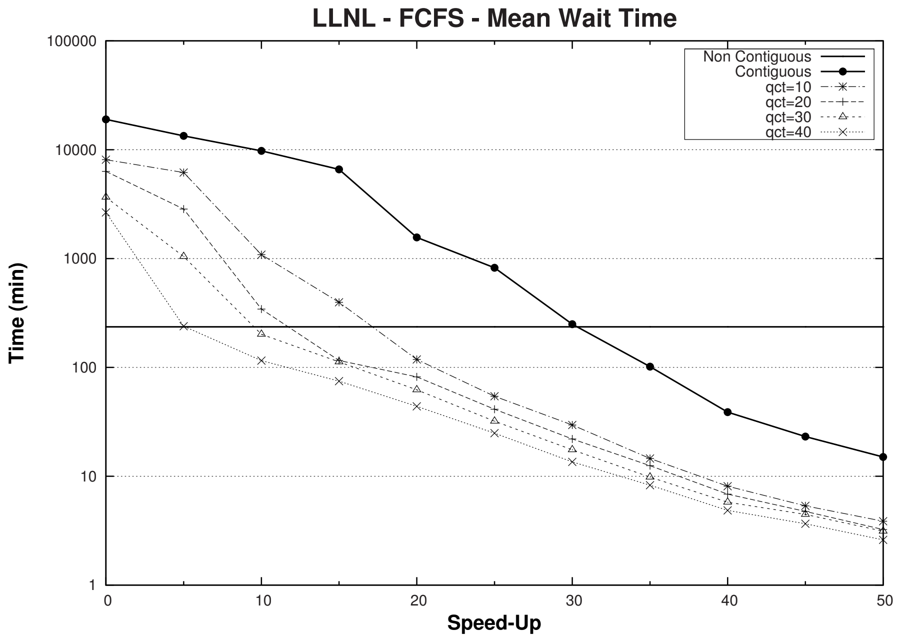


Fig. 4. Results of the experiments with the LLNL workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.



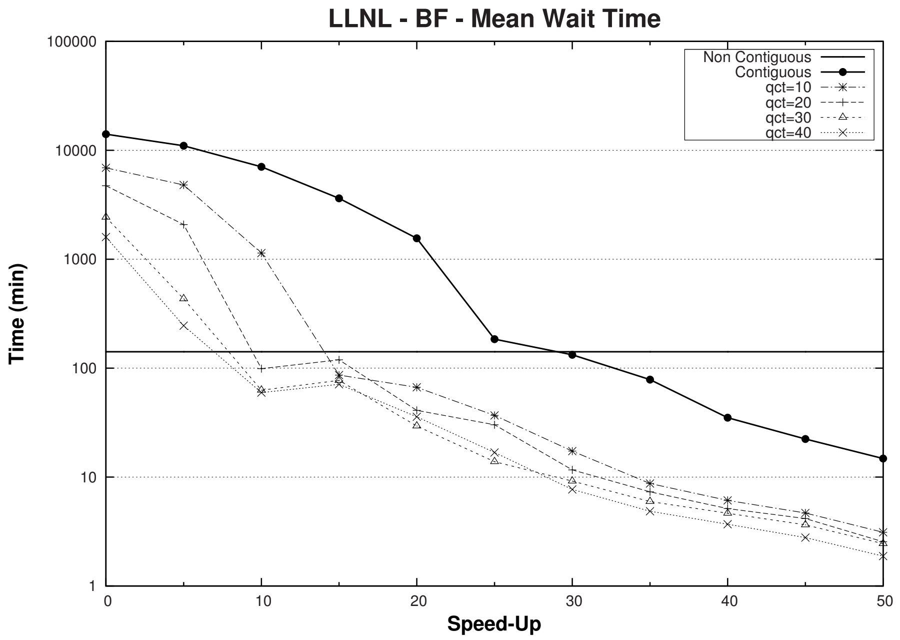
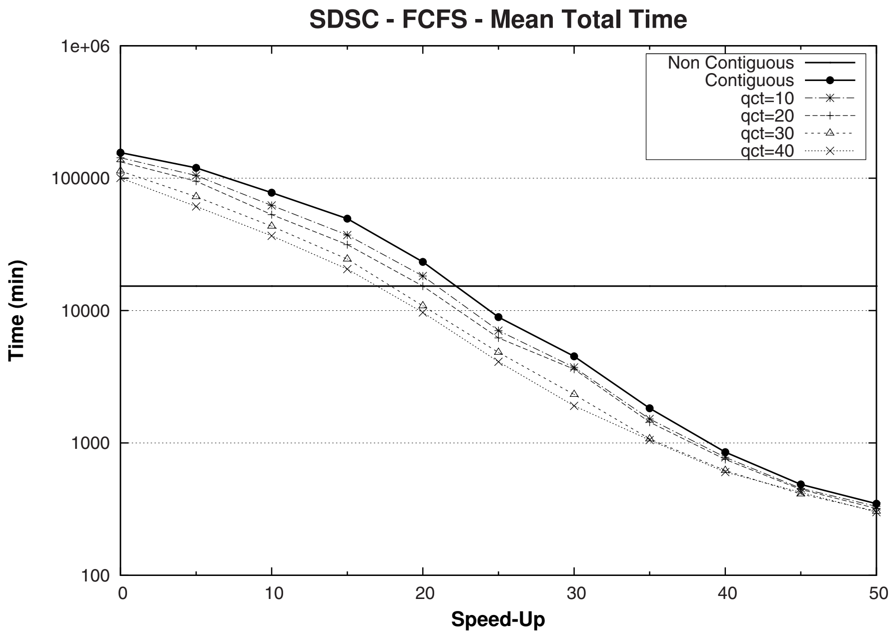


Fig. 5. Results of the experiments with the LLNL workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Wait Time at different speed-ups. The scale of the Y axis is logarithmic.



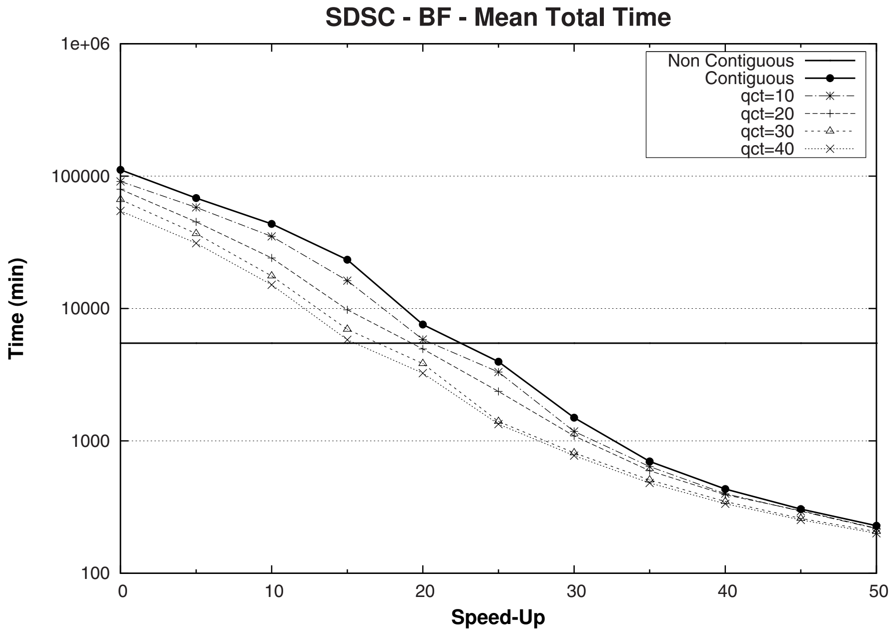
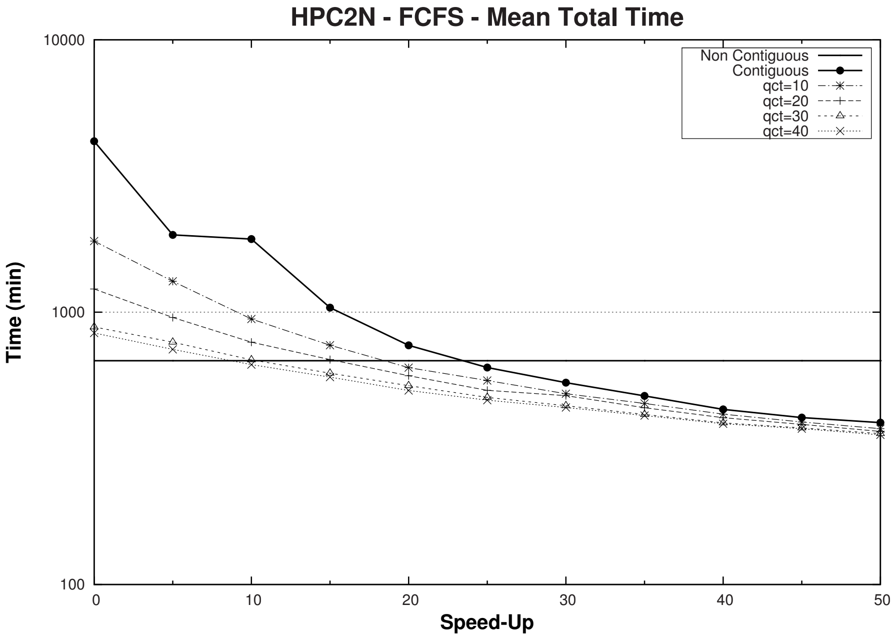


Fig. 6. Results of the experiments with the SDSC workloads for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.



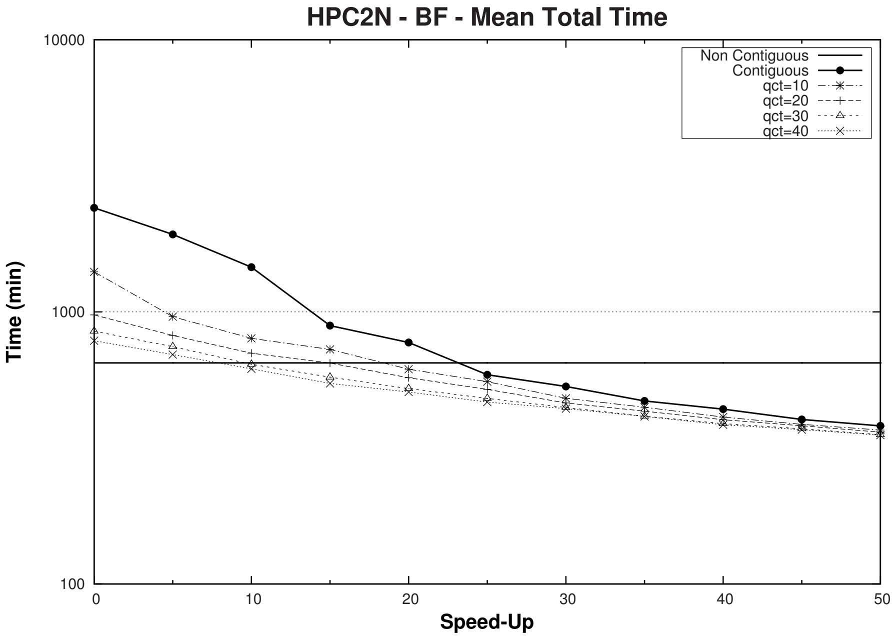


Fig. 7. Results of the experiments with the HPC2N workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.

speed-up when using a given value of *qct*, we will actually obtain better speed-ups, because in some cases the scheduler will obtain a contiguous allocation for the jobs.

Note that the increase of the *qct* parameter results in an equalization of the FCFS and backfilling performance reducing the difference between them. The reason is that the quasi-contiguous allocation strategy has a similar effect to the backfilling policy allowing the schedule of more jobs and thus, reducing the waiting time in the queue.

7 Conclusions and Future Work

Most current supercomputing sites are built around parallel systems shared between different users and applications. The optimal use of resources is a complex task, due to the heterogeneity in user and application demands: some users run short sequential applications, while others launch applications that use many nodes and need weeks to be completed.

Supercomputers are expensive to build and maintain, so that conscious administrators try to keep utilization as high as possible. However, the efficient use of a parallel computer cannot be measured only by the lack of unused nodes. Other utilization characteristics, although not that evident, may improve the general system performance.

In this paper we have studied the impact on performance of allocation and scheduling policies. We compared two scheduling techniques combined with three allocation algorithms in a k-ary n-tree network topology. Allocation algorithms that search for contiguous resources have an elevated cost in terms of system fragmentation, but also are able to accelerate the execution of applications. With the quasi-contiguous allocation, this acceleration is slightly penalized but the scheduling performance is significantly improved.

Experiments with actual workloads demonstrate that the cost of contiguous allocation is very high, but when the improvement of run time experienced by jobs is around 20-30%, this cost is compensated. Using relaxed versions of the contiguous allocation strategy (which we have called quasi-contiguous) this threshold lowers significantly, in such a way that in some cases speed-ups around 10% are enough to provide improvements in terms of scheduling efficiency.

This study has focused only in tree-based networks; the next step will be a performance study for other topologies (in particular, for k-ary n-cubes and k-ary n-tori). Because of the highly dependency of the allocation algorithms on the underlying topology, new quasi-contiguous allocation strategies should be developed for each new studied topology. We have provided application acceleration as a simulation parameter, although we know that the real acceleration depends heavily on the communication pattern of the applications, and on the way processes are mapped onto system nodes. For this reason, we plan to perform more complex simulations, in which the actual interchanges of messages are considered; to that end, we plan to integrate INSEE [16] into the scheduling simulator.

Finally, we plan to implement our allocation techniques into a real (commercial or free) scheduler in order to make real measurements in production environments with real applications.

References

1. 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)
2. Gupta, E.K.S., Srimani, P.K.: Subtori Allocation Strategies for Torus Connected Networks. In: Proc. IEEE 3rd Int'l Conf. on Algorithms and Architectures for Parallel Processing, pp. 287–294 (1997)
3. Choo, H., Yoo, S.M., Youn, H.Y.: Processor Scheduling and Allocation for 3D Torus Multicomputer Systems. IEEE Transactions on Parallel and Distributed Systems 11(5), 475–484 (2000)
4. Mao, W., Chen, J., Watson, W.I.: Efficient Subtorus Processor Allocation in a Multi-Dimensional Torus. In: HPCASIA 2005: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region, Washington, DC, USA, p. 53. IEEE Computer Society, Los Alamitos (2005)
5. Lo, V., Windisch, K., Liu, W., Nitzberg, B.: Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers. IEEE Transactions on Parallel and Distributed Systems 8, 712–726 (1997)
6. Petrini, F., Vanneschi, M.: Performance Analysis of Minimal Adaptive Wormhole Routing with Time-Dependent Deadlock Recovery. In: IPPS 1997: Proceedings of the 11th International Symposium on Parallel Processing, Washington, DC, USA, p. 589. IEEE Computer Society, Los Alamitos (1997)
7. Bhatele, A., Kale, L.V.: Application-specific Topology-aware Mapping for Three Dimensional Topologies. In: Proceedings of Workshop on Large-Scale Parallel Processing (held as part of IPDPS 2008) (2008)
8. Navaridas, J., Pascual, J.A., Miguel-Alonso, J.: Effects of Job and Task Placement on the Performance of Parallel Scientific Applications. In: Proc 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Weimar, Germany (February 2009)
9. Aridor, Y., Domany, T., Goldshmidt, O., Moreira, J.E., Shmueli, E.: Resource Allocation and Utilization in the Blue Gene/L Supercomputer. IBM Journal of Research and Development 49(2–3), 425–436 (2005)
10. Ansaloni, R.: The Cray XT4 Programming Environment, http://www.csc.fi/english/csc/courses/programming/
11. PWA: Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html
12. 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)
13. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. IEEE Trans. Parallel Distrib. Syst. 18(6), 789–803 (2007)

14. 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, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 67–90. Springer, Heidelberg (1999)
15. Tsafrir, D.: Modeling, Evaluating, and Improving the Performance of Supercomputer Scheduling. PhD thesis, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel (September 2006) Technical Report 2006–78
16. Ridruejo, F.J., Miguel-Alonso, J.: INSEE: An Interconnection Network Simulation and Evaluation Environment. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1014–1023. Springer, Heidelberg (2005)
17. NASA Advanced Supercomputer (NAS) division: Nas parallel benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html

Contention-Aware Scheduling with Task Duplication

Oliver Sinnen, Andrea To, and Manpreet Kaur

Department of Electrical and Computer Engineering, University of Auckland
Private Bag 92019, Auckland 1142, New Zealand
o.sinnen@auckland.ac.nz

Abstract. Scheduling a task graph onto several processors is a trade-off between maximising concurrency and minimising interprocessor communication. A technique to reduce or avoid interprocessor communication is task duplication. Certain tasks are duplicated on several processors to produce the data locally and avoid the communication among processors. Most algorithms using task duplication are for the classic model, which allows concurrent communication and ignores contention for communication resources. The recently proposed, more realistic contention model introduces contention awareness into task scheduling by assigning the edges of the task graph to the links of the communication network. It is intuitive that scheduling under such a model benefits even more from task duplication. This paper proposes a contention-aware task duplication scheduling algorithm, after investigating how to use task duplication in the contention model. An extensive experimental evaluation demonstrates the significant improvements to the speedup of the produced schedules.

1 Introduction

In the task scheduling area, a program is represented as a directed acyclic graph, called task graph, where the nodes represent the tasks and the edges represent the communications between the tasks. Scheduling such a task graph on a set of processors for fastest execution is a well known NP-hard optimisation problem [10] and many heuristics have been proposed [3,7,10,17].

Task duplication is a well known technique to reduce the necessary communication between processors. In this technique certain crucial tasks are executed on more than one processor. The data they procedure is then locally available on different processors and less communication has to be sent between the processors. Again, many algorithms have been proposed that incorporate this technique into scheduling [4,7,8,9].

The classic model used by most scheduling algorithms heavily idealises the target parallel system. It is assumed that all communication can happen at the same time and that all processors are fully connected, in other words there is no contention for communication resources. It is now more and more recognised that this classic model is not realistic and does not suffice for accurate and efficient task scheduling [1,5,15,16]. Contention aware scheduling algorithms depart from the classic model and schedule not only the tasks, but also the edges on the communication resources.

It is intuitive that avoiding or reducing interprocessor communication becomes more important under the contention model. Consequently, task duplication should be more beneficial under this model. To the authors' best knowledge however, no task duplication algorithm to be used under a contention model has been proposed. In this paper we propose a contention-aware task duplication scheduling algorithm. It works under the general contention model and its algorithmic components are based on state-of-the-art techniques used in task duplication and contention-aware algorithms. We investigate the changes to the scheduling model (Section 3) and discuss the proposed algorithm (Section 4). An extensive experimental evaluation shows that our algorithm is far superior to contention-aware algorithms that do not use task duplication and to task duplication algorithms under the classic model (Section 5). The next section gives a background on task scheduling, including the different models and basic algorithmic techniques.

2 Task Scheduling

The program to be scheduled is represented by a directed acyclic graph (DAG), called task graph, $G = (\mathbf{V}, \mathbf{E}, w, c)$. The nodes $\mathbf{V}$ represent the program's tasks and the edges $\mathbf{E}$ the communications between them. An edge $e_{ij} \in \mathbf{E}$ represents the communication from node n_i to node n_j. The positive weight $w(n)$ of node $n \in \mathbf{V}$ represents its computation cost and the non-negative weight $c(e_{ij})$ of edge $e_{ij} \in \mathbf{E}$ represents its communication cost.

The set $\{n_x \in \mathbf{V} : e_{xi} \in \mathbf{E}\}$ of all direct **predecessors** of n_i is denoted by **pred**(n_i) and the set $\{n_x \in \mathbf{V} : e_{ix} \in \mathbf{E}\}$ of all direct **successors** of n_i, is denoted by **succ**(n_i).

A schedule of a task graph on a target system consisting of a set $\mathbf{P}$ of **dedicated** processors is the association of a start time and a processor with each of its nodes: $t_s(n, P)$ denotes the **start time** of node $n \in \mathbf{V}$. Thus, the node's **finish time** is given by $t_f(n, P) = t_s(n, P) + w(n)$. The processor to which n is allocated is denoted by $proc(n)$. Further, let $t_f(P) = \max_{n \in \mathbf{V}:proc(n)=P}\{t_f(n, P)\}$ be the **processor finish time** of P and let $sl(\mathcal{S}) = \max_{n \in \mathbf{V}}\{t_f(n, proc(n))\}$ be the **schedule length** (or makespan) of $\mathcal{S}$, assuming $\min_{n \in \mathbf{V}}\{t_s(n, proc(n))\} = 0$. For such a schedule to be feasible, the following two conditions must be fulfilled for all nodes in G.

The **Processor Constraint** enforces that only one task is executed by a processor P at any point in time, which means for any two nodes n_i,$n_j \in \mathbf{V}$ that either $t_f(n_i, P) \leq t_s(n_j, P)$ or $t_f(n_j, P) \leq t_s(n_i, P)$ must be true.

The **Precedence Constraint** enforces that for every edge $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, the destination node n_j can only start after the communication associated with e_{ij} has arrived at n_j's processor P

$$t_s(n_j, P) \geq t_f(e_{ij}, proc(n_i), P). \tag{1}$$

$t_f(e_{ij}, P_{src}, P_{dst})$ is the edge finish time of e_{ij} communicated from P_{src} to P_{dst}, which is defined later, depending on the scheduling model.

2.1 Classic Scheduling

Traditionally, most scheduling algorithms have employed a strongly idealised model of the target parallel system [3,7,10,17].

Definition 1 (Classic System Model).
A parallel system $M_{classic} = (\mathbf{P})$ consists of a finite set of dedicated processors $\mathbf{P}$ connected by a communication network. This dedicated system has the following properties: i) local communication has zero costs; ii) communication is performed by a communication subsystem; iii) communication can be performed concurrently; iv) the communication network is fully connected.

Based on this system model, the edge finish time only depends on the finish time of the origin node and the communication time. The **edge finish time** of $e_{ij} \in \mathbf{E}$ is given by

$$t_f(e_{ij}, P_{src}, P_{dst}) = t_f(n_i, P_{src}) + \begin{cases} 0 & \text{if } P_{src} = P_{dst} \\ c(e_{ij}) & \text{otherwise} \end{cases} \tag{2}$$

Thus, communication can overlap with the computation of other nodes, an unlimited number of communications can be performed at the same time, and communication has the same cost $c(e_{ij})$, regardless of the origin and the destination processor, unless the communication is local.

2.2 List Scheduling

The scheduling problem is to find a schedule with minimal length. As this problem is NP-hard [10], many heuristics have been proposed for its solution. A heuristic must schedule a node on a processor so that it fulfils all resource and precedence constraints.

The best known scheduling heuristic is list scheduling as given in Algorithm 1. In this simple, but common, variant of list scheduling the nodes are ordered according to a priority in the first part of the algorithm. The schedule order of the nodes is important for the schedule length and many different priority schemes have been proposed [6,13,17]. A common and usually good priority is the node's **bottom level** bl, which is the length of the longest path leaving the node. Recursively defined it is

$$bl(n_i) = w(n_i) + \max_{n_j \in \mathbf{succ}(n_i)} \{c(e_{ij}) + bl(n_j)\} \tag{3}$$

2.3 Contention Aware Scheduling

The classic scheduling model (Definition 1) does not consider any kind of contention for communication resources. To make task scheduling contention aware,

Algorithm 1. List scheduling

```
1: Sort nodes $n \in \mathbf{V}$ into list $L$, according to priority scheme and precedence constraints.
2: for each $n \in L$ do
3:     Find processor $P \in \mathbf{P}$ that allows earliest finish time of $n$.
4:     Schedule $n$ on $P$.
```

and thereby more realistic, the communication network is modelled by a graph, where processors are represented by vertices and the edges reflect the communication links. The awareness for contention is achieved by edge scheduling [11], i.e. the scheduling of the edges of the DAG onto the links of the network graph, in a very similar manner to how the nodes are scheduled on the processors.

The network model proposed in [15] captures network [11,13] as well as endpoint contention [1,5]. This is achieved by using different types of edges and by using switch vertices in addition to processor vertices. Here, it suffices to define the topology network graph to be $TG = (\mathbf{P}, \mathbf{L})$, where $\mathbf{P}$ is a set of vertices representing the processors and $\mathbf{L}$ is a set of edges representing the communication links. The system model is then defined as follows.

Definition 2 (Target Parallel System – Contention Model). *A target parallel system $M_{TG} = (TG)$ consists of a set of processors $\mathbf{P}$ connected by the communication network $TG = (\mathbf{P}, \mathbf{L})$. This dedicated system has the following properties: i) local communications have zero costs; ii) communication is performed by a communication subsystem.*

The notions of concurrent communication and a fully connected network found in the classic model (Definition 1) are substituted by the notion of scheduling the edges $\mathbf{E}$ on the communication links $\mathbf{L}$. Corresponding to the scheduling of the nodes, $t_s(e, L)$ and $t_f(e, L)$ denote the **start** and **finish time** of edge $e \in \mathbf{E}$ on link $L \in \mathbf{L}$, respectively.

When a communication, represented by the edge e, is performed between two distinct processors P_{src} and P_{dst}, the routing algorithm of TG returns a **route** from P_{src} to P_{dst}: $R = \langle L_1, L_2, \ldots, L_l \rangle$, $L_i \in \mathbf{L}$ for $i = 1, \ldots, l$. The edge e is scheduled on each link of the route. For details on the scheduling of the edges on the links and the topology graph refer to [15].

It is important to realise that the edge scheduling only affects the scheduling of the tasks through a redefinition of the edge finish time, when compared with the classic model (eq. 2). Let $R = \langle L_1, L_2, \ldots, L_l \rangle$ be the route for the communication of $e_{ij} \in \mathbf{E}$ from P_{src} to P_{dst} if $P_{src} \neq P_{dst}$. The **edge finish time** of e_{ij} is

$$t_f(e_{ij}, P_{src}, P_{dst}) = \begin{cases} t_f(n_i, P_{src}) & \text{if } P_{src} = P_{dst} \\ t_f(e_{ij}, L_l) & \text{otherwise} \end{cases} \qquad (4)$$

Thus, the edge finish time $t_f(e_{ij}, P_{src}, P_{dst})$ is now the finish time of e_{ij} on the last link of the route, L_l, unless the communication is local. As nothing else changes for the scheduling of the tasks, most scheduling heuristics proposed for the classic model, can also be used under the contention model, thereby making them contention aware. This is in particular true for list scheduling [13].

3 Duplication in Contention Aware Scheduling

Scheduling a task graph is a trade-off between maximising the concurrency and minimising the interprocessor communication costs. It often happens that the advantage of executing tasks in parallel is negated by the associated interprocessor communication cost. It is intuitive that this is even more pronounced under the more realistic contention model, where contention can increase the communication delay.

Task duplication is a well known technique that tries to reduce the communication costs, by scheduling certain tasks on more than one processor. The function $proc(n)$ for the processor allocation of node n becomes a subset of P, denoted by $\mathbf{proc}(n)$. The communication from these duplicated nodes then becomes local on their allocated processors, avoiding costly interprocessor communication.

Many algorithms have been proposed using task duplication [4,7,8,9]. The irony is that most of them have been proposed for the classic model, even though avoiding interprocessor communication under the more realistic contention model can be more crucial. This paper proposes a novel task duplication algorithm for the contention model. In this section we will study the general consequence for the scheduling of the nodes and the next section proposes a contention aware task duplication algorithm. First, let us look at task duplication under the classic model.

Under the classic model, task duplication has an impact on the Precedence Constraint, eq. (1). Given the communication e_{ij}, the node n_j cannot start until *at least one* instance of the duplicated nodes of n_i has provided the communication e_{ij}. It is not necessary to define which instance of n_i is sending the data to n_j in case there is more than one instance that can provide it on time.

3.1 Under Contention Model

Task duplication under the contention model changes significantly. Under the contention model, it must be strictly defined from where a communication is sent if there are several instances of a sending task. Regard Figure 2 where the task graph of Figure 1(left) is scheduled under the contention model on four processors connected to a central ideal switch (Figure 1(right)). Ideal means there is no contention within the switch. The tasks A and B have been duplicated and only two communications are remote. Edge e_{AE} is scheduled on links L_2 and L_3 (route from P_2 to P_3) , and e_{AF} on links L_1 and L_4 (route from P_1 to P_4). In other words, both instances of A are sending out data, but each only one edge.

Because of the contention model, it is actually important that e_{AE} and e_{AF} are sent from different processors as can be observed in Figure 3, where both are sent from P_2. Due to contention on L_2, e_{AF} is delayed and therefore arrives one time unit later at P_4, which in turn increases the schedule length through F's later start time.

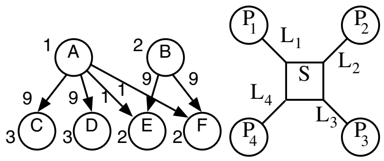


Fig. 1. Example task graph (left) and topology graph of four processor system (right)

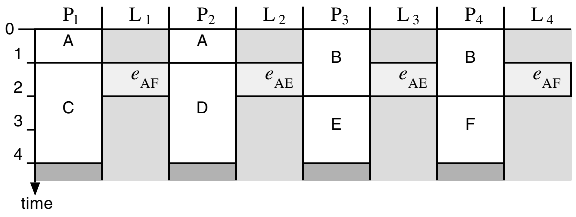


Fig. 2. Task duplication under contention model

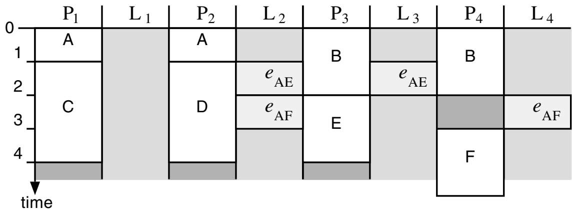


Fig. 3. Contention on L_2 delays communication e_{AF}, increases schedule length

The consequence from this observation is that it must be decided during the scheduling of the tasks and edges, which instance of a duplicated task sends the communication. As several instances of a node n_i might exist, e_{ij} might be sent several times to *different* processors, possibly from the same source processor.

As this duplication is done under the contention model, the finish time of the edge remains as defined in eq. (4), that is it corresponds to the finish time of the edge on the link entering the destination processor, for example in Figure 3 the finish time of e_{AF} is $f(e_{AF}, P_2, P_4) = f(e_{AF}, L_4) = 3$.

A scheduling algorithm must carefully choose from which task a communication is sent when several instances exist so that the communication edge can be scheduled and an accurate view of the contention is gained. Under the contention model, this choice is make by tentatively scheduling the edges on the links of the different routes to see from where the communication arrives first as will be seen in the following section [15].

4 Algorithm

The contention-aware task duplication scheduling algorithm proposed in this section is based on scheduling algorithms for the contention model and task duplication techniques used under the classic model. In the following we present and discuss its elements.

List scheduling. As the general algorithmic approach, list scheduling, as given in Algorithm 1, is chosen. List scheduling is easily adaptable to the contention model, as shown in [13]. In the first phase the nodes are ordered according to their bottom levels $bl(n)$, defined in (3), which was shown to be the superior node priority under the contention model in an extensive experimental evaluation [13]. Algorithm 2 outlines our proposed algorithm.

Algorithm 2. Contention-aware task duplication scheduling algorithm

```
 1: ▷ 1. Part:
 2: Sort nodes $n \in \mathbf{V}$ into list $L$, according to $bl(n)$
 3: ▷ 2. Part:
 4: for each $n \in L$ do
 5:     for each $P \in \mathbf{P}$ do
 6:         Tentatively schedule $n$, recursively duplicating $n$'s critical parent – record best
            finish time $t_f(n, P)$ and ancestors to be duplicated, if any
 7:     Let $P_{min}$ be processor where $n$ can finish earliest
 8:     Duplicate recorded ancestors of $n$ on $P_{min}$
 9:     Schedule $n$ on $P_{min}$
10:     Remove redundant tasks and their in-edges
```

Insertion technique. During list scheduling, each task can be scheduled between already scheduled tasks (insertion technique) or after the finish time of processor P (end technique). The same principle applies of course to the scheduling of the edges on the links. For the necessary tentative scheduling and the redundant task/edge removal (see below) the insertion technique is more suitable and hence employed.

Critical parent. An essential question for task duplication algorithms is which tasks should be duplicated. When a task n is scheduled on a processor P, the primary candidates for duplication are its predecessors $\mathbf{pred}(n)$, or parents. As task duplication algorithms have shown, it is usually not beneficial to duplicate all predecessors. The most important task to duplicate is the task from which the data transfer arrives the latest, called critical parent $cp(n)$ [4]. Under the contention model, this corresponds to the edge $e_{cp(n),n}$ with the highest finish time $f(e_{cp(n),n}, L_l)$ on the link L_l entering the processor P. If that communication $e_{cp(n),n}$ can be made local, task n might start earlier. Hence, our proposed algorithm considers the critical parent for duplication. The duplication is accepted if the task n can start earlier.

Recursive duplication. In some situations it can be more beneficial to not only duplicate the critical parent, but also considering the predecessors of the critical parent for duplication. Task duplication algorithms therefore consider the recursive duplication of the critical parent $cp(n)$, its critical parent $cp(cp(n))$ and so on [2]. This approach is adopted by our algorithm, whereby the recursive duplication goes as deep as it is most beneficial, i.e. as it reduces the start time of task n most.

Tentative scheduling. A characteristic aspect of scheduling under the contention model is the need to tentatively schedule edges on the communication links in order to obtain the data ready time of a task n, i.e. the time when all incoming edges have finished communication. For example, we search for the processor that allows task n_i's earliest finish time and n_i has the in-edges e_{li} and e_{ki}. Then, for each processor P, we must schedule the communication on the links of the route from $proc(n_l)$ and $proc(n_k)$ to P. That gives us an accurate data ready time of n_i on P. Before the next processor is considered, the edges must be removed from the schedule, hence tentative scheduling. With task duplication this tentative scheduling is even more involved as there might be more than one instance of n_l and n_k, as seen with task A in the example of Figure 2 and 3. Our algorithm therefore integrates tentative scheduling also on this level, i.e. the communication is tentatively scheduled from each instance of a predecessor task in order to find the best data provider.

Redundant task/edge removal. When a task n is duplicated on processor P, the original and other instances of n might have become redundant. This is the case, if one or more of these instances do not provide data to any predecessor. The redundant tasks can and should be removed from the schedule. Under the contention model, the removal of a task implies that also its in-edges can be removed from the links. Especially together with the insertion technique, the freed space can be used by subsequently scheduled tasks and their edges, potentially leading to shorter schedules. Our algorithm checks for and removes redundant tasks after the scheduling of each task.

Complexity. The complexity of contention-aware list scheduling with the insertion technique is $O(|\mathbf{V}|^2 + |\mathbf{P}||\mathbf{E}|^2 O(routing))$ [12]. $O(routing)$ is the complexity for finding the communication route in the network and its length, but is for many practically relevant systems $O(1)$. With our recursive task duplication the complexity increases to $O(|\mathbf{P}|^2(|\mathbf{V}|^3 + |\mathbf{V}||\mathbf{E}|^2 O(routing)))$.

5 Experimental Evaluation

Two questions need to be answered in the evaluation of the proposed algorithm: i) How do the schedules improve compared to a task duplication algorithm without contention awareness? ii) How does task duplication improve upon other contention-aware scheduling algorithms? To answer these questions, we have implemented four algorithms. The proposed contention-aware task duplication algorithm (CA-D) is compared with a contention-aware list scheduling (CA-LS) [13], which is essentially the same algorithm as CA-D, but without the duplication of tasks. Further, we implemented a task duplication (D) and a list scheduling algorithm (LS) under the classic model. Again, they are identical to CA-D and CA-LS, respectively, but without the contention awareness.

Schedules produced under the different models cannot be directly compared [14]. Usually, schedules under the contention-model are longer, but more realistic, resulting in shorter execution times. Hence to compare the schedule, we simulated contention for D and LS. This was done by rescheduling the D's and LS's schedules under the contention model [14]. To indicate this contention simulation we named D and LS in the following D-CS and LS-CS.

5.1 Setup

For the models of the parallel target systems we have chosen sets of processors (2, 8 and 15) connected to an ideal switch. Each processor has an out-going and an in-coming link connected to this switch, thus only one communication in each direct can take place at the same time. This corresponds to full-duplex communication ports and this model is also referred to the one-port model [1].

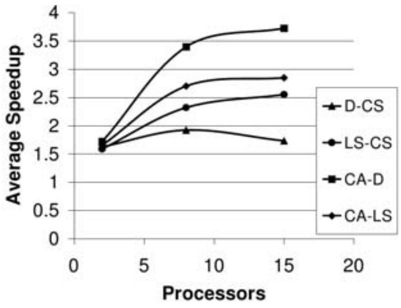
A large set of graphs was generated as the workload for the scheduling algorithms. This set comprised of graphs of seven types: In-trees, Out-trees, Series-Parallel (SP), Fork, Join, Fork-join and Random [12]. Within each type, graphs of different sizes were created (number of nodes= 20, 100, 500, 1000) with random node and edge weights, scaled to achieve different communication to computation ratios (CCR = 0.1, 1, 10) [12]. CCR is a measure for the importance of communication and is defined as the total edge weight over the total node weight $CCR = \frac{\sum_{e \in \mathbf{E}} c(e)}{\sum_{n \in \mathbf{V}} w(n)}$. In total about 2000 graphs were generated and scheduled.

5.2 Results

In this section the significant experimental results are shown and discussed. Regard Figures 4 and 5 that display the speedup over the number of processors for different graph types. The displayed values are average values across all different graphs of the same type. Speedup of a schedule $\mathcal{S}$ is defined as the sequential length of the graph over the schedule length $speedup(\mathcal{S}) = \frac{\sum_{n \in \mathbf{V}} w(n)}{sl(\mathcal{S})}$.

Contention aware (CA-D) vs. non-contention aware duplication (D-CS). The figures show that contention aware duplication (CA-D) is never worse than non-contention aware duplication (D-CS). In fact, CA-D produces greater speedup than D-CS for all graphs, except for fork graphs. The difference between the two algorithms is the greatest with SP graphs, where the speedup produced by CA-D on 15 processors is 120 percent greater than that of D-CS.

Figure 5(right) shows the average speedup across graph types produced by each algorithm for different CCR values on 15 processors. The average speedup values produced by the algorithms for high communication graphs (CCR = 10)



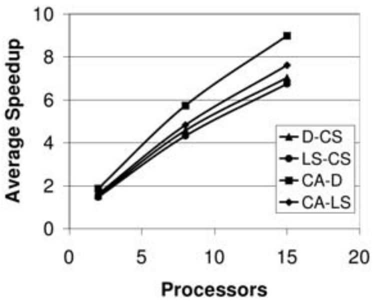
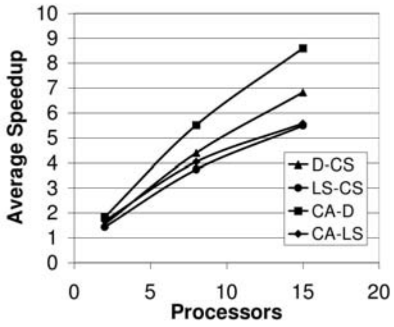


Fig. 4. Speedup over processors for SP-graphs (left) and random graphs (right)



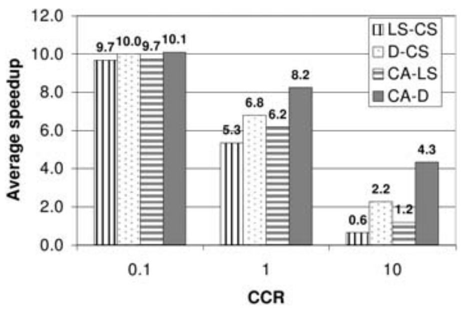


Fig. 5. Speedup over processors for out-trees (left) and speedup over CCR for all graphs on 15 processors (right)

show the greatest difference (95 percent) between contention aware duplication (CA-D) and non-contention aware duplication (D-CS). The difference is less, but still significant for medium communication graphs (contention aware duplication is 20 percent greater). As can be expected, contention aware duplication can excel most when the CCR value is medium to high, in other words when avoiding communication and contention is most important. To summarise, duplication under the contention model is significantly better than under the classic model.

Contention aware duplication (CA-D) vs. contention aware list scheduling (CA-LS). Task duplication has never been used in contention-aware algorithms. In this sub-section we are therefore evaluating if it improves the schedule length at least as much as it does under the classic model, so we compare CA-D with CA-LS, both contention aware algorithms, but only CA-D does duplication. As can be seen in the figures, CA-D has greater speedup on all numbers of processors for all graph types. Graphs with structures that benefit from task duplication (i.e., graphs where there is at least one node with more than one child) show the greatest difference in speedup. Speedup produced on

15 processors by CA-D is 54 percent greater than that of CA-LS for out-trees, 31 percent greater for SP graphs, and 18 percent greater for random graphs. Note that the difference between the non-contention aware algorithms D-CS and LS-CS is sometimes significantly less, e.g. for random-graphs. This is evidence supporting our hypothesis that task duplication is more important for scheduling under the contention model. To summarise, the duplication technique does significantly improve the list scheduling heuristic under the contention model for most task graphs, even more than under the classic model.

6 Conclusions

This paper proposed a novel contention-aware task duplication scheduling algorithm. It was studied how task duplication can be performed under the contention model. Based on this an algorithm was proposed using state-of-the-art scheduling techniques found in classic task duplication algorithms and other contention-aware algorithms.

An extensive experimental evaluation of the algorithm was performed, comparing the proposed algorithm with task duplication under the classic model and with a contention-aware algorithm without task duplication. This revealed very significant speedup gains, both compared to task duplication under the classic model and to other contention-aware scheduling algorithms without task duplication. As predicted, task duplication is even more beneficial under the contention model than under the classic model.

References

1. Beaumont, O., Boudet, V., Robert, Y.: A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In: HCW 2002, the 11th Heterogeneous Computing Workshop. IEEE Computer Society Press, Los Alamitos (2002)
2. Darbha, S., Agrawal, D.P.: Optimal scheduling algorithm for distributed-memory machines. IEEE Transactions on Parallel and Distributed Systems 9(1), 87–95 (1998)
3. Gerasoulis, A., Yang, T.: A comparison of clustering heuristics for scheduling DAGs on multiprocessors. Journal of Parallel and Distributed Computing 16(4), 276–291 (1992)
4. Hagras, T., Janeček, J.: A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Computing 31(7), 653–670 (2005)
5. Kalinowski, T., Kort, I., Trystram, D.: List scheduling of general task graphs under LogP. Parallel Computing 26, 1109–1128 (2000)
6. Kasahara, H., Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Transactions on Computers 33, 1023–1029 (1984)
7. Kruatrachue, B., Lewis, T.G.: Grain size determination for parallel processing. IEEE Software 5(1), 23–32 (1988)
8. Liou, J.-C., Palis, M.A.: A new heuristic for scheduling parallel programs on multiprocessor. In: 1998 International Conference on Parallel Architectures and Compilation Techniques, October 1998, pp. 358–365 (1998)

9. Sandnes, F.E., Megson, G.M.: An evolutionary approach to static taskgraph scheduling with task duplication for minimised interprocessor traffic. In: Proc. Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2001), Taipei, Taiwan, July 2001, pp. 101–108. Tamkang University Press (2001)
10. Sarkar, V.: Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors. MIT Press, Cambridge (1989)
11. Sih, G.C., Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Transactions,on Parallel and Distributed Systems 4(2), 175–186 (1993)
12. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley, Chichester (2007)
13. Sinnen, O., Sousa, L.: List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Computing 30(1), 81–101 (2004)
14. Sinnen, O., Sousa, L.: On task scheduling accuracy: Evaluation methodology and results. The Journal of Supercomputing 27(2), 177–194 (2004)
15. Sinnen, O., Sousa, L.: Communication contention in task scheduling. IEEE Transactions,on Parallel and Distributed Systems 16(6), 503–515 (2005)
16. Tam, A., Wang, C.L.: Contention-aware communication schedule for high speed communication 6(4), 339–353 (2003)
17. Wu, M.Y., Gajski, D.D.: Hypertool: A programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1(3), 330–343 (1990)

Job Admission and Resource Allocation in Distributed Streaming Systems

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, and Kun-Lung Wu

IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA

Abstract. This paper describes a new and novel scheme for job admission and resource allocation employed by the *SODA* scheduler in *System S*. Capable of processing enormous quantities of streaming data, *System S* is a large-scale, distributed stream processing system designed to handle complex applications. The problem of scheduling in distributed, stream-based systems is quite unlike that in more traditional systems. And the requirements for *System S*, in particular, are more stringent than one might expect even in a "standard" stream-based design. For example, in *System S*, the offered load is expected to vastly exceed system capacity. So a careful job admission scheme is essential. The jobs in *System S* are essentially directed graphs, with software "processing elements" (*PE*s) as vertices and data streams as edges connecting the PEs. The jobs themselves are often heavily interconnected. Thus resource allocation of individual PEs must be done carefully in order to balance the flow. We describe the design of the *SODA* scheduler, with particular emphasis on the component, known as *macroQ*, which performs the job admission and resource allocation tasks. We demonstrate by experiments the natural trade-offs between job admission and resource allocation.

1 Introduction

We consider distributed computer systems designed to handle large-scale data stream processing jobs. This area of research is relatively new. Early examples include Borealis, TelegraphCQ, STREAM, StreamBase and Aurora [1,3,5,15,23]. These systems mostly take relational model as a basis. They process voluminous quantities of incoming stream data, performing relational operations on them.

We have been involved in an ambitious project, started in 2003 at the IBM T. J. Watson Research Center, known as *System S* [2,8,9,11,18,19]. *System S* is a large-scale, distributed computer system designed to handle complex jobs involving enormous quantities of streaming data. The paradigm is significantly more general than a relational model-oriented streaming environment. A prototype of this system has been built and continues to evolve. The scheduler for *System S* is known as *SODA*.[1] In this paper, we will describe the *SODA* scheduler, focusing particularly on one key mathematical component known as

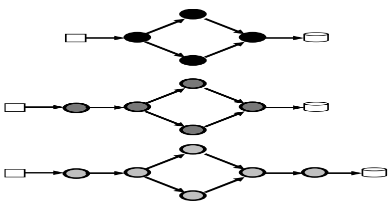
[1] *SODA* stands for Scheduling Optimizer for Distributed Applications.

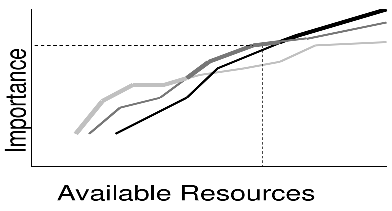
macroQ. Specifically, we will present the detailed mathematical formulation and algorithm of *macroQ*. This component deals with job admission and resource allocation, among other things, and is perhaps the most novel of the four mathematical components in *SODA* in terms of both functionality and design. Note that in a related but substantially different paper [18], we presented an overview of all the *SODA* components without mathematical details, and showed several SODA performance studies with System S applications.

The basic unit of computational work in *System S* is called a *processing element (PE)*. PEs can be arbitrary stream processing software. They are the basic execution containers in *System S*. The PEs are connected via *streams*, which flow from an output port of one PE to an input port of another. The PEs and streams are grouped into *jobs* which represent the basic unit of admittable work in the system. Hence a job is represented by a directed graph. In the current *System S* implementation, each job consists of one or more alternative data flow graphs called *templates*. These templates could be provided to *SODA* by the application developers. Each template represents a different implementation of the same job, perhaps to achieve a different level of solution quality. The logical nodes in a given template correspond to PEs, and the directed arcs correspond to streams. PEs thus both consume and produce streams.

The *macroQ* module of *SODA* in *System S* provides three critical functions, namely job admission, template selection and resource allocation. We will define a new notion, known as *importance*, which can be thought of intuitively as a function which measures the "benefit" produced by a particular job template when allocated a specific amount of processing resources. The *macroQ* module will attempt to maximize the total importance across all the competing job templates by optimizing importance as a function of allocated resources. For those jobs that get admitted it will choose a template alternative and an amount of resources to be allocted to the PEs in that template. Even though job admission and resource allocation in themselves are not new, they are significantly more challenging in the context of a distributed streaming system. In particular, *System S* has even more stringent requirements than a standard streaming system, and the offered load is expected to vastly exceed system capacity. Our main contribution is a novel approach that combines these three critical decisions in a unified framework. We elaborate on these issues in the next few paragraphs.

A key design assumption of *Systems S* is that there will be too much work. Thus *System S* hardware must be made to run at nearly full capacity nearly all of the time. This includes the processing nodes, which must be utilized as completely as possible. Moreover, some work simply will not fit. It is the role of the *macroQ* module to make intelligent decisions about job admission. To the best of our knowledge, there are no other schedulers implemented in any stream processing system that consider job admission. (In [20], the authors describe a scheduler for a hypothetical system with a simplified model of stream processing.) Some schedulers [13, 16] perform load shedding to deal with dynamically overloaded processing nodes, but this is an inherently different concept.


(a) Templates



(b) Importance

Fig. 1. Template Alternatives

A second role of the *System S* scheduler is to choose one of the alternative templates for each admitted job. Fig. 1(a) shows a job with three such templates. All the nodes in these data flow graphs are PEs, as noted above, except for the following left and right edge conventions: The left-hand node in each case is a dummy node, which may be used to "connect" this job to another. The right-hand node in each template represents disk storage, which can be regarded as a second type of dummy connector node, for persisted data. Both the left- and right-hand side streams are required by *SODA* to "match" in all templates, as are the dummy nodes, so that the identical inter-job connections may take place regardless of the template chosen.

The example in Fig. 1(a) is one possible scenario which results in multiple templates for a job. In this case, the first template would provide the basic job functionality, consisting of 4 PEs. The second template adds a preprocessing PE to achieve a higher quality of solution. The third further adds a post-processing PE, to achieve an even higher quality of solution. But there is a correlation between the benefit of doing a job and the total resources allocated to it. Each alternative template will provide the greatest overall benefit within some range of total resources allocated to the job. In *SODA* the notion of importance, defined formally in the next section, is used to quantify benefit. Fig. 1(b) illustrates importance as a function of allocated resources for the three job templates. In this case, at high allocated resource levels the third template dominates. (At this level, sufficient resources are available for both the preprocessing and post-processing PEs.) At medium allocated resource levels the second template dominates, and at low allocated resource levels the first template dominates.

A third function of the scheduler in *System S* is resource allocation of the PEs in the various accepted jobs. It is the interconnected (producer/consumer) nature of these PEs, potentially even across jobs, that makes this problem difficult: Flow imbalances can lead on one hand to buffer overflows (and loss of data), and on the other to under-utilization of processing nodes. The resources allocated to a PE which produces a stream affect the resources required for the PE(s) that consume that stream. The *macroQ* module optimizes and flow balances the *amount* of processing resources allocated to each PE in the jobs that are admitted.

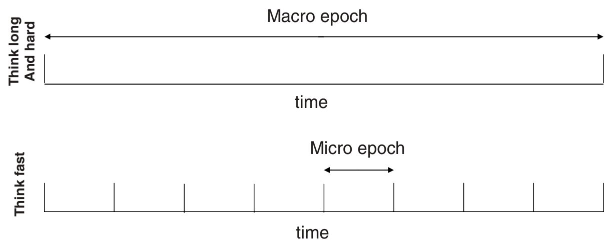


Fig. 2. Macro and micro epochs

It is the role of other *SODA* modules to take these processing goals and fractionally assign each PE to one or more acceptable processing nodes [18]. Thus the problem of determining *quantity* of PE resource allocations is effectively decoupled from the problem of determining *where* the PEs should be executed. In assigning the PEs in the chosen templates of the accepted jobs to processing nodes, there is a trade-off between the load on the processing nodes and stream traffic on the network. Assigning two PEs connected by a stream to the same processing node eliminates the contribution of that stream to network traffic, but may contribute instead to overloading the processing node. So *SODA* attempts to achieve a balanced placement that does not overload either network links or node capacities. In fact, it attempts to minimize a weighted average of six separate metrics associated with processing loads on the nodes and traffic on the network links. The assignment problem is made more complex by the addition of many special constraints imposed by *System S*. These include, among many others, hardware constraints for certain PEs and nodes (*resource matching*), security and license constraints, constraints that pairs of PEs be placed together (*colocation*), or that pairs of PEs be placed on distinct nodes (*ex-location*). Of course, many PEs may share a node. *SODA* attempts to provide each PE with a fraction of the processing power of any node to which it assigned, matching as closely as possible the overall PE flow balancing goals already computed.

Finally, in order to react quickly in a highly dynamic environment, *SODA* is an *epoch*-based scheduler. There are two kinds of epochs: macro epochs and micro epochs. Each macro epoch contains several micro epochs. Fig. 2 shows the temporal hierarchy between macro and micro epochs.[2] The *macroQ* component operates at the macro epoch level, and hence we focus mainly on macro epochs in this paper. At the beginning of each epoch, *SODA* obtains as input a snapshot of the current system state, including the jobs running on the system and the jobs waiting to be admitted. It then computes for most of an epoch, finally outputting its scheduling decisions at the end of the epoch. That is, it produces a list of accepted and rejected jobs. For the accepted jobs it produces a choice of templates and a set of fractional allocations of the PEs to processing nodes. Those decisions are enforced by *System S* during the following epoch, and the

[2] For reasonably sized *System S* installations the macro and micro epochs can also be solved sequentially.

entire process repeats indefinitely. Epoch lengths are a *SODA* settable parameter, but macro epochs on the order of a minute are typical. This is a reasonable compromise between the staleness of the input data and the time required for the mathematical components of *SODA* to make high quality decisions. Each macro epoch usually corresponds to five micro epochs, allowing *SODA* to respond quickly to changes in system.

Our contributions can be summarized as follows:

1. We provide the first stream processing scheduler in a working system that performs *job admission*. The choice to admit a job will depend on whether or not the optimal total importance occurs when that job is allocated a positive amount of resources, given certain natural constraints.
2. We provide a systematic way of optimally choosing one of several job *templates* for newly admitted jobs. The choice will depend on the relative importance of work which can be produced by these templates as well as that of the other potential work in the system.
3. We provide flow balanced resource allocations for each of the PEs in the chosen templates of accepted jobs, while simultaneously optimizing the overall allocation of resources in the system. In other words, each admitted job will get an appropriate total amount of resources based on its contribution to overall importance, and the PEs within that job will be allocated those resources in a balanced manner. Given the highly interconnected nature of the data flow graphs this is a difficult optimization problem.
4. We provide appropriate constructs to allow the scheduler to react quickly and intelligently to dynamic changes in the system, including the arrival and departure of jobs, nodes going up and down, and also changes in the relative importance of the work in the system.
5. We have designed a real-time scheduler which makes complicated decisions in each epoch, using algorithms that are deadline-aware.

The remainder of this paper is organized as follows. Section 2 contains preliminaries, including a glossary of new terms used by *SODA*, and by *macroQ* in particular. Section 3 contains an overview of the *SODA* scheduler itself, describing each of the four major mathematical components. In Section 4 we give the description, formulation and the solution approach to *macroQ*. (We are focusing here on *macroQ* because of its novelty, but also because of space limitations: A very complete description of all of the *SODA* components, associated infrastructure components and many other *SODA* capabilities is available [17].) Section 5 describes experiments showing the natural trade-offs associated with job admission and resource allocation. (We should point out that while alternative template selection is a current *macroQ* feature, the *System S* infrastructure does not yet support multiple templates. So we do not provide experiments illustrating this feature in the current paper.) Section 6 describes related work. Finally, Section 7 gives conclusions.

2 Preliminaries

In *macroQ*, we use a number of terms that have very specific meanings to the scheduler. We list these below, with explicit definitions. These concepts are critical to the discussions that follow. The first two items, the *value function* and *weight*, are the key components of the third item, *importance*. Roughly speaking, value functions measure benefit. Weights are used in their traditional sense as multiplicative "knobs", in this case accentuating or decentuating value. The product of the two is importance. Importance, in turn, is the metric that *macroQ* tries to maximize. The fourth item, the *resource function (RF)*, is essentially the means by which we iteratively compute this notion of importance. Finally, *rank*, the fifth item, is an orthogonal notion to importance. It is a priority metric assigned to each job. Jobs which produce little importance but have a better rank may get admitted *instead* of jobs which have more importance but have a worse rank. Some of these terms are not new in themselves, but the combination of them is novel.

1. *Value function:* Each derived stream produced by a potential *System S* job has a *value function* associated with it. The domain of this function might typically be the projected rate of the stream. Or it might instead be a stream quality measure, such as projected goodput. In theory it could be a cross product of a variety of quantity, quality and even other measures of benefit. The definition is intentionally general, though early *SODA* instances have employed simple rate-based value functions. Also note that value functions which are 0 everywhere will typically predominate: Although the notion is also intentionally general we expect to see non-trivial value functions mostly on terminal streams of various jobs. These are, of course, the "end products" of *System S* work, and one would thus naturally want to measure benefit there.
2. *Weight:* Each derived stream produced by a potential *System S* job also has a *weight* associated with it. Non-trivial weights will also typically be quite sparse, since we will see that the weight may as well be 0 unless the stream also has a non-zero value function.
3. *Importance:* Each derived stream produced by a potential *System S* job has an *importance* which is the product of the weight and the value function. Importance is therefore a function of the rate or quality of the stream, which in turn depends on the resources allocated to all the upstream PEs – in other words, those PEs which help to produce the stream. The summation of this importance over all derived streams is the *overall importance* being produced by *System S*, and this is what *macroQ* attempts to maximize. (Again, a large majority of streams will typically not contribute to this importance metric.) Consider Fig. 3, representing the flow graph of the same job in scenarios involving two different sets of weights. In Fig. 3(a), positive weights are at all the terminal, "starred" streams. But in Fig. 3(b) the second weight has been eliminated (changed to 0). It follows that the 2 PEs immediately upstream of that weight cannot do work which contributes to overall importance. *SODA*

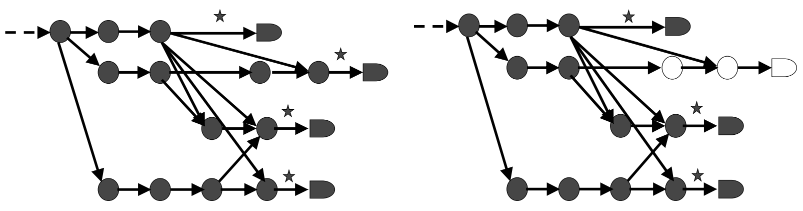
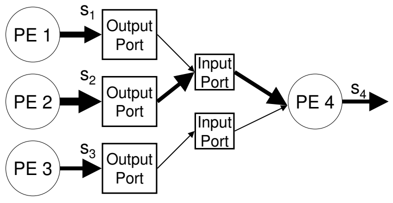

(a) All Weights (b) One Weight Removed

Fig. 3. Varying the Weights

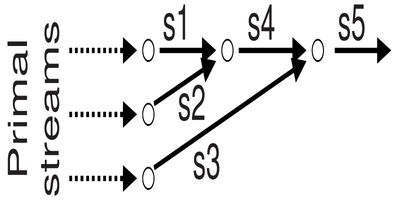
will therefore not allocate resources to them. (Other PEs, further upstream, do useful work in support of streams with positive weights. They may get fewer resources than they would in the upper half, of course.) Weights are thus a multiplier knob to turn on and off portions of a job and, more generally, a simple way to adjust relative importance.

4. *Resource function.*[3] If importance is the metric to be maximized, the natural question is how to compute it. The first part of the answer is as follows: Each derived stream s in *System S* (and by approximate terminology the PE that produces that stream) has an *RF* associated with it. The *RF* is multidimensional. If there are N input streams to the producer PE, then the *RF* has $N+1$ input parameters. There is one parameter for each of the input streams, each with the same domain as the value function. The final input dimension is the (computational) resources which may be allocated to the PE, in *millions of instructions per second (MIPS).* The output of this function for stream s is again in terms of the same domain. See, for example, Fig. 4(a). Assuming the domain to be rate-based, the *RF* for stream s_4 takes 4 parameters as input. The first three are the rates of streams s_1 through s_3, and the fourth is the MIPS allocated to PE 4. The output is the rate of stream s_4. (Some details are hinted at in the figure. Output ports filter the streams, and the output from PEs 1 and 2 are aggregated into the first input port, effectively decreasing the dimensionality of this *RF* by one.) The *RF* needs to be "learned" over time by a *SODA* infrastructure component known as the *Resource Function Learner (RFL).* The second part of computing importance involves iteratively traversing the data flow graphs from "left" to "right", ending in a final value function calculation. Consider Fig. 4(b). By topologically sorting [7] a directed acyclic graph, we can apply ready list scheduling [4,6] to compute the importance for stream s_5. In the figure three *RF*s are initially ready because they are fed by primal (external) streams. So we obtain the rates at streams s_1 through s_3. One additional *RF* becomes ready in each of the next two steps (because their inputs have been computed), and we obtain the rates at streams s_4 and s_5 in succession. Finally we apply the weighted value function at s_5 to obtain importance. (*SODA* can also handle data flow graphs with cycles, but we omit details for that case.)

[3] A paper about these *RF*s is forthcoming.



(a) Resource Function



(b) Calculation of Importance

Fig. 4. Using Resource Functions to Calculate Importance

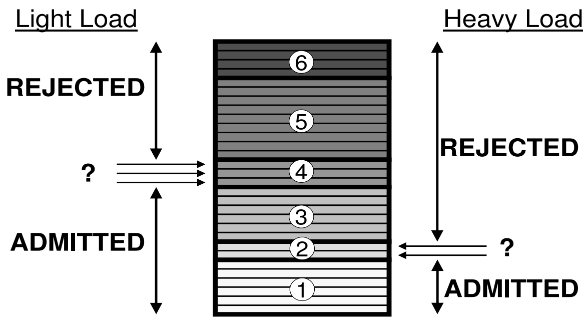


Fig. 5. Job Admission as a Function of Rank

5. *Rank:* Each job in *System S* has a *rank*, a positive integer which is used to determine whether the job should be run at all. A lower job rank is better than a higher one. (There are two seemingly irreconcilable camps on the issue of whether rank should improve with value or the reverse. Our motivation in using the convention we chose is twofold: First, it is common to say that something is "priority one", meaning it is most important. Second, one is inarguably the smallest positive integer, and thus we definitively will know that a job with rank one is most essential. On the other hand, it is certainly true that adopting this definition causes rank to be inversely related to the assigned rank number.)

   The rank of a job is set by a separate, independent component in *System S* based on a set of criteria, beyond the scope of this paper. The importance, on the other hand, determines the amount of resources to be allocated to each job that will be run. A lower job rank is better than a higher one. We will shortly provide a notion of *rank-legality* which will describe the possible subsets of jobs that can be admitted into *System S*. There is a specific job rank for which the following holds: All jobs with lower ranks are admitted, and all jobs with higher ranks are not admitted. Jobs with that rank may or may not be admitted, depending on the available resources and the importance associated with the (streams of the) jobs themselves. We call this property *rank-legality.* (This statement is a slight simplification, since one needs to account for inter-job dependencies. We will describe this notion of *revised rank* shortly.) Fig. 5 illustrates job admission based on revised rank. The *waterline* (that is, the cardinality of the highest admitted job rank) goes up in the case of lighter load conditions or with more processing power in the system. It goes down in heavier load conditions or with less processing power.

3 Overview of *SODA*

In this section we describe the four major mathematical components of *SODA*. We describe the solution approaches and motivate them from a practical standpoint, emphasizing how the solutions are dictated and/or guided by the *SODA* design philosophy.

To make the scheduling problem tractable, each *SODA* epoch is divided into four mathematical phases. Each of the four phases corresponds to a mathematical optimization module. The first two phases are known collectively as the *macro* model, while the second two are known as the *micro* model. The two temporally hierarchical levels and their goals are:

– *The macro model*, which chooses the jobs that will be admitted, the templates for those jobs, and the candidate nodes to which the PEs in those jobs and templates can be assigned. The choices made in the macro model are respected by the micro model during the micro epochs of the *next* macro epoch, making the decisions of the micro model easier and more effective.
– *The micro model*, which chooses the fractional allocations of the PEs in the jobs and templates that have been chosen by the macro model. Fractional allocations of PEs are 0 for a particular node *unless* that node has been chosen as a candidate node by the macro model. The micro model handles dynamic variability in the relative importance of work (via revised weights), and changes in the state of the system (via nodes and PEs that go up or down), without having to consider the difficult constraints handled in the macro model.

This decomposition is not perfect. Periodically there could be solutions from the macro model which are inconsistent with the constraints of the micro model. A “micro to macro” feedback loop would seem to be useful, but we have not seen examples where it is needed in practice.

Now we describe the individual decoupled quantity and where components for both the macro and micro models:

– *macroQ*, the *macro quantity* model, maximizes projected importance by deciding which jobs to admit, which templates to choose, and by computing flow balanced PE processing allocation goals (in MIPS), subject to job rank-legality, required jobs, minimum and maximum MIPS constraints. We describe *macroQ* in the next section.
– *macroW*, the *macro where* model, minimizes projected network traffic and load balances the nodes, allocating uniformly more candidate nodes than

PE goals dictate, subject to resource matching, specialized hardware, security, licensing, memory, PE exclusivity, maximum PEs per node, maximum degrees of parallelism for each PE, fixed PEs, mutual PE exclusion and colocation, legal fractional allocations and various incremental movement limits. It optimizes a weighted average of six separate metrics, three of which are averages of the utilizations of the nodes, the traffic in the network, and the bandwidth in and out of the nodes. The other three are maximum values on these same components. The overallocation allows more flexibility in handling micro epoch dynamics.
– *microQ*, the *micro quantity* model, maximizes projected importance, computing more accurate MIPS allocation goals for the PEs than those of *macroQ* by taking the candidate nodes into account. (Recall that *macroQ* does not know this information.) It also deals with revisions due to changes in node states, PE states and the like.
– *microW*, the *micro where* model, minimizes the differences between the goals output by *microQ* and achieved fractional allocations subject to various constraints on incremental movement and node changes, fixed PEs and so on. The output is thus a set of fractional assignments of the PEs to the nodes whose sum across all nodes is as close to the allocation goals as possible.

4 *MacroQ* Algorithm

The *macro quantity* model, *macroQ*, finds a set of jobs to admit during the next macro epoch. For each job it chooses a template from among the options given to it. Each template represents an alternative plan for performing the job. The jobs have ranks, and the jobs that are chosen by *macroQ* must respect a rank-legality constraint. Required jobs must be admitted. (Without loss of generality we can assume required jobs have rank 1.) Minimum and maximum PE MIPS constraints must also be respected. The goal of the *macroQ* model is to maximize the projected importance of the streams produced by the winning jobs and templates. In the process of solving the problem *macroQ* computes the optimal importance, the list of job and template choices, and finally the set of processing power goals (measured in MIPS) for each of the PEs within the chosen list. We formalize this below.

The problem formulation and the algorithm in *macroQ* are fairly elaborate. For the reader's convenience, Table 1 provides a summary of notation used, in order of appearance. And Fig. 6 provides a summary of the *macroQ* pseudo-code. Note that there are basically three nested loops.

– The outer loop, from line 3 to line 26, considers different levels of resolution granularity for the resource allocation problems that will be solved. A coarse level of granularity provides a quick solution, while a fine level provides an accurate solution. Because *SODA* is a real-time scheduler, *macroQ* must have a solution by the time the macro epoch completes. A quick, coarse solution will serve this purpose.

Table 1. Key *macroQ* Notation

Variable	Definition
J	Number of jobs offered
$\pi(j)$	Original rank of job j
N_j	Number of templates for job j
$\mathcal{J}$	Job list
T	Template list
$\mathcal{T}_{\mathcal{J}}$	Set of all template lists for job list $\mathcal{J}$
$\mathcal{L}$	Job/template list
$D_{(\mathcal{J},T)}$	Directed acyclic graph associated with job/template pair $(\mathcal{J},T)$
$\mathcal{P}_{(\mathcal{J},T)}$	Nodes (PEs) in $D_{(\mathcal{J},T)}$
$d_{(\mathcal{J},T)}$	Asymmetric distance function
$\mathcal{D}_{(\mathcal{J},T)}(p)$	Set of PEs which depend on PE p
$\pi_{(\mathcal{J},T)}(j)$	Revised rank for job j
$\hat{\mathcal{L}}$	Rank-legal job/template list
g_p	*mips* allotted to PE p
V_s	Composite value function for stream s
w_s	Weight for stream s
I_s	Importance function for stream s
G	Total MIPS in system
m_p	Minimum MIPS for PE p if admitted
M_p	Maximum MIPS for PE p if admitted
$\hat{\mathcal{J}}$	Jobs that must be admitted
$\bar{G}$	Number of resource units in discrete RAP
$\bar{m}_p$	Minimum resource units for PE p if admitted
$\bar{M}_p$	Maximum resource units for PE p if admitted
$C(\mathcal{J},T)$	Number of weak components for job/template list $(\mathcal{J},T)$
$\mathcal{I}_c$	Importance function for weak component c
$\bar{m}_c$	Minimum resource units for weak component c
$\bar{M}_c$	Maximum resource units for weak component c
L_r	Number of job/template alternatives examined of revised rank r

– The middle loop, from line 5 to line 24, decrements the possible revised rank waterlines, considering less and less jobs as it goes.
– The inner loop, from line 7 to line 23, is a divide and conquer approach based on the number of so-called *weak components* of the relevant data flow graphs. The overall resource allocation problem to be solved can be handled by solving an elaborate problem on each weak component, and then combining the solutions via a simpler problem across *all* components. We will describe these in more detail later in this section. problems are solved as we go along.

Ultimately we output the best solution discovered, on line 27.

4.1 Notation

Let J denote the number of jobs being considered, indexed by j. Each job j has a *rank* $\pi(j) \in \mathbb{N}$. (Here, $\mathbb{N}$ represents the natural numbers.) We adopt the

```
1: Set $OPT = \infty$
2: Set $OK$ = false
3: while $OK$=false do
4:    Pick resolution granularity $\bar{G}$
5:    for $r = R$ to 1 by -1 do
6:       Create list $(\mathcal{J}, T)_1, ..., (\mathcal{J}, T)_{L_r}$ of rank-legal job/templates with waterline $r$
7:       for $l = 1$ to $l = L_r$ do
8:          Compute $C_{(\mathcal{J},T)}$ weak components
9:          for $c = 0$ to $c = C_{(\mathcal{J},T)} - 1$ do
10:            NSDP scheme to solve component $c$ RAP with granularity $\bar{G}$
11:         end for
12:         Compute number of components with concave importance functions
13:         if all are concave then
14:            Galil-Megiddo scheme to solve inter-component RAP with granularity $\bar{G}$
15:         else if none are concave then
16:            DP scheme to solve inter-component RAP with granularity $\bar{G}$
17:         else
18:            Fox/DP scheme to solve inter-component RAP with granularity $\bar{G}$
19:         end if
20:         if $\mathcal{I} > \mathcal{OPT}$ then
21:            OPT=$\mathcal{I}$
22:         end if
23:      end for
24:   end for
25:   Evaluate $OK$
26: end while
27: Output $OPT$
```

Fig. 6. *macroQ* Pseudocode

(slightly unnatural) convention that lower numbers indicate better ranks. Thus the best possible rank is 1. Each job j comes with a small number of possible job *templates*. This number may be 1. It *will* be 1 if the job has already been instantiated, because we assume that the choice of a template is fixed throughout the "lifetime" of a job. It is, however, the role of the *macroQ* model to make this choice for jobs that are newly admitted. Let N_j denote the number of templates for job j, indexed by t.

Any subset $\mathcal{J} \in 2^J$ will be called a *job list*. For each job list $\mathcal{J}$ a function $T : \mathcal{J} \rightarrow \mathbb{N}$ satisfying $T(j) \leq N_j$ for all j will be called a *template list*. Denote the set of all template lists for $\mathcal{J}$ by $\mathcal{T}_{\mathcal{J}}$. Finally, define the *job/template list* to be the set $\mathcal{L} = \{(\mathcal{J}, T) | \mathcal{J} \in 2^J, T \in \mathcal{T}_{\mathcal{J}}\}$. A major function of *macroQ* is to make a "legal and optimal" choice of a job/template list.

We will make the assumption, for ease of exposition, that no cycles exist in the directed flow graphs for a job and template choice. *SODA* can actually handle intra- and inter-job cycles, but the details are somewhat complex.

Each job/template list $(\mathcal{J}, T)$ gives rise to a directed acyclic graph $D_{(\mathcal{J},T)}$ whose nodes $\mathcal{P}_{(\mathcal{J},T)}$ are the PEs in the template and whose directed arcs are the streams. (This digraph is "glued" together from the templates of the various jobs in the list, and we omit the exact details. These PE nodes may come from multiple jobs.) Assigning length one to each of the directed arcs, there is an obvious notion of an asymmetric distance function $d_{(\mathcal{J},T)}$ between pairs of relevant PEs. Note that $d_{(\mathcal{J},T)}(p,q) < \infty$ means that PE p precedes PE q, or, equivalently, that q depends on p. Let $\mathcal{D}_{(\mathcal{J},T)}(p)$ denote the set of PEs $q \in D_{(\mathcal{J},T)}$ for which q depends on p. This notion of dependence gives rise, in turn, to the notion of dependence between the relevant jobs: Given jobs $j, j' \in \mathcal{J}$, we will say that j' depends on j provided there exist PEs q and p, belonging to j' and j, respectively, for which $d_{(\mathcal{J},T)}(p,q) < \infty$. Let $\mathcal{D}_{(\mathcal{J},T)}(j)$ denote the set of jobs $j' \in \mathcal{J}$ for which j' depends on j.

We now define a revised job rank notion based on a particular job/template list $(\mathcal{J},T)$ by setting

$$\pi_{(\mathcal{J},T)}(j) = \begin{cases} \min_{j' \in \mathcal{D}_{(\mathcal{J},T)}(j)} \pi(j') & \text{if } j \in \mathcal{J} \\ \pi(j) & \text{otherwise.} \end{cases}$$

This is well-defined. We can define the notion of a *rank-legal* job/template list $(\mathcal{J},T)$ as follows: We insist that $j \in \mathcal{J}$ and $j' \notin \mathcal{J}$ implies that $\pi_{(\mathcal{J},T)}(j) \leq \pi_{(\mathcal{J},T)}(j')$. (This is equivalent to the statement that there is a value for which all jobs with lower revised ranks will be admitted and all jobs with higher revised ranks will not be admitted.) Let $\hat{\mathcal{L}}$ denote the set of rank-legal job/template lists.

Define the decision variable g_p to be the resource allocation, in MIPS, given to PE p. As noted, any derived stream s associated with job/template list $(\mathcal{J},T)$ has a *value function*. The stream, in turn, is created by a unique PE p associated with $(\mathcal{J},T)$. The PE p gives rise to a set $\{q_1,...,q_{k_p}\}$ of k_p PEs q_i for which $p \in \mathcal{D}_{(\mathcal{J},T)}(q_i)$. This set includes p itself. We have also introduced the notions of learned *RFs* which can be iteratively composed to create a function from the processing power tuple $(g_{q_1},...,g_{q_{k_p}})$ to the domain of the value function. And so the composition of these recursively unfolded functions with the value function yields a mapping V_s from the tuple $(g_{q_1},...,g_{q_{k_p}})$ to the non-negative real numbers for stream s. This function is called the *composite value function* for s. Multiplied by a weight w_s for stream s it becomes a *stream importance* function I_s mapping $(g_{q_1},...,g_{q_{k_p}})$ to the non-negative real numbers $[0,\infty)$. Finally, aggregating all the stream importance functions together for all streams which are created by a given PE p yields a *PE importance function* $\mathcal{I}_p$.

Let G denote the total amount of *System S* processing power, in MIPS. Let m_p denote the minimum amount of processing power which can be given to PE p if it is admitted, and M_p denote the maximum amount of processing power which can be given to PE p if it is admitted. Suppose that the set $\hat{\mathcal{J}}$ represents the jobs that must be admitted.

4.2 Mathematical Formulation

We seek to maximize the overall importance, which is the sum of the PE importance functions across all possible rank-legal job/template lists. The objective is therefore to find

$$\max_{(\mathcal{J},T)\in\hat{\mathcal{L}}} \sum_{p\in\mathcal{P}_{(\mathcal{J},T)}} \mathcal{I}_p(g_{q_1},...,g_{q_{k_p}})$$

subject to the following constraints:

$$\sum_{p\in\mathcal{P}_{(\mathcal{J},T)}} g_p \leq G, \tag{1}$$

$$m_p \leq g_p \leq M_p \qquad \forall p \in \mathcal{P}_{(\mathcal{J},T)}, \tag{2}$$

$$\hat{\mathcal{J}} \subseteq \mathcal{J} \tag{3}$$

Constraint 1 is the resource allocation constraint. It ensures that all of the resource is used if it is useful and possible to do so. Constraint 2 requires a PE p to be within some minimum and maximum range if it is admitted. Constraint 3 insists that required jobs are admitted.

4.3 Solution Approach

We discretize the above continuous resource allocation problem by dividing the total amount of resource G into $\bar{G}$ equal size atomic units of "resolution" $G/\bar{G}$ MIPS each. Assume that this value $\bar{G}$ is given. For each PE p let $\bar{m}_p = \lfloor m_p\bar{G}/G \rfloor$ and $\bar{M}_p = \lceil M_p\bar{G}/G \rceil$ represent the discrete analogues of the minimum and maximum MIPS constraint terms. Also assume a fixed rank-legal job/template list $(\mathcal{J},T) \in \hat{\mathcal{L}}$ containing all the required jobs $\mathcal{J}$. Partition the PEs and streams into $C_(\mathcal{J},T)$ weak components and fix one such component c.

We consider, using the natural change in notation, the corresponding discrete resource allocation problem of maximizing $\sum_{p\in c}\mathcal{I}_p(\bar{g}_{q_1},...,\bar{g}_{q_{k_p}})$ subject to the constraints $\sum_{p\in c}\bar{g}_p \leq \bar{G}$ and $\bar{m}_p \leq \bar{g}_p \leq \bar{M}_p$ for all $p \in c$. This problem can be solved by a scheme known as *Non-Serial Dynamic Programming (NSDP)* [10]. NSDP is a complex dynamic programming scheme designed specifically to handle difficult (non-separable) resource allocation problems. (See line 10 of Fig. 6.) As part of the solution methodology we obtain the optimal values $\mathcal{I}_c(\bar{g}_c)$ for every $\bar{g}_c$ between 1 and $\bar{G}$, *as well as* the PE MIPS allocations that constitute this optimal solution. We can thus regard $\mathcal{I}_c$ as a *component importance function* of the resources $\bar{g}_c$ allotted to component c. Set $\bar{m}_c = \sum_{p\in c}\bar{m}_p$ and $\bar{M}_c = \sum_{p\in c}\bar{M}_p$.

Note that the objective function can be regarded as a "black box", calculated by iterative *RF* compositions followed by a weighted value function calculation. To make this as efficient as possible the *macroQ* code has itself been carefully optimized. Careful analyses are performed to determine which sub-graph calculations are strictly necessary and which are redundant. A cache of previous results is also employed. Also, *macroQ* code is aware of time and is given a deadline by *SODA*. So it occasionally takes "shortcuts", using a partially greedy scheme instead of a full NSDP algorithm. This fits the design philosophy: *SODA* is a *real-time* scheduler.

Having performed this NSDP on *each* component we now consider the problem of optimizing over *all* components. The good news here is that the problem is a *separable* resource allocation problem: We wish to maximize $\sum_c\mathcal{I}_c(\bar{g}_c)$ such that $\sum_c \bar{g}_c \leq \bar{G}$ and $\bar{m}_c \leq \bar{g}_c \leq \bar{M}_c$ for all c. Separability here means that each summand is a function of a single decision variable, and such resource allocation problems are inherently easier to solve.

In fact, if the component importance functions happen to be *concave* the problem can be solved by one of three algorithms: These are the schemes by Fox, Galil and Megiddo, and Frederickson and Johnson, which can be regarded as fast, faster and (theoretically) fastest, respectively. If the component importance functions, on the other hand, are not concave, the problem may still be solved by dynamic programming. See [10] for details on all of these algorithms. Also see lines 14, 16 and 18 of Fig. 6.

It turns out that concavity is not an uncommon condition for our component importance functions. So we test each component for concavity and adopt one of three approaches, depending on the results.

– If all component importance functions are concave we solve the resource allocation problem by the Galil and Megiddo algorithm. This algorithm is quite fast and easier to code than the Fredrickson and Johnson scheme.
– If all the component importance functions are not concave we solve the resource allocation problem by dynamic programming.
– In other cases we solve the concave portion of the problem by the Fox algorithm (because it provides the needed intermediate values) and then solve the remainder of the problem by dynamic programming.

At the end of this step we have computed the optimal MIPS allocations for each PE. But this can be regarded as just the inner loop of a three step nested process. In the central loop we evaluate all rank-legal templates. In the outer loop we evaluate successively finer resolution granularities. Again, see Fig. 6.

The evaluation of all rank-legal templates is obviously exponential [7] in nature, though the problem is generally not large: *SODA* only evaluates alternative for *new* jobs. Once a template decision has been reached it lasts for the remaining epochs of the job. And most jobs, in fact, only have a single template. The rank-legality constraints adds another exponential term, but this process can also be streamlined if time is an issue. The code loops through each rank value, working from the highest rank (R) to lowest rank (1): For any given rank value it assumes all higher revised rank jobs will not be admitted, all lower revised rank jobs will be admitted, and has to decide which jobs of that revised rank will be admitted. For all but the highest revised rank these jobs were admitted in the previous calculation. The code computes their importance divided by their resource allocations and orders the jobs accordingly. If a full exponential evaluation will not complete in time the code admits jobs of that revised rank based on this ordering. The case where there are a large number of jobs of the highest revised rank is obviously less satisfactory. And this case includes the case where all jobs have the same revised rank. The code performs a greedy scheme if pressed for time, but the results may be less than optimal. The philosophy is that an imperfect *macroQ* solution is better than no solution at all. In the pseudo-code we let L_r the be the number of job/template alternatives examined of revised rank r, whether linear or exponential.

Table 2. *mips* x 100

%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	786	882	462	540	558	318	336	786	870	462	702	264	366	786	870	516	558	294	336
95	780	840	462	540	558	318	336	780	780	462	696	264	366	780	780	516	558	294	336
90	786	792	462	540	558	318	336	780	780	462	612	264	366	696	780	516	558	294	
85	696	792	372	540	468	318	318	696	780	372	612	264	366	696	780	516	468	294	
80	690	786	366	534	462	312	312	690	780	366	606	258	366	696	774	510		288	
75	684	780	366	534	354	270	312	626	768	366	606	258	360	684	774	510			
70	690	786	366	534	444	312	312	690	774	366	606	258	360	690		510			
65	696	792	372	540	468	318	336	696	780	390	612	264	366			516			
60	696	792	372	540	468	318	324	696	780	372	612	264	366						
55	696	792	462	540	552	318	336	696	780		612	264							
50	690	786	366	534	426	312	312	692	774		606								
45	690	792	366	534	462	318	312	696	780										
40	696	792	462	540	558	318	336	696											
35	626	710	354	494	426	300	314	626											
30	510	558	344	426	354	272	292	544											
25	510	558	344	426	354	272	286												

Table 3. Importance

%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	402	389	136	262	222	279	13	402	389	136	261	250	291	402	390	304	222	279	13
95	402	388	136	262	222	279	13	402	388	136	261	250	291	402	388	304	222	279	13
90	402	388	136	262	222	279	13	402	388	136	259	250	291	400	388	304	222	279	
85	400	388	129	262	216	279	10	400	388	129	259	250	291	400	388	304	216	279	
80	395	383	126	260	212	275	9	395	388	126	257	245	291	400	383	301		275	
75	389	377	126	260	215	274	9	389	377	126	257	245	286	389	383	301			
70	395	383	126	260	205	275	9	395	383	126	257	245	286	395		301			
65	400	388	129	262	216	279	13	400	388	129	259	250	291			304			
60	400	388	129	262	216	279	11	400	388	129	259	250	291						
55	400	388	136	262	221	279	13	400	388		259	250							
50	395	383	126	260	178	275	9	395	383		257								
45	395	388	126	260	212	279	9	400	388										
40	400	388	136	262	222	279	13	400											
35	359	355	125	259	178	273	10	389											
30	301	333	116	253	170	262	6	380											
25	301	333	116	253	170	262	5												

The resolution granularity loop is simple in nature: *macroQ* starts with a coarse resolution to obtain a quick solution. Then it uses the time already spent to estimate the finest resolution it believes it can safely solve in the remaining time, subject to a reasonable minimum MIPS value. It reports the best importance found, and this is typically based on the finer resolution.

5 Experiments

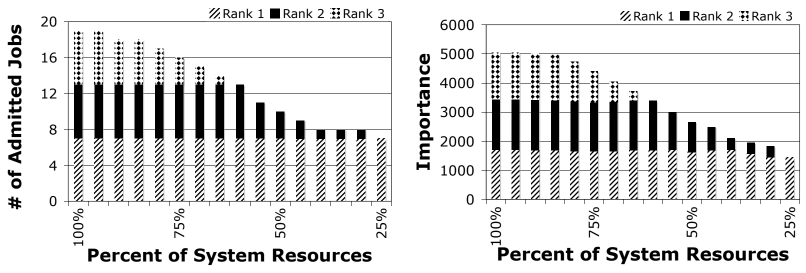
In this section we experimentally evaluate *SODA* performance, focusing on the functions of job admission and resource allocation. Note that the trade-offs between the two can be quite subtle. In order to better reveal these subtle trade-offs, a carefully chosen set of well-controlled experiments were conducted. In these experiments, a variety of job submission scenarios were simulated. For the complete

SODA performance with real applications running on *System S*, we refer readers to [17,18].

Now we will describe the experimental setup. The largest system installation we consider has 100 processing nodes with a rating of 11,000 MIPS each. In the experiments we examine the effect of removing 5 processing nodes (and thus 5% of the processing power) from the system at a time. The jobs presented to *macroQ* always remain the same: There are 19 jobs (labeled A through S), consisting of 7 required jobs of rank 1, 6 optional jobs of rank 2, and 6 optional jobs of rank 3. The jobs are not interconnected, so rank and revised rank are identical for each job. The experiments are designed so that at 100 processing nodes the jobs will nearly (but not quite) use all the resources in the system when each is allocated their maximum useful resources. This occurs, as per the previous section, when each of the component importance functions becomes flat as a function of allocated resources. In fact, when *macroQ* is run on the full 100 processing nodes all 19 jobs are admitted and the average utilization of the processors is 97%.

Fig. 7(a) shows the number of jobs admitted by rank as the number of processing nodes decreases in 5% increments from 100 nodes to 25 nodes. At 95% all jobs are still admitted, though the processor utilization now is 100%. From there on the utilization remains at 100%, as one would expect based on *macroQ*'s design: The system is overloaded. At 90% 1 job of rank 3 is rejected, and all of the rank 3 jobs are gone by the 60 processing node level. But rank 1 and 2 jobs remain during the 65% to 100% range. Thus the rank waterline is 3. In the 30% to 60% range the rank waterline is 2. All rank 1 jobs are admitted, but more and more rank 2 jobs are rejected as the processing power decreases. At 25 processing nodes only the required rank 1 jobs are admitted. The system is fully stressed at this point, and a *macroQ* run at 20% of the processing nodes would not find a feasible solution: There would not be sufficient processing power to admit all of the required jobs even at their minimum acceptable resource allocations.

Fig. 7(b) shows the contribution to overall importance by rank as the number of processing nodes decreases from 100 to 25. Importance is a decreasing function of system resources, as should be the case. But between 100 and 85 nodes the importance curve is actually quite flat: The component importance curves turn



(a) Admitted Jobs by Rank (b) Importance by Rank

Fig. 7. Effect of Rank on Admission

out to be concave or close to concave, and there are sufficient resources available so that the solution lies near the flat part of each curve. Job S is rejected at 85% and 90%. But its importance is low and its resource requirements is high. One can see this in Tables 2 and 3. The first table shows the allocated resources (in hundreds of MIPS) by individual jobs as the number of processing nodes decreases. The second table shows the corresponding importance values. Job S contributes an importance of only 13 at 33600 MIPS, so it is clearly highly expendable, and being of rank 3 it is jettisoned as soon as the offered load exceeds available resources. The effect on overall importance is minimal. The only other job with a poor ratio of importance to resources is job G. Job G is a twin of Job S, but it is required and *macroQ* cannot reject it. Observe in Fig. 7(b) that importance does start to decrease linearly from 80 down through 25 processing nodes. The available MIPS dictate that the solution to the component resource allocation problems occur on the steeper portion of each importance curve.

Examine Table 2 in the 3 ranges of resource allocations for which the admitted jobs remains identical. (The 100% and 95% rows have this property. So do the 90% and 85% rows. And finally, so do the 40%, 35% and 30% rows.) If the component importance curves are concave for each admitted job in these ranges, the separable resource allocation problems in *macroQ* would solve where the first differences (effectively the derivatives) at each job would be as close to equal as possible. And that would, in turn, imply that the resource allocations for each job would be monotone non-increasing as the number of processors decrease. In fact, this is the case in each of the three ranges, as an examination of the relevant columns shows.

Overall, however, the allocated resources for each job will not be monotone as the number of processors decrease. Consider, for example, the column for Job C in Table 2. The MIPS allocated are high in the 85% to 100% range, because the system is not heavily overloaded. As the number of processors decrease the MIPS allocated to Job C exhibits somewhat oscillatory behavior, decreasing through 70%, then increasing back to its maximum useful allocation at 55%, and so on. This behavior is primarily due to the changes in admission of the *other* jobs. As jobs get rejected the allocations they would have received become available, and *macroQ* will distribute these to the jobs that remain. (A secondary reason for the lack of monotonicity is the slight deviations from concavity.) At the 25 and 30 processor levels the system is truly stressed and there the job is given minimum acceptable MIPS allocations.

We have focused on the *macroQ* problems of job admission and resource allocation in these experiments. Extensive experimental analyses of the overall performance of *SODA* can be found in [18,17].

6 Related Work

Stream processing systems have been an active area of research in recent years. Example systems include Borealis [1], TelegraphCQ [5], STREAM [3], Aurora, StreamBase [15] and Medusa [23] and so on. These systems are mostly based on relational model and process voluminous quantities of incoming stream data. In contrast, *System S* is much more general in terms of programming model. It does not assume a relational model and it allows arbitrarily complex operators.

Most of these stream processing systems are designed to be run on more than one node, and thus there has also been work on scheduling and load-balancing the stream operations. While these scheduling approaches have some of the flavor of the work we have presented here, none targets our problem exactly. We describe some of these related scheduling approaches here.

The FIT algorithm [16] is a load-shedding algorithm which intelligently drops load. Determining where best to drop load can be quite a complex problem, since dropping at a particular operator has an effect on the downstream operators, sometimes an unintended one. In some cases, shedding load on a particular operator increases the resources for other operators on that node, and so could *increase* load at nodes downstream. FIT cleverly addresses this problem in a distributed way, but without a global notion of importance. The *SODA* scheduler provides this same functionality as part of its resource allocation and scheduling, and does so in a way that takes into account the processing graph for a job and the total system objectives.

In [21, 22], the authors address the problem of variance in stream rates. Both papers describe a way to distribute the load so that changes in input rate have a smaller chance of overloading the system. However, they do not address the case when the system is overloaded, and make no decisions about job admission. In [14], the authors provide a scheduling algorithm for a wide-area network that places operators so as to minimize network latency. In the local area network that we address, bandwidth, not network latency, is the main concern. In addition, their work does not address the problem of job admission. Others [12] also address scheduling to minimize latency.

The STREAM project [13] has goals somewhat similar to those presented in this paper. Their system handles queries in an SQL-like language. When resources are tight, they revise queries by dropping packets and/or changing internal parameters. Finally, in [20], the authors address admission control problem in a hypothetical stream processing systems. Their model assumes a linear processing graph. In other words, input stream is processed, successively, by a series of operators. Thus, no operator takes input from more than one source stream.

7 Conclusions

In this paper we have described the *SODA* scheduler for large-scale distributed stream processing applications. We have focused on one component, *macroQ*, in particular. This component is responsible for the two key functions of job admission and resource allocation. We have provided an introduction to *System S*, an introduction to the scheduler in general, and to the *macroQ* component of *SODA* in particular. We have evaluated the subtle trade-offs between job admission and resource allocation via simulation experiments, showing the capabilities of the scheduler.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings of Conference on Innovative Data Systems Research (2005)
2. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., Venkatramani, C.: SPCA distributed, scalable platform for data mining. In: Proceedings of the Workshop on Data Mining Standards, Services and Platforms (2006)
3. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin 26 (2003)
4. Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J.: Scheduling in Computer and Manufacturing Systems. Springer, Heidelberg (1993)
5. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proceedings of Conference on Innovative Data Systems Research (2003)
6. Coffman, E.: Computer and Job-Shop Scheduling Theory. John Wiley and Sons, Chichester (1976)
7. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. McGraw Hill, New York (1985)
8. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.: SPADE: The System S declarative stream processing engine. In: Proceedings of the ACM International Conference on Management of Data (2008)
9. Hildrum, K., Douglis, F., Wolf, J., Yu, P.S., Fleischer, L., Katta, A.: Storage optimization for large-scale stream processing systems. ACM Transactions on Storage 3(4) (2008)
10. Ibaraki, T., Katoh, N.: Resource Allocation Problems. MIT Press, Cambridge (1988)
11. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation and evaluation of the linear road benchmark on the stream processing core. In: Proceedings of the ACM International Conference on Management of Data (2006)
12. Lakshmanan, G., Strom, R.: Biologically-inspired distributed middleware management for stream processing systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 223–242. Springer, Heidelberg (2008)
13. Motwani, R., Widom, J., Arasu, A., Babcokc, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: CIDR (2003)
14. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: IEEE ICDE, Washington, DC, USA. IEEE Computer Society, Los Alamitos (2006)
15. StreamBaseSystems, http://www.streambase.com
16. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: Efficient load shedding techniques for distributed stream processing. In: Proceedings of the International Conference on Very Large Data Bases Conference, pp. 159–170 (2007)
17. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer., L.: A scheduling optimizer for distributed applications: A reference paper. Technical Report 24453, IBM Research Report (2007)

18. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer, L.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Proceedings of Middleware Conference (2008)
19. Wu, K.-L., Yu, P.S., Gedik, B., Hildrum, K.W., Aggarwal, C.C., Bouillet, E., Fan, W., George, D.A., Gu, X., Luo, G., Wang, H.: Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In: Proceedings of the International Conference on Very Large Data Bases Conference (2007)
20. Xia, C.H., Towsley, D., Zhang, C.: Distributed resource management and admission control of stream processing systems with max utility. In: ICDCS (2007)
21. Xing, Y., Hwang, J.-H., Çetintemel, U., Zdonik, S.: Providing resiliency to load variations in distributed stream processing. In: Proceedings of the International Conference on Very Large Data Bases Conference, pp. 775–786. VLDB Endowment (2006)
22. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic load distribution in the Borealis stream processor. In: IEEE ICDE, Washington, DC, USA, pp. 791–802. IEEE Computer Society, Los Alamitos (2005)
23. Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa projects. IEEE Data Engineering Bulletin 26(1) (2003)

Scalability Analysis of Job Scheduling Using Virtual Nodes

Norman Bobroff[1], Richard Coppinger[2], Liana Fong[1], Seetharami Seelam[1], and Jing Xu[3]

[1] IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
{bobroff,llfong,sseelam}@us.ibm.com

[2] IBM Systems and Technology Group, Poughkeepsie, NY 12601, USA
coppinge@us.ibm.com

[3] University of Florida, Gainesville, FL 32611, USA
jxu@acis.ufl.edu

Abstract. It is important to identify scalability constraints in existing job scheduling software as they are applied to next generation parallel systems. In this paper, we analyze the scalability of job scheduling and job dispatching functions in the IBM LoadLeveler job scheduler. To enable this scalability study, we propose and implement a new virtualization method to deploy different size LoadLeveler clusters with minimal number of physical machines. Our scalability studies with the virtualization show that the LoadLeveler resource manager can comfortably handle over 12,000 compute nodes, the largest scale we have tested so far. However, our study shows that the static resource matching in the scheduling cycle and job object processing during the hierarchical job launching are two impediments for the scalability of LoadLeveler.

1 Introduction

Job scheduling software is a key piece of system software to maximize the utilization of parallel computing systems. As these systems increase in size with one generation of systems having more processors and compute power than the previous generation, the performance of job scheduling becomes crucial to optimize the overall system utilization. To support the current and next generation of massively parallel systems (MPP), the job scheduler must scale in several dimensions. It must be able to manage a large number of jobs and compute resources, quickly match resources to a job, and rapidly dispatch the job on those resources. During the last year, we have analyzed the scalability of IBM Tivoli workload scheduler LoadLeveler in the context of the IBM DARPA HPCS program [2]. This paper presents the method and results of the study, as well as insights about scheduling and dispatching in the context of LoadLeveler.

An essential requirement for a scalability study is access to a representative large scale system. Although we indeed have access to fifty 8-processor pSeries machines, the goal is to study scalability beyond a thousand nodes. This resource limitation is overcome by developing a lightweight virtualization mechanism that creates hundreds of LoadLeveler nodes on each physical machine. This technique allows testing on up to a 12,000 node LoadLeveler cluster using fifty physical nodes. Virtualization is applied to study the scalability of LoadLeveler in its capability in resource monitoring, identifying and scheduling jobs to resources, and in dispatching jobs to the allocated resources. The following contributions are made:

– Introduce lightweight virtualization technology that isolates and executes multiple instances of LoadLeveler node daemons on a physical machine, creating a large cluster with minimal physical machine and memory requirements (Section 3).
– Analyze the scalability of job scheduling algorithms for sequential and parallel jobs and isolate the performance of different phases in the scheduling algorithm. Static resource matching is determined to be a scalability problem and approaches to address this problem are described (Section 4.2).
– Investigate the scalability of hierarchical job launching and identify scalability hot-spots with processing job object at various levels of the hierarchical tree (Section 4.3).

2 LoadLeveler Overview

LoadLeveler [3] is a distributed job scheduling product of IBM. It is based on the licensed code from the Condor system [13] in mid 1990. The architectural framework of LoadLeveler, as shown in Figure 1, retains the core structure of Condor. The *Central Manager* (CM) consists of two functional units: the *Collector* and the *Negotiator*. The Collector receives resource information sent by a daemon called *StartD* running on the machine.[1] LoadLeveler ensures there is only one StartD on each machine with responsibility for reporting the machine state, resources and attributes, utilization, and managing presence heartbeats. The Negotiator applies this resource information to allocate machines matching the execution requirements of user jobs.

Jobs are submitted to LoadLeveler through the *Scheduling daemon* (SchedD). SchedD is responsible for maintaining a local job queue and persisting job state, as well as coordinating the activities of assigning execution nodes to the job, and launching the job. Multiple SchedD machines can be defined for a cluster to eliminate bottlenecks with large job submissions requirements. SchedD informs the Negotiator about each job arrival and the Negotiator applies scheduling algorithms to allocate computational and other resources to jobs. The resource assignments are returned to the SchedD which forwards the job launch information to the StartDs on the allocated executing machines. Execution at the node is managed by the local StartD which forks a *Starter* process to initiate and control the job. Concurrent execution of multiple jobs or tasks at the node is enabled by forking multiple Starters.

[1] The terms ”machine” and ”compute node” are used interchangeably.

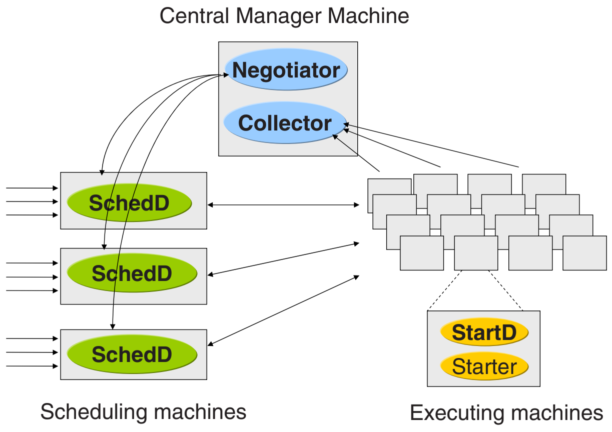


Fig. 1. LoadLeveler Architecture

LoadLeveler continually evolves to support new hardware architectures, and leverage novel software features. Examples include high bandwidth interconnection switches (e.g. SP2 switch [12], InfiniBand [10]) hardware multi-threading, and Blue Gene [6,5]. Supported software features include AIX's WLM [1], and various Linux distributions [4]. Recent development has emphasized support for highly parallel applications running on large scaled clusters with high speed interconnection instead of flocks of workstations [7].

The following sub-sections describe in more detail scheduling in Negotiator, and hierarchical communication scheme for scalable job dispatching. This material provides the necessary background for our scalability studies covered later in this paper:

2.1 Scheduling in Negotiator

The Negotiator of CM processes incoming jobs in two sequential phases: scheduling requested resources to jobs and coordinating with SchedD to dispatch jobs to assigned machines for execution.

During the scheduling phase, Negotiator logically performs the following steps:

1. Select machines that have the capabilities to match the requirements of the job; Exemplary capabilities include machine architecture type (x86, POWER), job class definitions, and high-speed switch connectivity.
2. Select capable machines that have the dynamic capacities to assign to the job; Exemplary capacities include unused job class, unfilled multiprogramming level, and spare switch adapters.
3. Assign the machine(s) to the job based on specific scheduling algorithms and administrative policies; for example, backfill [9] and fairshare are two of scheduling algorithms in LoadLeveler.

At the end of scheduling phase, Negotiator sends successful machines-to-job assignments to SchedD, which dispatches the job to all assigned machines through the StartDs on the machines.

2.2 Hierarchical Scheme for Job Dispatching

In a large cluster, information and command propagation from a single machine to a large number of other machines is parallelized (e.g. using multi-threading) for faster communication. However, the number of open communication connections on a single machine is a potential bottleneck because of limited resources such as communication buffers and OS data structures. The hierarchical scheme provides the benefit of parallel communication operations by dividing the connections through a spanning tree.

The hierarchical communication scheme implemented by LoadLeveler [8] is shown in Figure 2 with an exemplary fan-out of three. When a job is scheduled by the Negotiator a job object of assigned machines and job specific information is sent to SchedD. SchedD constructs a hierarchical spanning tree of machines using the configurable fan-out parameter. and sends the tree structure and job information to the root node or master StartD. The master StartD repackages the job object into job objects customized exclusively for the subtree headed by each immediate child StartD and forwards the information. This process is repeated down the tree to the leaf nodes. Communication of the job object to all compute nodes is flagged as successful when an acknowledgement from each node in the tree is received at the master StartD. Then, a subsequent communication is sent using the tree to command each node to start the job. The start command prompts each StartD to launch a Starter process which locally manages job execution.

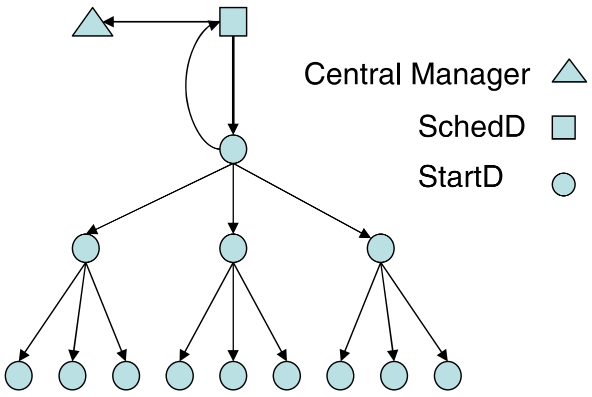


Fig. 2. Hierarchical Communication Structure

3 Virtual Node Method for Scalability Analysis

Analyzing the performance and scalability of LoadLeveler requires a cluster of at least a few thousand compute nodes. Building and maintaining such a large scale cluster for this analysis alone is not cost effective. A StartD instance is the component in LoadLeveler that represents a compute node. From the perspective of the CM the size of the cluster is the number of StartD processes reporting to the CM. Thus, the key to creating a compute cluster whose apparent size is greater than the number of physical machines is to allow multiple StartD to execute independently and in mutual isolation on each physical node. However, the design point of LoadLeveler is a single StartD process for each physical machine node. Communication ports are the identical for all StartD as specified in a central configuration file and can not be shared among StartD instances on the same machine. Furthermore, each StartD must appear to both the CM and other StartDs that it is located at a dedicated and unique ip address. A secondary issue is that the configuration file read by each StartD also contains information about spool and log file locations which cannot be shared between StartD. So the challenge is to provide an environment to StartD where it is on a private network.

One approach is to fully virtualize the platform hardware at each physical compute node using a hyper-visor to execute multiple operating system images, each running a single StartD. This provides the requisite isolation of network bindings and configuration settings. The drawback is the memory, disk, and processor overhead of using the operating system as a isolation container. A StartD process requires about 25MB of memory including the Starter process, so an OS container is inefficient. A better approach is lightweight virtualization of just the network layer so that multiple StartDs execute in isolation within a single OS image. It is reasonable to expect that a lightweight solution on a physical machine with 4 GB of memory is capable of hosting approximately 160 'virtual' compute nodes, more when using memory swap space on disk. A further practical challenge to lightweight virtualization is that the initial implementation cannot involve modification of product code. The product group is willing to make minor modifications to network interface binding to simplify network isolation, when substantial benefit of emulating large clusters is demonstrated. Such benefit is initially displayed first using an IPTables approach which motivated a minor (¡ 10 lines of code) change to the IP binding in LoadLeveler as explained next.

The adopted methodology for lightweight virtualization of StartD nodes is now described. The initial hurdle is to generate a unique configuration file for each virtual StartD. Activation of a StartD process causes it to read a configuration file from a fixed location. The configuration file defines daemon communication intervals, locations for log files, spool directories, and IP ports on which StartD listens for external traffic. Subsequently StartD spawns a thread that binds to its set of listening ports. As part of job launch, StartD also forks off Starter processes which also rely on this configuration file. Thus, the configuration file cannot be modified while the StartD process is active. Fortunately, the LoadLeveler development team identified an an environment variable

LOADL_CONFIG which when set is recognized by StartD as an override to the global configuration file. The original intent of this variable is to manage jobs in multi-cluster LoadLeveler environment [3]. It is used here to provide each StartD with a unique set configuration parameters by setting LOADL_CONFIG to a corresponding file prior to instantiating the StartD process. However, the network conflict issue remains as the StartDs still share a common IP address and conflict over the binding to communications ports.

Tying the communication of each StartD to a different IP requires a flexible method to create multiple IP addresses and use them for CM, SchedD, and StartD communications. Because code modification is not permitted in the proof of concept phase, the approach taken to network isolation is based on iptables. Iptables maps ports and addresses at the ip layer of the transport stack. For a large scale system this requires complex setup and imposes significant performance overhead. The iptables approach is interesting and a potential solution to other lightweight virtualization problems [11]. It is successfully used to demonstrate value of virtualized StartD to the product group, but because it is not used as an ongoing solution the discussion is presented in the appendix A.

The adopted solution, implemented with minor code changes, introduces a private network between the CM node, SchedD node, and the compute nodes. Many network performance related studies use this method of creating a private network. The basic idea is to create IP aliases to the network interfaces of the physical machines hosting CM, SchedD and StartDs. The alias adapters appear in the output of the Linux command /sbin/ifconfig. All aliases are created within the same subnet so that the machines can communicate without routing. A single alias of the network interface is built for the CM and another for the SchedD node. On StartD nodes, an alias of the network interface is created for each StartD that runs on the machine. Since LoadLeveler uses hostnames to map to ip addresses, a unique hostname is assigned to every IP address and the hostname to ip mappings are placed in the /etc/hosts file of every machine. For example, to execute 500 StartDs on a physical node 500 network adapter aliases with consecutive ip addresses are created. An alias for the CM and SchedD leads to a total of 502 entries in the /etc/hosts files of the physical nodes. Having large /etc/hosts file has performance implications discussed in the experimental section 4.

With the aliased network adapters in place, communications endpoints need to be bound from every StartD to a corresponding aliased ip address. The original code binds an endpoint to the global listening port specified in the configuration file and physical machine ip address. The modified code issues a bind to the global listening port and an ip address from an aliased network adapter. The aliased adapter hostname for each StartD is passed from a new environment variable (LOADL_HOSTNAME). With this modification all StartD instances use the globally specified listening ports, together with unique ip addresses. Figure 3 summarizes the communications setup. Less than 10 lines of code are needed to support this expanded endpoint binding functionality. The procedure isolates

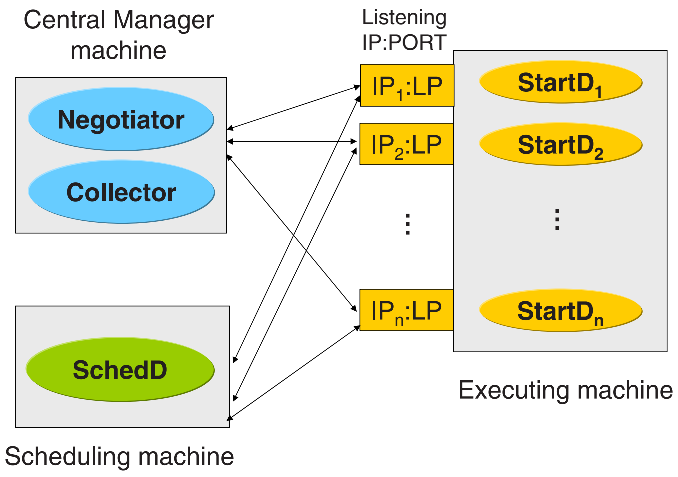


Fig. 3. LoadLeveler with virtualized StartDs – Binding to IP:Port combination

StartDs from each other while preserving common listening ports, and makes them appear to the central manager as distinct machines.

Virtualized StartDs provide a lightweight platform for studying scalability but have other limitations. Each StartD thinks it manages all resources of a node thus, the total amount of resource reported at the CM is the actual resource on the node multiplied by the total number of StartDs. As a result, the quantity reported at CM may not be used to study the turn around time of real jobs that require some amounts of static and dynamic resources on the nodes. This study is restricted to the scalability of scheduling at the Central Manager and hierarchical job launching.

4 Performance and Scalability Studies

The performance of LoadLeveler scheduling and job dispatching is evaluated in two interesting limiting cases: 1) A parallel job requiring all compute nodes in the system. 2) A single node job on an occupied cluster. The study explores the applicability of the multiple StartD per node approach and more importantly identifies bottlenecks in the LoadLeveler implementation that limit scalability. The study conveniently separates into two sections along the lines of LoadLeveler functions. Job processing from submission to launch at the compute nodes consists of the sequential and independent cycles of scheduling and dispatching. According to LoadLeveler implementation (Section 2), the former occurs in the Negotiator which executes on the CM machine, while the latter is distributed involving communicating processes on SchedD and between the StartD on each compute node. Based on our analysis, possible solutions are suggested to mitigate scalability problems and improve the performance issues.

4.1 Methodology

LoadLeveler is treated as a black box and performance data is extracted from log file messages which are recorded with microsecond timing. Care is taken that logging does not interfere with system performance as certain flags cause log performance to overwhelm actual function. Because the granularity of control of debug flags is coarse an option such as `D_NEGOTIATE` generates thousands of messages for a single scheduling operation in a large system. Although each log event takes about two microseconds to time-stamp and format and is written in the background the aggregate effect causes noticeable delay. The log file data is used to verify that log events do not impede scheduling operation.

The job description used to drive the experiments contains matching constraints on job class and the number of nodes. Typically, the executable is a shell script that invokes a 60 second `sleep` command. An MPI executable invoking `sleep` on each node is also used.

Experiments are performed in two different cluster computing environments. One is a homogeneous collection of 50 IBM pSeries 575 machines running IBM AIX version 6 and connected by a high speed SP switch. Each machine has eight dual core IBM POWER5+ processors and 32GB of memory. A heterogeneous and smaller cluster of machines is also used to collect data. This cluster typically contains 6 to 8 compute nodes consisting of partitions on an IBM pSeries 575 and IBM POWER blade servers connected by a ten gigabit ethernet switch. The blades have two dual core IBM POWER5 processors, 4 to 16GB memory, and run Red Hat Enterprise Linux version 5. Each cluster dedicates a machine for CM (scheduling and resource manager) and SchedD (dispatching and job life cycle management). Each StartD is configured to have a single Starter so that only one task is executed per virtual node.

4.2 Scheduling Analysis

The time to schedule a job depends on the number of nodes, the parallelism of the job, and the complexity of matching job requirements to compute node resources. The study starts by quantifying the dependence of single node job scheduling time on the number of compute nodes, then moves to large parallel jobs. In both cases the compute nodes are unoccupied and a single job is placed in the LoadLeveler queue. A timing event is issued in the log file when the assignment of the job to a compute node is complete and the job is dequeued from the submit queue.

Single node job scheduling analysis. Figure 4 shows scheduling time for a single node job where the independent variable is the number of compute nodes (N). Because the nodes are initially unoccupied, the expected result is that scheduling time is independent of the number of N, instead, the data show a linear dependence on N. This result is understood in terms of the three scheduling steps described in Section 2. First, the scheduler scans all machines to locate candidate nodes that have the capability to execute the job. The capability scan is an O(N) operation that produces a bit map of all machines. The bit map is applied in the subsequent machine matching steps.

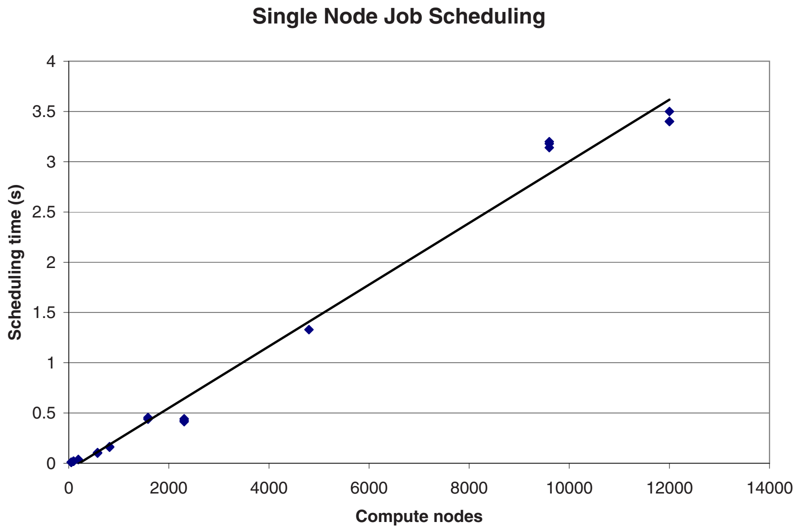


Fig. 4. Scheduling time for a single node job

The second and third scheduling steps assign the job to the compute node with the highest priority. In LoadLeveler, the priority of each compute node is defined by the system administrator using a formula syntax. The default priority scheme, used here, decreases the priority of a node linearly with the short term (five minute) average processor utilization of each machine. The scheduler maintains a priority sorted list of all compute nodes. List order is updated when jobs are scheduled or terminated as well as by periodic updates of node resource consumption reported by StartDs. The list entry for each compute node is a summary of the latest reported resource consumption information about the nodes, e.g. utilization, memory, job class, network adapters, multi-programming level. In the second step, the scheduler takes the top priority node and checks the corresponding bit map entry from the first step. If true, the latest resource information is checked to see if the node has the current capacity to execute the job. If not, the scan of the compute node list continues until a match is found. A machine match results in a job assignment to that node. In this experiment, the second step time is constant time because the cluster is unoccupied and the machine at the top of the priority list is always available.

Thus, the single node scheduling time is determined by the O(N) behavior of the first step of the scheduling process. The data in Figure 4 confirm this and show how virtual StartDs enable experiments on the 50 physical node cluster to 12,000 nodes. The data are linear and the scheduling time per node is about 250us for the static resource matching step. Furthermore, the resource manager component of the CM performed remarkably well as 12,000 nodes are brought up, identifying all resources in less than two minutes.

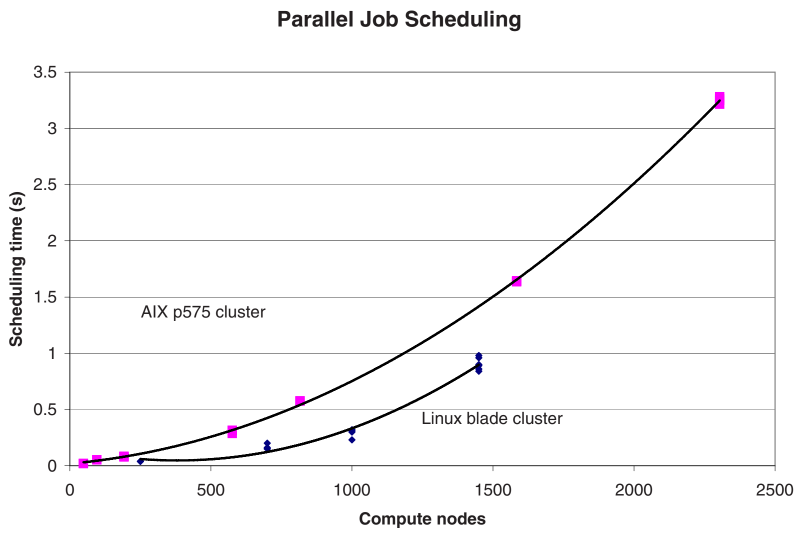


Fig. 5. Scheduling time for parallel job requesting all compute nodes

This result points to the importance of improving the performance of the capability scan step of the scheduling cycle. However, it is useful to recognize that in many scheduling conditions, generation of the bit map in the capability scan is an optimization. The resource matching of the second step is more extensive and costly than that of step one and should be avoided for nodes that cannot potentially execute the job. Such a situation occurs with heterogeneous compute resources as in an ad hoc cluster of workstations where the bit map may be sparse, significantly reducing the number of machines tested in the dynamic matching of the second step. Also, in a highly utilized cluster there is a good chance that a job is not initially scheduled and needs to be retried. The bit map is retained with the job and is not recreated in subsequent cycles.

There is still room for enhancements. One observation is that the capability scan step can be decoupled from the scheduling cycle. Results of the scan are based on static information about the jobs and compute nodes. So it can be performed when a job arrives at the CM prior to the scheduling cycle. Binding the capability scan to the scheduling cycle makes sense when the scheduler supports ad hoc clusters of workstations with intermittent availability or connectivity. A decoupled bit map can become out of sync with the cluster state. But this is not an issue for high performance compute clusters.

A further observation is that clusters are frequently homogeneous. In this case, every bit in the capability map has the same result for a given job. Here, the scan result has no added benefit to the machine list matching step. So the scheduling cycle can have distinct operational states depending on the heterogeneity. Selection of the state is inferred dynamically based on cluster workload. When recent job submissions generate a bit map that is largely ones, the scheduler folds the capability scan logic of step one into the matching of step two which eliminates the need to scan the entire cluster. If the machine assignment step fails frequently, the capability scan step is reconstituted as a distinct step to regain the advantage of having a single scan for multiple scheduling cycles.

Parallel job scheduling analysis. The next study investigates a parallel job requesting all N cluster nodes. This study targets the second scheduling step because the capability scan is performed once while machine matching and job assignment are exercised N times.

In this experiment, the system is initially unoccupied and a single job requesting all nodes is submitted. Figure 5 shows results for the two clusters. The first observation is that the data for the both clusters has a second order component that significantly impacts performance above 1000 nodes. This behavior is unexpected as the scheduling steps should be linear in the number of nodes.

4.3 Dispatching

As described in Section 2.2, SchedD starts the job dispatch process by constructing a *job object*. All information necessary to execute the task on the assigned compute nodes is contained in the job object. While many task details such as the binary executable location are common to all nodes, node specific details such as which network interface to use are also included. The job object also contains the structure of the hierarchical communication tree used to distribute and communicate job information and status between the compute nodes and the SchedD. The job object is forwarded from SchedD to the master StartD to initiate job dispatching. Subsequent responsibility for constructing the communications tree and propagating job dispatching information from the master StartD node to the compute nodes on the tree belongs to the StartDs at each compute node. The StartD has no *a priori* information about the tree structure. It decodes the job object passed to it and locates its children. Then, a new job object is created for each child customized to contain only information required for the child's subtree. This process continues until the tree is fully constructed.

Dispatching performance is studied in a large scale environment created by running multiple instances of StartDs on each compute node. The starting point is to establish the equivalence of logical and physical StartDs within the context of hierarchical communication. There is extensive communication between StartD instances during the job dispatching cycle. The pattern and concurrency of the communication processing is expected to change when multiple StartD execute on a common physical platform. For example, the concurrency is limited by the number of available free processing units. Validation starts by comparing a fully parallel cluster with a single StartD on each of 48 nodes to a single physical node with 48 instances of StartD.

The first experiment compares dispatch time as a function of the fan-out. The measured dispatch time is the interval that starts when the master StartD receives the job object and ends with an acknowledgement at the StartD from all nodes on the tree. The results are presented in Figure 6. The lower solid line corresponds to

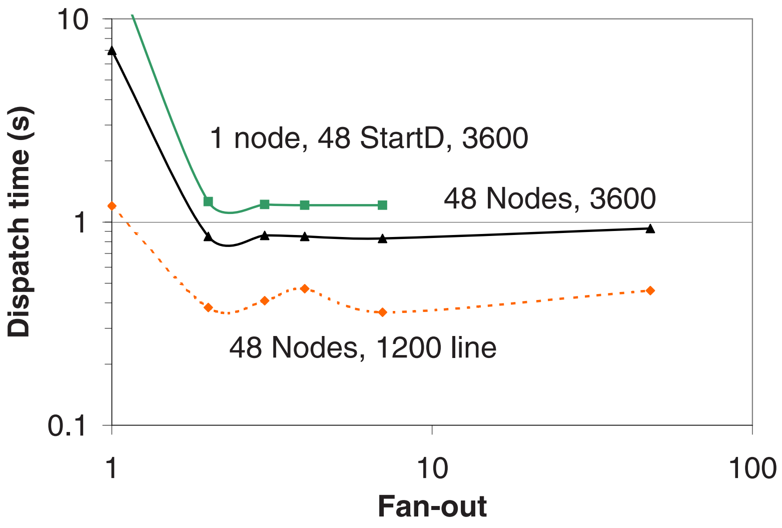


Fig. 6. Compare logical to physical StartD at 48 nodes. The 1200, 3600 indicate size of /etc/hosts file.

the fully parallel cluster of 48 machines and the '1200' on the label indicates the number of entries in the /etc/hosts file. The intermediate dashed line is the same experiment as the lower line except that the /etc/hosts file has 3600 lines. Data for the single machine, multiple StartD case is at the top.

The qualitative behavior of dispatch time is expected based on the tree architecture. The number of levels L in a tree of fan-out F with N nodes is

$$L = \left\lceil \frac{\log(1 + N(F-1))}{\log(F)} \right\rceil .$$

Performance is poor for a fan-out of unity as this degenerate case serializes communication over the 48 levels. A binary tree reduces the number of levels from 48 to 6 and the multi-processor platform concurrently executes StartDs. Additional benefit is expected as fan-out grows because of concurrency and logarithmic reduction in tree depth. The trend is expected to reverse when fan-out exceeds machine concurrency and communication becomes serialized.

Figure 6 demonstrates that for up to 48 StartD the behavior of the two configurations with fan-out is comparable. This is an important step in validating the virtual StartD methodology that allows many StartD on each physical node. However, the expected gain from increasing fan-out beyond a binary tree is absent. This suggest some operation is limiting performance and is largely independent of the fan-out or depth of the communication tree. The log file is investigated for more information, but the 48 node job does not provide a clue to the cause.

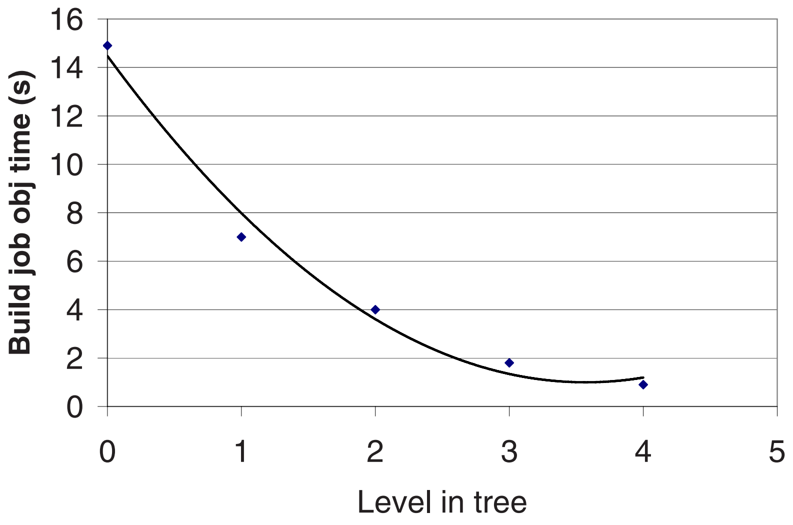


Fig. 7. Time to build job object, 2304 node binary tree

In an effort to identify the origin of the problem, a parallel job requesting 1000 nodes is executed with fan-out of 2 in the environment of 2304 compute nodes by running 48 StartD on each of the 48 physical nodes. The StartD log file shows significant processing occurs at each StartD prior to forwarding the modified job objects to the F children. In particular, the processing time is proportional to the number of compute nodes remaining on this branch of the tree. This decreases approximately as 1/F at each level down the tree (i.e. as N, N/F, ..., N/F^L, for the L^{th} level), as shown in Figure 7. It is apparent that time spent de-serializing the objects from the communications buffers and repackaging and serializing job object for each child overwhelms the potential performance gains expected from increased fan-out. To support heterogeneous clusters the external data representation (XDR, RFC-1832 (1995)) standard is used to encode/decode every field in the job object and is a large component of the observed overhead. Because most fields are read for the sole purpose of repackaging and copying from parent to child there is no reason for these fields to be decoded/encoded at each level of the tree. Structuring the HC messages to substitute much of the XDR activity with buffer copies is a potential optimization.

This discovery shows that attention to all aspects of a hierarchical communications scheme is required to achieve the anticipated gains. The layout of the data structure used to transfer information in the tree needs to be easy to parse and rebuild. This raises the question of whether a job object format is possible so that decoding and repackaging occurs in F rather than N operations. The cost of the initial construction remains proportional to N, but is incurred once at the SchedD machine instead of on the compute nodes.

This example of performance problem detection and analysis highlights the advantage of using lightweight virtualization to create a large scale system for testing job scheduling and launching. It exposes design issues not apparent at the physical cluster size available to developers. The 2304 node cluster is leveraged further by measuring the dispatch time for a 2304 node job with the tree structures of fan-out 2 to 13 with levels of 12 and 4, respectively. The time is about 100 seconds compared with 60 seconds obtained by linear extrapolation from the 48 node system of Figure 6. This is not an unreasonable prediction error for a factor of 48 scale up in a computer system.

An additional performance consideration revealed in the experiments related to Figure 6 is the effect the /etc/hosts file size. The IP alias for multiple StartDs on a node require entries in the hosts file increasing the processing time of IP lookup. Investigation using the secondary test cluster suggests that significant time is spent searching the /etc/hosts file for IP resolution. The figure demonstrates the 48 physical node data is significantly improved when /etc/hosts is reduced to 1200 from 3600 lines. Unfortunately, for large systems the size of /etc/hosts is considerable large. The speed of /etc/hosts lookups is also operating system dependent.

5 Concluding Remarks

A lightweight virtualization methodology is introduced to LoadLeveler and applied to study the scalability of job scheduling and dispatching in large scale parallel systems using modest number of physical nodes. The study identifies static resource matching in scheduling and job object processing in dispatching as potential scalability bottlenecks. and proposed solutions to their performance. Further research needs to be applied to investigate whether results observed here continue to demonstrate the same functional scaling in larger systems.

Acknowledgments

We are extremely grateful to Linda Cham for tirelessly helping us with our most last minutes requests to build the LL library for various platforms. We also thank Waiman Chan, Alexander Druyan, Nathan Falk, Kevin Gildea, Kailash Marthi, and Ravindra Sure for numerous valuable discussions. We express our gratitude to David Olshefski and Gong Su for their valuable insight on iptables setup. 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. AIX 5l workload manager (wlm),
   http://www.redbooks.ibm.com/redbooks/pdfs/sg245977.pdf
2. Darpa high productivity computing systems project,
   http://www.darpa.mil/ipto/programs/hpcs/hpcs.asp

3. IBM tivoli workload scheduler loadleveler, http://publib.boulder.ibm.com/-infocenter/clresctr/vxrx/index.jsp
4. Linux distributions, http://www.linux.org/dist/
5. 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)
6. Aridor, Y., Domany, T., Goldshmidt, O., Kliteynik, Y., Shmueli, E., Moreira, J.E.: Multitoroidal interconnects for tightly coupled supercomputers. IEEE Trans. Parallel Distrib. Syst. 19(1), 52–65 (2008)
7. Pruyne, J., Livny, M.: A worldwide flock of condors: Load sharing among workstation clusters. Journal on Future Generations of Computer Systems (1996)
8. Moreira, J.E., Chan, W., Fong, L.L., Franke, H., Jette, M.A.: An infrastructure for efficient parallel job execution in terascale computing environments. In: Supercomputing 1998: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, pp. 1–14. IEEE Computer Society, Los Alamitos (1998)
9. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel Distrib. Syst. 12(6), 529–543 (2001)
10. Pfister, G.F.: An introduction to the InfiniBand architecture. In: Jin, H., Cortes, T., Buyya, R. (eds.) High Performance Mass Storage and Parallel I/O: Technologies and Applications, ch. 42, pp. 617–632. IEEE Computer Society Press/Wiley, New York (2001)
11. Ryu, K.D., Daly, D., Seminara, M., Song, S., Crumley, P.G.: Agent multiplication: An economical large-scale testing environment for system management solutions. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, April 2008, pp. 1–8 (2008)
12. Stunkel, C.B., Shea, D.G., Aball, B., Atkins, M.G., Bender, C.A., Grice, D.G., Hochschild, P., Joseph, D.J., Nathanson, B.J., Swetz, R.A., Stucke, R.F., Tsao, M., Varker, P.R.: The sp2 high-performance switch. IBM System Journal 34(2), 185–204 (1995)
13. Tannenbaum, T., Wright, D., Miller, K., Livny, M.: Condor – a distributed job scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Linux. MIT Press, Cambridge (2001)

A Appendix Iptables for StartD Virtualization

Iptables is an approach to orchestrate the communication between the CM, SchedD and StartDs without code modification. It is a generic network packet manipulation technology that enables packet filtering, network address translation, and packet mangling. Iptables is used extensively in building Internet firewalls, redirecting traffic between servers, sharing public IP addresses.

The difficulty with StartD is that its IP endpoint sockets are bound to the listening port specified in the 'LoadL-config' file and the ip associated with the machine hostname in the network interface. Thus, while each StartD within a physical machine is assigned a unique listening port by directing it to a unique configuration file (e.g. 'LoadL-config.nnn') the ip address of the endpoint listen socket binding is hard coded. This is fine as the StartD listeners within each physical machine no longer conflict with each other. However, the CM and SchedD assume all StartD in the cluster listen on the same port number at each machine. What is required is a way for the multiple StartD on each physical machine having unique ports but a shared ip to appear to the CM and SchedD that they are at common port but unique ip addresses. In concept, the resolution is to use iptables to:

– Map outgoing packets from the StartD endpoints (unique-port, common-ip) to appear to originate at (common-port, unique-ip).
– Map outgoing packets from CM and SchedD sent to (common-port, unique-ip) to be sent to (unique port, common-ip)

This is accomplished defining iptable rules that remap these endpoint bindings transparent to the LoadLeveler code. In Figure 8(a), when CM needs to communicate to a StartD i $(1 \leq i \leq n)$, it sends the message to the respective destination IP address DIP_i but to a fixed port number LP_p. When this communication arrives at the StartD node, the packet is trapped by this iptable forwarding rule and forwards it to the appropriate port LP_i based on DIP_i. A similar rule coordinates communication originating from the SchedD to the StartDs.

StartDs are made to appear to the CM that they originate from unique IP addresses using the following three steps:

– The configuration file of each StartD is setup so that it uses a private port to communicate with the CM. Although CM is not actually listening on this port, this change is needed to identify which packets belong to which StartD at the iptables layer. When the StartD initiates a communication to the CM, its packets have this private port as the destination port in their header.
– Create an iptable rule that captures all outgoing packets from any StartD to the CM (DIP_{cm}), and based on the CM destination port number (LP_i) on the packet header, assigns a corresponding StartD IP address as the packet source IP address SIP_i, as shown in Figure 8(b). With this method, each outgoing packet to the CM is correctly labeled with the StartD that generated the packet but its destination port is not the actual port where CM is listening for StartD communications.
– Create an iptable rule on the CM node that redirects traffic destined to the list of private ports $(LP_1, LP_2, \ldots, LP_n)$ to the public port (LP_p) on which CM is listening.

The above methods create a large LoadLeveler cluster with minimal compute and memory resources. The iptables setup is automated with a the help of a few Perl and shell scripts. These scripts are parameterized so that the required number of StartDs may be activated on different physical machines. This setup is used to exercise the scheduling algorithms in the CM up to several thousand computes nodes.

A major limitation of iptables is their performance. Anecdotal evidence suggests that large numbers of iptables rules degrades the network performance

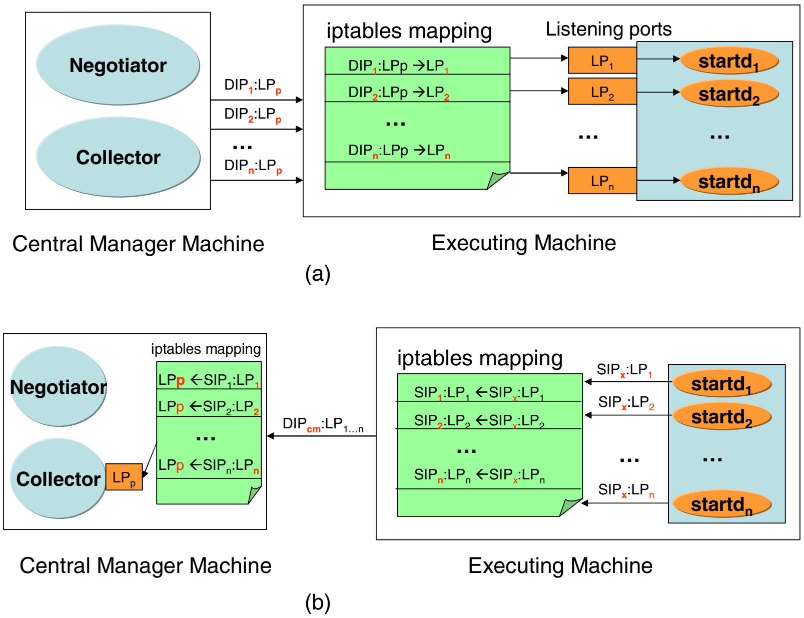


Fig. 8. LoadLeveler with virtualized StartDs – Iptables

because these rules are processed sequentially for every packet. Since network performance is a critical component of the hierarchical communication performance of LoadLeveler, the iptables approach is less favored than source code modifications described in the main text to orchestrate communications between CM, SchedD and StartDs.

Competitive Two-Level Adaptive Scheduling Using Resource Augmentation

Hongyang Sun[1], Yangjie Cao[2], and Wen-Jing Hsu[1]

[1] School of Computer Engineering, Nanyang Technological University, Block N4, Nanyang Avenue, Singapore 639798
{sunh0007,hsu}@ntu.edu.sg

[2] School of Electronic and Information Engineering, Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, Shanxi, 710049, P.R. China
caoyj@stu.xjtu.edu.cn

Abstract. As multi-core processors proliferate, it has become more important than ever to ensure efficient execution of parallel jobs on multiprocessor systems. In this paper, we study the problem of scheduling parallel jobs with arbitrary release time on multiprocessors while minimizing the jobs' mean response time. We focus on non-clairvoyant scheduling schemes that adaptively reallocate processors based on periodic feedbacks from the individual jobs. Since it is known that no deterministic non-clairvoyant algorithm is competitive for this problem, we focus on resource augmentation analysis, and show that two adaptive algorithms, AGDEQ and ABGDEQ, achieve competitive performance using $O(1)$ times faster processors than the adversary. These results are obtained through a general framework for analyzing the mean response time of any two-level adaptive scheduler. Our simulation results verify the effectiveness of AGDEQ and ABGDEQ by evaluating their performances over a wide range of workloads consisting of synthetic parallel jobs with different parallelism characteristics.

Keywords: Malleable jobs, Mean response time, Non-clairvoyant algorithm, Online scheduling, Resource augmentation analysis, Two-level adaptive scheduling, Simulations.

1 Introduction

With the proliferation of multi-core computers and the increasing use of multiprocessor systems, more software developers have started programming in parallel and migrating the existing sequential applications to the parallel platforms. One imminent challenge for the operating system is thus to schedule the parallel applications to fully exploit the multiprocessor resources.

In this paper, we study the problem of scheduling a set of parallel jobs with arbitrary release time on multiprocessors. The objective is to minimize the jobs' mean response time, where the response time of a job is defined to be the duration between its release and its completion. We consider *malleable* jobs [17] that have changing degrees of parallelism and can execute with a varying number of allocated processors [12, 13, 18]. The task of the operating system scheduler is to decide the number of processors allocated to each job at any time. Since information about the jobs' characteristics is generally unavailable to the system, we assume that the schedulers are *online non-clairvoyant*, that is, they know nothing about the job's release time, work and future parallelism when making scheduling decisions.

To date, many excellent results for online scheduling have been obtained using *competitive analysis* [9], in which the performance of a scheduling algorithm is described in terms of its competitive ratio against the optimal scheduler. However, since it has been shown in [25] that any deterministic online non-clairvoyant algorithm is $\Omega(n^{1/3})$-competitive with respect to the mean response time even for scheduling sequential jobs on a single processor, some recent studies along this line have focused on *resource augmentation analysis* [21, 26], in which the online algorithm is augmented with extra resources as compared to the adversary, either in the form of faster processors or more processors. In this case, the online algorithm is said to be s-speed c-competitive if its performance with s times of extra resources is no worse than c times that of the optimal. The rationale for resource augmentation is that the traditional competitive analysis for an online algorithm can lead to a large competitive ratio because the online algorithm, being non-clairvoyant, cannot recover from sometimes even a small mistake made on certain worst-case job instances. The extra resources for the online algorithm compensates for their non-clairvoyance on these worst-case scenarios. Hence, if an algorithm achieves competitive result with moderate increase in processor resources, then it is likely to perform comparably to the optimal on most practical workloads. The readers may wish to refer to [21, 26, 28, 27] for more elaborate interpretations of resource augmentation. Basically, the goal is to achieve competitive performance for an algorithm with minimal extra resources.

Perhaps the simplest online non-clairvoyant scheduler for parallel jobs is Equi (Equi-Partitioning)[14, 13], which divides the total number of processors evenly among all active jobs at any time. Using a sophisticated analysis, Edmonds [13] proved that Equi is $(2+\epsilon)$-speed $O(1)$-competitive with respect to the mean response time of any set of jobs. Recently, Edmonds and Pruhs [16] proposed Laps (Latest Arrival Processor Sharing), a variant of Equi that divides the total number of processors among a certain portion of the latest released jobs. They showed that Laps is $(1+\epsilon)$-speed $O(1)$-competitive with sufficiently large ϵ. The analysis of Equi and Laps employs a technique called *amortized local competitive argument*, which bounds the amortized performance of an algorithm at any local time through a carefully designed potential function, and it has become a useful technique for analyzing scheduling algorithms (see, e.g. [28, 27, 5, 4, 23, 11]). In this paper, we extend the amortized local competitive argument and provide a simple framework to analyze the mean response time for a set of perhaps less well-known but also quite effective schedulers called *two-level adaptive schedulers* [1, 18, 32].

The theoretical study of two-level adaptive schedulers was initiated by Agrawal, et al. [1]. Unlike Equi, which allocates processors to jobs without considering the jobs' utilization of the allocated resources, the two-level adaptive schedulers take a corrective approach by collecting statistics from jobs' *past* executions and using them to guide the future processor allocations. Since no knowledge about the future is assumed, they are also non-clairvoyant in nature. Specifically, the scheduling of jobs by a two-level adaptive scheduler can be decomposed into two parts: at the system level, an *OS allocator* decides the processor allocations for jobs; at the job level, a *task scheduler* schedules the tasks of each job with the allocated processors. In order to allocate processors more effectively, each task scheduler provides feedback to the OS allocator indicating the job's future processor desire. The processors are reallocated periodically by the OS allocator after each *scheduling quantum* based on the feedback from the jobs. The length of the scheduling quantum is usually set to be long enough to amortize the overheads incurred by the processor reallocations and bookkeepings for scheduling, but it should not be too long to make the feedback relevant.

Using the above two-level adaptive scheduling framework, Agrawal et al. [1] proposed AG (Adaptive Greedy) task scheduler, which calculates the processor desire for a job in each scheduling quantum with a simple multiplicative-increase multiplicative-decrease strategy based on the execution statistics of the job in the immediate previous quantum. They analyzed the performance of AG in terms of an individual job's running time and processor utilization. He et al. [18] combined AG task scheduler with DEQ (Dynamic Equi-Partitioning) [33, 24] OS allocator, which is a variant of EQUI that never allocates more processors to a job than the job's processor desire. They called the resulting two-level scheduler AGDEQ, and showed that it is $O(1)$-competitive with respect to the mean response time of any set of *batched* parallel jobs (i.e., all jobs are released at time 0). Furthermore, they showed that AGDEQ simultaneously guarantees $O(1)$-competitiveness for the makespan of arbitrarily released jobs. Sun and Hsu [32] later proposed a task scheduler ABG (Adaptive B-Greedy), which directly utilizes the job's past parallelism to calculate the processor desire and improves upon AG in terms of its desire stability. They also showed similar mean response time and makespan performances for the two-level scheduler ABGDEQ.

In this paper, we show that, for parallel jobs with *arbitrary* release time, AGDEQ and ABGDEQ are competitive with respect to the mean response time with $O(1)$ times faster processors. Compared to EQUI, which is competitive for the mean response time [14, 13], but not competitive for the makespan [29], the results of AGDEQ and ABGDEQ show that the two-level adaptive schedulers achieve both fairness and efficiency for executing parallel applications on multiprocessor systems. Moreover, we provide a framework for analyzing the mean response time of any algorithm that can be formulated as a two-level adaptive scheduler. The analysis given in this paper and in [18] offers a convenient technique for analyzing the mean response time of a wide spectrum of scheduling algorithms that utilize parallelism feedbacks from the jobs, while the analysis in the previous results [13, 15, 29, 30, 16] are applied more specifically to EQUI and its variants.

In addition, we also conduct simulations and compare the mean response time of AGDEQ and ABGDEQ with that of EQUI. The simulation results verify the effectiveness of the two-level adaptive schedulers over a wide range of workloads consisting of synthetic parallel jobs with different parallelism characteristics.

The rest of this paper is organized as follows. Section 2 formally introduces the job model and the objective function. Section 3 discusses two-level adaptive scheduling in general, and describes AGDEQ and ABGDEQ algorithms in details. Section 4 provides the mean response time analysis framework, and its applications to AGDEQ and ABGDEQ. Our simulation results are presented in Section 5. Section 6 discusses some related work, and Section 7 concludes the paper.

2 Job Model and Objective Function

We adopt the job model used by Edmonds et al. [13, 16], which allows a parallel job to have time-varying parallelism modeled by multiple phases of speedup functions. However, unlike in [13, 16], where each phase of a job admits an arbitrary non-decreasing but sub-linear speedup, we consider a simpler model where each phase has a linear speedup function up to a certain degree of parallelism, beyond which no further speedup can be gained.[1] Specifically, we consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n jobs to be scheduled on P processors. Each job J_i in the job set contains p_i phases $\langle J_i^1, J_i^2, \ldots, J_i^{p_i}\rangle$, and each phase J_i^p has an amount of *work* w_i^p, and a linear *speedup function* $\mathit{\Gamma}_i^p$ up to a certain degree of parallelism h_i^p, where $h_i^p \geq 1$. The *span* l_i^p of phase J_i^p is therefore $l_i^p = w_i^p / h_i^p$. The phase is *parallelizable* if $h_i^p = \infty$, and it is *sequential* if $h_i^p = 1$. The total work of job J_i is denoted by $w_i = \sum_{p=1}^{p_i} w_i^p$, and the total span of the job is $l_i = \sum_{p=1}^{p_i} l_i^p$. At any time t, suppose that job J_i is in its p-th phase and is allocated $a_i(t)$ processors of speed s, the effective speedup or execution rate of the job is thus given by $\mathit{\Gamma}_i^p(a_i(t)) = a_i(t) \cdot s$ if $a_i(t) \leq h_i^p$ and $\mathit{\Gamma}_i^p(a_i(t)) = h_i^p \cdot s$ if $a_i(t) > h_i^p$.

A scheduling algorithm ALG for any set $\mathcal{J}$ of jobs specifies the number $a_i(t)$ of processors allocated to each job J_i at any time t. In order for the schedule to be valid, we require that at any time t the total processor allocation is not more than the total number of processors, i.e., $\sum_{i=1}^n a_i(t) \leq P$. Let r_i denote the *release time* of job J_i. Let c_i^p denote the completion time of the p-th phase of job J_i, and let $c_i = c_i^{p_i}$ denote the *completion time* of job J_i. We also require that a valid schedule must complete all jobs in finite amount of time and can not begin to execute a phase of a job unless it has completed all its previous phases, i.e., $r_i = c_i^0 < c_i^1 < \ldots < c_i^{p_i} < \infty$, and $\int_{c_i^{p-1}}^{c_i^p} \mathit{\Gamma}_i^p(a_i(t))dt = w_i^p$ for all $1 \leq p \leq p_i$.

The job J_i is said to be *active* at time t if it is released but not completed at t, i.e., $r_i < t < c_i$. The *response time* R_i of the job is the duration between the completion time and the release time of the job, i.e., $R_i = c_i - r_i$. The *total response time* $R(\mathcal{J})$ of the entire job set $\mathcal{J}$ is thus given by $R(\mathcal{J}) = \sum_{i=1}^n R_i$, or alternatively can be expressed as $R(\mathcal{J}) = \int_0^\infty n_t dt$, where n_t is the number of active jobs at time t. The *mean response time* $\overline{R}(\mathcal{J})$ of the job set is therefore $\overline{R}(\mathcal{J}) = R(\mathcal{J})/n$.

[1] See Section 7 for our discussion on the chosen job model.

Our objective is to minimize the mean response time $\overline{R}(\mathcal{J})$ of the job set $\mathcal{J}$, for which we use *resource augmentation analysis* [21, 26]. Specifically, we equip the online algorithm Alg with processors of speed s, where $s > 1$, while the optimal algorithm is only given processors of unit speed. In this case, Alg is said to be s-speed c-competitive with respect to the mean response time if there exists a constant b such that it satisfies $\overline{R}_{\text{ALG}}(\mathcal{J}) \leq c \cdot \overline{R}^*(\mathcal{J}) + b$ for any job set $\mathcal{J}$, where $\overline{R}^*(\mathcal{J})$ denotes the mean response time of the optimal scheduler.

Let $l(\mathcal{J})$ denote the total span of job set $\mathcal{J}$, i.e., $l(\mathcal{J}) = \sum_{i=1}^{n} l_i$. Then a simple lower bound for the total response time of job set $\mathcal{J}$ is its total span, that is, $R^*(\mathcal{J}) \geq l(\mathcal{J})$, since it takes at least l_i time to complete job J_i using any scheduler on unit-speed processors.

3 Two-Level Adaptive Scheduling: From Agdeq to Abgdeq and beyond

In this section, we present how two-level adaptive scheduling can be used to schedule parallel jobs. We first introduce the basic concept of two-level adaptive scheduling, followed by detailed descriptions of two specific two-level algorithms, namely Agdeq and Abgdeq. We end this section with a remark on the general applicability of two-level adaptive schedulers.

3.1 Two-Level Adaptive Scheduling

In two-level adaptive scheduling, the execution of jobs is decomposed into two parts: a system-level *OS allocator* decides the processor allocations for jobs; and a *task scheduler* for each job in the user level executes the job with the allocated processors. The processors are reallocated by the OS allocator after each *scheduling quantum*, which is usually set to be a fixed amount of time, say L time units. In order for the OS allocator to allocate processors to jobs more effectively, the task scheduler should provide feedback to the OS allocator indicating the job's processor *desire* for each quantum, typically based on the execution statistics of the job in previous quanta. For a scheduling quantum q, let $d_i(q)$ and $a_i(q)$ denote the processor desire and the processor allocation for job J_i, respectively. We assume that the task scheduler always executes the job based on the model given in Section 2. Hence, as far as the task schedulers are concerned, their only difference lies in the strategies for estimating the processor desires. In [19], the interaction between task scheduler and the OS allocator is referred to as the *processor request-allocation protocol.* At each quantum in this protocol, we say that a job J_i is *satisfied* if its processor allocation is no less than its processor desire, i.e., $a_i(q) \geq d_i(q)$. Otherwise, the job is *deprived* if $a_i(q) < d_i(q)$. In addition, the notions of satisfied and deprived are extended from quantum to time. A job is said to be satisfied (deprived) at time t if t is within a satisfied (deprived) quantum for the job. Finally, to ease our analysis, when a new job is released in the middle of a quantum, it is not scheduled until the beginning of the next quantum.

3.2 AGDEQ

AGDEQ (Adaptive-Greedy Dynamic Equi-Partitioning) is a two-level adaptive scheduler proposed by He et al. [18] that combines the task scheduler AG [1] and the OS allocator DEQ [33, 24]. In this subsection, we will describe the two parts in detail.

The task scheduler AG estimates the processor desire for a job in a scheduling quantum based on the job's execution characteristics in the immediate previous quantum. Specifically, let t_q denote the time when quantum q starts, then the work $w_i(q)$ completed for job J_i in quantum q is given by $w_i(q) = \int_{t_q}^{t_q+L} \Gamma_i^{p_t}(a_i(q))dt$, where L is the length of the scheduling quantum and p_t is the phase job J_i is executing at time t. The execution of job J_i is said to be *efficient* in quantum q if the work $w_i(q)$ completed is at least δ fraction of the maximum amount of work that can be done in the quantum, i.e., $w_i(q) \geq \delta a_i(q)sL$, where $0 < \delta < 1$ is called the *utilization threshold*; otherwise it is *inefficient* if $w_i(q) < \delta a_i(q)sL$. Based on the efficient and inefficient classification as well as the satisfied and deprived classification for quantum q, the processor desire for the next quantum $q+1$ is calculated using a multiplicative-increase multiplicative-decrease strategy as shown in Algorithm 1, assuming that the OS allocator never allocates more processors than its desire.[2]

Algorithm 1. AG(δ)

```
1: if $w_i(q) < \delta a_i(q)sL$ then
2:     $d_i(q+1) = d_i(q)/2$ //inefficient
3: else if $a_i(q) = d_i(q)$ then
4:     $d_i(q+1) = d_i(q) \cdot 2$ //efficient and satisfied
5: else
6:     $d_i(q+1) = d_i(q)$ //efficient and deprived
```

The rationale of the AG algorithm is as follows [1]. If the allocated processors in quantum q are not utilized efficiently, then the parallelism of the job may not be as high. Therefore, the processor desire will be reduced by a factor of 2 for the next quantum $q+1$ (line 1 and line 2). If the allocated processors are utilized efficiently and the processor desire is satisfied, then the parallelism of the job could be even higher. Thus, the processor desire will be increased by a factor of 2 (line 3 and line 4). Lastly, if the allocated processors are utilized efficiently but the desire is deprived, then it is not known whether the processors could still be efficiently utilized had the desire been satisfied. Therefore, the processor desire is not changed for the next quantum (line 5 and line 6). The processor desire for the initial quantum when the

[2] In this paper, we simplify AG algorithm by setting its multiplicative factor to 2, while in [1], a tuning parameter ρ is defined that can take on any value greater than 1. The simplification is justified by the fact that the multiplicative factor is mainly related to the processor waste of the job in deductible quanta [1], which we show in this paper is actually irrelevant to the jobs' mean response time, provided that the initial processor desire is set sufficiently high.

job is first scheduled is set to be the total number P of processors.[3] Following the terminologies in [1], a job is referred to as *accounted* in quantum q if the job is both deprived and efficient. Otherwise, the job is *deductible*.

Now, we describe the OS allocator Deq, which is a variants of Equi that partitions the total number of processors equally among all active jobs. In Deq, however, if a job desires for less processors than the equal share, it will not be allocated more processors than its desire, and the remaining processors will instead be given to the other jobs with higher desires. Let $\mathcal{J}(t)$ denote the set of active jobs at time t when a new quantum begins. Based on the processor desires from all jobs in $\mathcal{J}(t)$, Deq allocates the processors as shown in Algorithm 2.

Algorithm 2. Deq($\mathcal{J}(t), P$)

```
1: if $\mathcal{J}(t) = \emptyset$ then
2:     return
3: $S = \{J_i \in \mathcal{J}(t) : d_i(t) \leq P/|\mathcal{J}(t)|\}$
4: if $S = \emptyset$ then
5:     for each $J_i \in \mathcal{J}(t)$ do
6:         $a_i(t) = P/|\mathcal{J}(t)|$ //deprived jobs get current equal share
7:     return
8: else
9:     for each $J_i \in S$ do
10:        $a_i(t) = d_i(t)$ //satisfied jobs get their desires
11:    Deq($\mathcal{J}(t) - S, P - \sum_{J_i \in S} a_i(t)$)
```

As can be seen in the pseudocode, if a job's processor desire is not more than the equal share $P/|\mathcal{J}(t)|$ of processors, the job will be satisfied (line 3 and line 10). After that, the equal share will be recalculated excluding the jobs already satisfied and the processors already allocated. The remaining processors will then be allocated to the rest of the jobs by recursively calling the main procedure (line 11) until all jobs' processor desires are satisfied or they exceed the equal share. In the latter case, each remaining job will be deprived and get the current equal share of processors (lines 4-7). As was shown in [12, 18], if there are deprived jobs for a quantum, then all P processors must have been allocated by Deq, and each deprived job will have the same number of allocated processors, which is higher than the initial equal share $P/|\mathcal{J}(t)|$. Note that in this paper, as in [12, 13, 18, 16], we assume that the number of processors allocated to a job can be non-integral. The fractional allocation can be considered as time-sharing a processor among the jobs.

[3] Note that in [1], the initial desire is set to be 1. This more conservative strategy is to ensure that jobs do not waste a lot of processors, especially in deductible quanta, which intuitively could affect the mean response time of the job set, since the wasted processors of a job could have been well utilized by the other jobs to speed up their executions [18]. However, we show in this paper that the mean response time of the jobs is actually independent of their deductible waste. This phenomenon can also be observed in the analysis of Equi, which fares poorly in terms of its processor utilization, yet it still achieves good performance in terms of mean response time [13].

3.3 ABGDEQ

Abgdeq (Adaptive B-Greedy Dynamic Equi-Partitioning) was proposed by Sun and Hsu [32] and it combines OS allocator Deq with task scheduler Abg, which directly calculates the average parallelism of the job in a quantum, and uses it as the processor desire for the next quantum. Specifically, let t_q denote the time when quantum q starts, then the work $w_i(q)$ completed for job J_i in quantum q is $w_i(q) = \int_{t_q}^{t_q+L} \Gamma_i^{p_t}(a_i(q))dt$, and the span $l_i(q)$ reduced for job J_i in quantum q is $l_i(q) = \int_{t_q}^{t_q+L} \Gamma_i^{p_t}(a_i(q))/h_i^{p_t}dt$, where L is the length of the scheduling quantum and p_t is the phase job J_i is executing at time t. Although the instantaneous parallelism of job J_i at any time during quantum q may vary, its average parallelism $A_i(q)$ in the quantum is given by $A_i(q) = w_i(q)/l_i(q)$. Abg directly sets the processor desire for quantum $q + 1$ to the average parallelism of quantum q, i.e., $d_i(q+1) = A_i(q)$.[4] This strategy makes the processor desire more representative of the job's average processor requirement, and eliminates the desire instability problem of Ag when the parallelism of the job stays constant for sufficiently long time (see Section 5). Again, the initial desire is set to be the total number P of processors. In addition, job J_i is said to be *under-allocated* in quantum q if the average parallelism is at least the processor allocation, i.e., $A_i(q) \geq a_i(q)$, otherwise it is *over-allocated* if $A_i(q) < a_i(q)$. Following the terminologies from Ag, a job is *accounted* if it is both deprived and under-allocated, and otherwise it is *deductible*.

3.4 Remark

Beyond Agdeq and Abgdeq, the general concept of two-level adaptive scheduling can represent a rich class of scheduling algorithms with various other feedback mechanisms and allocation policies. For instance, Equi can be considered as a special type of two-level adaptive scheduler with variable quantum length (a quantum only expires if a job completes or a new job is released) and an oblivious OS allocator (which always divides the processors equally among the active jobs regardless of each job's processor desire). In the following section, we will present a general framework for analyzing the mean response time on this class of schedulers.

4 Mean Response Time Analysis

In this section, we present a general framework for the mean response time analysis of the two-level adaptive schedulers, and apply it to Agdeq and Abgdeq algorithms. We begin this section by introducing a few concepts and notations.

[4] In [32], the processor desire for quantum $q + 1$ is set to be a linear combination of the average parallelism and the processor desire of quantum q, i.e., $d_i(q + 1) = (1-v)A_i(q)+vd_i(q)$, where v is called the *convergence rate*. In this paper, we set the convergence rate to be $v = 0$, hence make the processor desire achieve one-quantum convergence towards the job's average parallelism when the latter is constant.

4.1 Preliminaries

At any time t, a job J_i executed by the online algorithm can be characterized by certain properties. For example, a job can be classified according to "satisfied", "deprived", "accounted", or "deductible" as described in the preceding section. Our analysis relies on identifying two such properties A and B for the set of active jobs. Let $\mathcal{J}(t)$ denote the set of active jobs at time t, and let $\mathcal{J}_A(t)$ and $\mathcal{J}_B(t)$ denote the sets of active jobs at time t that satisfy property A and property B, respectively. Throughout the execution of job J_i, let $a_A(J_i)$ denote the total processor allocation when job J_i satisfies property A, i.e., $a_A(J_i) = \int_0^\infty a_i(t)s \cdot [J_i(t) \in \mathcal{J}_A(t)]dt$, and let $t_B(J_i)$ denote the total amount of time when job J_i satisfies property B, i.e., $t_B(J_i) = \int_0^\infty s \cdot [J_i(t) \in \mathcal{J}_B(t)]dt$, where s denotes the processor speed of the online algorithm and $[x]$ is 1 if proposition x is true and 0 otherwise. In addition, we also require the notion of total amount of time for the entire job set $\mathcal{J}$ that satisfies property B, which is defined to be $t_B(\mathcal{J}) = \sum_{i=1}^n t_B(J_i)$. To simplify our notations, we let $n_t = |\mathcal{J}(t)|$ denote the number of active jobs at time t, and let $n_t^A = |\mathcal{J}_A(t)|$ and $n_t^B = |\mathcal{J}_B(t)|$ denote the number of active jobs at time t that satisfy property A and property B, respectively. As far as this paper is concerned, the two properties A and B are chosen such that $\mathcal{J}_A(t)$ and $\mathcal{J}_B(t)$ are disjoint, i.e., $\mathcal{J}_A(t) \bigcap \mathcal{J}_B(t) = \emptyset$, and they cover the whole set of active jobs, i.e., $\mathcal{J}_A(t) \bigcup \mathcal{J}_B(t) = \mathcal{J}(t)$. Hence, we have $n_t^A + n_t^B = n_t$.

We also introduce the notions of *t-prefix* and *t-suffix* for jobs to ease our analysis. For an online algorithm, define the t-prefix $J_i(\overleftarrow{t})$ of job J_i to be the portion of the job executed before time t, and the t-suffix $J_i(\overrightarrow{t})$ to be the portion executed after time t. Specifically, if the online algorithm is executing the p-th phase of job J_i at time t, then $J_i(\overleftarrow{t})$ consists of the first $p-1$ phases $\langle J_i^1, J_i^2, \ldots, J_i^{p-1}\rangle$ of job J_i, followed by part of the p-th phase with work $\int_{c_i^{p-1}}^t \Gamma_i^p(a_i(t))dt$; and $J_i(\overrightarrow{t})$ begins with the rest of the p-th phase with work $\int_t^{c_i^p} \Gamma_i^p(a_i(t))dt$, followed by the remaining phases $\langle J_i^{p+1}, J_i^{p+2}, \ldots, J_i^{p_i}\rangle$ of job J_i. In addition, we extend the definitions of t-prefix and t-suffix from a job to a job set such that $\mathcal{J}(\overleftarrow{t}) = \{J_1(\overleftarrow{t}), J_2(\overleftarrow{t}), \ldots, J_{n_t}(\overleftarrow{t})\}$ and $\mathcal{J}(\overrightarrow{t}) = \{J_1(\overrightarrow{t}), J_2(\overrightarrow{t}), \ldots, J_{n_t}(\overrightarrow{t})\}$. We let $\mathcal{J}^*(\overleftarrow{t}) = \{J_1^*(\overleftarrow{t}), J_2^*(\overleftarrow{t}), \ldots, J_{n_t^*}^*(\overleftarrow{t})\}$ and $\mathcal{J}^*(\overrightarrow{t}) = \{J_1^*(\overrightarrow{t}), J_2^*(\overrightarrow{t}), \ldots, J_{n_t^*}^*(\overrightarrow{t})\}$ denote the t-prefix and t-suffix of job set $\mathcal{J}$ executed by the optimal scheduler, respectively, where n_t^* is the number of active jobs at time t under the optimal scheduler.

4.2 Analysis Framework

Our analysis framework adopts the *amortized local competitive argument* [27], which bounds the amortized performance of an online algorithm at any time through a potential function. In addition, we also extend the *two-step analysis* used in [18] for bounding the mean response time of batched parallel jobs (i.e., when all jobs arrive at time 0). In the case of two-level adaptive schedulers, we first analyze the task scheduler by bounding two specific properties of it on an individual job. We then apply the amortized local competitive argument to the OS allocator. Finally, by combining the analysis of the task scheduler and the OS allocator, we can obtain the mean response time performance of the two-level algorithm. Specifically, our analysis develops as follows.

Step (1): For the task scheduler, choose two properties A and B. Then bound the processor allocation $a_A(J_i)$ to any job J_i in terms of the total work w_i of the job, as well as the amount of time $t_B(J_i)$ for any job J_i in terms of the total span l_i of the job, on processors of any speed s, where $s > 0$. That is, find coefficients γ_1, γ_2 and constant γ_3 such that

$$a_A(J_i) \leq \gamma_1 \cdot w_i \ , \tag{1}$$

$$t_B(J_i) \leq \gamma_2 \cdot l_i + \gamma_3 \ . \tag{2}$$

Step (2): For the OS allocator, with properties A and B chosen in Step (1) in mind, find a potential function $\Phi(t)$, a processor speed s', and coefficients c_1 and c_2 such that on processors of speed $s = s' + \epsilon$ for any $\epsilon > 0$, the execution of the job set satisfies the following

- *Boundary Condition*: $\Phi(0) = 0$ and $\Phi(\infty) \geq 0$;
- *Arrival Condition*: $\Phi(t)$ does not increase when new jobs arrive;
- *Completion Condition*: $\Phi(t)$ does not increase when jobs complete;
- *Running Condition*:

$$\frac{dR(\mathcal{J}(t))}{dt} + \frac{d\Phi(t)}{dt} \leq c_1 \cdot \frac{dR^*(\mathcal{J}^*(t))}{dt} + c_2 \cdot \frac{dt_B(\mathcal{J}(t))}{dt} \ , \tag{3}$$

where $\frac{dR(\mathcal{J}(t))}{dt} = \lim_{\Delta t \to 0} \frac{R(\mathcal{J}(\overleftarrow{t+\Delta t})) - R(\mathcal{J}(\overleftarrow{t}))}{dt}$ denotes the change of total response time under the online algorithm in an infinitesimal interval of time Δt, and apparently we have $\frac{dR(\mathcal{J}(t))}{dt} = n_t$. Similarly, the change of total response time under the optimal algorithm satisfies $\frac{dR^*(\mathcal{J}^*(t))}{dt} = n_t^*$, and the change of total amount of time for the job set satisfying property B under the online algorithm is given by $\frac{dt_B(\mathcal{J}(t))}{dt} = sn_t^B$. In addition, $\frac{d\Phi(t)}{dt} = \lim_{\Delta t \to 0} \frac{\Phi(t+\Delta t) - \Phi(t)}{dt}$ denotes the change of potential function in interval Δt. Note that we assume Δt is infinitesimally small such that no new job arrives, and no job completes, or makes a transition between two phases, or experiences processors reallocation under both the online algorithm and the optimal.

The form of the potential function $\Phi(t)$ may not be unique, but it usually depends on the processor allocation $a_A(J_i)$ and the coefficient γ_1 given in Inequality (1). In addition, coefficient γ_1 can also affect the choice of processor speed s' for Inequality (3) to be satisfied. In the next subsection, we will provide a concrete example on choosing the potential function and the processor speed for the analysis of the OS allocator DEQ. Now, combining the results of Step (1) and Step (2), we can obtain the performance for the two-level algorithm. First, summing over all jobs for Inequality (2), we get $t_B(\mathcal{J}) \leq \gamma_2 \cdot l(\mathcal{J}) + \gamma_3 n$. Since $l(\mathcal{J})$ is a lower bound on the total response time of job set $\mathcal{J}$, integrating Inequality (3) over time, we have

$$\begin{aligned} R(\mathcal{J}) &\le c_1 \cdot R^*(\mathcal{J}) + c_2 \cdot t_B(\mathcal{J}) \\ &\le c_1 \cdot R^*(\mathcal{J}) + c_2\gamma_2 \cdot l(\mathcal{J}) + c_2\gamma_3 n \\ &\le (c_1 + c_2\gamma_2) \cdot R^*(\mathcal{J}) + c_2\gamma_3 n \ . \end{aligned}$$

The mean response time $\overline{R}(\mathcal{J})$ thus satisfies $\overline{R}(\mathcal{J}) \le (c_1 + c_2\gamma_2) \cdot \overline{R}^*(\mathcal{J}) + c_2\gamma_3$, which suggests that the two-level algorithm is $(s' + \epsilon)$-speed $(c_1 + c_2\gamma_2)$-competitive with respect to the mean response time for any job set $\mathcal{J}$, provided that c_2 and γ_3 are constants.

4.3 Performance of AGDEQ

We now apply the framework outlined in the preceding subsection to analyze the mean response time of two-level adaptive scheduler AGDEQ. We choose property A and property B for AGDEQ to be "accounted" and "deductible", respectively.

We first focus on the task scheduler AG, whose total accounted allocation and the total deductible time have been bounded in [1] on unit-speed processors. Using the same argument [1], we can show similar results for AG on speed s processors. The following lemma gives the performance bounds.

Lemma 1. *Suppose that* AG *schedules a job J_i with work w_i and span l_i on speed s processors. Then the total accounted allocation to the job satisfies $a_A(J_i) \le w_i/\delta$, and the total deductible time of the job satisfies $t_B(J_i) \le 2l_i/(1-\delta) + 2L$, where $\delta < 1$ is* AG*'s utilization threshold and L is the length of the scheduling quantum.*

Proof sketch. The total accounted allocation $a_A(J_i)$ to job J_i can be directly inferred from the definition of accounted quantum. Specifically, since an accounted quantum q for job J_i is also efficient, we have $a_i(q)sL \le w_i(q)/\delta$. Let AC denote the set of accounted quanta for the job. The total accounted allocation thus satisfies $a_A(J_i) = \sum_{q\in AC} a_i(q)sL \le \sum_{q\in AC} w_i(q)/\delta \le w_i/\delta$.

The total deductible time $t_B(J_i)$ of job J_i can be bounded by considering the total inefficient time and the total efficient-and-satisfied time, separately. The former is no more than $l_i/(1-\delta)$ by considering the reduction of the job's span in each inefficient quantum. The latter can be related to the former, because there exists a correspondence between the set of inefficient quanta and the set of efficient-and-satisfied quanta due to the multiplicative-increase multiplicative-decrease strategy [3]. By setting the initial processor desire to P, the total efficient-and-satisfied time turns out to be no more than $l_i/(1-\delta) + L$. The total deductible time of the job is thus bounded by summing up the two terms as well as the additional waiting time of the job after its arrival, which is at most the quantum length L. □

We now turn to the analysis of the OS allocator DEQ using amortized local competitive argument. We adopt the potential function used by Lam et al. [23] in the context of online speed scaling and tailor it to suit the mean response time analysis of two-level adaptive schedulers. Specifically, at any time t, let $n_t(z)$ denote the number of jobs whose remaining accounted allocation is at least $\gamma_1 z$ under AGDEQ, i.e., $n_t(z) = \sum_{i=1}^{n_t}[a_A(J_i(\overrightarrow{t})) \geq \gamma_1 z]$, and let $n_t^*(z)$ denote the number of jobs whose remaining work is at least z under the optimal, i.e., $n_t^*(z) = \sum_{i=1}^{n_t^*}[w(J_i^*(\overrightarrow{t})) \geq z]$. Apparently, $n_t(z)$ and $n_t^*(z)$ are staircase-like decreasing functions of z, and Figure 1(a) shows an example of $n_t(z)$ and $n_t^*(z)$ at a given time t. The potential function is defined to be

$$\Phi(t) = \eta \int_0^\infty \left[\left(\sum_{i=1}^{n_t(z)} i \right) - n_t(z) n_t^*(z) \right] dz \ , \tag{4}$$

where $\eta = \frac{2\gamma_1}{\epsilon P}$. For convenience, define $\phi_t(z) = \left(\sum_{i=1}^{n_t(z)} i\right) - n_t(z)n_t^*(z)$. We can now check the four conditions for Step (2) of the analysis framework given in the preceding subsection.

- Boundary Condition: at time 0, no job exists in the system. The terms $n_t(z)$ and $n_t^*(z)$ are both 0 for all z. Therefore, we have $\Phi(0) = 0$. At time ∞, no job remains in the system, so again we have $\Phi(\infty) = 0$. Hence, the boundary condition holds.

- Arrival Condition: suppose that a new job with work w' arrives at time t. Let t^- and t^+ denote the instances right before and after the job arrives. Hence, we have $n_{t^+}^*(z) = n_{t^-}^*(z) + 1$ for $z \leq w'$ and $n_{t^+}^*(z) = n_{t^-}^*(z)$ for $z > w'$. Similarly, $n_{t^+}(z) = n_{t^-}(z) + 1$ for $z \leq a'/\gamma_1$ and $n_{t^+}(z) = n_{t^-}(z)$ for $z > a'/\gamma_1$, where a' is the total accounted allocation to the job. Figure 1(b) illustrates the changes of $n_t(z)$ and $n_t^*(z)$ in this case. Note that $a'/\gamma_1 \leq w'$ from Step (1) of the analysis. Thus, it is obvious that for $z > w'$, we have $\phi_{t^+}(z) = \phi_{t^-}(z)$. For $z \leq w'$, we consider two cases.

Case 1: for $z \leq a'/\gamma_2$, we have $\phi_{t^+}(z) - \phi_{t^-}(z) = \left(\sum_{i=1}^{n_{t^-}(z)+1} i\right) - (n_{t^-}(z) + 1)\left(n_{t^-}^*(z) + 1\right) - \left(\sum_{i=1}^{n_{t^-}(z)} i\right) + n_{t^-}(z)n_{t^-}^*(z) = -n_{t^-}^*(z) \leq 0$.

Case 2: for $a'/\gamma_2 \leq z \leq w'$, we have $\phi_{t^+}(z) - \phi_{t^-}(z) = \left(\sum_{i=1}^{n_{t^-}(z)} i\right) - n_{t^-}(z)\left(n_{t^-}^*(z) + 1\right) - \left(\sum_{i=1}^{n_{t^-}(z)} i\right) + n_{t^-}(z)n_{t^-}^*(z) = -n_{t^-}(z) \leq 0$.

Hence, $\Phi(t^+) = \eta \int_0^\infty \phi_{t^+}(z)dz \leq \eta \int_0^\infty \phi_{t^-}(z)dz = \Phi(t^-)$, and the arrival condition holds.

- Completion Condition: when a job completes under AGDEQ or the optimal algorithm, the potential function $\Phi(t)$ remains unchanged, because in such cases, $n_t(z)$ or $n_t^*(z)$ only reduces by 1 for $z = 0$. Hence, the completion condition also holds.

At this point, the first three conditions hold true regardless of the OS allocator used and the properties A and B chosen. It remains to check the running condition, which typically depends on the specific OS allocator as well as the properties A and B. The following lemma shows the running condition of DEQ with property A and property B being "accounted" and "deductible", respectively.

Lemma 2. *Suppose that* DEQ *schedules a job set $\mathcal{J}$ on speed s processors with* AG. *Then the running condition in Inequality (3) is satisfied with potential*

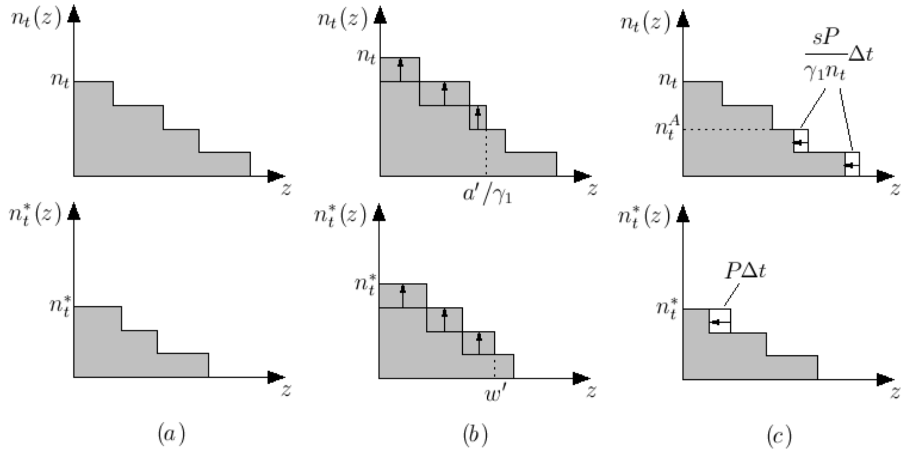


Fig. 1. (a) An example of $n_t(z)$ and $n^*_t(z)$ at a given time t. (b) The changes of $n_t(z)$ and $n^*_t(z)$ after a new job arrives. (c) The changes of $n_t(z)$ and $n^*_t(z)$ in an infinitesimally small interval of time Δt.

function in Equation (4), processor speed $s = 2\gamma_1 + \epsilon$, and coefficients $c_1 = 2s/\epsilon$ and $c_2 = 2/\epsilon$, provided that property A and property B are chosen to be "accounted" and "deductible", respectively.

Proof. As mentioned in Section 3.2, DEQ ensures that accounted jobs, which are deprived by definition, get at least P/n_t processors at any time t. In the worst case, these n^A_t accounted jobs at time t have the most remaining accounted processor allocation among the n_t active jobs, while the optimal scheduler executes the job with the least remaining work using all P processors. As a result, which can be seen from Figure 1(c), each of the bottom n^A_t horizontal stripes of $n_t(z)$ shrinks by $sP\Delta t/(\gamma_1 n_t)$, and the top horizontal stripe of $n^*_t(z)$ shrinks by $P\Delta t$ in interval Δt. The change of the potential function can then be bounded by

$$\begin{aligned}
\frac{d\Phi(t)}{dt} &= \frac{\eta}{\Delta t}\int_0^\infty \left[\left(\sum_{i=1}^{n_{t+\Delta t}(z)} i\right) - n_{t+\Delta t}(z)n^*_{t+\Delta t}(z) - \left(\sum_{i=1}^{n_t(z)} i\right) + n_t(z)n^*_t(z)\right] dz \\
&\le \frac{\eta}{\Delta t}\int_0^\infty \left[\left(\sum_{i=1}^{n_{t+\Delta t}(z)} i\right) - \left(\sum_{i=1}^{n_t(z)} i\right)\right] dz \\
&\quad + \frac{\eta}{\Delta t}\int_0^\infty \left[n_t(z)\left(n^*_t(z) - n^*_{t+\Delta t}(z)\right) + n^*_t(z)\left(n_t(z) - n_{t+\Delta t}(z)\right)\right] dz \\
&\le \frac{2\gamma_1}{\epsilon P\Delta t}\left(-\frac{n^A_t(n^A_t+1)}{2}\cdot\frac{sP}{\gamma_1 n_t}\Delta t + n_t P\Delta t + n^*_t\frac{sPn^A_t}{\gamma_1 n_t}\Delta t\right) \\
&\le \frac{2\gamma_1}{\epsilon}\left(1 - \frac{x_t^2 s}{2\gamma_1}\right)n_t + \frac{2s}{\epsilon}n^*_t \ , \qquad (5)
\end{aligned}$$

where $x_t = n_t^A/n_t$, and $0 \le x_t \le 1$. Since a job is either accounted or deductible, we have $n_t^B = (1 - x_t)n_t$. It can be easily verified that the running condition holds for all values of x_t by substituting Inequality (5), $s = 2\gamma_1 + \epsilon$, $c_1 = 2s/\epsilon$ and $c_2 = 2/\epsilon$ into Inequality (3). □

Now, we can establish the mean response time performance of two-level adaptive scheduler AGDEQ in the following theorem.

Theorem 1. *AGDEQ is $(\frac{2}{\delta} + \epsilon)$-speed $\left(2 + \frac{4}{\delta(1-\delta)\epsilon}\right)$-competitive with respect to the mean response time of any job set, where $\delta < 1$ is AG's utilization threshold.*

Proof. The theorem follows by combining the analysis given in Section 4.2 and the results of task schedulers AG in Lemma 1 and the result of OS allocator DEQ in Lemma 2. □

4.4 Performance of ABGDEQ

In this subsection, we show the mean response time of two-level adaptive scheduler ABGDEQ. Again, we choose property A and property B to be "accounted" and "deductible", respectively.

The performance of task scheduler ABG relies on a certain characteristic of the job, which is called *transition factor* in [32] and denoted as C_L for a given quantum length L. Roughly speaking, the transition factor of a job characterizes how fast the job's parallelism changes with time in the worst case, and hence reflects the degree of difficulty to schedule it in an adaptive fashion. The following lemma bounds the performance of ABG on speed s processors. The proof follows closely that of Lemma 1 and can be found in [32].

Lemma 3. *Suppose that ABG schedules a job J_i with work w_i and span l_i on speed s processors. Then the total accounted allocation to the job satisfies $a_A(J_i) \le 2w_i$, and the total deductible time of the job satisfies $t_B(J_i) \le (C_L + 1)l_i + 2L$, where C_L is the transition factor of the job and L is the length of the scheduling quantum.* □

The following theorem gives the mean response time performance of ABGDEQ.

Theorem 2. *ABGDEQ is $(4+\epsilon)$-speed $\left(2 + \frac{10+2C_L}{\epsilon}\right)$-competitive with respect to the mean response time of any job set, where $C_L \ge 1$ is the maximum transition factor of the jobs in the job set.*

Proof. Since we can apply the analysis of DEQ to ABGDEQ as well, combining the results of Lemma 3 and Lemma 2, the theorem follows. □

4.5 Discussions

As suggested in Section 3.4, we can formulate EQUI as a two-level adaptive scheduler, where EQUI serves as the OS allocator itself that interacts with an arbitrary task scheduler using variable quantum length. To analyze its mean response time, we choose properties A and B as follows. At any time t, a job J_i satisfies property A if its processor allocation is no more than the maximum parallelism of the phase the job is currently executing, i.e., $a_i(t) \leq h_i^{p_t}$. Otherwise, the job satisfies property B if $a_i(t) > h_i^{p_t}$. In this case, we can easily show that the coefficients γ_1 and γ_2 are both equal to 1, which when combined with a similar analysis in Lemma 2, can lead to the mean response time performance of $(2+\epsilon)$-speed $(2+\frac{6}{\epsilon})$-competitiveness. This demonstrates the generality of two-level adaptive scheduling as well as the usefulness of our analysis framework. Note that the results in this paper are obtained by augmenting the online schedulers with extra-speed processors. Slightly larger competitive ratios can be obtained by giving them extra number of processors as shown in [13, 16].

It is also worth noting that the extra resources required by AGDEQ and ABGDEQ as well as their competitive ratios are more than that of EQUI, which implies that the two-level adaptive schedulers have inferior mean response time performance in the worst case. The same phenomenon can be observed when comparing the competitive ratios of AGDEQ, ABGDEQ and EQUI for scheduling batched parallel jobs [14, 18, 32]. The reason is because two-level adaptive schedulers only utilize the history of the job to generate feedbacks and we assume that the job's future parallelism need not be correlated to its past. Hence, in the worst case, the adversary can always make the future parallelism of the job deviate from its processor desire, e.g., by forcing the job to have high parallelism when its processor desire is low or vice versa. Thus, compared to EQUI, the OS allocator of AGDEQ and ABGDEQ can be tricked into making poorer decisions, resulting in worse processor distributions.

In practice, however, the worst-case scenario is not likely to occur. Therefore, we expect that AGDEQ and ABGDEQ should perform comparably to or even better than EQUI, especially when the parallelism of the job does not change frequently, hence the correlation between the future parallelism and the past can be well exploited by the adaptive strategies of AGDEQ and ABGDEQ. Moreover, the practical performances of the two-level adaptive schedulers may also depend upon the specific parallelism characteristics of the jobs, the length of the scheduling quantum selected and the amount of system overhead incurred, etc., which are omitted in the theoretical analysis. We will evaluate the impacts of these factors in the next section by carrying out simulations.

5 Simulations

In this section, we conduct simulations on two-level adaptive schedulers AGDEQ and ABGDEQ using synthetic parallel jobs with various parallelism characteristics. To better understand adaptive scheduling, we first study how task schedulers AG and ABG respond to these parallelism characteristics in terms of their processor desire estimation. We then focus on the mean response time of AGDEQ and ABGDEQ by comparing them with EQUI on various workloads and by studying the impacts of different quantum length and system overhead. Since resource augmentation employed in the previous sections only serves as an analysis tool for deriving the performance bound of an algorithm, for a fair comparison, we assign unit-speed processors to all algorithms in our simulations instead of giving them different extra-speed processors.

5.1 Synthetic Parallel Jobs

We construct synthetic parallel jobs with different types of parallelism characteristics. Specifically, we identify five distinct parallelism variation curves, which are specified by Step, Impulse, Ramp, Poly(I) and Poly(II) functions, respectively, and they describe precisely how the parallelism changes with time. Figure 2(a) shows these five parallelism variation curves with each one containing the same underlining work and span, hence the same average parallelism. Among them, the Step function can represent part of a data-parallel job that contains constant and stable parallelism. The Impulse function, with drastically increased parallelism after a sequential phase, can approximate part of an irregular parallel job containing transient and spiky parallelism profile. The Ramp, Poly(I) and Poly(II) functions, which are constructed by polynomials of degree 1, 3 and 1/3 respectively in our simulation, can represent parts of a parallel job whose parallelism increases at different levels of steepness. Since the exact parallelism characteristics of the actual applications are generally unknown, we believe that these types of parallelism variation curves can represent a wide range of parallelism structures, which are useful for evaluating scheduling algorithms.

Besides the increasing parallelism curves as shown in Figure 2(a), we can also have parallelism structures specified by the corresponding decreasing curves, which together with the increasing curves form the basic building blocks for our parallel job construction. In our simulations, each *block* contains a pair of increasing and decreasing curves of the same type with the average parallelism chosen uniformly from 1 to 200 and the length fixed at 250. In addition, to study the impacts of different parallelism variations on scheduling algorithms, we only generate homogeneous jobs, where each job contains over 500 blocks of the same type interconnected by sequential phases with the same length. Note that the concept of a block used here should be distinguished from that of a phase introduced in Section 2. While a block describes the parallelism structure of a job over a period of time, a phase refers to a segment of the job in which the parallelism is constant.

5.2 Transient Response

To better understand the behavior of two-level adaptive scheduling algorithms, we first focus on task schedulers AG and ABG in this subsection by studying their *transient response* to different parallelism variation curves.[5]

[5] The term "transient response" stems from electrical/control engineering, and often refers to the response of a system to changes in the input signal from a steady state. In this case, we use transient response to describe how a task scheduler responds to different parallelism variations in terms of its processor desire estimation.

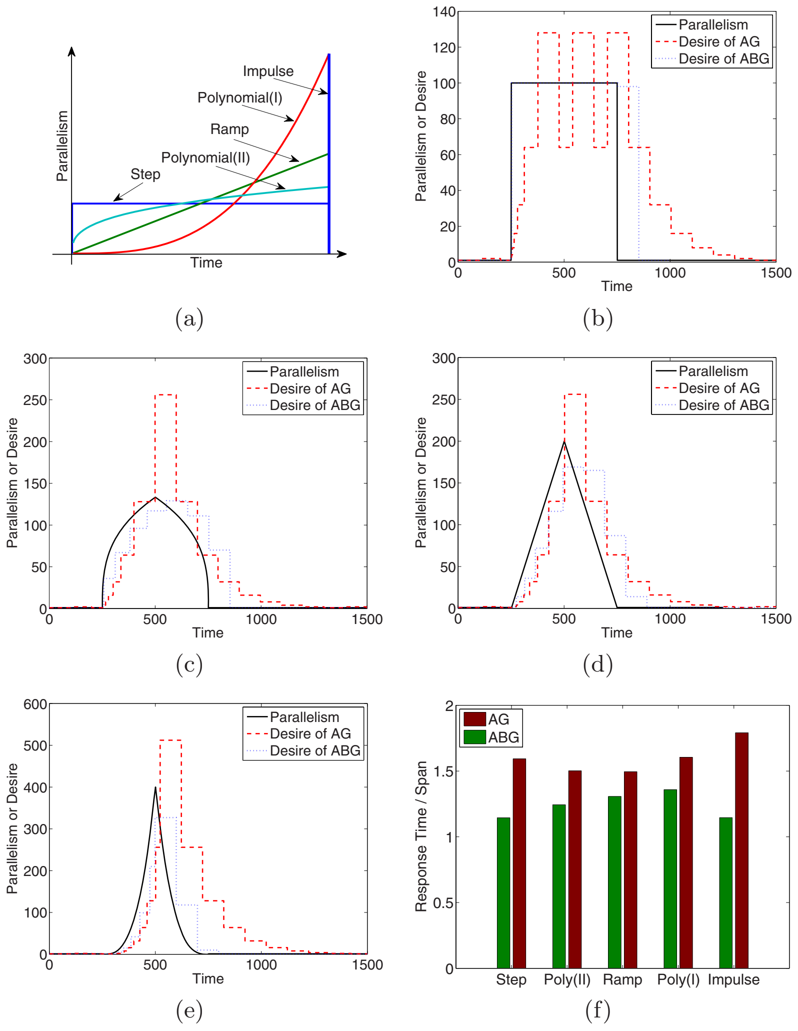


Fig. 2. (a) Five different parallelism variation curves, represented by the Step, Poly(II), Ramp, Poly(I) and Impulse functions. (b)-(e) Transient responses of AG and ABG on the Step, Poly(II), Ramp, and Poly(I) functions. (f) The response time ratios of AG and ABG on five parallel jobs with same average parallelism but different parallelism variations.

Figures 2(b)-2(e) demonstrate the transient response of AG and ABG on four parallelism variations given in the previous subsection. (The response of the Impulse function is similar to that of the Step function and is not shown.) The length of the scheduling quantum is set to 100, which is scaled in the figure to restore the original parallelism variation. We assume that the desires for both schedulers start and end at a steady state with value of 1, and are satisfied by the

OS allocator at all time. As shown in these figures, both adaptive schedulers can efficiently adjust the processor desires based on the parallelism changes, although AG and ABG exhibit different transient responses with respect to different parallelism variations. For the Step function, AG is able to gradually catch up with the parallelism change but suffers from desire instability even when the parallelism remains constant. In contrast, ABG rapidly approaches the parallelism within a quantum, and thereafter provides stable desires by directly utilizing the average parallelism of the job. For the other functions, both AG and ABG respond gradually to the parallelism variations with ABG in general following more closely the changes of the parallelism and thus taking shorter time to reach steady state. This is due to ABG's more effective processor desire calculation.

To confirm the quality of feedbacks observed in the transient responses, we also measure the response time of AG and ABG on five parallel jobs with the same average parallelism but different parallelism variations. Figure 2(f) shows the performances of AG and ABG on each of the five jobs in terms of the job's response time normalized by its span. We can see clearly that ABG indeed outperforms AG for all parallelism variations. This is especially true on the Step and Impulse functions, where ABG shows clear advantage over AG with more stable and efficient feedbacks.

5.3 Mean Response Time

In this subsection, we study the mean response time of two-level adaptive schedulers AGDEQ and ABGDEQ. We simulate a system with 1000 processors, and generate a wide range of workloads by varying the number of jobs and their parallelism variations. In each experiment, jobs are released according to the Poisson process within the span of the first arrived job so that all jobs would be released before any could complete. Hence, the load of the system will increase with the number of jobs used for each experiment. As with [19], we define the load of the system to be the sum of the average parallelism of all jobs normalized by the total number of processors. In our simulations, the number of jobs is varied from 1 to 100 for each parallelism variation. The mean response time of AGDEQ and ABGDEQ are compared to that of EQUI. In addition, we also study the impacts of quantum length and system overhead on the performances of the two-level adaptive schedulers.

(1) Performance comparison.
As can be observed in Figure 3, AGDEQ and ABGDEQ generally achieve better performances than EQUI on jobs with all parallelism variations. When the system has light workload with a small number of jobs, EQUI performs better because in this case all jobs can be easily satisfied on the given processors. With increased workload, however, both AGDEQ and ABGDEQ outperform EQUI, and eventually tend to converge to EQUI at extremely heavy workloads, where each job gets very few processors most of the time and hence the advantage of adaptive scheduling is diminished. This suggests that two-level adaptive scheduling is more effective

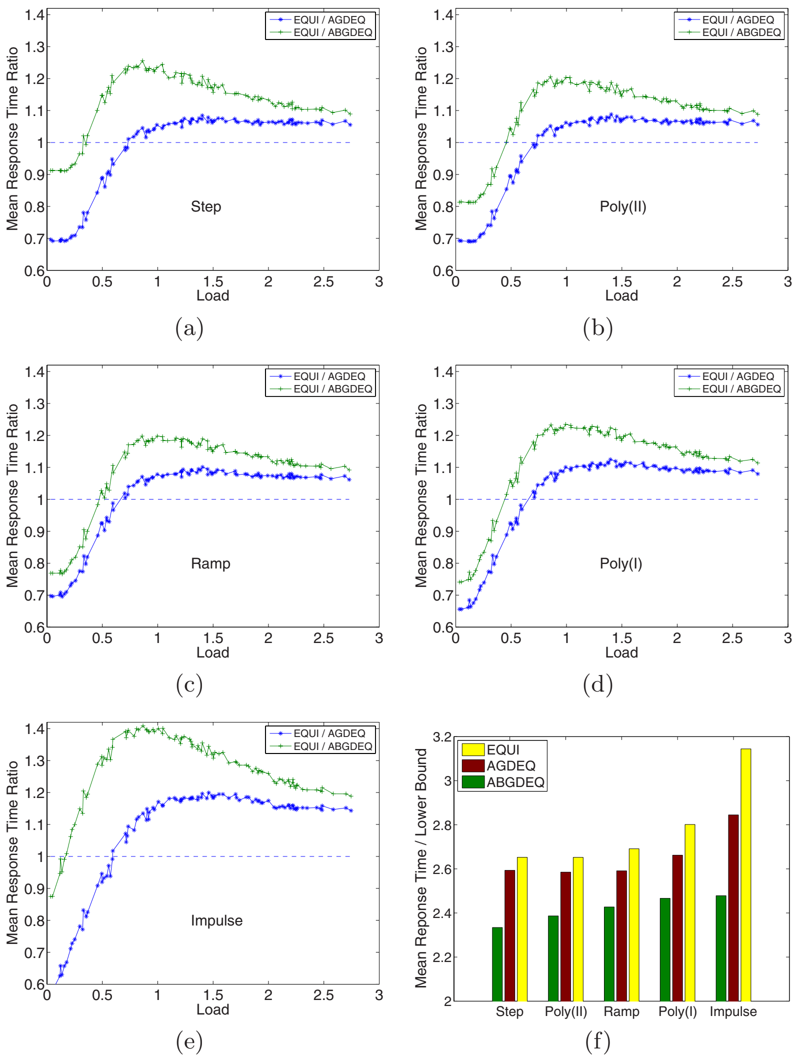


Fig. 3. (a)-(e) Mean response time ratios of Equi over Agdeq and Equi over Abgdeq on different workloads and parallel jobs with Step, Poly(II), Ramp, Poly(I), and Impulse parallelism curves. (f) Average mean response time normalized by the theoretical lower bound for Equi, Agdeq and Abgdeq over the entire workload range on jobs with the five parallelism variations.

under moderate workloads with many parallel jobs competing for but not overwhelmed by the limited processor resources. Moreover, Figure 3 also shows that the performance of Abgdeq is always better than that of Agdeq, which is again due to task scheduler Abg's more effective processor desire feedbacks.

(2) Impact of parallelism variations.
The impact of different parallelism variations on the performances of AGDEQ, ABGDEQ, and EQUI are shown in Figure 3(f), which gives the average performances of the three algorithms over our entire workload range in terms of their mean response time normalized by the theoretical lower bound. Roughly speaking, the performances of all three algorithms are closely related to the degree at which the parallelism varies. Specifically, the Impulse function contains the most drastic parallelism variation and therefore incurs the worst performance for all algorithms. The other functions present better performances for the algorithms with smoother parallelism variations. Furthermore, we can also see that the performance of ABGDEQ is relatively insensitive to different parallelism variations, while EQUI is affected the most as the parallelism variations change from Step to Impulse.

(3) Impacts of quantum length and overhead.
In the preceding simulations, we fixed the quantum length to 100 and ignored the system overhead. However, in two-level adaptive scheduling, the length of the scheduling quantum is an important system parameter, which together with

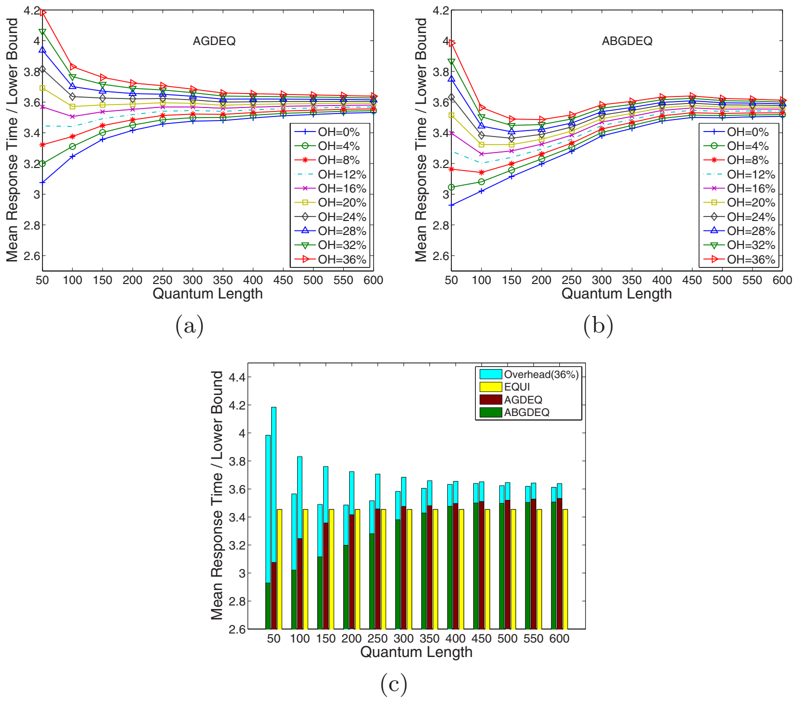


Fig. 4. (a)-(b) Impacts of different quantum length and scheduling overhead on the performances of AGDEQ and ABGDEQ. (c) Performance comparison among EQUI, AGDEQ, and ABGDEQ on medium to heavy workloads with and without overhead (36%) for different quantum length.

the scheduling overhead can significantly affect the mean response time performance. To better understand their impacts, we conduct a set of simulations by changing the quantum length and the scheduling overhead. Specifically, the quantum length is varied from 50 to 600 in steps of 50. The scheduling overhead is changed from zero (i.e., OH= 0%) up to 36% in terms of the smallest quantum length 50 at an increment of 4% each time. Figure 4 shows the simulation results on medium to heavy workloads using jobs whose parallelism variation follows the Step function. Similar outcomes are observed using jobs with the other types of parallelism variations. We can see that as the scheduling overhead increases, both AGDEQ and ABGDEQ have significantly worse performances when the quantum length is small, while the impact of the overhead becomes less severe with larger quantum length. When the quantum length is large enough, the performances of AGDEQ and ABGDEQ become generally stable and are slightly worse than that of EQUI, which is hardly affected by the overhead. However, when the system has relatively small overhead, both adaptive schedulers do outperform EQUI with suitably chosen quantum length. In this sense, two-level adaptive schedulers are quite sensitive to the length of scheduling quantum and the amount of system overhead. Hence, when implementing these algorithms on different platforms, attentions should be paid to choosing an appropriate quantum length based on the scheduling overhead of the system in order to offer desirable performances.

6 Related Work

The problem of scheduling a set of fully parallelizable jobs on multiprocessors is equivalent to scheduling sequential jobs on a single processor. For the latter problem, Motwani et al. [25] showed that, for batched jobs, RR (Round Robin) is $(2-2/(n+1))$-competitive with respect to the mean response time. When jobs can have arbitrary release time, however, they showed that every deterministic non-clairvoyant algorithm is $\Omega(n^{1/3})$-competitive and every randomized non-clairvoyant algorithm is $\Omega(\log n)$-competitive. Using resource augmentation analysis, Kalyanasundaram and Pruhs [21] proved that the deterministic non-clairvoyant algorithm SETF (Shortest Elapsed Time First) is $(1+\epsilon)$-speed $(1+1/\epsilon)$-competitive, which was later improved by Berman and Coulston [7] to $2/s$ when $s \geq 2$. Kalyanasundaram and Pruhs [22] also showed that the randomized non-clairvoyant algorithm RMLF (Randomized Multi-Level Feedback) is $O(\log n \log\log n)$-competitive against an adaptive adversary. Becchetti and Leonardi [6] improved the competitive ratio of RMLF to $O(\log n)$ when the adversary is oblivious, hence matching the lower bound in this case. In addition, it is well-known that the clairvoyant algorithm SRPT (Shortest Remaining Processing Time) is optimal for this problem [10].

For parallel jobs with changing execution characteristics, Edmonds [13] proved that EQUI is $(2+\epsilon)$-speed $O(1)$-competitive with respect to the mean response time of the jobs. Edmonds and Pruhs [16] recently proposed LAPS (Latest Arrival Processor Sharing), which in a sense combines EQUI and SETF, and proved that it is $(1+\epsilon)$-speed $O(1)$-competitive for sufficiently large ϵ, hence achieving *almost fully scalable* [28, 27], i.e., the least possible extra resources required to be competitive. For the relatively easier case, where jobs are released in a batched fashion, Edmonds et al. [14] showed that Equi is $(2+\sqrt{3})$-competitive. Deng et al. [12] showed that Deq with jobs' instantaneous parallelism as feedback is 2-competitive for parallel jobs with single phase and 4-competitive for multiple-phase jobs. The latter ratio was recently improved to 3 by He et al. [20]. In addition, Edmonds et al. [15] also extended the analysis of Equi to the TCP protocol in Internet congestion control. Robert and Schabanel [29, 30] applied variations of Equi to other job models and objective functions.

For two-level adaptive schedulers, modeling a parallel job as a directed acyclic graph (dag), Agrawal et al. [1, 3] proposed two algorithms, namely Ag (Adaptive Greedy) and As (Adaptive Work-Stealing), which are based on centralized scheduling and distributed work stealing, respectively. They proved that Ag and As achieve nearly linear speedup and waste a relatively small number of processor cycles for each individual job. He et al. [18] later combined task schedulers Ag and As with the OS allocator Deq to form two-level schedulers. They proved that the resulting algorithms Agdeq and Asdeq are both $O(1)$-competitive with respect to the mean response time for batched parallel jobs. In addition, He et al. [19] also showed that when the system is heavily-loaded, the two-level algorithms can be coupled with Rr to achieve similar results. Observing that Ag can cause unstable processor desires although the parallelism of the job is constant, Sun and Hsu [32] proposed Abg (Adaptive B-Greedy) task scheduler, which guarantees stability of the processor desires along with other control-theoretic properties. They also proved the mean response time of batched parallel jobs for the two-level scheduler Abgdeq in terms of the jobs' parallelism transition.

Several empirical studies on two-level adaptive scheduling are also known in the literature. Sen [31] presented experimental results on a dynamic desire estimation algorithm for the Cilk work-stealing scheduler [8], which inspired the research presented in [1, 3]. Agrawal, He and Leiserson [2] compared Asdeq with Equi through simulations, and confirmed that the former has superior performance. He, Hsu and Leiserson [19] evaluated the performance Agdeq under a wide range of workloads, and revealed that it actually performs much better in practice than predicted by the theoretical bounds. Sun and Hsu [32], also through simulations, confirmed that Abgdeq does improve upon Agdeq for batched parallel jobs.

7 Conclusion

In this paper, we have analyzed the mean response time of two-level adaptive schedulers Agdeq and Abgdeq on parallel jobs with arbitrary release time and changing degrees of parallelism. We have shown through a general analysis framework that both Agdeq and Abgdeq are competitive with respect to the mean response time using $O(1)$ times faster processors. In addition, we have also conducted simulations over a wide range of workloads using parallel jobs with different parallelism variations. The simulation results have verified the effectiveness of Agdeq and Abgdeq with appropriately chosen quantum length.

Compared to the job model used in this paper, Edmonds et al. [13, 16] have assumed a more general model, in which each phase of a job can admit an arbitrary non-decreasing and sub-linear speedup. However, to analyze the mean response time of EQUI and LAPS, Edmonds et al. reduced any set $\mathcal{J}$ of jobs with non-decreasing and sub-linear speedups to a set $\mathcal{J}'$ of jobs that consist of only fully parallelizable and strongly sequential phases, where a phase is *fully parallelizable* if its speedup function Γ satisfies $\Gamma(a) = a$ for all $a \geq 0$ and it is *strongly sequential* if $\Gamma(a) = 1$ for all $a \geq 0$. One implicit assumption used in this reduction is that the online algorithm is not able to distinguish a newly constructed phase from the original phase because it is non-clairvoyant. Thus, the same number of processors will be allocated to $\mathcal{J}'$ and $\mathcal{J}$ at any time. However, such reduction does not directly apply to the type of adaptive schedulers considered in this paper because their future processor allocations do depend on the past parallelism of the jobs. It will be interesting to see if similar reductions are possible for these adaptive schedulers.

Another problem with existing task schedulers AG and ABG is that they both require comprehensive statistics about a job's execution in the current quantum in order to estimate its processor desire for the next quantum. Collecting such statistics can be difficult in real systems, and even if possible, might incur significant overheads, which as have been shown in our simulations can have adverse effect on the system performance. It will be useful to design task schedulers that use incomplete information about a job's execution (e.g., obtained through samplings) to estimate its processor desires while still guaranteeing desirable performances.

Acknowledgments

The first author would like to thank Yuxiong He for many discussions and much help on two-level adaptive scheduling.

References

[1] Agrawal, K., He, Y., Hsu, W.-J., Leiserson, C.E.: Adaptive scheduling with parallelism feedback. In: PPoPP, New York City, NY, USA, pp. 100–109 (2006)

[2] Agrawal, K., He, Y., Leiserson, C.E.: An empirical evaluation of work stealing with parallelism feedback. In: ICDCS, Lisbon, Portugal, pp. 19–29 (2006)

[3] Agrawal, K., He, Y., Leiserson, C.E.: Adaptive work stealing with parallelism feedback. In: PPoPP, San Jose, CA, USA, pp. 112–120 (2007)

[4] Bansal, N., Chan, H.L., Lam, T.W., Lee, L.K.: Scheduling for speed bounded processors. In: ICALP, Reykjavik, Iceland, pp. 409–420 (2008)

[5] Bansal, N., Pruhs, K., Stein, C.: Speed scaling for weighted flow time. In: SODA, New Orleans, LA, USA, pp. 805–813 (2007)

[6] Becchetti, L., Leonardi, S.: Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. Journal of the ACM 51(4), 517–539 (2004)

[7] Berman, P., Coulston, C.: Speed is more powerful than clairvoyance. Nordic Journal of Computing 6(2), 181–193 (1999)

[8] Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5), 720–748 (1999)
[9] Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, New York (1998)
[10] Brucker, P.: Scheduling Algorithms. Springer, New York (2001)
[11] Chan, H.-L., Edmonds, J., Lam, T.-W., Lee, L.-K., Marchetti-Spaccamela, A., Pruhs, K.: Nonclairvoyant speed scaling for flow and energy. In: STACS, Freiburg, Germany, pp. 409–420 (2009)
[12] Deng, X., Gu, N., Brecht, T., Lu, K.: Preemptive scheduling of parallel jobs on multiprocessors. In: SODA, Philadelphia, PA, USA, pp. 159–167 (1996)
[13] Edmonds, J.: Scheduling in the dark. In: STOC, Atlanta, GA, USA, pp. 179–188 (1999)
[14] Edmonds, J., Chinn, D.D., Brecht, T., Deng, X.: Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. In: STOC, El Paso, TX, USA, pp. 120–129 (1997)
[15] Edmonds, J., Datta, S., Dymond, P.: TCP is competitive against a limited adversary. In: SPAA, San Diego, CA, USA, pp. 174–183 (2003)
[16] Edmonds, J., Pruhs, K.: Scalably scheduling processes with arbitrary speedup curves. In: SODA, New York, NY, USA, pp. 685–692 (2009)
[17] Feitelson, D.G.: Job scheduling in multiprogrammed parallel systems (extended version). IBM Research Report RC19790 (87657) 2nd Revision (1997)
[18] He, Y., Hsu, W.-J., Leiserson, C.E.: Provably efficient two-level adaptive scheduling. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2006. LNCS, vol. 4376, pp. 1–32. Springer, Heidelberg (2007)
[19] He, Y., Hsu, W.-J., Leiserson, C.E.: Provably efficient online non-clairvoyant adaptive scheduling. In: IPDPS, Long Beach, CA, USA, pp. 1–10 (2007)
[20] He, Y., Sun, H., Hsu, W.-J.: Adaptive scheduling of parallel jobs on functionally heterogeneous resources. In: ICPP, Xi'an, China, p. 43 (2007)
[21] Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. In: FOCS, Milwaukee, WI, USA, pp. 214–221 (1995)
[22] Kalyanasundaram, B., Pruhs, K.: Minimizing flow time nonclairvoyantly. In: FOCS, Miami Beach, FL, USA, p. 345 (1997)
[23] Lam, T.W., Lee, L.-K., To, I.K.-K., Wong, P.W.H.: Speed scaling functions for flow time scheduling based on active job count. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 647–659. Springer, Heidelberg (2008)
[24] McCann, C., Vaswani, R., Zahorjan, J.: A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. ACM Transactions on Computer Systems 11(2), 146–178 (1993)
[25] Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. In: SODA, Austin, TX, USA, pp. 422–431 (1993)
[26] Phillips, C.A., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation (extended abstract). In: STOC, El Paso, TX, USA, pp. 140–149 (1997)
[27] Pruhs, K.: Competitive online scheduling for server systems. ACM SIGMETRICS Performance Evaluation Review 34(4), 52–58 (2007)
[28] Pruhs, K., Torong, E., Sgall, J.: Online scheduling. In: Handbook of scheduling: Algorithms, models, and performance analysis, ch. 15, CRC Press, Boca Raton (2004)
[29] Robert, J., Schabanel, N.: Non-clairvoyant batch set scheduling: Fairness is fair enough. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 741–753. Springer, Heidelberg (2007)

[30] Robert, J., Schabanel, N.: Non-clairvoyant scheduling with precedence constraints. In: SODA, San Francisco, CA, USA, pp. 491–500 (2008)
[31] Sen, S.: Dynamic processor allocation for adaptively parallel jobs. Master's thesis, Massachusetts Institute of technology (2004)
[32] Sun, H., Hsu, W.-J.: Adaptive B-Greedy (ABG): A simple yet efficient scheduling algorithm. In: SMTPS in conjunction with IPDPS, Miami, FL, USA, pp. 1–8 (2008)
[33] Tucker, A., Gupta, A.: Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In: SOSP, New York, NY, USA, pp. 159–166 (1989)

Job Scheduling with Lookahead Group Matchmaking for Time/Space Sharing on Multi-core Parallel Machines

Xijie Zeng and Angela C. Sodan

University of Windsor, Windsor ON N9B 3P4, Canada
zengx@uwindsor.ca, acsodan@uwindsor.ca

Abstract. Multi-core nodes of parallel machines may only provide gradual performance improvement per application due to competition on resources like the cache. As shown in our earlier work, spreading out applications over as many nodes as possible or letting different applications with potentially complementary characteristics (semi time) share each node by allocating different cores to them may provide better performance. In the latter case, groups of jobs may be necessary to obtain balanced resource utilization due to different sizes of jobs. We present a scheduler G-LOMARC-TS which can match groups of jobs and consider both space- and time-sharing allocation. Since matchmaking may select jobs further down in the waiting queue, fairness in regards to possible delays of the other jobs is watched and delays are kept within certain bounds. This results in a large number of possible combinations. A number of heuristics to select the most promising combinations make it possible to deal with the NP-completeness of the problem. We show that our scheduler improves utilization of high-load phases by about 27% and subsequently average response times by about 36% (and 53% for long jobs) compared to space sharing scheduling for normal workloads. Additionally the scheduler can handle much higher workloads than a space-sharing scheduler.

Keywords: space sharing, semi time sharing, lookahead matchmaking, job groups.

1 Introduction

Multi-core nodes in cluster are becoming widespread though the additional cores may only provide gradual performance improvement due to competition on shared resources such as memory, network, and potentially caches. In earlier work, we have shown that better results may be obtained if rather using the additional cores for other applications with complementary characteristics [15][17][21], as also found by other researchers [19]. We call such resource allocation *semi time sharing* since the cores are partitioned among the applications, i.e. space-shared, but the other resources may be shared.

Our LOMARC scheduler [17] first proposed such semi time sharing on nodes with hyperthreaded CPUs, assuming that the second virtual CPU per CPU/node would either be allocated to a second application or remain unused. We proved that many combinations of the NAS benchmarks ran very well together. Under the conditions mentioned, LOMARC improved average response times by 30% to 50%. In other work, we showed that even if the choice is between using fewer nodes exclusively (space sharing) vs. more nodes coscheduled with another application (semi time sharing), semi time sharing may perform better [16]. We also found that pairing communication-bound applications may yield acceptable slowdown [16][21]. While our work focused on computation and communication performance, the study in [19] focused on memory and I/O, confirming the benefits from coscheduling. However, there are also cases where an application can exploit the performance potential of all cores per node well and space sharing may perform better.

To use the cores per node most effectively, more general schedulers are needed that can allocate the cores intelligently for high resource utilization. Similar to adaptive approaches which reduce job sizes under high load [14], we can expect that higher resource utilization decreases average response times in spite of increasing individual runtimes, because load and subsequently average wait times are significantly decreased.

The LOMARC scheduler [17] applied lookahead matchmaking among waiting and running jobs to find jobs with complementary usage of multiple resources, using simple heuristics to select the best combination. However, no fairness considerations were applied in regards to the impacts of the partial reordering of the waiting queue.

Our main goals for the work presented in this paper are to

– Provide a more flexible approach for nodes with multiple cores and for applications which may either exploit multiple cores per node exclusively or share them with another application. We can consider this approach as semi adaptive by assuming a fixed number of processes that can be allocated differently to nodes and cores.
– Provide a framework which, in spite of partial reordering of the waiting queue that is necessary to find suitable matches, is fair to individual jobs and does not impose overly high delays on individual jobs.
– Match groups of jobs since jobs typically have very different job sizes and only matching two jobs may therefore not provide sufficient potential for utilization gain.

The main contributions of this paper and our new G-LOMARC-TS scheduler are:

– Matching groups of jobs among waiting and/or running jobs.
– Applying several heuristics that are likely to extract the most promising groups since finding the optimum group among the many possibilities is an NP-complete problem.
– Including important special cases like bursts of serial jobs.
– Using a metric for selection of the best group which considers the absolute gain in utilization (nodes saved over a certain time interval) and subsequently the global interest of all jobs.

– Supporting both space and time sharing and choosing between them according to the utilization gain and current machine load.
– Providing fairness to individual jobs by using a maximum slack factor vs. the originally estimated response time as a constraint for the reordering.
– Providing more chances to fit a job into the machine by including the options of 1) matching it with running jobs and 2) reducing its node requirement via space sharing.

G-LOMARC-TS is implemented as an extension of the coarse-grain preemption scheduler Scojo-PECT [5]. We demonstrate via simulation with synthetic workloads that G-LOMARC-TS performs significantly better than pure space sharing and that group matching contributes significantly to the improvements and is therefore essential.

The paper discusses related work in Section 2. Section 3 defines the machine and application model. Our G-LOMARC-TS algorithm is described in Section 4, including utilization-gain and slowdown metrics. Experimental results are shown in Section 5, and Section 6 gives a summary and conclusion.

2 Related Work

Existing approaches to time sharing for parallel jobs are gang scheduling [5] and loosely coordinated coscheduling [14]. Gang scheduling allocates globally synchronized time slices, while keeping all jobs in memory to make slice switches fast. Loosely coordinated coscheduling uses distributed algorithms to approximate coordinated execution which is necessary to avoid idling in communication. Though gang scheduling may better pack jobs into the machine, it does not improve utilization of the individual resources such as network or disk. Loosely coordinated coscheduling has the potential of improving utilization by switching between jobs to hide resource-access latencies. However, jobs need to be fairly synchronous or very coarse-grain to make coordinated execution possible or unimportant, respectively. Proposals were made to relax gang scheduling and merge time slices with computation-bound and I/O-bound jobs [20] or to switch from gang scheduling to loosely coordinated coscheduling for coarse-grain or I/O-bound jobs [1]. Both approaches depend on the dynamic availability of suitable job combinations but can adapt to different phases in the program execution. However, the probability of finding suitable dynamic job combinations decreases if groups of jobs need to be formed.

In regards to semi time sharing, studies found complementary characteristics of the coscheduled jobs to perform better due to balancing the usage of system resources [17][19]. Spreading out jobs to different nodes and semi time sharing the resources per node among multiple jobs may perform better than dedicated allocation of all node resources to one job [15][16][19]. The experiments in [19] were carried out with only 4 nodes and therefore limited communication but the experiments in [15][16] on up to 64 nodes show that the benefits of semi time sharing vs. dedicated allocation scale to larger number of nodes. Characteristics which were found important for job combinations are whether the job is CPU-bound, cache-bound, memory-bound, network-bound, or disk-bound [17][19] but also which access patterns are applied [11][21]. For example, CPU-bound jobs were shown to match well with network-bound or memory-bound jobs. Several studies on hyperthreaded and multi-core CPUs showed that scheduling multiple processes of the same job per node provides limited benefits or does not work well in certain cases. Thus, in [2], jobs became only between 1.2 and 1.5 times faster by running another process on a second AMD-Opteron core. Potentially high resource contention on hyperthreaded CPUs from processes of the same application with same resource requirement (such as the cache) were shown before in [8][9]. In other cases, resource sharing per node can provide a benefit rather than performance degradation: intra-node communication through shared memory and cache may be faster than inter-node communication which is a benefit if a large percentage of messages are transferred via intra-node communication [4] (50% of the messages were found to be intra-node).

Thus, the LOMARC scheduler [17] matches jobs with complementary characteristics at job start times, while partially reordering the waiting queue to find suitable matches. The work in [19] proposes a scheduler which space-partitions each node into one half for memory-bound and one half for the other jobs.

Fairness is discussed in several papers. The work in [12] measures overall fairness (in retro) of a job scheduler by considering the actual start time of a job vs. its virtual start time without effects from later arriving jobs. The slack approach in [18] tries to maintain relative fairness among jobs by dynamically calculating possible delays in the presence of different job priorities.

3 Machine and Application Model

We assume that the target machine is a cluster with multiple multi-core CPUs per node. Though multiple CPUs and multiple cores per node can significantly increase performance, they do not simply multiply the performance by the number of CPUs/cores due to the contention effects on shared resources, but rather typically provide less performance gain than additional nodes. Processes running on the same node share the network, disk, and the memory. Processes running on the same CPU additionally share the memory access paths and potentially the cache. In regards to the cache, some multi-core CPUs share the L2 cache (such as the Intel Core Duo, IBM POWER5, and Ultra SPARC T1/T2), whereas other multi-core CPUs have private L2 caches per core (AMD Opteron, Intel Itanium, IBM POWER6, and Ultra SPARC IV). We model the contention effects as application slowdown (Section 4.9), and differentiate between CPU and core slowdowns, with CPU slowdowns typically being lower. This also implies that the allocation to CPUs and cores matters if fewer processes run per node than there are CPUs/cores available. In the few cases where resource sharing among processes of the same application provides a benefit, the slowdown would turn into a speedup.

We assume that jobs and workload have the following characteristics:

- Jobs consist exclusively of processes (no threads) which is still the dominant approach applied by users [2].
- Upon job submission, the number of processes (the job size) is specified but the allocation to nodes, CPUs, and cores is left to the scheduler. The number of processes is therefore fixed (no molding in the sense of changing the job size).
- Runtime estimation and sufficient characteristics information to calculate slowdowns are available, provided by users, historical databases, or compilers. General research progress made by compilers, application profilers, and by prediction from historical information suggests that roughly correct runtime estimation by the system is becoming realistic for future schedulers (see e.g. [3]). More optimistic is the assumption about slowdown estimations being available which is farer in the future. This assumption helps to explore possible benefits obtainable from such information and its exploitation in advanced space/time-sharing job schedulers.
- The workload includes a large percentage of serial jobs and of parallel jobs with power-of-two sizes, as observed in the analysis of job traces [10]. Also bursts of submissions (submission of jobs with potentially similar characteristics in close time proximity) are possible.

4 G-LOMARC-TS Scheduling Algorithm

4.1 Scheduling Objectives

Terms used in the following discussion of formulas are listed and explained in Table 1.

The objective of the scheduler is to obtain best possible utilization in phases of high load, while keeping fairness acceptable. Higher utilization in high-load

Table 1. Terms used throughout formulas in paper

Term	Meaning
S_i	Size (all processes) of Job i
T_i	Runtime of Job i if scheduled individually with 1 process per node
PPN_i	Number of processes of Job i per node
$T_{makespan}$	Total runtime of workload
M	Machine size in number of nodes
N_{core}	Number of cores per node
U_{core}	Core utilization
U_{node}	Node utlization
U_{gain}	Utilization gain for comparison of node usage
$R_{est,i}$	Estimated response time of Job i in FCFS order at submission time
R_i	Response time of Job i
F_{slack}	Slack factor used for fairness check

phases likely leads to better average response times [13]. Under utilization, we understand, as the basic metric, core utilization which relates the number of cores used to a certain time span. The other possible metric for utilization, namely node utilization, considers the number of nodes used. Note, that overall/average core utilization remains equal to the submitted load, independent of the scheduling policy, as long as the scheduler is not saturated (i.e. can handle the offered load and jobs do not queue up). Schedulers with little utilization support would delay jobs until phases with lower load, whereas the machine may be idle or very lowly loaded in such phases for schedulers with high utilization support. However, nodes are saved if jobs are better packed onto the cores per node, i.e. if cores are better utilized in high-load phases. This subsequently reduces node utilization.

Thus, we make improvement in core utilization in phases of high load the primary objective and fairness a constraint. This requires to formally define high-load phases, core and node utilization, utilization gain, fairness, and the scheduler impact on fairness–which we do in the following.

To formalize utilization, we assume that N_{wait} is the number of jobs in the waiting queue, N_H is a threshold for rating as high load, tp_i is a time period between load changes (due to job termination, job start, and slice switches), and t_l, t_j, and t_k are certain time points of load changes, un_i and uc_i are the number of used nodes and used cores, respectively, during the time period tp_i. High-load phases $Phase_H$ are defined as:

$$\begin{aligned} Phase_{H,l} = [&t_l \text{ with } N_{wait} \geq N_H \ \&\& \ N_{wait} < N_H \text{ for } t_{l-1}, t_j \text{ with } N_{wait} < N_H \\ &\&\& \text{ any } t_k \text{between l and} j \text{ has } N_{wait} \geq N_H] \end{aligned} \tag{1}$$

Node utilization U_{node}, is the percentage of used-nodes time over the makespan.

$$U_{node} = \sum_{i \text{ in time periods}} (tp_i * un_i)/(T_{makespan} * M) \tag{2}$$

Core utilization U_{core}, is the percentage of used-cores time over the makespan.

$$U_{core} = \sum_{i \text{ in time periods}} (tp_i * uc_i)/(T_{makespan} * M * N_{core}) \tag{3}$$

Then, U_{core} during a high-load phase is the percentage of used-cores time over this phase, and U_{node} during a high-load phase is the percentage of used-nodes time over this phase.

Utilization gain U_{gain} (for details, see Section 4.6) is measured as the fraction of saved usage of nodes and used as the metric per individual scheduling decision. As discussed above, saving nodes corresponds to higher core utilization.

In regards to fairness, there exist different definitions of fairness in the literature. We consider any delay versus the response time without reordering and coscheduling as a negative impact on fairness. Since our scheduling is basically FCFS with conservative backfilling, estimation of response times via simulation at submission time is possible, and we record the estimated response

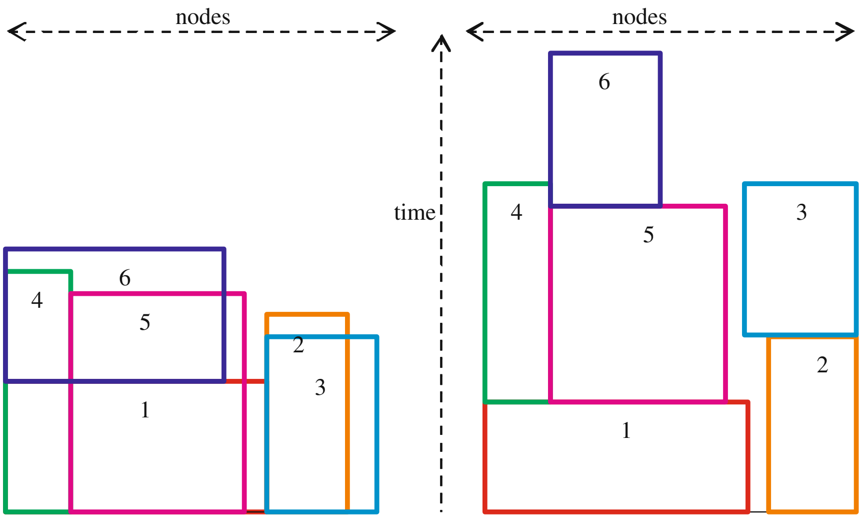


Fig. 1. Utilization gain with G-LOMARC-TS from increased core utilization for Job 1 to Job 6 (left) compared to space sharing (right)

times ($R_{est,i}$). Fairness is then provided by limiting the response-time changes to $R_{max,i}$ (response time can increase due to the matchmaking reordering), calculated via slack factor (F_{slack}):

$$R_{max,i} = F_{slack} * R_{est,i} \tag{4}$$

Finally, we define our objective as maximizing core utilization U_{core} during high-load phases $Phase_H$, while maintaining $R_i \leq R_{max,i}$. The optimization is approximated by heuristics, making per-job-group decisions, and calculating utilization gain per decision.

Figure 1 shows how high core utilization leads to saved nodes usage if 6 jobs are scheduled by our G-LOMARC-TS compared to space sharing (note that for Job 6, number of processes per job per node (PPN) is 2 with G-LOMARC-TS and 4 with space sharing; i.e, the node requirements are different under the two scheduling schemes).

4.2 General G-LOMARC-TS Scheduling Idea

Our G-LOMARC-TS supports both space sharing and semi time sharing which we define more precisely as follows:

– *Space sharing*: Resources are allocated in a dedicated manner, but the machine (its "space") is shared, i.e., different parallel jobs may potentially run at the same time if resource requirements permit them to be allocated to different subsets of compute nodes.

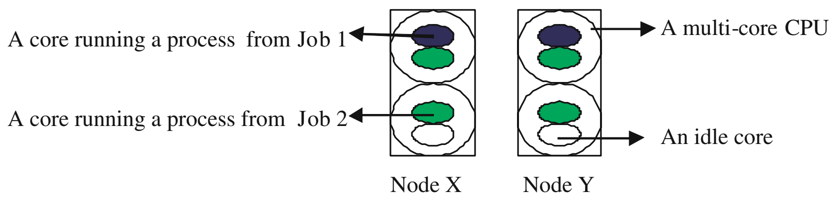

A core running a process from Job 1
A core running a process from Job 2
A multi-core CPU
An idle core
Node X
Node Y

Fig. 2. Coscheduling processes of different parallel jobs per node

– *Semi time sharing*: The CPUs/cores per SMP node of a cluster are allocated to different jobs as illustrated in Figure 2. Semi time sharing does not share and switch the core as done under standard time sharing. However, other resources like memory, network, disk and potentially caches (Section 3) are simultaneously shared. Since all cores remain responsive to communication, no coordination of parallel processes is required as necessary under standard time sharing.

Processes of the same job have similar characteristics, contention on certain resources is very likely for these processes if they share resources on the same node. However, different jobs may have different characteristics and complementary resource usage, leading to less contention. Thus, our G-LOMARC-TS scheduler supports the option of coscheduling, i.e. scheduling processes from different applications on the same nodes, see Figure 2. To make such coscheduling effective, job combinations with high complementary resource usage should be created. This does typically not apply if only pairing the first 2 jobs in the waiting queue. Rather the scheduler needs to search among waiting and running jobs for suitable matches. This means lookahead in the waiting queue vs. FCFS order. If jobs move ahead by being coscheduled with the first job in the waiting queue, typically the runtimes increase (due to contention). This delays the first and other jobs originally in front of the coscheduled ones in the queue. Runtimes also increase for running jobs if waiting jobs are coscheduled with them. Thus, the matchmaking leads to partial reordering of the waiting queue and potential delays for running and waiting jobs. This is the reason why we need to include the fairness criterion as described in Section 4.1 to avoid severe delays or push-backs for individual jobs.

However, in some cases the processes of the same job may run well together and may even benefit from intra-node communication (Section 3). In such cases, space sharing is the better option. Thus, our scheduler supports both coscheduling and individual scheduling.

In regards to coscheduling, the simplest approach is pairing two jobs as applied in the original LOMARC scheduler. However, sizes of jobs can be very different. Therefore grouping multiple jobs for coscheduling can increase the chance for utilization gain. We have the following cases of forming groups: 1) matching a waiting job with several other waiting jobs, 2) matching a waiting job with several running jobs, and 3) matching a running job with several waiting jobs.

4.3 Time vs. Space Scheduling

In the following, we explain the space sharing and semi time sharing options of G-LOMARC-TS in more detail. Space and semi time sharing can involve different numbers of processes of the same job per node if we have more than 2 cores per node. For example, with 4 cores per node, we can have 1 or 2 processes per job per node with semi time sharing, and we can have 1, 2, or 4 processes per node per job with space sharing. Which number is chosen depends on the self slowdown caused by resource contention of processes of the same job (details are discussed in Section 4.9). If the self slowdown is severe, fewer processes per node per job are meaningful. If the self slowdown is low or if there is even speedup, more processes per node per job are better. For space sharing, the number of processes per node is always chosen to utilize the space well, balancing runtime with used cores by employing a threshold on acceptable self slowdown.

In our scheduler, short jobs are only scheduled via space sharing because short jobs are not considered worth the effort of matchmaking. Otherwise, decisions between space and semi time sharing are made adaptively when trying to schedule the first job in the waiting queue. If the workload is low in the sense that all jobs fit into the machine with one process per node, space sharing is more beneficial because obtaining the best runtimes per job. A special case is that there may not be enough space to schedule a job under space sharing but it may be possible to start the job by coscheduling it with running jobs. Otherwise, whether space or semi time sharing is applied depends on which option provides better utilization gain for the current scheduling decision.

If a job's self slowdown is low, it may benefit more with space sharing where only self slowdown involves. However, if a job can find matched jobs with complementary resource requirements and consequently with low coscheduling slowdown caused by processes of different jobs (described in Section 4.9), it may gain more with semi time sharing.

4.4 Scheduling Algorithm

The main part of the G-LOMARC-TS scheduling algorithm is shown in Figure 3. The algorithm description is generalized to work with any number of cores per node (though our evaluation uses 4 cores per node). The scheduler tries different scheduling possibilities: space sharing and semi time sharing matching the first waiting job with other waiting jobs or running jobs. Space sharing is used if the load is low. Serial jobs are treated specially as described in Section 4.5, before attempting other forms of coscheduling. Then, the 3 forms of semi time sharing (Case 1 to Case 3) and space sharing with full usage of all cores per node (Case 5) are compared. The option with the highest utilization gain is selected. If all of the latter options fail, space sharing with partial usage of the cores per node (Case 4) is applied.

For coscheduling, groups are formed. A *group* is composed of a *primary job* and one or multiple *matched jobs*. The primary job can be the first waiting job or a running job. (Details about forming groups are described in 4.5.)

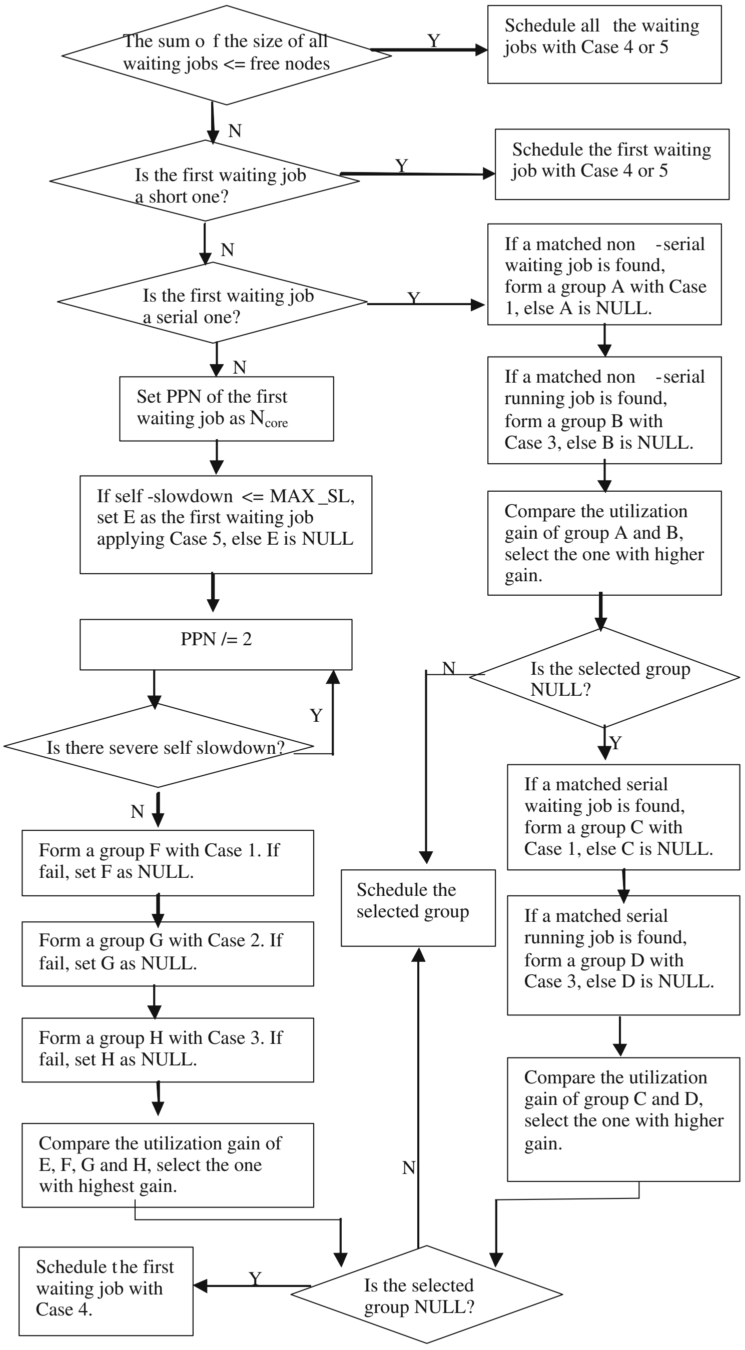

The sum of the size of all waiting jobs <= free nodes
Y
Schedule all the waiting jobs with Case 4 or 5
N
Is the first waiting job a short one?
Y
Schedule the first waiting job with Case 4 or 5
N
Is the first waiting job a serial one?
Y
If a matched non-serial waiting job is found, form a group A with Case 1, else A is NULL.
If a matched non-serial running job is found, form a group B with Case 3, else B is NULL.
Compare the utilization gain of group A and B, select the one with higher gain.
Is the selected group NULL?
N
Y
If a matched serial waiting job is found, form a group C with Case 1, else C is NULL.
If a matched serial running job is found, form a group D with Case 3, else D is NULL.
Compare the utilization gain of group C and D, select the one with higher gain.
N
Set PPN of the first waiting job as N_core
If self-slowdown <= MAX_SL, set E as the first waiting job applying Case 5, else E is NULL
PPN /= 2
Is there severe self slowdown?
Y
N
Form a group F with Case 1. If fail, set F as NULL.
Form a group G with Case 2. If fail, set G as NULL.
Form a group H with Case 3. If fail, set H as NULL.
Compare the utilization gain of E, F, G and H, select the one with highest gain.
Schedule the selected group
N
Is the selected group NULL?
Y
Schedule the first waiting job with Case 4.

Fig. 3. Flow chart for main part of scheduling algorithm

Then, the first waiting job can be scheduled/coscheduled in the following ways:

- Case 1 (semi time sharing): coscheduled in a group with the first waiting job as the primary job and several other waiting jobs as matched jobs (one waiting : multiple waiting).
- Case 2 (semi time sharing): coscheduled in a group with the first waiting job as the primary job and several running jobs as matched jobs (one waiting : multiple running).
- Case 3 (semi time sharing): coscheduled in a group with a running job as the primary job and several waiting jobs including the first one as matched jobs (one running : multiple waiting).
- Case 4 (space sharing): scheduled individually with $PPN = 1$ or 2 or 4 ... or $M - 2$
- Case 5 (space sharing): scheduled individually with $PPN = M$.

A similar algorithm is applied when attempting to backfill jobs. However, only Case 1, Case 4, and Case 5 are applied.

Note that groups of jobs are scheduled as a whole and adding individual jobs later is not considered. Nevertheless, a job may be matched more than once over its runtime since the group may be disbanded and the job become an individual job again. Disbandment of a group happens under the following conditions:

- The primary job terminates, while at least one matched job is still running.
- All matched jobs terminate, while the primary job is still running.

This also means that the scheduler makes no attempt to add jobs to a running group if some of the matched jobs terminate.

4.5 Group Formation

Forming groups is an NP-complete problem due to the many possibilities to combine jobs with different runtimes and sizes and due to slowdowns depending on the job combination. To make the problem tractable, we apply intelligent heuristics to form groups.

As mentioned above, a group is composed of a primary job and one or multiple matched jobs which may be waiting or running jobs. Per node, only two jobs are coscheduled (one is the primary job, the other one is one of the matched jobs). If the primary job is a running job, we choose of the least delayed jobs (since coscheduling implies slowdown).

After the primary job has been decided, the matched jobs are selected with the following steps (if the primary is the first waiting job, the matched jobs are running or other waiting jobs; if the primary is a running job, the matched jobs are waiting jobs):

1. Pre-selection: If a job and the primary job do not slow down each other severely (less than a threshold), the job is selected as a candidate for matched jobs.

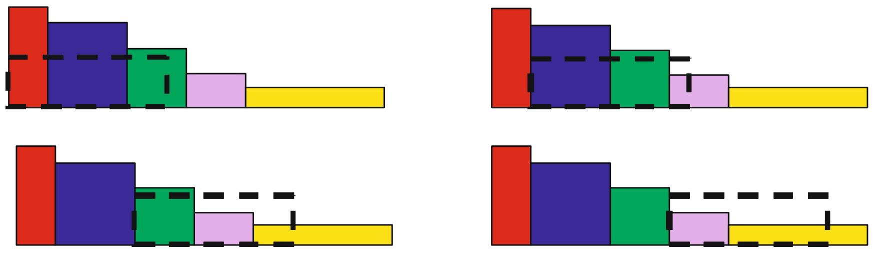


Fig. 4. "Window" with shape of primary job sliding over first block to search for a set of matched jobs

2. Sorting: Candidate jobs are sorted in increasing order of delay if they are running jobs; if they are waiting jobs, their original FCFS order is kept. Then the sorted candidate jobs are divided into blocks, and the jobs per block are sorted in decreasing order of their remaining runtime.[1]
3. First block: A "window" with the same node requirements as the primary job "slides" over the first block (see Figure 4). Each time, the set of jobs within the window's range is selected as matched jobs (though we permit the aggregated node requirements of the matched jobs to be slightly larger than the window). The set with the highest utilization gain achieved is selected as the best group. Likely, most of the matched jobs in the best group are from the first block if including the fairness constraint.
4. Other blocks: If the primary job is not coscheduled over all its nodes (there is still space left) in the best group, jobs from the remaining blocks may be added if this leads to an increase in utilization gain. Each new group which increases the utilization gain is stored.
5. Purify: Matched jobs which slow down the primary job most but do not contribute to the utilization gain are removed from the group.
6. Fairness check: A group which causes any other job to be delayed severely (more than R_{max}) is discarded. Groups kept in Step 4 are tested for fairness in decreasing order of utilization gain until a group passes the check.

4.6 Utilization-Gain Calculation

If the cores per node are better utilized (more processes running per node, fewer idle cores), fewer nodes will be used to run a specific number of jobs. Thus, as discussed in Section 4.1, this means that high core utilization leads to saved node usage. Utilization gain is calculated as the ratio of saved nodes to used nodes which we first explain on the basis of the example shown in Figure 5. Figure 5 (left) shows the resource requirements of Job 1, 2 and 3 when no coscheduling scheme is applied, i.e. the runtime does not have any slowdown. Figure 5 (right)

[1] The order could also be chosen as least delayed vs. their estimated response time but experimental results show virtually no difference.

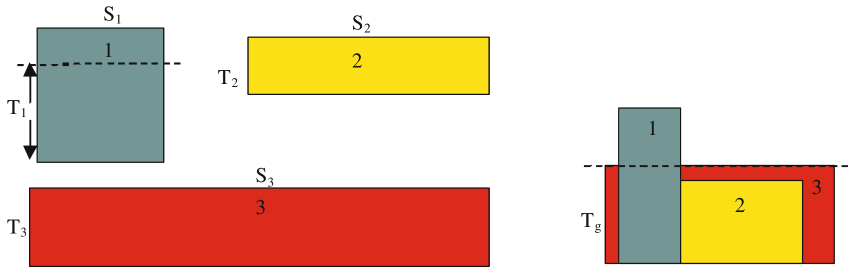


Fig. 5. Node usage of Job 1 to Job 3 with space sharing (left) and semi time sharing, i.e. coscheduled (right)

shows a group formed by these three jobs with Job 3 as the primary job and Job 1 and 2 as matched jobs. In this group, each job has two processes per node, i.e., the node requirement is half of its process number and the runtime is extended by the slowdown. After T_g time, Job 3 in the group finishes running and the group is consequently disbanded. All the work of Job 2 and 3 is done during the group execution and part of the work of Job 1 is done. Let us assume that, in order to do the same amount of work, Job 1 has to spend T_1 time without any coscheduling scheme, Job 2 spends T_2, and Job 3 spends T_3. We compare the node usage of the group to the sum of the node usages of the three jobs without coscheduling and calculate the utilization gain U_{gain} of the group as

$$\begin{aligned} U_{gain} &= (T_1 * S_1 * N_{cores} + T_2 * S_2 * N_{cores} + T_3 * S_3 * N_{cores} - \\ &\quad T_g * S_3 * N_{cores}/2) \ / \ (T_g * S_3 * N_{cores}/2) \\ &= (T_1 * S_1 + T_2 * S_2 + T_3 * S_3 - T_g * S_3/2)/(T_g * S_3/2) \end{aligned} \tag{5}$$

To generalize the calculation for all groups, we assume that a primary Job A coschedules with N matched jobs (Job 1, 2, 3, ... Job n). Suppose that $SL_{A,i}$ is the slowdown of Job A when A coschedules with Job i; $SL_{i,A}$ is the slowdown of Job i when it coschedules with A. There are Q time periods t_q $(1 \leq q \leq Q)$ during the coscheduling of the jobs between changes in the coscheduling status (jobs terminating). $Q \leq N$ because the group is disbanded if all matched jobs or the primary job have terminated. To make the formula easier to understand, and without loss any generalization, we assume that the sum of the node requirements of all matched jobs is less than or equal to that of the primary job (the real algorithm permits that the node requirement of the matched jobs is slightly greater than the primary job).

In each time period q, there are F_q matched jobs running and the indexes of the matched jobs are x_1, x_2, x_3, ..., x_{F_q}. SL_q $(1 \leq q \leq Q)$ is the maximum slowdown of Job A when A coschedules with the matched F_q jobs running in time period q. Then, $SL_q = max\{SL_{A,x_1}, SL_{A,x_2}, \ldots, SL_{A,x_{F_q}}\}$ and the sum of all time periods t_{total} is $t_{total} = \sum_{1 \leq q \leq Q} t_q$.

Then, the general formula for calculating the utilization gain is:

$$U_{gain} = (\sum_{1 \leq q \leq Q} (\sum_{i\ in [x_1, x_2, \ldots, x_{F_q}]} (t_q * S_i / SL_{i,A}) + t_q * S_A / SL_q) - \\ t_{total} * S_A / PPN_A) \ / \ (t_{total} * S_A / PPN_A) \quad (6)$$

Note that utilization gain is defined as a relative gain. This means that matching shorter jobs is given preference if other parameters are the same. Experimental results show that this gives a minor benefit vs. using absolute gain though it does not make any substantial difference.

4.7 Incorporation into Scojo-PECT

Scojo-PECT [5] employs preemption to support scheduling of shorter jobs even in the presence of long-running. Scojo-PECT preempts jobs to swap space which is easy to support in the machine environment and avoids the memory pressure which gang scheduling imposes. Scojo-PECT does not require hard-to-support checkpointing but subsequently imposes the constraint that preempted jobs are later restarted on the same resources as migration is not possible without checkpointing [14]. To make preemption to disk affordable and avoid that jobs are delayed because of problems to get access to their resources again, Scojo-PECT employs coarse-grain time slices and preempts all jobs. Jobs are sorted according to job type based on their runtime (we currently support short, medium, and long jobs), and scheduled in different virtual machines with a time slice per job type / virtual machine. The slice time for each job type is determined on the basis of typical job-type mixes and the administrator's policies and can be recalculated in regular time intervals. One slice for each type is scheduled per time interval (since short jobs backfill into other slices in most cases, their slice is only scheduled if short jobs are waiting), and the slice times can be decided at the beginning of each interval. This permits controlling the resource allocation via different policies at different times of the day or via adaptive allocation which considers the current load of the machine [13]. In the context of this paper, the relative slice times are kept static.

Jobs per job type are scheduled in FCFS. Additionally, the typical backfilling is applied. Backfilling means that jobs can move ahead in the queue if they do not delay other jobs as specified by the backfilling approach. Scojo-PECT can either use EASY or conservative backfilling. In the presented work, we use conservative backfilling which requires that none of the jobs in the queue are delayed.

Since the separation of jobs into different types is likely to increase the fragmentation because job sizes and job runtimes tend to be correlated, Scojo-PECT employs additionally safe non-type slice backfilling. This means that preempted or waiting jobs of a different type may be backfilled into a slice-with this backfilling only being valid until the end of the slice-if they do not delay any job of the slice type or of their own type according to the backfilling approach applied.

If setting time slices (resource shares) for equal service, Scojo-PECT provides similar service to medium and long jobs as standard space sharing with priorities but improves overall response times by about 50% by serving short jobs better [5].

The basic framework remains to be Scojo-PECT, and G-LOMARC-TS is applied per virtual machine. Only jobs of the same type are matched, though non-type slice backfilling is still applied. Thus, G-LOMARC-TS schedules jobs per virtual machine and does not even need to know about the existence of time slices. Groups are preempted and resumed as well as non-type slice backfilled like individual jobs. By representing this at job level, groups remain transparent to Scojo-PECT. FCFS is kept as basic scheduling order per virtual machine with constrained reordering as discussed above. The FCFS scheduling order and the compressed (keeping backfilled jobs in their backfill position even if jobs terminate earlier than estimated) conservative backfilling permit estimation of response times via simulation. As explained in Section 4.1, the estimated response times and the slack factor define the constraints.

4.8 Basic Job Creation

For the evaluation of our scheduler, we use the Lublin-Feitelson statistical workload model [8] which is the best available synthetic workload model (it includes power-of-two sizes, sequential jobs, correlations between runtimes and sizes, and varying inter-arrival times at different times of the day). Though the Lublin-Feitelson workload model was derived from statistical evaluation of 3 real-life workload traces, it generalizes the workload generation to the point that different machine sizes can be chosen. However, the Lublin-Feitelson model assumes that applications are run with 1 process per compute node though we need to model a hierarchical structure with multiple cores and subsequently the possibility of multiple processes per node. Thus, using the number of nodes as machine size would create a load which is too low. Multiplying the number of nodes by the number of cores per node would create a machine load which is too high because the additional cores add less performance gain than independent nodes [2]. Our goal is to create a workload with a similar load (utilization) and similar job/size characteristics as the original workload to have a similarly realistic model of the real world. In detail this means:

– Keep the runtimes of jobs the same, while letting jobs double or quadruple the number of processes by exploiting several cores per node or leaving the process number unchanged, depending on the modeled self slowdown (see Section 4.9 for definitions of SL_{scr} and SL_{sno}). If the self slowdown of having 4 processes per node is less than a threshold MAX_SL (Table 6), the job size is quadrupled. If the self slowdown of having 2 processes per node is less than a smaller threshold $SELF\_SL\_2$ (Table 6), the job size is doubled. Thus, we adjust to both multiple cores per node and subsequent resource contention. Note that the modification of the workload corresponds to our model of space sharing as defined in Section 4.3 and used for the evaluation.
– Keep the percentage of serial jobs the same.
– Keep the percentage of power-of-two size jobs the same.

Note that the proper modification of the workload can be verified (which we did) by checking the average response times or average relative response times which should remain similar to the combination of the original machine and workload model if applying space sharing per virtual machine.

The rationale for our modification is that with more cores being available, users would likely run applications with more processes and tackle larger problem sizes. Moreover, since average runtimes were found to depend on the relative work submitted to the machine and not on the shape of the jobs, this is one of the feasible options of adjustment [13].

The detailed modification applied per job is described in the following formula ($nSize$ is the new size and $oSize$ is the original size):

$$nSize = \begin{cases} oSize & oSize = 1 \mid\mid (SL_{sno} > SELF\_SL\_2 \\ & \&\&\ SL_{scr} * SL_{sno} > MAX\_SL) \\ oSize * 2/SL_{sno} & oSize > 1 \&\& SL_{sno} \leq SELF\_SL\_2 \\ & \&\&\ SL_{scr} * SL_{sno} > MAX\_SL \\ oSize * 4/SL_{scr}/SL_{sno} & oSize > 1 \&\& SL_{scr} * SL_{sno} \leq MAX\_SL \end{cases} \quad (7)$$

4.9 Slowdown Modeling

As mentioned above, competition on shared resources like memory, network, disk, caches (if cache shared among cores) causes slowdown. This not only applies if different applications share resources (*coscheduling slowdown* SL_{cos}) but also if processes of the same application share resources per node. Slowdowns can differ depending on whether the processes run on different CPUs (*node self slowdown* SL_{sno}) or different cores of the same CPU (*core self slowdown* SL_{scr}). For simplification, we include any relative runtime changes due to changing the number of nodes used by exploiting different numbers of cores/CPUs per node in the self slowdown. We also include any potential memory contention in the slowdowns (analysis of workload traces from [7] suggests that typically 95% of the jobs need $\leq$ 50% of the memory per node, i.e. that memory contention is not a major issue if only 2 jobs are coscheduled). Slowdowns depend on the applications' characteristics in regards to the usage of resources and require proper slowdown metrics. Since resource usage characteristics are not available and would already require assumptions and since the slowdown metric goes beyond the scope of this paper, we chose to directly model slowdowns statistically based on available experimental data.

To derive the statistical distribution of slowdowns, we took data from different experimental sources. Thus, we used data from [2] which investigates the performance gain from using dual-core vs. single-core AMD nodes in the Cray Red Storm system at Sandia National Laboratories. If comparing the normalized grind times of the PARTISN benchmark (the only benchmark with all data needed) for the same number of processes on single-core CPUs (T_{single}) and dual-core CPUs (T_{dual}), we obtain the slowdown as $SL_{scr} = 2 * T_{dual}/T_{single}$. The calculated slowdowns are shown in Table 2 (showing only machine sizes relevant to our simulation).

Table 2. SL_{scr} calculated from data for the PARTISN benchmark in [2]

Benchmark and configuration	32P/16N vs. 32P/32N	64P/32N vs. 64P/64N	128P/64N vs. 128P/128N	256P/128N vs. 256P/256N
Diffusion: 24^3 problems	1.31	1.46	1.30	1.42
Transport: 24^3 problems	1.05	1.19	1.00	1.16
Diffusion: 48^3 problems	1.61	1.51	1.48	1.57
Transport: 48^3 problems	1.29	1.15	1.09	1.21

Table 3. SL_{scr} and SL_{sno} for NAS benchmarks, as measured in [15]. * means that the application could not be run on 8 nodes but needed to run on 9 nodes.

Allocation of 16 processes to nodes	IS	EP	FT	CG	LU	BT*	MG	SP*
8N vs. 16N, multi-core per node	1.37	1.05	1.27	1.04	1.08	1.11	1.22	1.47
8N vs. 16N, multi-CPU per node	1.35	1.05	1.16	0.99	0.99	1.00	1.01	1.01

From [15], we also obtained data for SL_{scr} by investigating several NAS benchmarks. Though the experiments only involved 8 and 16 nodes, the data range is similar. The same paper also measured SL_{sno} shown in Table 3. Because the data for SL_{scr} is similar to Table 2, we consider SL_{sno} from Table 3 to be generally valid. Note that the data in Table 2 shows that slowdowns are not very sensitive to job size but more dependent on problem size and the problem itself. The latter two, however, are exactly what we capture with a statistical model.

For data in regards to SL_{cos}, we refer to [21] which investigates the coscheduling slowdown for combinations of NAS benchmarks and combinations of synthetic benchmarks (with different communication patterns, different communication percentages and different message sizes), run on 8, 32, and 64 nodes.

Taking the data from Table 2, Table 3, and from [21] (for SL_{cos}) as the typical spread of possible slowdowns, we calculated the distribution of slowdowns and classified them into different ranges as shown in Table 4.[2] For example, in regards to SL_{scr}, 17% of the data above falls into the range [1.2, 1.3).

Table 4. Modeled distribution of slowdowns

range	SL_{scr}	SL_{sno}	SL_{cos}
$[0.9, 1.0)$	0%	25%	0%
$[1.0, 1.1)$	25%	45%	68%
$[1.1, 1.2)$	17%	12%	17%
$[1.2, 1.3)$	17%	5%	7%
$[1.3, 1.4)$	13%	13%	3%
$[1.4, 1.5)$	17%	0%	2%
$[1.5, 1.6)$	8%	0%	1%
$[1.6, 1.7)$	3%	0%	1%
$[1.7, 1.8)$	0%	0%	1%

[2] Minor adjustments of rounded values are done to obtain 100%.

To simplify the modeling and the slowdown calculation, processes of the same job are currently allocated to cores of different CPUs per node, while processes of different jobs are allocated to the cores of the same CPU. This captures the most frequent cases that processes of the same job run better on different CPUs rather than on the cores of the same CPU (future extensions toward more differentiated performance considerations are possible). If there are N_{core} processes of the same job, they occupy all cores in a node.

5 Experimental Results

5.1 Experimental Set-up

We perform the evaluation via discrete event simulation with the workload model described in Section 4.8. Each test with the Lublin-Feitelson workload model is run with 3 random workloads (each 10,000 jobs) and results are averaged. The cluster used in the simulation has two dual-core CPUs (4 cores totally) per node. Table 5 shows the characteristics of the workloads. Workload $W1$ is the workload created with the original slightly adjusted Lublin parameters (since our scheduler currently involves 5% overhead,[3] we have reduced the workload in our scheduler vs. the original workload by 5% via slightly increasing the inter-arrival times). We also test a busier Workload $W2$ which sets the α parameter in the inter-arrival time distribution to a smaller value and subsequently creates shorter inter-arrival times.

In regards to response-time estimation, we apply an adjustment by a factor of 0.75 to reflect that the estimates do not consider the benefits from non-type slice backfilling and coscheduling and jobs therefore run on average faster than estimated.

The parameters of Scojo-PECT are set to 30% relative time share for medium jobs and 70% relative time share for long jobs, 60 sec overhead per time slice

Table 5. Workload characteristics

Parameter	Value
$W1$ (normal load)	$\alpha = 10.33 \rightarrow Load = 10.6$
$W2$ (high load)	$\alpha = 9.83 \rightarrow Load= 13.0$
Machine size M	128
Percentage of short jobs N_S	64%
Percentage of medium jobs N_M	19.5% (54% of medium and long)
Percentage of long jobs N_L	16.5% (46% of medium and long)
Work of short jobs W_S	0.5%
Work of medium jobs W_M	26.0%
Work of long jobs W_L	73.5%
Percentage of serial jobs	24%
Percentage of jobs with power-of-two sizes	75%

[3] If jobs continue to run in the next slice, they do not actually need to be preempted but this reduction in overhead is currently not considered.

for preemption/resumption of the jobs, and 1h intervals for scheduling one short (optional), one medium and one long time slice. Jobs are classified as short if their runtime is $\leq$ 10 minutes, as medium if their runtime is $\leq$ 3 hours, and as long otherwise.

To evaluate the performance of our algorithm, we compare to the following approaches:

– SSP: Standard space sharing (only one job per node with 1, 2 or 4 processes, depending on the self slowdown as discussed in Section 4.3)
– $GLTS$: full group and time/space sharing G-LOMARC-TS

$$PPN = \begin{cases} 1 & jobsize = 1 \mid\mid (SL_{sno} > SELF\_SL\_2 \\ & \&\& \ SL_{scr} * SL_{sno} > MAX\_SL) \\ 2 & jobsize > 1 \&\& SL_{sno} \leq SELF\_SL\_2 \\ & \&\& \ SL_{scr} * SL_{sno} > MAX\_SL \\ 4 & jobsize > 1 \&\& SL_{scr} * SL_{sno} \leq MAX\_SL \end{cases} \tag{8}$$

– CST: Only coscheduling and matchmaking two jobs, while taking the best suitable match
– $CSTWS$: Only coscheduling and matchmaking two jobs, while taking the first match

We also experiment with different variants of G-LOMARC-TS:

– FBO: Only matchmaking in the first block
– NS: No sorting per block
– NH: No sorting and no blocks, while selecting the first suitable group
All scheduling approaches use conservative backfilling to support prediction. Table 6 shows all scheduler parameters used in our experiments.

Table 6. Scheduler parameters used in the experiments

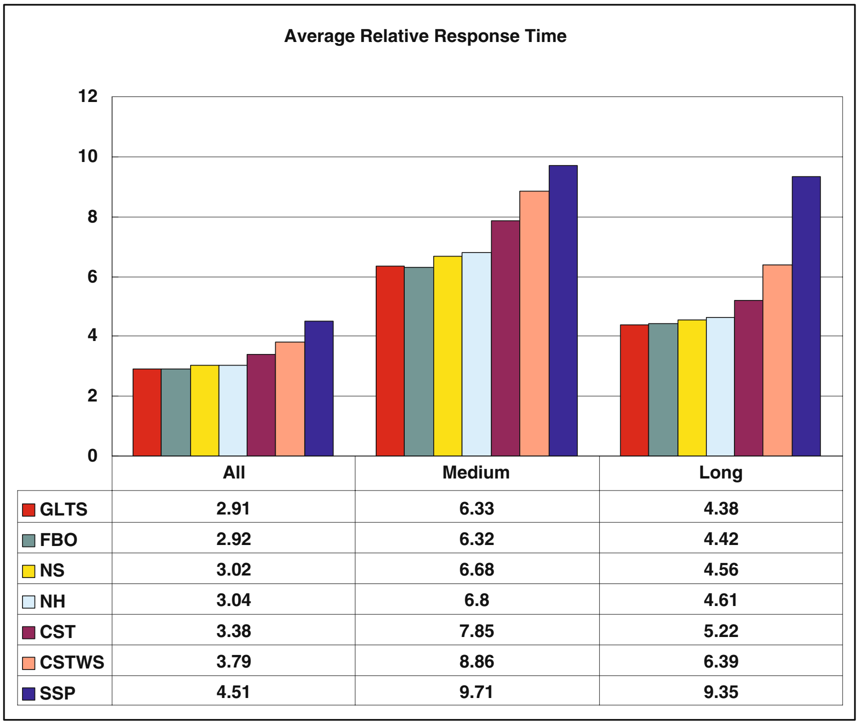
Parameter	Value	Explanation
MAX_SL	1.25	Maximum slowdown that a job should experience
$SELF\_SL\_2$	1.12	Maximum self slowdown with 2 processes of a job per node
$MIN\_UTILGAIN$	0.45	Minimum utilization gain a group should achieve
$BLOCK\_SIZE$	16	Number of jobs per block
$MATCHED\_LARGER$	0.125	$X - Y \leq MATCHED\_EXCEED\_PRIM * Y$ if X is the sum of the node requirements of all matched jobs and Y is the node requirement of the primary job
$RUNNING\_RPIM\_NUM$	8	Number of running jobs which are considered as primary jobs
$MATCHED\_LONGER$	3,000	Maximum time in seconds by which a matched job can be longer than the the primary job in a group
F_{slack}	1.5	Maximum slack factor for a job
N_H	12	Threshold rating high-load phases

We use the following metrics for comparison:

– Average bounded relative response time[4] (RR): response time in relation to pure runtime (without time slicing) while using cut-offs for very short jobs (only relevant for all-job evaluation)
– Core utilization U_{core} during high-load phases (see Section 4.6)

5.2 General Performance Results

The performance results (measured in RR) for G-LOMARC-TS compared to space sharing (SSP) and matchmaking for only two jobs (CST and $CSTWS$) are shown in Figure 6. The results show that the full algorithm of G-LOMARC-TS ($GLTS$) compared to SSP performs by 34.9% better for medium jobs, by 53.2% for long jobs, and 35.5% for all jobs. Note that this means that long jobs benefit more. Compared to matching only two jobs with best match (CST), the improvement is 19.4% for medium jobs, 16.1% for long jobs, and 13.9% for all jobs. Compared to matching only two jobs with the first suitable match



	All	Medium	Long
🟥 GLTS	2.91	6.33	4.38
▨ FBO	2.92	6.32	4.42
🟨 NS	3.02	6.68	4.56
☐ NH	3.04	6.8	4.61
🟪 CST	3.38	7.85	5.22
🟧 CSTWS	3.79	8.86	6.39
🟦 SSP	4.51	9.71	9.35

Fig. 6. *RR* for different schedulers and G-LOMARC-TS variants with Workload $W1$

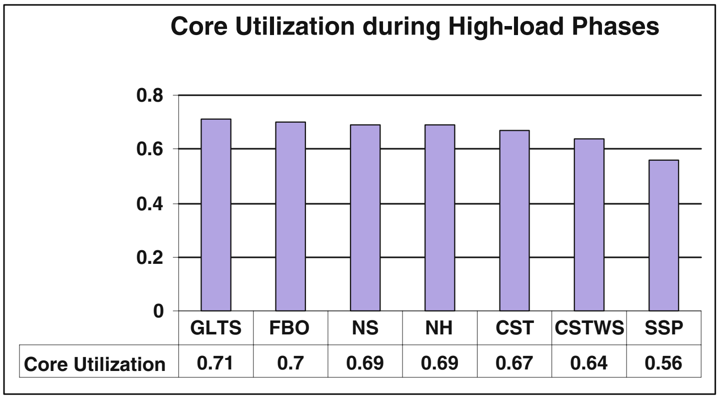
[4] The bounded relative response time is often called bounded slowdown. We avoid this term to avoid confusion with the slowdown due to resource contention.

($CSTWS$), the improvement is 28.6% for medium jobs, 31.5% for long jobs, and 23.2% for all jobs. This demonstrates that G-LOMARC-TS significantly improves average relative response times and that group matchmaking contributes significantly to the improvements.

Looking into the details of group matchmaking, we find an average of 2.5 jobs per group and about 1,500 (slightly depending on the concrete workload) groups being formed. Since the number of matched jobs is decided by the size of the primary job, on average, a job cannot match with many other jobs, and there are cases with only two jobs in a group as well as groups with more jobs. However, as discussed, we still obtain a significant improvement from group matchmaking. Since only medium and long jobs are coscheduled and they account for 36% of all jobs, this means that 41.7% of the jobs that are eligible for coscheduling actually run in a group. Though not shown, the relative results for response times look similar.

Looking into the effect on different job sizes for $GLTS$ compared to SSP, we find that job size has some impact on the benefits obtained. Medium-sized jobs (between 10% and 50% of machine size) gain most: about 40% for medium jobs and about 60% for long jobs, whereas narrow and wide jobs gain less (about 25% for narrow medium, 35% for wide medium and 45% for both narrow and wide long jobs). The likely explanation is that medium-sized jobs are easier to match with primary jobs than wide jobs and, thus, gain on average more. Narrow jobs may not gain much because the Lublin-Feitelson workload model does not include bursts, i.e. matching groups of serial jobs does not come fully into effect. However, narrow and wide jobs still experience a significant benefit (which also applies to maximum RR and 95% percentiles).

Figure 7 shows core utilization for high-load phases. We see that $GLTS$ improves core utilization by 26.8% vs. SSP and by 6.0% and 10.1% vs. CST and $CSTWS$. This demonstrates that the source of the relative response time improvements is the increased core utilization in high-load phases and that the



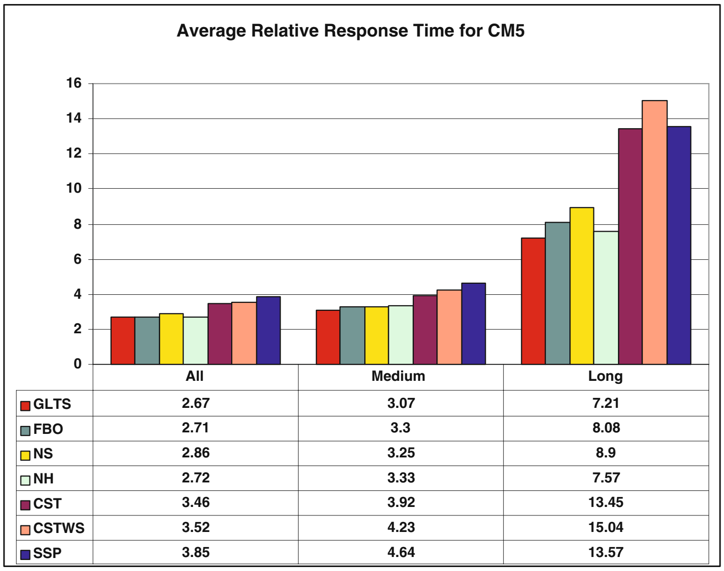
	GLTS	FBO	NS	NH	CST	CSTWS	SSP
Core Utilization	0.71	0.7	0.69	0.69	0.67	0.64	0.56

Fig. 7. Core utilizations during high-load phases for different schedulers and G-LOMARC-TS variants with Workload $W1$

increase vs. *SSP* is significant. Due to saved nodes from better core utilization in high-load phases, overall node utilization decreases by about 9%, with more improvement for long jobs (10%) than for medium jobs (4%).

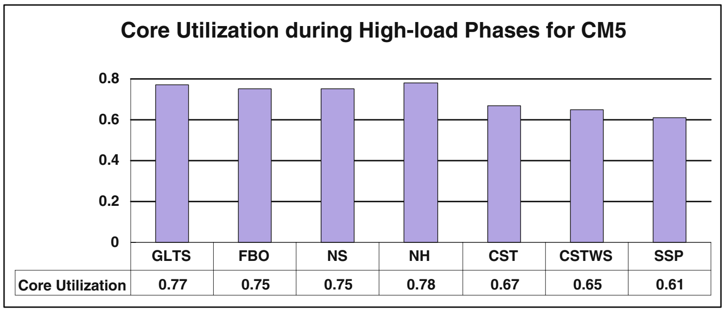
In regards to fairness, the distribution of delays shows that on average jobs finish at their predicted response time. The 75% percentile is a delay factor of 1.3 for both *M* and *L* jobs and the 95% percentile is a factor of 1.4. Thus, the scheduler meets its goal of supporting fairness well.

Finally, we present results for the CM5 trace of the Feitelson workload archive [6] in Figure 8 and Figure 9. The results show similar relative benefits, i.e. confirm



	All	Medium	Long
GLTS	2.67	3.07	7.21
FBO	2.71	3.3	8.08
NS	2.86	3.25	8.9
NH	2.72	3.33	7.57
CST	3.46	3.92	13.45
CSTWS	3.52	4.23	15.04
SSP	3.85	4.64	13.57

Fig. 8. *RR* for different schedulers an G-LOMARC-TS variants with CM5 trace



	GLTS	FBO	NS	NH	CST	CSTWS	SSP
Core Utilization	0.77	0.75	0.75	0.78	0.67	0.65	0.61

Fig. 9. Core utilization during high-load phases for different schedulers an G-LOMARC-TS variants with CM5 trace

the usefulness of the G-LOMARC-TS scheduler. The CST and $CSTWS$ variants perform poorly for long jobs which means that group matchmaking is even more beneficial for the CM5 trace than for the Lublin-Feitelson workload.

5.3 Impact of Different Heuristics and Machine Load

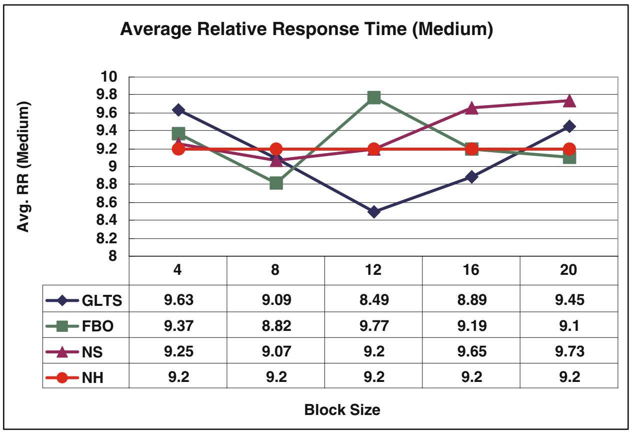
Figure 6 and Figure 8 also include results of several variants of G-LOMARC-TS. However, we do not see much impact from using the full matchmaking algorithm ($GLTS$) vs. simplifications which only match within the first block (FBO), do not sort (NS), or only selecting the first suitable group (NH). The reason is that the number of candidate jobs for matchmaking is less than 4 in 85% of the cases.

For Lublin-Feitelson Workload $W2$, the situation looks different. The results in Figure 10 show that $GLTS$ achieves the best results with optimum group size. NH does not depend on block size and is better than $GLTS$ with small and large block size for medium jobs but becomes significantly worse than $GLTS$ with optimum block size. FBO is much worse than $GLTS$ if the block size is small (because fewer jobs are considered) and becomes similar to $GLTS$ if block sizes are large enough. NS is especially worse than $GLTS$ for the optimum block size. Notable is that the optimum block size is 12 for both medium and long jobs. If the block size is too small, not enough matching candidates are available in the first block and more jobs are selected from the other blocks. If the block size is too large, jobs may be matched from the end of the first block. In both cases, jobs from further down the queue may move ahead and delay the jobs further up in the queue. Under Workload $W2$, the average group size is 2.6 (almost unchanged) but about 2,000 groups are formed which is about 25% more than for the basic workload. This is due to the fact that the waiting queues become longer and more matching candidates are available.

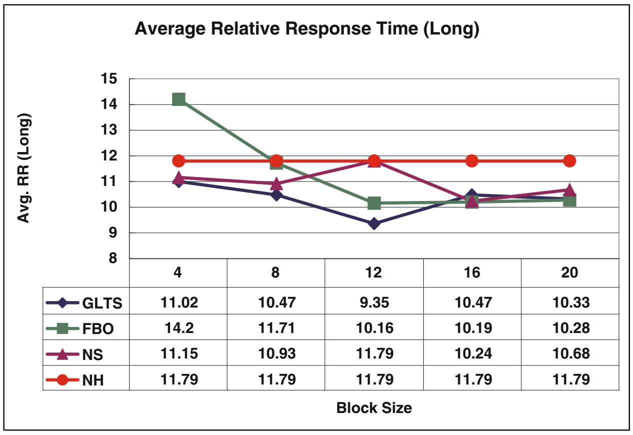
SPP cannot even handle Workload $W2$ and jobs queue-up as shown by an increased makespan–which is not the case for G-LOMARC-TS. Correspondingly, with SPP, the average relative response times become 17.5 (medium jobs), 36.74 (long jobs), and 10.65 (all jobs) which is much longer than G-LOMARC-TS. This demonstrates that our scheduler not only runs significantly better if the workload is normal ($W1$) but also can handle a much higher workload ($W2$) since the increased utilization makes it possible to run more jobs over the same time period.

5.4 Fairness vs. Utilization

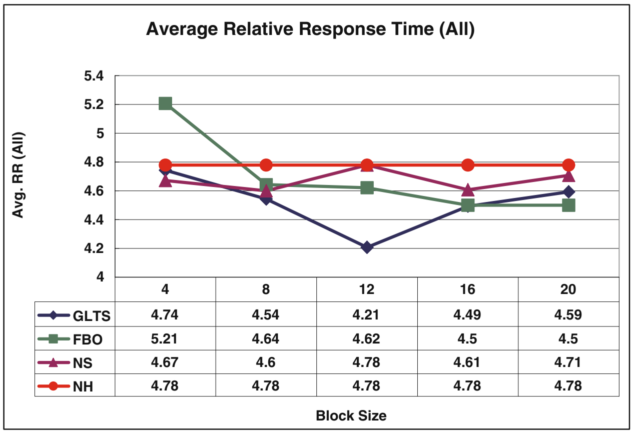
Next we investigate the trade-off between optimization for highest utilization gain and fairness by running the G-LOMARC-TS with different slack factors. The results are shown in Figure 11. If the slack factor is low, more possible groups are rejected which leads to higher average relative response times. With increasing slack factor, more groups pass the fairness check and average relative



	4	8	12	16	20
GLTS	9.63	9.09	8.49	8.89	9.45
FBO	9.37	8.82	9.77	9.19	9.1
NS	9.25	9.07	9.2	9.65	9.73
NH	9.2	9.2	9.2	9.2	9.2

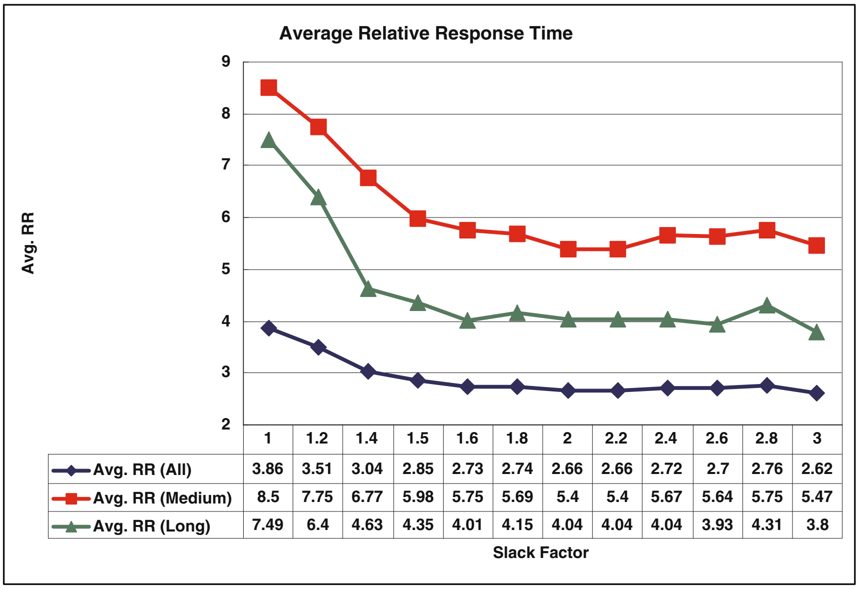


	4	8	12	16	20
GLTS	11.02	10.47	9.35	10.47	10.33
FBO	14.2	11.71	10.16	10.19	10.28
NS	11.15	10.93	11.79	10.24	10.68
NH	11.79	11.79	11.79	11.79	11.79



	4	8	12	16	20
GLTS	4.74	4.54	4.21	4.49	4.59
FBO	5.21	4.64	4.62	4.5	4.5
NS	4.67	4.6	4.78	4.61	4.71
NH	4.78	4.78	4.78	4.78	4.78

Fig. 10. Average relative response time for G-LOMARC-TS and different variants with different block sizes under Workload $W2$



Slack Factor	1	1.2	1.4	1.5	1.6	1.8	2	2.2	2.4	2.6	2.8	3
◆ Avg. RR (All)	3.86	3.51	3.04	2.85	2.73	2.74	2.66	2.66	2.72	2.7	2.76	2.62
■ Avg. RR (Medium)	8.5	7.75	6.77	5.98	5.75	5.69	5.4	5.4	5.67	5.64	5.75	5.47
▲ Avg. RR (Long)	7.49	6.4	4.63	4.35	4.01	4.15	4.04	4.04	4.04	3.93	4.31	3.8

Fig. 11. Average relative response time for Workload $W1$ with different slack factors F_{slack}

response times improve. This is true up to a certain slack factor from which on the results become approximately equal. This can be explained by most suitable groups pass the check if the slack factor reaches a certain value. As can be seen from the figure, the threshold is $F_{slack} = 1.5$ (used in all experiments above). Thus, larger slack factors than that certain value make no sense. Smaller slack factors may be chosen if fairness is rated higher than relative response times.

6 Summary and Conclusion

We have presented the G-LOMARC-TS scheduler which incorporates both space sharing and semi time sharing on clusters with multi-core nodes. G-LOMARC-TS employs group matchmaking and constraints the matchmaking by fairness to individual jobs. G-LOMARC-TS is integrated with the coarse-grain preemptive Scojo-PECT scheduler by applying G-LOMARC-TS per virtual machine managed in Scojo-PECT. Relative response times and core utilization are significantly improved with G-LOMARC-TS. At the same time, node utilization is decreased by packing the jobs better onto cores per node, and the scheduler can therefore handle heavier workloads than space sharing per virtual machine, i.e. increase the saturation point. The results also demonstrate that group matchmaking contributes significantly to the improvements.

References

[1] Alves, F., Silva, B., Scherson, I.D.: Concurrent Gang – Towards a Flexible and Scalable Gang Scheduler. In: Proc. 11th Symp. on Computer Architecture and High Performance Computing, Natal, Brazil (September 1999)

[2] Brightwell, R., Underwood, K.D., Vaughan, C.: An Evaluation of the Impacts of Network Bandwidth and Dual-Core Processors on Scalability. In: Proc. Internat. Supercomputing Conference, Dresden, Germany (June 2007)

[3] Carrington, L., Wolter, N., Snavely, A., Bailey Lee, C.: Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications. In: Proc. DoD Users Group Conference, Williamsburg, Virginia. IEEE, Los Alamitos (2004)

[4] Chai, L., Gao, Q., Panda, D.K.: Understanding the Impact of Multi-Core Architecture in Cluster Computing–A Case Study with Intel Dual-Core System. In: Proc. CCGRID, Rio de Janeiro, Brazil. IEEE, Los Alamitos (2007)

[5] Esbaugh, B., Sodan, A.C.: Coarse-Grain Time Slicing with Resource-Share Control in Parallel-Job Scheduling. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC 2007. LNCS, vol. 4782, pp. 30–43. Springer, Heidelberg (2007)

[6] Feitelson Workload Archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html (last retrieved January 2008)

[7] Feitelson, D.G., Rudolph, L., Schwiegelsohn, U., Sevcik, K.C., Parkson, W.: 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)

[8] Jung, C., Lim, D., Lee, J., Han, S.: Adaptive Execution Techniques for SMT Multiprocessor Architectures. In: Proc. ACM SIGPLAN PPoPP, Chicago, Illinois (June 2005)

[9] Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R.: An Empirical Study of Hyper-Threading in High Performance Computing Clusters. In: Proc. Linux HPC Revolution, St. Petersburg, Florida (2002)

[10] Lublin, U., Feitelson, D.G.: The Workload on Parallel Supercomputers-Modeling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing 63(11), 1105–1122 (2003)

[11] Moscibroda, T., Mutlu, O.: Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In: Proc. 16th USENIX Security Symp., Boston, Mass. (2007)

[12] Sabin, G., Sadayapan, 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)

[13] Sodan, A.C.: Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability–First Experiences. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 259–279. Springer, Heidelberg (2009)

[14] Sodan, A.C.: Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling–A Survey. Concurrency & Computation: Practice & Experience 17(15), 1725–1781 (2005)

[15] Sodan, A.C., Gupta, G., Deshmeh, A., Zeng, X.: Benefits of Semi Time Sharing and Trading Space vs. Time in Computational Grids. Technical Report 08-020, University of Windsor, Computer Science (May 2008)

[16] Sodan, A.C., Gupta, G.: Time vs. Space Adaptation with ATOP-Grid. In: Proc. ACM Workshop on Adaptive and Reflective Middleware (ARM), Melbourne, Australia (November 2006)
[17] Sodan, A.C., Lan, L.: LOMARC Lookahead Matchmaking for Multiresource Coscheduling on Hyperthreaded CPUs. IEEE Trans. on Parallel and Distributed Systems 17(11), 1360–1375 (2006)
[18] Talby, D., Feitelson, D.G.: Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In: Proc. Internat. Symp. on Parallel Processing & Symp. on Parallel and Distributed Processing (IPPS/SPDP), Puerto Rico. IEEE, Los Alamitos (1999)
[19] Weinberg, J., Snavely, A.: Symbiotic Space-Sharing on SDSC's DataStar System. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2006. LNCS, vol. 4376, pp. 192–209. Springer, Heidelberg (2007)
[20] Wiseman, Y., Feitelson, D.G.: Paired Gang Scheduling. IEEE Trans. Parallel and Distributed Systems 14(6), 581–592 (2003)
[21] Zeng, X., Shi, J., Cao, X., Sodan, A.C.: Grid Scheduling with ATOP-Grid under Time Sharing. In: Proc. CoreGrid Workshop on Grid Middleware, Dresden. Springer, Heidelberg (2007)

Adaptive Scheduling for QoS Virtual Machines under Different Resource Allocation – Performance Effects and Predictability

Angela C. Sodan

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

Abstract. Virtual machines have become an important approach to provide performance isolation and performance guarantees (QoS) on cluster servers and on many-core SMP servers. Many-core CPUs are a current trend in CPU design and require jobs to be parallel for exploitation of the performance potential. Very promising for batch job scheduling with virtual machines on both cluster servers and many-core SMP servers is adaptive scheduling which can adjust sizes of parallel jobs to consider different load situations and different resource availability. Then, the resource allocation and resource partitioning can be determined at virtual-machine level and be propagated down to the job sizes. The paper investigates job re-sizing and virtual-machine resizing, and the effects which the efficiency curve of the jobs has on the resulting performance. Additionally, the paper presents a simple, yet effective queuing-model approach for predicting performance under different resource allocation.

Keywords: adaptive job scheduling, molding, utilization, prediction, queuing model.

1 Introduction

Virtual machines (VMs) have become an important approach in resource provisioning and providing quality-of-service on cluster servers and shared-memory servers [8]. Shared-memory servers will likely gain more significance with the emergence of many-core CPUs as a trend in CPU design to reduce power consumption and to continue performance growth of CPUs in spite of physical limits imposed on the performance improvement of individual cores [17]. Thus, we will likely see CPUs with tens of cores in the near future. To exploit the performance of many-core CPUs per individual program, parallelization of applications will be a must [24]. (System) virtual machines support to safely share servers among different applications or user groups, and are likely to become more important on many-core servers with large numbers of cores. Virtual machines provide functional separation (with potentially even different operating systems) but also a framework for performance guarantees (QoS) if controlling resource allocation among virtual machines. We assume in the following that resource allocation is realized via partitioning the available cluster nodes or cores among the different virtual machines, with potentially dynamically changing partition sizes. Furthermore, per virtual machine, jobs are assumed to be scheduled with space sharing, i.e. dedicated allocation of CPUs. Our experimental analysis is based on cluster workloads but would easily translate to many-core servers that are run with batch scheduling.

If resource allocation per virtual machine changes over time, this has implications on suitable sizes of parallel jobs. Adaptive job scheduling which can adjust job sizes according to the current load and resource availability is already well investigated in the literature and will likely become an important approach for scheduling not only on cluster servers but also on many-core/virtual-machine servers. Adaptive job scheduling may mean deciding the job size at start time (molding jobs) or adjusting it at runtime (malleable jobs). Though the latter is more flexible it requires special support in the runtime system of the application, whereas start-time adaptation was found to be applicable to the majority of jobs [3]. In this paper, we therefore only consider size adaptation at job start-time.

The presented work studies the effects of changing the (virtual) machines size and the size allocation of the jobs, including the effects of the jobs' efficiency/scalability curve. The evaluations are mainly done by keeping the original scheduler unchanged and adjusting all job sizes equally, i.e. without looking into specific scheduling contexts. This permits investigation of effects independent of the scheduling algorithm, whereas previous research mixed efficiency and scheduler considerations. Thus, as one of the contributions, this paper looks separately into the benefits obtained from size changes under constant efficiency (work-conserving adaptation) and different levels of efficiency changes (non-work-conserving adaptation). The experiments were performed with 1) a standard scheduler and FCFS and priority policies and 2) with our Scojo-PECT preemptive scheduler [5][20][19] which can assign different priorities to different job type (currently defined on the basis of runtimes) via different time shares but is FCFS per job type. Though benefits may be expected from mere reduction in sizes by easier fitting of jobs and smart scheduling approaches, the results demonstrate that the change in efficiency and subsequently work load constitutes the dominating performance factor.

2 Related Work

Most adaptive approaches apply molding only. The approach of Cirne and Berman [3] molds jobs at the time of job submission, whereas later research showed performance improvements [23] by setting limits for the maximum job size in dependence on the current system load and on the job's size requests and by making decisions at job start time rather than submission time. Both approaches applied the Downey scalability model [4] which describes typical application scalability/efficiency curves and includes the possibility to model different scalability by modifying the corresponding scalability factor σ. In [23], the evaluations of the proposed adaptive scheduling algorithm were done with different

σ values. However, in our work, we not only test different efficiency but separate efficiency effects from the scheduling algorithm. Most of these approaches exploit adaptation with the goal to adapt to varying system load. The approach by Naik et al. [13] also adapts resource allocation at runtime and attempts to schedule all jobs from the queue, though setting a limit for medium and long jobs to keep space for short jobs. Other approaches apply limits on job sizes in relation to machine size to keep space for future arrivals [9][15]. In addition to load adjustments, some approaches additionally try to reduce fragmentation in dependence on the specific resource/scheduling situation [14][22].

The two basic approaches to decide about the job sizes are resource-based partitioning and efficiency-based partitioning [6]. Resource-based partitioning typically comes in the form of EQUI partitioning which means assigning the same number of resources to each job. This approach yields suboptimal performance in the general case as it does not consider how well the jobs use the resources [2][12]. Efficiency-based partitioning exploits the efficiency characteristics of the applications and allocates more resources to jobs that make better use of them, which typically leads to the overall best results [2][12]. Similar to resource-based partitioning, efficiency-based partitioning may be applied in the form of providing equal efficiency to all jobs in the system (EQUI-EFF).

3 Work-Conserving and Non-Work-Conserving Job-Size Adaptation—Myths and Reality

3.1 Space Sharing and Scojo-PECT Time Sharing

For our experiments, we use a standard space-sharing scheduler which employs either a FCFS policy or priority scheduling. Priorities are based on runtime classes, and classes with shorter runtime receive higher priority. To avoid starvation, the implementation which is used here ages jobs to the next higher priority level if their wait time exceeds 10 times their runtime.

Scojo-PECT [5] employs preemption to support scheduling of shorter jobs in the presence of longer-running jobs. Scojo-PECT preempts jobs to swap space which is easy to support in the machine environment. This avoids the memory pressure which gang scheduling imposes and the hard-to-support checkpointing which is necessary for migration. However, Scojo-PECT subsequently imposes the constraint that preempted jobs are later restarted on the same resources. To make preemption to disk affordable and to avoid that jobs are delayed because of problems in getting access to their resources again, Scojo-PECT employs coarse-grain time slices that preempt all jobs. Jobs are sorted per job class/type, and slices associated with job types. The slice time for each job type is determined on the basis of typical job-type mixes and the administrator's policies and can be recalculated in regular time intervals. One slice for each type is scheduled per interval (since short jobs backfill into other slices in most cases, their slice is only scheduled if short jobs are waiting), and the slice times can be decided at the beginning of each interval. This permits controlling the resource allocation via different policies at different times of the day or via adaptive allocation which considers the current load of the machine [19]. In the context of this paper, the relative slice times of different job classes are kept static. Jobs per job type are scheduled in FCFS, and either EASY or conservative backfilling is applied. Since the separation of jobs into different types is likely to increase fragmentation because job sizes and job runtimes tend to be correlated, Scojo-PECT employs additionally safe non-type slice-backfilling. This means that preempted or waiting jobs of a different type may be backfilled into a slice–with this backfilling only being valid until the end of the slice–if they do not delay any job of the slice type or any of their own type jobs according to the backfilling approach applied. In [5] and [20], this optimization is shown to be crucial for good performance.

All schedulers are configured to support 3 job classes based on their runtime: short (S), medium (M), and long (L) jobs. The original classification is kept if resizing the jobs. We use A to denote results for all jobs. The backfilling approach applied is for all schedulers conservative backfilling.

3.2 Test Setup

Thus, the following schedulers were tested:

- Standard space-sharing with FCFS ($FCFS$)
- Standard space-sharing with priorities ($Prio10$)
- Scojo-PECT coarse-grain time sharing with separation of job types ($PECT$)

The parameters of Scojo-PECT were set to 30% relative time share for M jobs and 70% for L jobs, 60 sec overhead per time slice for preemption/resumption of the jobs, and 1 h time intervals for scheduling one S (optional), one M, and one L time slice. Jobs are classified as S if they run $\leq$ 10 minutes, as M if they run $\leq$ 3 hours, and as L otherwise.

The schedulers were tested with the Lublin-Feitelson workload model [10], setting the original machine size Mz to 128 CPUs, and with the CM5 trace from the Feitelson workload archive [7]. The Lubin-Feitelson model creates one process per node. The CM5 trace has 32 CPUs per node and sizes only come in multiple of 32 (i.e. do not differentiate the use per node)—thus, all sizes (including the original machine size measured in 1024 CPUs) were divided by 32 to map them to the model of one process per node. In each test run, 10,000 jobs were simulated.

Job size was modified by certain factors F_A in the range between 1.5 and 0.3. For the basic tests without modification of the scheduling algorithm, the same F_A was applied to all jobs per experiment. Job sizes were always rounded up. Note that the Lublin-Feitelson model creates about 25% serial jobs which never change their size. The jobs with maximum size never grow beyond this size and can only become smaller. The job sizes and corresponding runtimes created by the Lublin-Feitelson model or the trace were taken as the original sizes/runtimes with F_A=1.0. Runtimes are considered to be correct estimates.

An efficiency model was built on top of the created workload and the following efficiency parameters tested:

– Equal efficiency for all resource allocations, which means that the jobs can be size adapted in a work-conserving manner (WC)
– Efficiency described via typical efficiency speedup curves, which were modeled with a simple phase-wise linear approximation between pairs of 4 sizes: a) 1, b) 0.5 * original size, c) original size, and d) min {2 * original size, machine size}, using efficiencies of
  1. 1, 0.8, 0.65, and 0.4 ($E1$)
  2. 1, 0.75, 0.65, and 0.5 ($E2$)
  3. 1, 0.7, 0.65, and 0.6 ($E3$)

This is similar to the idea of Secvik's modeling approach presented in [16] and sufficient for the purpose of our experiments to only study the principle effects of different efficiencies. With the same argument, the efficiency is assumed to be the same for all jobs. In other work, we develop efficiency/scalability models for applications on multi-core CPUs [11].

For tests with different resource allocation to virtual machines, the original machine size is modified by a factor F_M, while still using the job sizes and runtimes generated for the original machine sizes, adjusted by specified F_A factor. Resizing of virtual machines is assumed to be done without any performance impact (slowdown) from other virtual machines which may share the server.

Throughout the paper, the evaluation uses average bounded relative response times RR (often called bounded slowdowns in the literature[1]). RR which relates response times to runtimes is calculated vs. the pure runtime of the job (without time slicing) and always vs. the original runtime without size modification. In some cases, average response times are shown. We found that the relative effect of the changes which we study were almost identical for R and RR, and RR is shown for reasons of how this research evolved. With RR_A denoting bounded relative response times under adaptive resource application, the graphs evaluate relative worsening, i.e. $RR_A/RR - 1$, if $F_A > 1$, and relative improvement, i.e., $RR/RR_A - 1$, if $F_A < 1$ to obtain a balanced presentation (rather than one end providing results in the range [0,1] which are hard to differentiate).

3.3 Formal Results for Work-Conserving and Non-Work-Conserving Job Re-shaping

Before presenting the results of the experiments, below some simple theoretical considerations are presented to help explain some effects in the experiments.

Theorem 1. We assume that all job runtimes T are equal, have equal original size Sz with $Sz = Mz$ (machine size), and constant and equal efficiency under different sizes. Additionally, we assume off-line scheduling of a fixed set of jobs. This means that jobs can be reshaped in a work-conserving manner. N_J is the number of jobs resized to fit together into memory rather than being scheduled serially, and N is the overall number of jobs in the system, $N_G = N/N_J$ the

[1] The term slowdown is avoided in this paper since it is also used for contention effects under time sharing.

number of groups under re-shaping, then serial scheduling gives average response times ϕR of

$$\phi R_{Serial} = T * (\sum_{i=1,N} i)/N = T * (N * (N + 1)/2)/N = T * (N + 1)/2 \tag{1}$$

and re-shaped-job adaptive scheduling gives

$$\phi R_{Adaptive,WC} = N_J * T * (\sum_{i=1,N_G} i)/N_G \tag{2}$$

If rewriting (1) for better comparison, this gives:

$$\phi R_{Serial} = T * (\sum_{i=1,N_J} i)/N_J + N_J * T * (\sum_{i=1,N_G-1} i)/N_G \tag{3}$$

Proof. Under serial scheduling, the first job's response time is T, the second one has a wait time W of T and a runtime of T, i.e. a response time of $R = T + W = 2T$, etc., which means that $\phi R_{Serial} = (\sum_{i=1,N}(i * T))/N = T * \sum_{i=1,N} i/N$ (q.e.d.)

Under reshaping, the runtime per job changes to $N_J * T$. The response time is equal for all the jobs in the group, with the first group having an average response time of $N_J * T$, the second group a wait time of $N_J * T$ and a runtime of $N_J * T$, i.e. an average response time of $2 * N_J * T$, etc., which means that $\phi R_{Adaptive,WC} = (\sum_{i=1,N_G}(iN_J * T))/N_G = N_J * T * \sum_{i=1,N_G} i/N_G$ (q.e.d.)

However, the runtime of the overall group is equal to the sum of the serial runtimes of those jobs, i.e. the average wait time for the next group of jobs is the same under both approaches. Thus, (1) can be transformed into (3).

Comparing (2) and (3) shows that the term $N_J * T * (\sum_{i=1,N_G-1} i)/N_G$ is common. Thus, for large N_G, the difference becomes small. However, the difference can matter if N_G is very small. This means high load makes the difference insignificant, whereas low load makes it relevant. In regards to the detailed difference, the remaining term in (2) is $t_{r,Serial} = T * (\sum_{i=1,N_J} i)/N_J$ and the remaining term in (3) is $t_{r,Adaptive,WC} = N_J * T$. Thus, $t_{r,Serial} < t_{r,Adaptive,WC}$ for $N_J > 1$ since $(N_J + 1)/2 < N_J$ for $N_J > 1$. This means that serial scheduling performs typically slightly better and that, for small N_G, the difference between serial and adaptive scheduling is more significant if N_J is larger.

Theorem 2. If reshaping is non work-conserving, i.e. jobs run with better efficiency if becoming smaller or lower efficiency if becoming larger, the runtimes are affected by the change in efficiency E, with $E_{F_A=1.0}$ being original and $E_{F_A=X}$ the efficiency after adaptation:

$$\phi R_{Adaptive,E} = N_J * T * E_{F_A=1.0}/E_{F_A=X} * (\sum_{i=1,N_G} i)/N_G \tag{4}$$

$$\begin{aligned}\phi R_{Adaptive,E} = N_J * T * E_{F_A=1.0}/E_{F_A=X} +& \\ N_J * T * E_{F_A=1.0}/E_{F_A=X} * (\sum_{i=1,N_G-1} i)/N_G&\end{aligned} \tag{5}$$

If comparing (3) and (4), $t_{r,Serial} = T * (\sum_{i=1,N_J} i)/N_J$ is not necessarily less than $t_{r,Adaptive,E} = N_J * T * E_{F_A=1.0}/E_{F_A=X}$ since $(\sum_{i=1,N_J} i)/N_J = (N_J+1)/2$ may not be less than $E_{F_A=1.0}/E_{F_A=X} * N_J$ if N_J is small and $E_{F_A=X}$ is much better than $E_{F_A=1.0}$. Thus, as can be easily seen from the formulation in (5), with larger N_G, adaptation reduces average response time by approximately $E_{F_A=1.0}/E_{F_A=X}$.

Thus, if assuming $E_{F_A=0.5} = 0.8$ and $E_{F_A=1.0} = 0.65$ (which corresponds to $E1$), $N_J = 2$, and $N = 2$ ($T_G = 1$)), $\phi R_{Adaptive,E} = E_{F_A=1.0}/E_{F_A=0.5} * N_J = 1.625$, whereas $\phi R_{Serial} = 1.5$. ϕR_{Serial} can be expected to be better than $\phi R_{Adaptive,E}$ under low load and to be increasingly worse with increasing load which is shown with the following calculations:

- $N_J = 2$ and $N = 4$ ($T_G = 2$): $\phi R_{Serial} = 2.5$ and $\phi R_{Adaptive,E} = 1.625 * 1.5 = 2.4375$ which means that adaptation is already slightly better
- $N_J = 2$ and $N = 6$ ($T_G = 3$):
  $\phi R_{Serial} = 3.5$ and $\phi R_{Adaptive,E} = 1.625 * 2 = 3.25$
- $N_J = 2$ and $N = 100$ ($T_G = 50$):
  $\phi R_{Serial} = 50.5$ and $\phi R_{Adaptive,E} = 1.625 * 25.5 = 41.4$ which means their ratio (1.21) is already close to the ratio of the efficiencies (1.23).

The writing as presented in (5) shows average runtime ϕT as the first term and average wait time ϕW as the second term. As can be seen from comparison to (2), with reshaping, ϕT becomes longer by $N_J * E_{F_A=1.0}/E_{F_A=X}$ and ϕW is reduced by $E_{F_A=1.0}/E_{F_A=X}$. The latter dominates for larger N_G.

In realistic scheduling, we face additionally packing problems, different runtimes and sizes, and potentially different efficiency. Moreover, submissions are dynamic. However, the above considerations give some basic clues about the expected performance behavior.

3.4 Experimental Results for Work-Conserving and Non-Work-Conserving Job Re-shaping

Before showing adaptation results which mostly look into relative performance under different adaptation, a note about absolute performance under $F_A = 1.0$. If setting resource shares for equal service, Scojo-PECT provides similar service to M and L jobs as the priority scheduler but improves overall response times by about 45% by serving S jobs better. The simple FCFS scheduler performs by about 60% worse for all jobs and by about 20% worse if only considering M and L jobs.

In the following, we investigate performance under different job re-shaping, while keeping the machine size at original size. Note that only the sizes of all jobs are changed and no special adaptive scheduler is used.

In regards to re-shaping, we can expect that

- If decreasing jobs sizes, the smaller sizes might provide better options for packing. If increasing job sizes, packing possibilities might worsen.

– A counter effect under work-conserving re-shaping as per (3) shows in phases of low load, while performance is approximately the same if the load is very high.
– Improvements under non-work-conserving re-shaping according to (4) show if reducing the job sizes, while performance is worsened if increasing the job sizes.

In our experiments, we explored the corresponding aggregate effects on RR for the 3 schedulers $FCFS$, $Prio10$, and $PECT$. The results are shown in Figure 1, Figure 2, Figure 3, and Figure 4, differentiated for M, L, and A jobs.

As we can see, under work-conserving re-shaping, the sizes make no relevant difference in regards to RR if scheduling with $FCFS$ or $PECT$ (though RR is shown, the same applies to R). Obviously any effects from better packing and extended wait times under low load cancel each other out, though improvements in packing might have been expected to make more of a difference. For $Prio10$, smaller sizes provide a benefit of up to a factor of 1.4.

The likely explanation is that packing and backfilling do not behave significantly different and that mostly the load of the jobs in the system matters (which is further investigated in Section 6.1.

Since, however, the overall load and the length of the queue may matter, Figure 5 shows results for the original load L_1 with $U = 0.76$, and for reduced workloads L_2 with $U = 0.66$ and L_3 with utilization $U = 0.56$ (loads are modified by increasing interarrival times via the α parameter from 10.33 to 10.65 and 10.9). Figure 5 also includes results for the interarrival times being changed

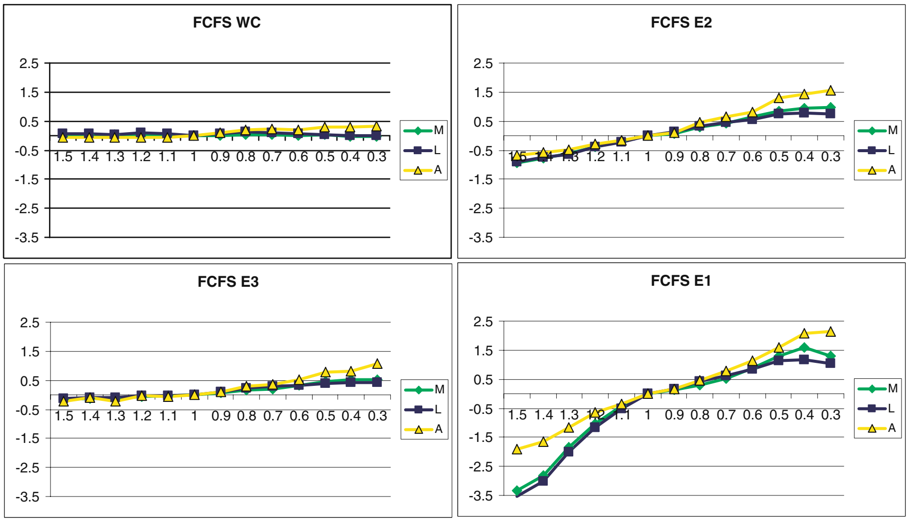


Fig. 1. RR improvement for Lublin-Feitelson workload under $FCFS$ with different efficiency, shown over different size-modification factors

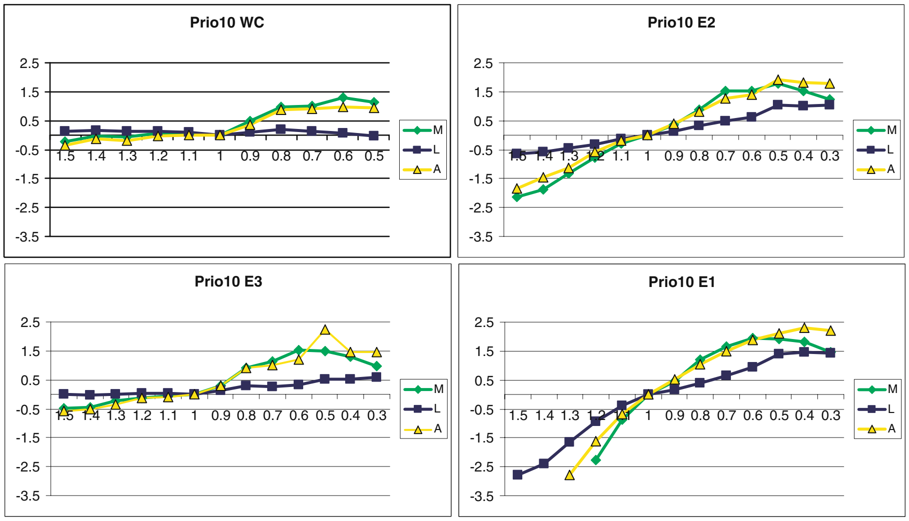


Fig. 2. *RR* improvement for Lublin-Feitelson workload under *Prio*10 with different efficiencies, shown over different size-modification factors

to exponential without daily cycles which significantly reduces average queue lengths ϕQ_{system} of jobs in the system (waiting or running) though they are almost independent of F_A (ϕQ_{system} is about 13 for M and 31 for L with $L1$; about 5 for M and 15 for L with $L3$; and reduces to about half if interarrival times are exponential without daily cycles). However, the RR results are all similar. Even if queue lengths become shorter, F_A only changes the number of jobs fitting into the machine by maximally a factor of 2 ($N_J = 2$), i.e. $(N_J + 1)/2 = 1.5$ which means that the difference for small N_G is low. Thus, the results behave as expected for the theoretic consideration that were done for the simple off-line case with equal job sizes.

Finally, since conservative and EASY backfilling may have different effects with job reshaping, Figure 6 compares results for conservative and EASY backfilling. Work-load conserving reshaping provides a little, though not very relevant benefit under EASY backfilling (because of being able to push some shorter jobs ahead) but overall the relative improvements are very similar. Thus, the effect of the backfilling approach has no major impact.

In all experiments shown, non-work-conserving job re-shaping with all factors $F_A < 1.0$ would suggest an improvement in R and RR which is indeed the case. The improvements are higher if the efficiency improvement is higher, i.e. best for $E1$ and lowest for $E3$. If looking at R from the theoretic consideration in (5), $E_{1,F_A=0.5}/E_{1,F_A=1.0} = 0.8/0.65 = 1.23$. The real improvements due to higher efficiency should be lower since not all jobs change sizes (such as serial jobs always remaining serial). However, the measured R_L improves by a factor of 2.28 and the measured R_M by a factor of 1.67. Correspondingly, the measured RR_L improves

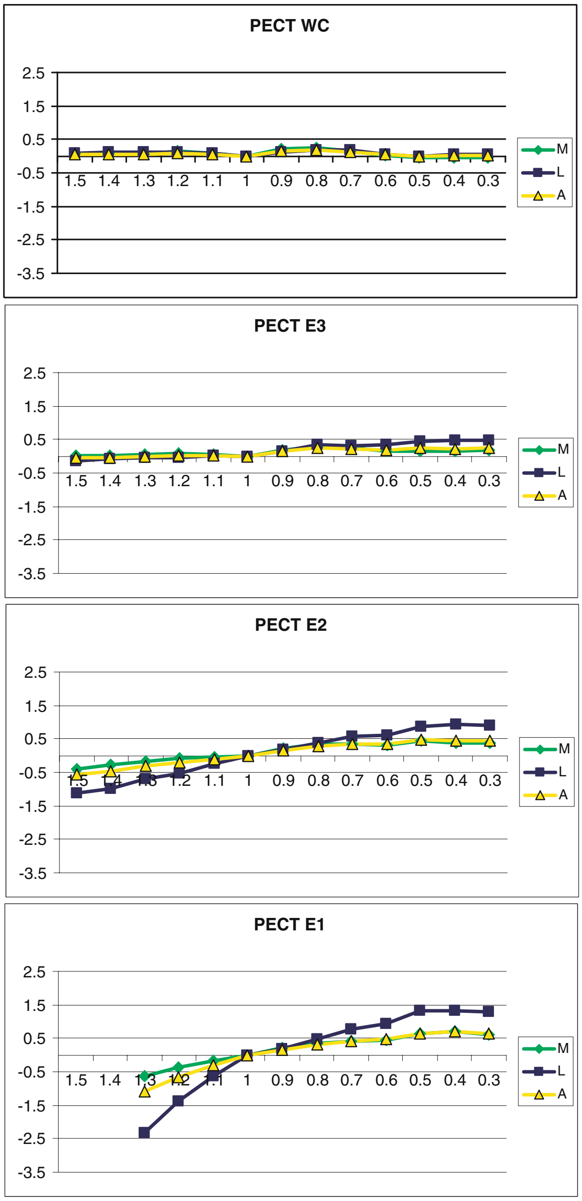


Fig. 3. *RR* improvement for Lubin-Feitelson workload under *PECT* with different efficiencies, shown over different size-modification factors

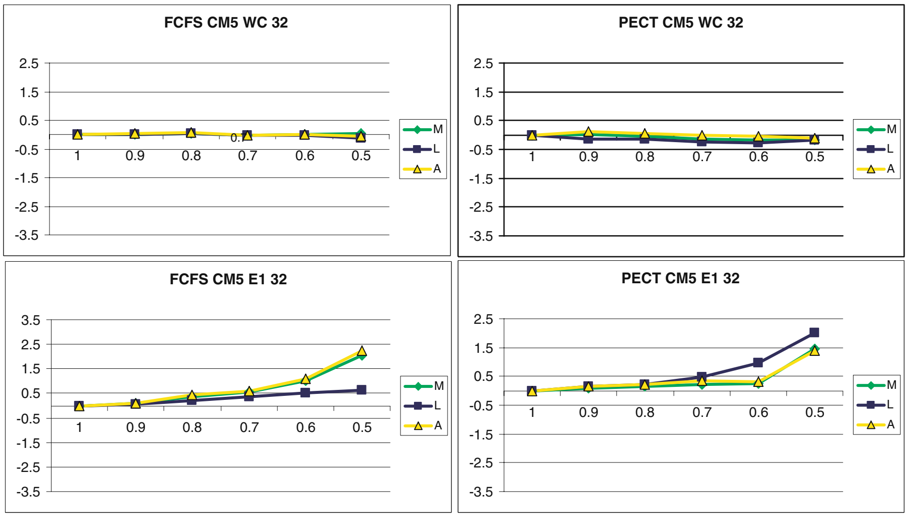


Fig. 4. RR improvement for CM5 trace under $FCFS$ and $PECT$ with work-conserving and with $E1$ efficiency, shown over different size-modification factors

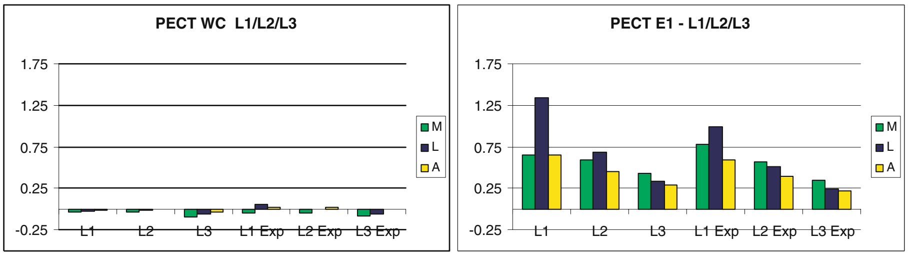


Fig. 5. RR improvement for $F_A = 0.5$ vs. $F_A = 1.0$ with different loads, and either standard interarrival times of model or exponential interarrival times without daily cycles (Exp)

by a factor of 2.34, RR_M by a factor of 1.65, and RR_A by a factor of 1.65. Thus, the changes in R and RR are almost identical. The higher-than-expected benefits may be partially due to better packing but likely also to less queue-up of work in the systems, as will be discussed in Section 6.1. To check the effect of the load on the improvements, again lower Loads $L2$ and $L3$ were tested (see Figure 5). Load $L2$ and $L3$ also show improvements though they are relatively lower than for $L1$ This can be explained by less difference in the work queuing up. Indeed, if looking at average wait times, they are—if changing from $F_A = 1.0$ to $F_A = 0.5$ and using $L1$—reduced from 9.3h to 2.4h for M jobs and from 33h to 7.4h for L jobs, which is much more than the predicted factor of 1.23 from (4).

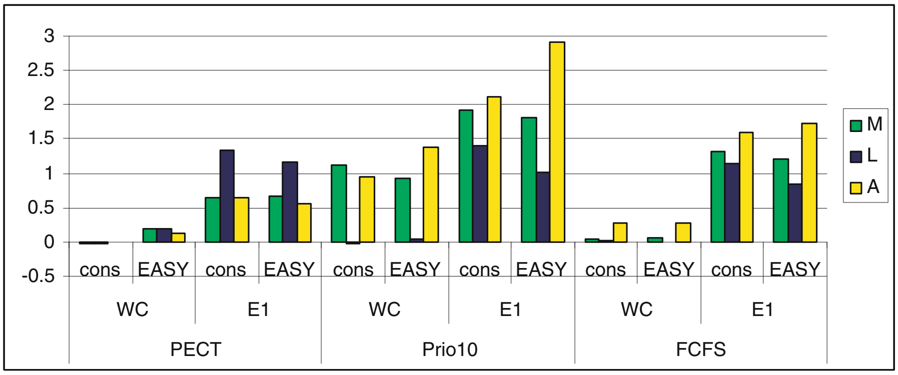


Fig. 6. RR improvement for $F_A = 0.5$ vs. $F_A = 1.0$ under conservative and EASY backfilling, for $PECT$, $Prio10$, and $FCFS$ with WC and $E1$ efficiency

Looking at the effect on runtimes, the change from F_A is limited: with $F_A = 1.0$, ϕT is 4.55h for M and 7.3h for L jobs and, with $F_A = 0.5$ and $E1$, ϕT is 5.9h for M and 10.3h for L jobs—which is a factor of 1.3 for M and a factor of 1.41 for L jobs. Predicted would be a factor of 1.625 for both (from $2 * E_{F_A=1.0}/E_{F_A=0.5}) = 2 * 0.65/0.8 = 1.625$) for adaptable jobs. With 25% serial jobs this reduces to a factor of 1.47. However, the lower load, especially for M jobs, also makes more non-type slice backfilling possible which improves service and reduces runtimes more than expected.

4 Adaptive Scheduling with Efficiency and Load Considerations

4.1 Scheduling with Adaptation to Load

As seen in Section 3.4, non-work-conserving scheduling provides significant benefits already from improved efficiency. Note that the scheduler simply re-shaped all jobs. In the following, we put more intelligence into the scheduler (Scojo-PECT) to consider different load situations and create a truly adaptive scheduler. The adaptive scheduler is applied separately and independently per job type M and L. Since $F_A = 0.5$ performed best in the basic experiments, we use this factor in the following experiments with an adaptive scheduler. Adaptation is performed according to the following steps:

1. The resource needs are calculated as the sum of the sizes of all waiting jobs (assuming that ideally all jobs should be allocated). The factor used for adaptation $F_{dyn,A}$ in the dynamic scheduling context is then adjusted as

$$max(F_{try,A}, F_{A,min}) \leq F_{dyn,A} \leq min(F_{try,A}, F_{A,max}) \qquad (6)$$

with $F_{A,min} = 0.5$ and $F_{A,max} = 1.0$ in the following.

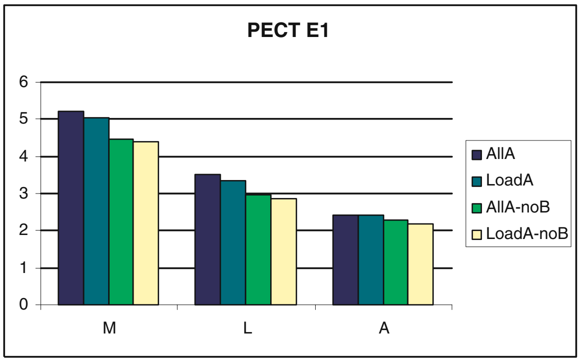


Fig. 7. *RR* for *PECT* with static $F_A = 0.5$, with (*AllA*) and without (*AllA_noB*) application of the same factor to backfilled jobs, and for adaptive *PECT* with dynamic F_{dynA} between $F_{A,min} = 0.5$ and $F_{A,max} = 1.0$, with (*LoadA*) and without (*LoadA_noB*) application to backfilled jobs

2. All jobs of a specific job type which the scheduler tries to fill into the machine, are then reshaped by $F_{dyn,A}$ unless the jobs are relatively long-running in their own jobs class. Thus, if the original job runtime is > 45 min (M jobs) or > 7 h (L jobs), the factor $F_{A,min}$ is used for reshaping.

The results of applying this approach (called *LoadA*) are shown in Figure 7. The improvement is relatively low. However, not adapting backfilled jobs made a difference and best results were obtained if combining both, load adaptive resizing and keeping the original size for backfilling (*LoadA_noB*). However, the improvements are still moderate. (Note that further improvements may be obtained by looking at each scheduling situation in detail to reduce fragmentation. However, previous work [22] suggests that the improvements would be minor.)

The obtained moderate additional improvements with dynamic job-size adaptation suggest that a substantial part of the benefits obtained in previous research with adaptive schedulers may been due to improved efficiency.

5 Job Re-shaping for Virtual Machines of Adaptive Size

In the following, we show results from changing the size of the machine by a factor F_M which corresponds to different resource allocation to virtual machines if partitioning core numbers among them. At the same time, the sizes of the jobs per virtual machines are adjusted by a static factor F_A. We tested resulting virtual-machine sizes of 96, 112, 144, and 160 nodes, using the Lublin-Feitelson workload for 128 nodes and $E1$ efficiency. The results are shown in Figure 8.

The sizes delivering similar results as $F_A = 1.0$ does for 128 nodes are $F_A = 0.5$ (still somewhat higher) for 96 nodes, $F_A = 0.75$ for 112 nodes, $F_A = 1.07$ for 144 nodes, and $F_A = 1.21$ for 160 nodes. The results demonstrate that certain

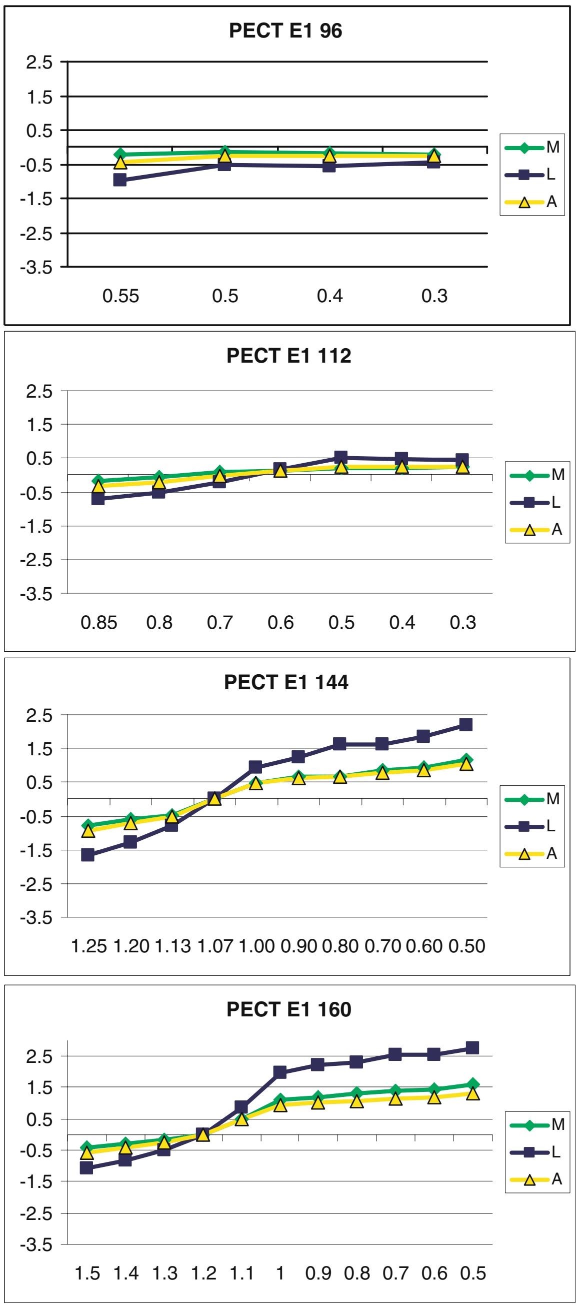


Fig. 8. *RR* with different resizing of the machine combined with different re-shaping of the workload, using *PECT* and *E*1

performance (QoS) can be kept over different resource allocations per virtual machine if re-shaping the jobs accordingly.

The sizes delivering similar results as $F_A = 1.0$ does for 128 nodes are $F_A = 0.5$ (still somewhat higher) for 96 nodes, $F_A = 0.75$ for 112 nodes, $F_A = 1.07$ for 144 nodes, and $F_A = 1.21$ for 160 nodes. The results demonstrate that certain performance (QoS) can be kept over different resource allocations per virtual machine if re-shaping the jobs accordingly.

6 Predicting R under Varying Resource Allocation

6.1 Dependence of R and RR on Utilization

Trying to generalize the performance in dependence on different virtual machine sizes, different job-reshaping factors, and different efficiencies, Figure 9 and Figure 11 plot RR and Figure 10 and Figure 12 plot R in dependence on the measured utilization for the corresponding test runs. Though M and L jobs perform differently, we observe a clear correlation between relative response times and utilization per job type. The correlation is stronger for L jobs but still reasonably clear for M jobs. This permits the conclusion that utilization changes from running jobs at better efficiency are the major source of improvements in adaptive job scheduling and that utilization makes a good predictor for performance if changing virtual-machine and job sizes (cf. (5)).

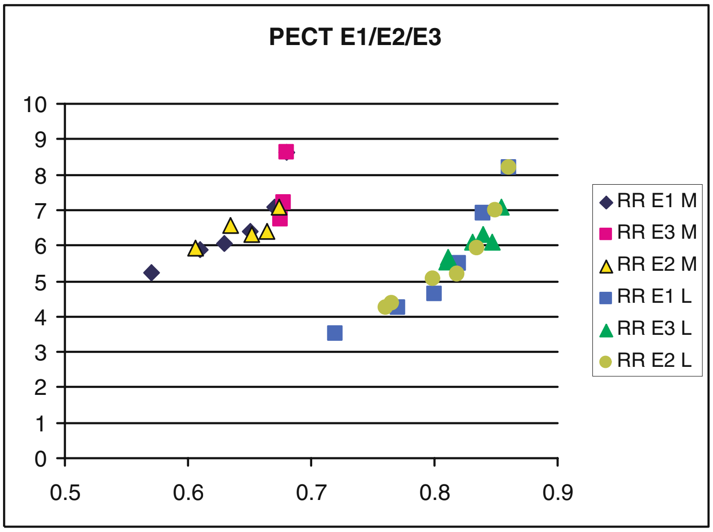


Fig. 9. RR for $PECT$ and $E1$, $E2$, and $E3$ with different F_A, shown over corresponding measured utilization

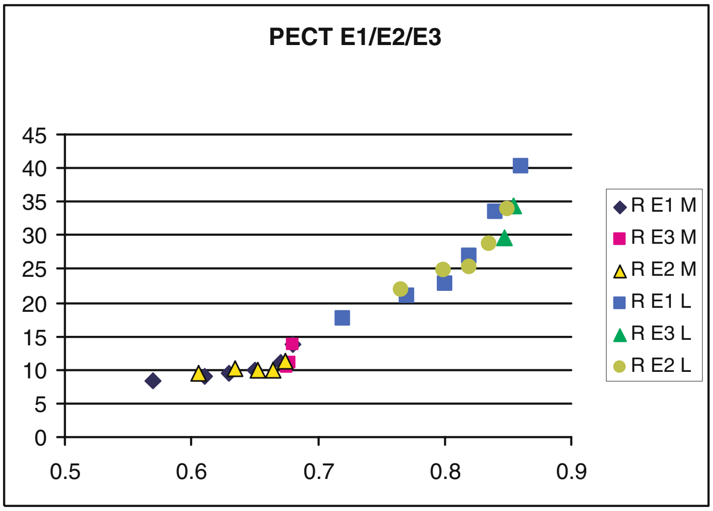


Fig. 10. R for $PECT$ and $E1$, $E2$, and $E3$ with different F_A, shown over corresponding measured utilization

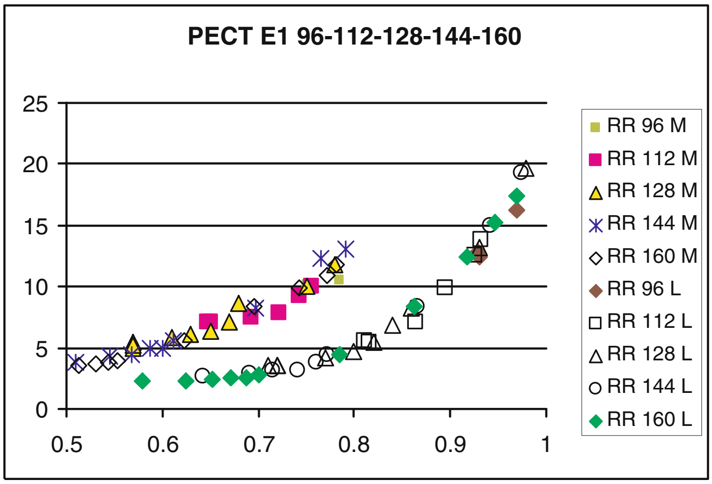


Fig. 11. RR for $PECT$ and $E1$ with different virtual-machine sizes and different F_A, shown over corresponding measured utilization

This is a nice property since utilization is easy to predict if knowing the efficiency changes and the mix of the sizes (especially the percentage of serial jobs), as utilization corresponds to the submitted load (work over time).

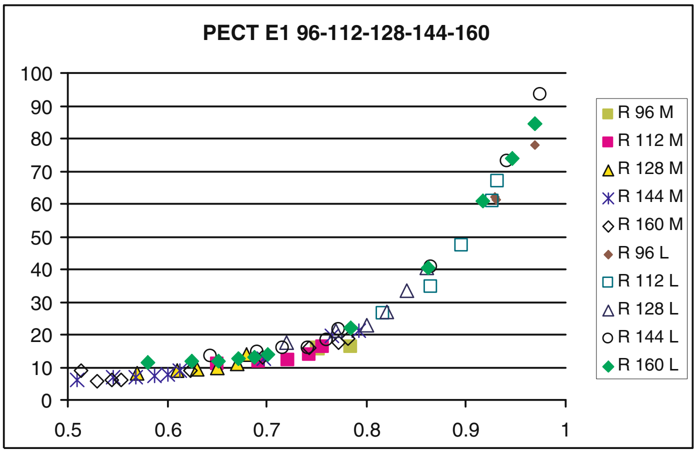


Fig. 12. R for $PECT$ and $E1$ with different virtual-machine sizes and different F_A, shown over corresponding measured utilization

The explanation of the response times not matching the simplified off-line case is that we are dealing with dynamics in interarrival times, daily cycles, job runtimes, and job sizes which can temporarily lead to very long queues. Thus, the proper approach is a queuing model which, however, is hard to establish due to the many contributing statistical distributions.

6.2 A Simple Predictive Model

If knowing queue lengths, response times can be predicted via Little' Law which is independent of the specific statistical distributions and the scheduling policies and says that

$$\phi Q_{system} = \lambda * \phi R \tag{7}$$

with λ being the average arrival rate and ϕQ_{system} the average number of jobs in the system (waiting and running). However, the law applies to a single-server system. We approximate the packing of multiple jobs into the machine as a variation of service time to obtain a single-server model. This simplification appears to be feasible, considering the predictions shown in Figure 13.

The results show a few cases of prediction from the standard model and the exponential interarrival times without daily cycles. As we can see, the predictions are very accurate—a little too high for the standard interarrival model and a little too low for the exponential interarrival model.

Our final goal is predicting response times for different resource allocations F_A and different F_M from a base resource allocation which in our case is $F_A = 1.0$ and $Mz = 128$. As mentioned above, the change in utilization can be directly calculated from the workload. Since the queuing model can capture the multi-node server behavior, we use an M/G/1 model to predict R from the utilization, which means

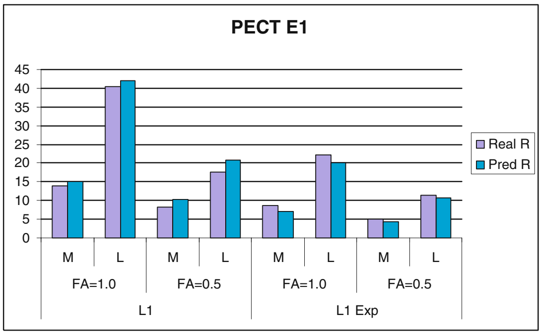


Fig. 13. Real and predicted R (from Q_{system}) for $PECT$ and $E1$ with different F_A. Both standard model and exponentially distributed interarrival times without daily cylces are shown.

$$\phi Q_{system} = U + U^2(1 + C_S^2)/(2 * (1 - U))) \tag{8}$$

With this simplified model, all variations of daily cycles, sizes, backfilling, and runtimes are mapped to the variation coefficient of the service time C_S^2. The concrete parameter is obtained from fitting the curve for $F_A = 1.0$ which reflects predicting from a base-case allocation.

The results from applying this simple prediction approach based on (7) and (8) to our workload are shown in Figure 14 and Figure 15.

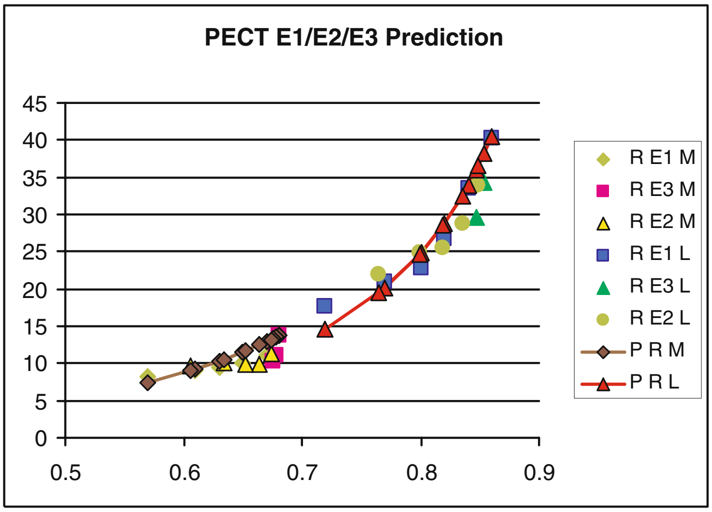


Fig. 14. Real R and predicted P R (from U) for $PECT$ and $E1$ with different F_A, shown over utilization

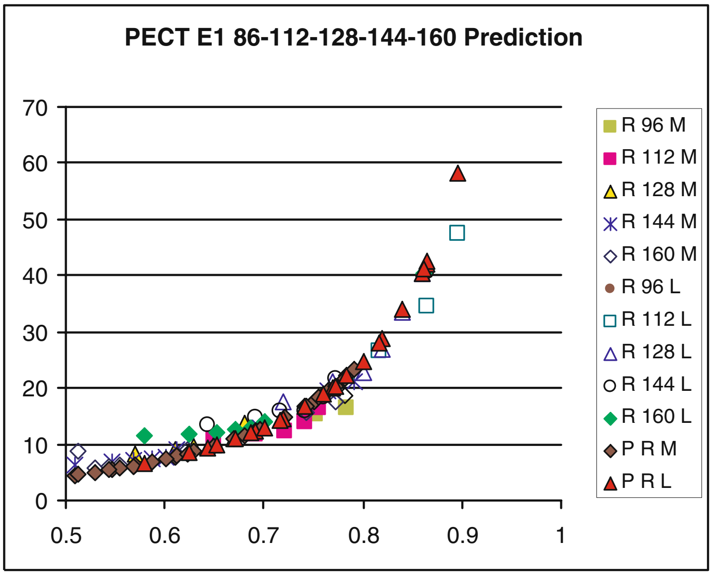


Fig. 15. Real R and predicted $P\ R$ (from U) for $PECT$ with different virtual-machine sizes and different F_A, shown over utilization

Considering that the actual R curves shows some irregularities, the prediction is a good fit (and worked better than all more detailed models tried). However, the curves match a little less well if utilization decreases and would be completely off if utilization increases beyond 0.9. The reason is that variation increases with lower utilization and decreases with very high utilization. For utilization beyond 0.9, the wait times are in the order of days, i.e. daily cycles have little impact, and backfilling opportunities can be expected to saturate. However, the extremely long wait times make such allocations anyway undesirable. Thus, the simple prediction model works well for practically relevant cases.

7 Summary and Conclusion

The paper has presented extensive experiments for investigating the effects on response times that were obtained if merely reshaping all jobs statically to run with smaller sizes (and correspondingly longer runtimes), with or without changing the (virtual) machine size. The results show that under fixed machine size and FCFS scheduling, the results surprisingly remained the same if reshaping the jobs in a work-conserving manner. However, significant benefits were obtained if jobs reshaped to smaller size ran at higher efficiency. Any further benefits obtained from an adaptive scheduler which makes context-dependent decisions and decides job sizes dynamically at job start time according to the machine load were relatively small. Thus, most of the benefit from adaptive schedulers appear to be due to efficiency gains.

Simple formulas looking at the off-line schedules of a series of jobs can only partially explain the effects from different resource allocation but dynamic measurement of system utilization proves to show a strong correlation to the obtained response times under varying virtual-machine and job-size adjustments. A simple queuing model was presented which can, in the range of typical system utilization, effectively predict average response times under varying resource allocation for both different job sizes and different virtual-machine sizes.

References

[1] Barsanti, L., Sodan, A.C.: Adaptive Job Scheduling via Predictive Job Resource Allocation. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2006. LNCS, vol. 4376, pp. 115–140. Springer, Heidelberg (2007)

[2] Chiang, S.-H., Vernon, M.K.: Dynamic vs. Static Quantum-Based Parallel Processor Allocation. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1996 and JSSPP 1996. LNCS, vol. 1162, pp. 200–223. Springer, Heidelberg (1996)

[3] Cirne, W., Berman, F.: When the Herd is Smart-Aggregate Behavior in the selection of Job Request. IEEE Trans. on Par. and Distr. Systems 14(2), 181–192 (2003)

[4] Downey, A.: A Model for Speedup of Parallel Programs. Technical Report CSD-97-933, Univ. of California Berkeley (January 1997)

[5] Esbaugh, B., Sodan, A.C.: Coarse-Grain Time Slicing with Resource-Share Control in Parallel-Job Scheduling. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC 2007. LNCS, vol. 4782, pp. 30–43. Springer, Heidelberg (2007)

[6] Feitelson, D.G., Rudolph, L., Schwiegelsohn, U., Sevcik, K.C., Parsons, W.: 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 Workload Archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html (last retrieved January 2008)

[8] Foster, I., Tuecke, S.: The Different Faces of IT as Service. ACM Queue, 27–34 (July/August 2005)

[9] Ghosal, D., Serazzi, G., Tripathi, S.K.: The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. IEEE Trans. Software Engineering 17(5), 443–453 (1991)

[10] Lublin, U., Feitelson, D.G.: The Workload on Parallel Supercomputers-Modelling the Characteristics of Rigid Jobs. Journal of Parallel and Distributed Computing 63(11), 1105–1122 (2003)

[11] Machina, J., Sodan, A.C.: Predicting Cache Needs and Cache Sensitivity for Applications in Cloud Computing on CMP Servers with Configurable Caches. In: Workshop on System Management Techniques, Processes, and Services (SMTPS) of IPDPS, Proc. IPDPS, Rome, Italy. IEEE, Los Alamitos (2009)

[12] McCann, C., Zahorjan, J.: Processor Allocation Policies for Message Passing Parallel Computers. In: Proc. SIGMETRICS Conf. Measurement & Modeling of Computer Systems, May 1994, pp. 208–219 (1994)

[13] Naik, V.K., Setia, S.K., Squillante, M.K.: Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems. Journal of Parallel and Distributed Computing 46(1), 28–47 (1997)
[14] Parsons, E.W., Sevcik, K.C.: Implementing Multiprocessor Scheduling Disciplines. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 166–192. Springer, Heidelberg (1997)
[15] Rosti, E., Smirni, E., Serazzi, G., Dowdy, L.W.: Analysis of Non-Work-Conserving Processor Partitioning Policies. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 165–181. Springer, Heidelberg (1995)
[16] Sevcik, K.C.: Characterization of Parallelism in Applications and Their Use in Scheduling. Performance Evaluation Review 17, 171–180 (1989)
[17] Sodan, A.C., Machina, J., Deshmeh, A., Macnaughton, K., Esbaugh, B.: Parallelism in Multithreaded and Multicore CPUs. Conditionally accepted for IEEE Computer
[18] Sodan, A.C.: Dynamic Job Scheduling for Computational Grids. In: Grid Computing Research Progress. Nova Science Publisher, Inc., Hauppauge (2008)
[19] Sodan, A.C.: Autonomic Share Allocation and Bounded Prediction of Response Times in Parallel Job Scheduling for Grids. In: Workshop on Adaptive Grid Computing (NCA-AGC), Proc. IEEE Int. Symp. on Network Computing and Applications (NCA), Cambridge, Mass., July 2008, pp. 307–314 (2008)
[20] Sodan, A.: Service Control and Service Prediction with the Preemptive Parallel Job Scheduler Scojo-PECT. Submitted to journal
[21] Sodan, A.C., Lan, L.: LOMARC Lookahead Matchmaking for Multiresource Coscheduling on Hyperthreaded CPUs. IEEE Trans. on Parallel and Distributed Systems 17(11), 1360–1375 (2006)
[22] Sodan, A.C., Huang, X.: Adaptive Time/Space Sharing for Workload Adaptation and Fragmentation Reduction. IJHPCN 4(5/6), 256–269 (2006)
[23] Srinivasan, S., Subramani, V., Kettimuthu, R., Holenarsipur, P., Sadayappan, P.: Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In: Sahni, S.K., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 174–183. Springer, Heidelberg (2002)
[24] Sutter, H.: The Free Lunch is Over-A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (March 2005)
[25] Weinberg, J., Snavely, A.: Symbiotic Space-Sharing on SDSC's DataStar System. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2006. LNCS, vol. 4376, pp. 192–209. Springer, Heidelberg (2007)

Limits of Work-Stealing Scheduling

Željko Vrba[1,2], Håvard Espeland[1,2], Pål Halvorsen[1,2], and Carsten Griwodz[1,2]

[1] Simula Research Laboratory, Oslo
[2] Department of Informatics, University of Oslo

Abstract. The number of applications with many parallel cooperating processes is steadily increasing, and developing efficient runtimes for their execution is an important task. Several frameworks have been developed, such as MapReduce and Dryad, but developing scheduling mechanisms that take into account processing *and* communication requirements is hard. In this paper, we explore the limits of work stealing scheduler, which has empirically been shown to perform well, and evaluate load-balancing based on graph partitioning as an orthogonal approach. All the algorithms are implemented in our Nornir runtime system, and our experiments on a multi-core workstation machine show that the main cause of performance degradation of work stealing is when very little processing time, which we quantify exactly, is performed per message. This is the type of workload in which graph partitioning has the potential to achieve better performance than work-stealing.

1 Introduction

The increase in CPU performance by adding multiple execution units on the same chip, while maintaining or even lowering *sequential* performance, has accelerated the importance of parallel applications. However, it is widely recognized that shared-state concurrency, the prevailing parallel programming paradigm on workstation-class machines, is hard and non-intuitive to use [1]. Message-passing concurrency is an alternative to shared-state concurrency, and it has for a long time been used in distributed computing, and now also in modern parallel program frameworks like MapReduce [2], Oivos [3], and Dryad [4]. However, message passing frameworks also have an increasing importance on multi-core architectures, and such parallel program runtimes are being implemented and ported to single multi-core machines [5–8].

In this context, we have experimented with different methods of scheduling applications defined by process graphs, also named process networks, which explicitly encode parallelism and communication between asynchronously running processes. Our goal is to find an efficient scheduling framework for these multi-core parallel program runtimes. Such a framework should support a wide range of complex applications, possibly using different scheduling mechanisms, and use available cores while taking into account the underlying processor topology, process dependencies and message passing characteristics.

Particularly, in this paper, we have evaluated the *work-stealing* load-balancing method [9], and a method based on *graph partitioning* [10], which balances the load across CPUs, and reduces the amount of inter-CPU communication as well as the cost of migrating processes. Both methods are implemented and tested in Nornir [8], which is our parallel processing runtime for executing parallel programs expressed as Kahn process networks [11].

It has been theoretically proven that the work-stealing algorithm is optimal for scheduling *fully-strict* (also called *fork-join*) computations [12]. Under this assumption, a program running on P processors, 1) achieves P-fold speedup in its parallel part, 2) using at most P times more space than when running on 1 CPU. These results are also supported by experiments [13, 14]. Saha et al. [14] have presented a run-time system aimed towards executing fine-grained concurrent applications. Their simulations show that work-stealing scales well on up to 16 cores, but they have not investigated the impact of parallelism granularity on application performance. Investigation of this factor is one of the contributions of this paper.

In our earlier paper [8] we have noted that careful static assignment of processes to CPUs can match the performance of work-stealing on finely-granular parallel applications. Since static assignment is impractical for large process networks, we have also evaluated an automatic scheduling method based on graph partitioning by Devine et al. [10], which balances the load across CPUs, and reduces the amount of inter-CPU communication as well as the cost of process migration. The contributions of this paper on this topic are two-fold: 1) showing that graph partitioning can sometimes match work-stealing when workload is very fine-grained, and 2) an investigation of *variation* in running time, an aspect neglected by the authors.

Our main observations are that work stealing works nice for a large set of workloads, but orthogonal mechanisms should be available to address the limitations. For example, if the work granularity is small, a graph partitioning scheme should be available, as it shows less performance degradation compared to the work-stealing scheduler. The graph-partitioning scheme succeeds in decreasing the amount of inter-CPU traffic by a factor of up to 7 in comparison with the work-stealing scheduler, but this reduction has no influence on the application running time. Furthermore, applications scheduled with graph-partitioning methods exhibit unpredictable performance, with widely-varying execution times between consecutive runs.

The rest of this paper is structured as follows: in section 2 we describe the two load-balancing strategies and compare our work-stealing implementation to that of Intel's Threading Building Blocks (TBB),[1] which includes an industrial-strength work-stealing implementation. In section 3 we describe our workloads, methodology and present the main results, which we summarize and relate to the findings of Saha et al. [14] in section 4. We conclude in section 5 and discuss broader issues in appendices.

[1] `http://www.threadingbuildingblocks.org/`

2 Dynamic Load-Balancing

We shall describe below the work-stealing and graph-partitioning scheduling methods. We assume an $m : n$ threading model where m user-level **processes** are multiplexed over n kernel-level **threads**, with each thread having its own run queue of ready processes. The affinity of the threads is set such that they execute on different CPUs. While this eliminates interference between Nornir's threads, they will nevertheless share their assigned CPU with other processes in the system, subject to standard Linux scheduling policy.[2]

2.1 Work Stealing

A work-stealing scheduler maintains for each CPU (kernel-level thread) a queue of ready processes waiting for access to the processor. Then, each thread takes ready processes from the *front* of its own queue, and also puts unblocked processes at the front of its queue. When the thread's own run queue is empty, the thread steals a process from the *back* of the run-queue of a randomly chosen thread. The thread loops, yielding the CPU (by calling `sched_yield`) before starting a new iteration, until it succeeds in either taking a process from its own queue, or in stealing a process from another thread. All queue manipulations run in constant-time ($O(1)$), independently of the number of processes in the queues.

The reasons for accessing the run queues at different ends are several [15]: 1) it reduces contention by having stealing threads operate on the opposite end of the queue than the thread they are stealing from; 2) it works better for parallelized divide-and-conquer algorithms which typically generate large chunks of work early, so the older stolen task is likely to further provide more work to the stealing thread; 3) stealing a process also migrates its future workload, which helps to increase locality.

The original work-stealing algorithm uses non-blocking algorithms to implement queue operations [9]. However, we have decided to simplify our scheduler implementation by protecting each run queue with its own lock. We believed that this would not impact scalability on our machine, because others [14] have reported that even a *single, centralized queue* protected by a *single, central lock* does not hurt performance on up to 8 CPUs, which is a decidedly worse situation for scalability as the number of CPUs grows. Since we use locks to protect the run queues, and our networks are static, our implementation does not benefit from the first two advantages of accessing the run queues at different ends. Nevertheless, this helps with increasing locality: since the arrival of a message unblocks a proces, placing it at the front of the ready queue increases probability that the required data will remain in the CPU's caches.

Intel's TBB is a C++ library which implements many parallel data-structures and programming patterns. TBB's internal execution engine is also based on

[2] It is difficult to have fully "idle" system because the kernel spawns some threads for its own purposes. Using POSIX real-time priorities would eliminate most of this interference, but would not represent a realistic use-case.

work-stealing, and it uses a non-blocking queue which employs exponential back-off in case of contention. However, the scheduler is limited to executing only fully-strict computations, which means that a process must run to completion, with the only allowed form of blocking being waiting that children processes exit.[3] Thus, the TBB scheduler is not applicable to running unmodified process networks, where processes can block on message send or receive.

2.2 Graph Partitioning

A graph partitioning algorithm partitions the vertices of a weighted graph into n disjoint partitions of approximately equal weights, while simultaneously minimizing the cut cost.[4] This is an NP-hard problem, so heuristic algorithms have been developed, which find approximate solutions in reasonable time.

We have implemented in Nornir the load-balancing algorithm proposed by Devine et al. [10]. This is one of the first algorithms that takes into account not only load-balance and communication costs, but also costs of process migration. The algorithm observes weights on vertices and edges, which are proportional to the CPU time used by processes and the traffic volume passing across channels. Whenever a significant imbalance in the CPU load is detected, the process graph is repartitioned and the processes are migrated. In our implementation, we have used the state-of-art PaToH library [16] for graph and hypergraph partitioning.

When *rebalancing* is about to take place, the process graph is transformed into an *undirected rebalancing graph*, with weights on vertices and edges set such that the partitioning algorithm minimizes the cost function given by the formula $\alpha t_{comm} + t_{mig}$. Here, α is the number of computation steps performed between two rebalancing operations, t_{comm} is the time the application spends on communication, and t_{mig} is time spent on data migration. Here, α represents a trade-off between good load-balance, small communication and migration costs and rebalancing overheads; see appendix B for a broader discussion in the context of our results.

Constructing the rebalancing graph consists of 4 steps (see also figure 1):

1. Vertex and edge weights of the original graph are initialized according to the collected accounting data.
2. Multiple edges between the same pair of vertices are collapsed into a single edge with weight α times the sum of weights of the original edges.
3. n new, zero-weight nodes, $u_1 \dots u_n$, representing the n CPUs, are introduced. These nodes are fixed to their respective partitions, so the partitioning algorithm will not move them to other partitions.
4. Each node u_k is connected by a *migration edge* to every node v_i iff v_i is a task currently running on CPU k. The weight of the migration edge is set to the cost of migrating data associated with process v_i.

[3] For example, the reference documentation (document no. 315415-001US, rev. 1.13) explicitly warns against using the producer-consumer pattern.

[4] Sum of weights of edges that cross partitions.

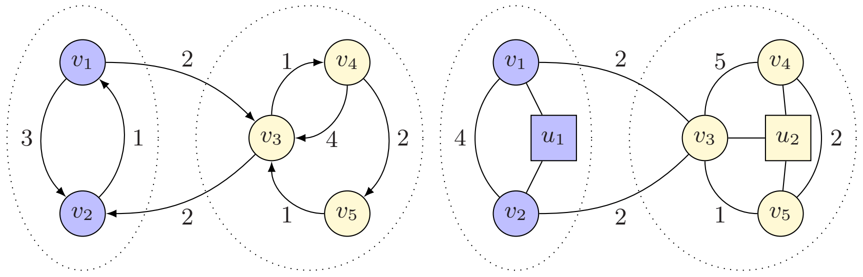


Fig. 1. An example of transforming a process graph into a rebalancing graph with $\alpha = 1$. Current partitions are delimited by ovals and distinguished by nodes of different colors.

For the *initial partitioning* phase, performed before the network starts running, the process graph is transformed into an undirected graph as described above, but with small differences: 1) unit weights are assigned to channels and processes, as the actual CPU usage and communication intensities are not known, and 2) the additional CPU nodes (u_k) and migration edges are omitted. Partitioning this graph gives an initial assignment of processes to CPUs and is a starting point for future repartitions.

Since our test applications have quickly shifting loads, we have implemented a heuristic that attempts to detect load imbalance. The heuristic monitors the idle time τ *collectively* accumulated by all threads, and invokes the load-balancing algorithm when the idle time has crossed a preset threshold. When the algorithm has finished, process and channel accounting data are set to 0, in preparation for the next load-balancing. When a thread's own run-queue is empty, it updates the collective idle time and continues to check the run-queue, yielding (`sched_yield`) between attempts. Whenever *any* thread succeeds in dequeuing a process, it sets the accumulated idle time to 0.

After repartitioning, we avoid bulk migration of processes. It would require locking of all run-queues, migrating processes to their new threads, and unlocking run-queues. The complexity of this task is linear in the number of processes in the system, so threads could be delayed for a relatively long time in dispatching new ready processes, thus decreasing the total throughput. Instead, processes are only reassigned to their new threads by setting a field in their control block, but without physically migrating them. Each thread takes ready processes *only* from its own queue, and if the process's run-queue ID (set by the rebalancing algorithm) matches that of the thread's, the process is run. Otherwise, the process is reinserted into the run-queue to which it has been assigned by the load-balancing algorithm.

3 Comparative Evaluation of Scheduling Methods

We have evaluated the load-balancing methods on several synthetic benchmarks which we implemented and run on Nornir. The programs have been compiled as

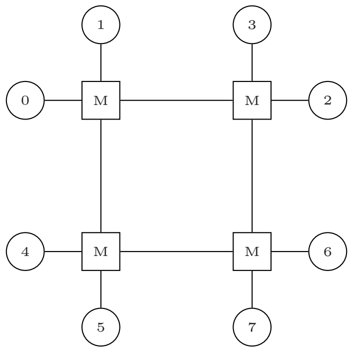


Fig. 2. Topology of the machine used for experiments. Round nodes are cores, square nodes are NUMA memory banks. Each CPU has one memory bank and two cores associated with it.

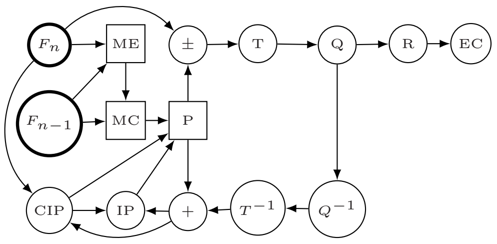
64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). Since PaToH is distributed in binary-only form, these flags have no effect on the efficiency of the graph-partitioning code. The benchmarks have been run on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs (see figure 2), 64 GB of RAM, running linux kernel 2.6.27.3. Each experiment has been repeated 10 consecutive times, with collection of accounting data turned on.

3.1 Description of Workloads

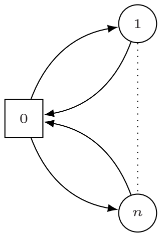
Figure 3(a) shows a process network implementing an **H.264** video-encoder, and it is only a slight adaptation of the encoder block diagram found in [17]. The blocks use an artificial workload consisting of loops which consume the amount of CPU time which would be used by a real codec on average. To gather this data, we have profiled x264, an open-source H.264 encoder, with the *cachegrind* tool and mapped the results to the process graph. Each of P, MC and ME stages has been parallelized as shown in figure 3(b) because they are together using over 50% of the processing time. The number of workers in each of the parallelized stages varies across the set $\{128, 256, 512\}$.

k-means is an iterative algorithm used for partitioning a given set of points in multidimensional space into k groups; it is used in data mining and pattern recognition. To provide a non-trivial load, we have implemented the MapReduce topology as a process network (see Figure 3(c)), and subsequently implemented the Map and Reduce functions to perform the k-means algorithm. The number of processes in each stage has been set to 128, and the workload consists of 300000 randomly-generated integer points contained in the cube $[0, 1000)^3$ to be grouped into 120 clusters.

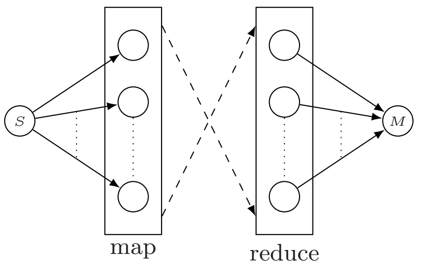
The two **random networks** (see figure 3(e) for an example) are randomly generated directed graphs, possibly containing cycles. To assign work to each process, the workload is determined by the formula nT/d, where n is the number of messages sent by the source, T is a constant that equals ~ 1 second of



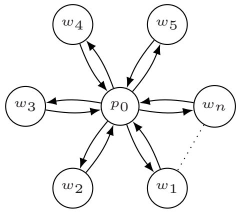
(a) H.264 process network. F_n and F_{n-1} are inputs to the network (current and previous frames)



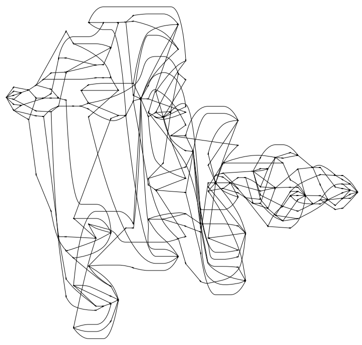
(b) Parallelization of each of P, MC, and ME stages



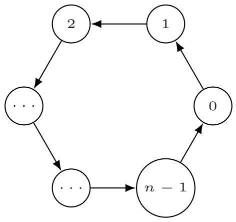
(c) MapReduce topology. Dashed lines denote that each process from the map stage is connected to each process in the reduce stage.



(d) Worker processes connected in a scatter/gather topology to a central process



(e) Random graph with 212 nodes, 333 edges and 13 cycles



(f) Ring topology

Fig. 3. Process networks used for benchmarking

CPU-time, and d is the work division factor. In effect, each message sent by the source (a single integer) carries $w = T/d$ seconds of CPU time. The workload w is distributed in the network (starting from the source process) with each process reading n_i messages from all of its in-edges. Once all messages are read, they are added together to become the t units of CPU-time the process is to consume before distributing t to its n_o forward out-edges. Then, if a process has a back-edge, a message is sent/received, depending on the edge direction, along that channel. As such, the workload w distributed from the source process will equal the workload w collected by the sink process. Messages sent along back-edges do not contribute to the network's workload; their purpose is solely to generate more complex synchronization patterns. We have used two networks: RND-A has 239 nodes, 364 edges and no cycles; RND-B has 213 nodes, 333 edges and 13 cycles. The work division factor has been varied over the set $\{1, 10, \ldots, 10000, 20000, \ldots, 90000\}$.

In the **ring** benchmark, n processes, $0 \ldots n-1$, are created and connected into a ring topology (see figure 3(f)); in our benchmark we have used $n = m = 1000$. Process 0 sends an initial and measures the time it takes to make m round-trips, while other processes just forward messages and do no other processing otherwise.

The **scatter/gather** network has a single central process (p_0) connected to n worker processes (see Figure 3(d)). The central process scatters m messages to the workers, each performing a set amount of work w for each message. When complete, a message is sent from the worker process to the central process, and the procedure is repeated for a given number of iterations. This topology corresponds to the communication patterns that emerge when several MapReduce instances are executed such that the result of the previous MapReduce operation is fed as the input to the next. We have fixed $n = 50$ and varied the work amount $w \in \{1000, 10000, 20000, \ldots, 10^5\}$.

3.2 Methodology

We use real (wall-clock) time to present benchmark results because we deem that it is the most representative metric since it accurately reflects the real time needed for task completion, which is what the end-users are most interested in. We have also measured system and user times (`getrusage`), but do not use them to present our results because 1) they do not reflect the reduced running time with multiple CPUs, and 2) resource usage does not take into account sleep time, which nevertheless may have significant impact on the task completion time.

In the Kahn process network formalism, processes can use only blocking reads and can wait on message arrival only on a single channel at a time. However, to obtain more general results, we have carefully designed the benchmark programs so that they execute correctly even when run-time deadlock detection and resolution is disabled. This is an otherwise key ingredient of a KPN run-time implementation [8], but it would make our observations less general as it would incur overheads not present in most applications.

The benchmarks have been run using 1, 2, 4, 6, and 8 CPUs under the work-stealing (WS) and graph-partitioning policies (GP). For the GP policy, we have varied the idle-time parameter τ (see section 2.2) from 32 to 256 in steps of 8. This has generated a large amount of raw benchmark data which cannot be fully presented in the limited space. We shall thus focus on two aspects: running time and amount of local and remote communication, i.e., the number of messages that have been sent between the processes on the same (resp. different) CPU. For a given combination of the number of CPUs, and work division, we compare the WS policy against the *best* GP policy.

We have evaluated the GP policy by varying the idle time parameter τ over a range of values for each given work division d. A point in the plot for the given d corresponds to the experiment with the median running time belonging to the best value of τ. The details of finding the best τ value are somewhat involved, and are therefore given in appendix.

Computing the median over a set of 10 measurements would generate artificial data points.[5] In order to avoid this, we have discarded the last of the 10 measurements before computing the median.

Since a context-switch also switches stacks, it is to be expected that cached stack data will be quickly lost from CPU caches when there are many processes in the network. We have measured that the cost of re-filling the CPU cache through random accesses increases by $\sim$ 10% for each additional hop on our machine (see figure 2). Due to an implementation detail of Nornir and Linux's default memory allocation policy, which first tries to allocate physical memory from the same node from which the request came, all stack memory would end up being allocated on a single node. Consequently, context-switch cost would depend on the node a process is scheduled on. To average out these effects, we have used the `numactl` utility to run benchmarks under the interleave NUMA (non-uniform memory access) policy, which allocates physical memory pages from CPU nodes in round-robin manner. Since most processes use only a small portion of the stack, we have ensured that their stack size, in the number of pages, is relatively prime to the number of nodes in our machine (4). This ensures that the "top" stack pages of all processes are evenly distributed across CPUs.

3.3 Results

The **ring** benchmark measures scheduling and message-passing overheads of Nornir. Table 1 shows the results for 1000 processes and 1000 round-trips, totalling 10^6 [send $\rightarrow$ context switch $\rightarrow$ receive] transactions. We see that GP performance is fairly constant for any number of CPUs, and that contention over run-queues causes WS performance to drop as the number of CPUs increases from 1 to 2. The peak throughput in the best case (1 CPU, no contention) is $\sim$ 750000 transactions per second. This number is approximately doubled, i.e., transaction cost halved, when accounting mechanisms are turned off. Since detailed accounting data is essential for GP to work, we have run also WS experiments with accounting turned on, so that the two policies can be compared against a common reference point.

The **k-means** program, which executes on a MapReduce topology, is an example of an application that is hard to schedule with automatic graph partitioning.

[5] Median of an *even* number of points is defined as the average of the two middle values.

Table 1. Summary of ring benchmark results (1000 processes and 1000 round-trips). t_n is running time on n CPUs.

	t_1	t_2	t_4	t_6	t_8
GP	1.43	1.65	1.78	1.58	1.71
WS	1.33	2.97	2.59	2.66	2.86

If the idle time parameter τ is too low, repartitioning runs overwhelmingly often, so the program runs *several minutes*, as opposed to 9.4 seconds under the WS method. When the τ is high enough, repartitioning runs only once, and the program finishes in 10.2 seconds. However, the transition between the two behaviours is discontinuous, i.e., as τ is slowly lowered, the behaviour suddenly changes from good to bad. Because of this, we will not consider this benchmark in further discussions.

From figure 4 it can be seen that the WS policy has *the least median running time for most workloads*; it is worse than the GP policy only on the ring benchmark (not shown in the figure; see table 1) and the RND-B network when work division is $d \geq 30000$. At this point, performance of message-passing and scheduling becomes the limiting factor, so the running time increases proportionally with d. On the **H.264** benchmark, the GP policy shows severe degradation in performance as the number of workers and the number of CPUs increases. The root cause of this is the limited parallelism available in the H.264 network; the largest speedup under WS policy is ~ 2.8. Thus, the threads accumulate idle time faster than load-balancing is able to catch-up, so repartitioning and process migration frequently takes place (90 – 410 times per second, depending on the number of CPUs and workers). The former is not only protected by a global lock, but its running time is also proportional with the number of partitions (CPUs) and the number of nodes in the graph, as can be clearly seen in the figure.

As the **RND-B** benchmark is the only case where GP outperforms WS (when $d > 30000$), we have presented further data of interest in figure 5. We can see that WS achieves consistently better peak speedup and at lower d than GP, achieving almost perfect linear scalability with the number of CPUs. Furthermore, we can see that the peak GP speedup has no correlation with peaks and valleys of the proportion of locally sent messages, which constitute over 70% of all message traffic on any number of CPUs. We can also see that the proportion of local traffic under WS decreases proportionally with the increase in the number of CPUs.

Furthermore, we see that WS achieves peak speedup at $d = 100$, which is the largest d value before message throughput starts increasing. Similarly, the peak speedup for GP is at $d = 1000$, which is again at the lower knee of the message throughput curve, except on 8 CPUs where the peak is achieved for $d = 10000$. At $d \geq 30000$, the throughput of messages under the GP policy becomes greater than throughput under the WS policy, which coincides with upper knee of the throughput curve and the point where speedup under GP speedup becomes greater than WS speedup.

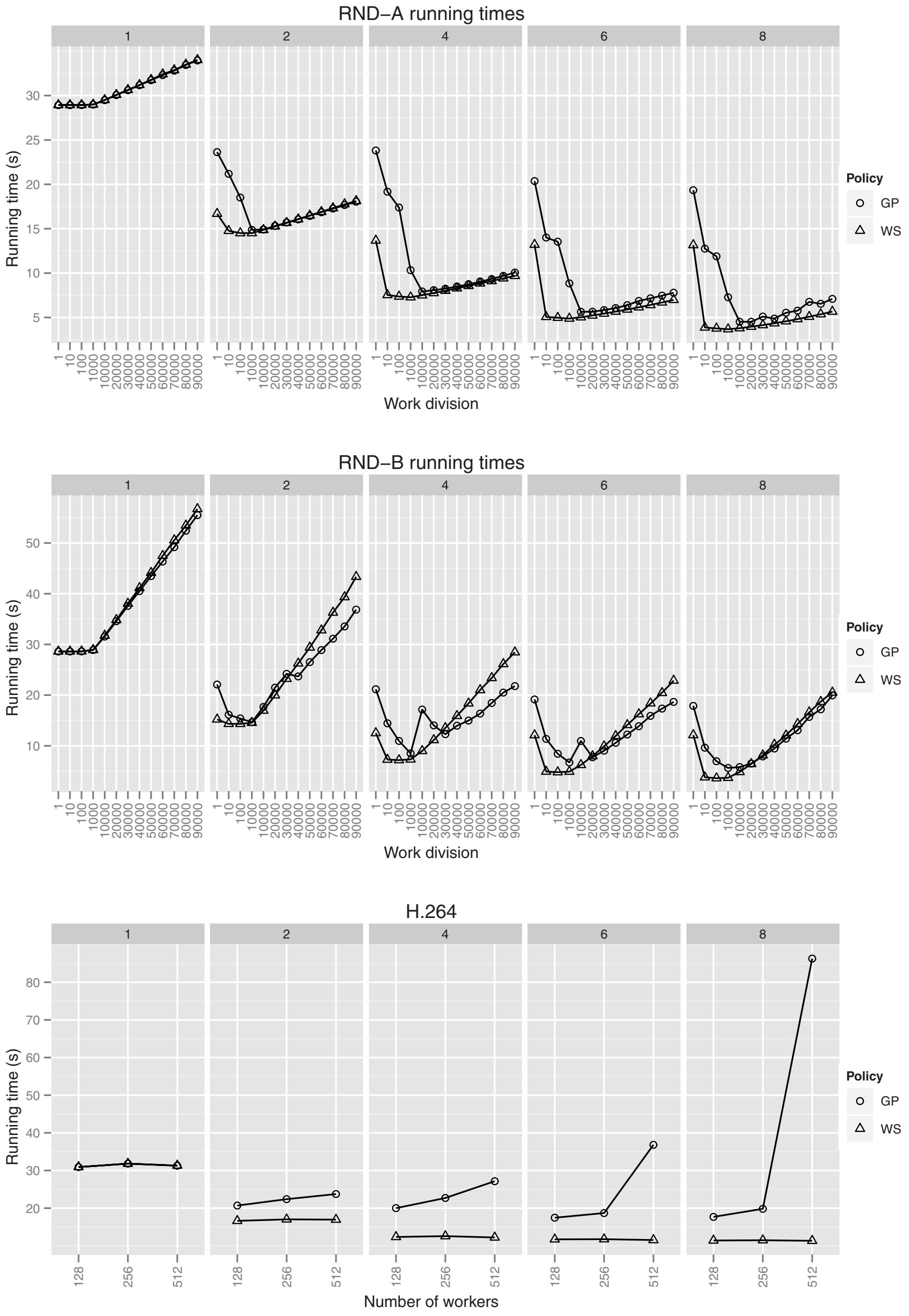


Fig. 4. Median running times for WS and GP policies on 1,2,4,6 and 8 CPUs

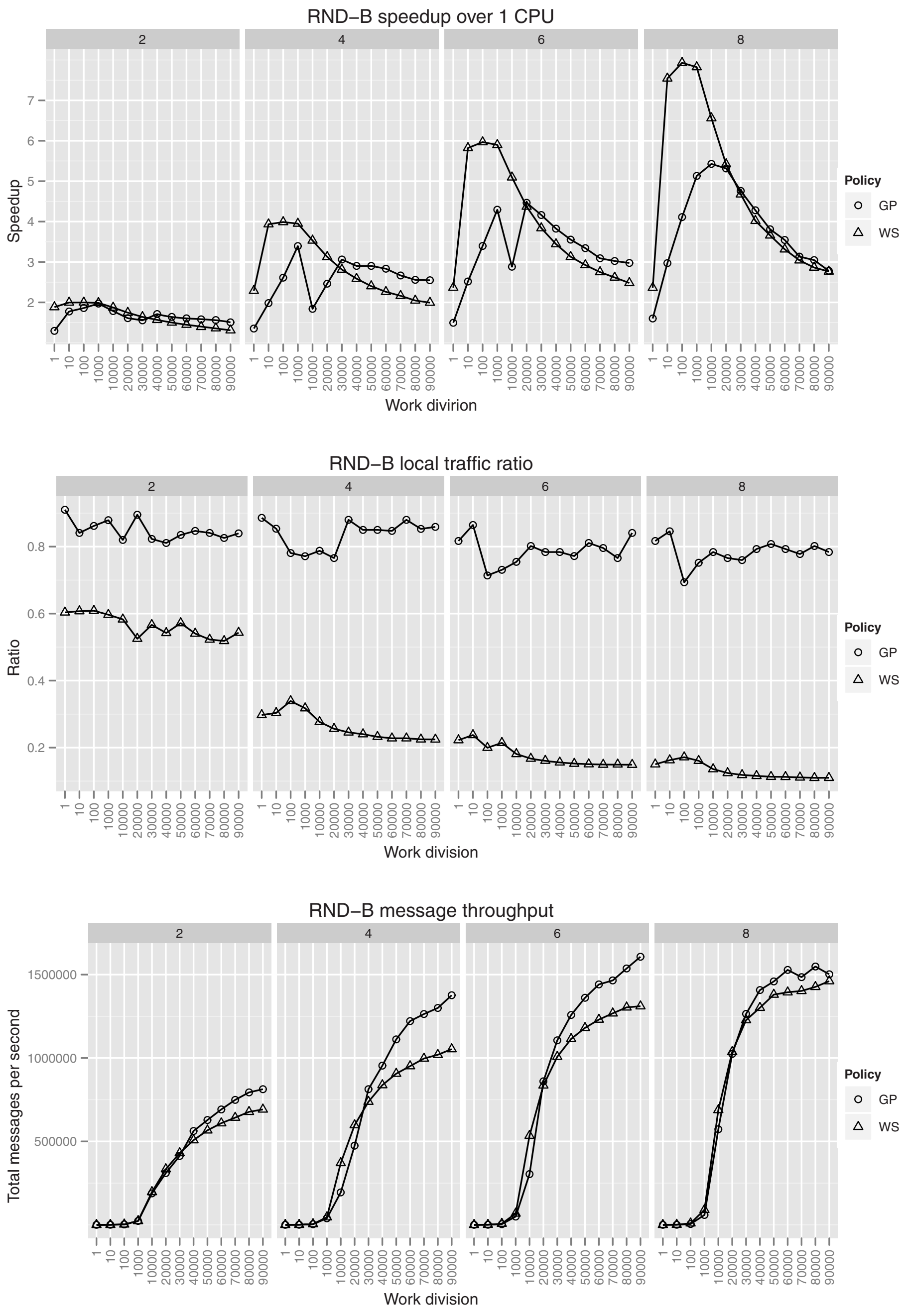


Fig. 5. Further data on RND-B benchmark. The local traffic ratio is calculated as $l/(l+r)$ where l and r are volume of local and remote traffic. Message throughput is calculated as $(l+r)/t$, t being the total running time.

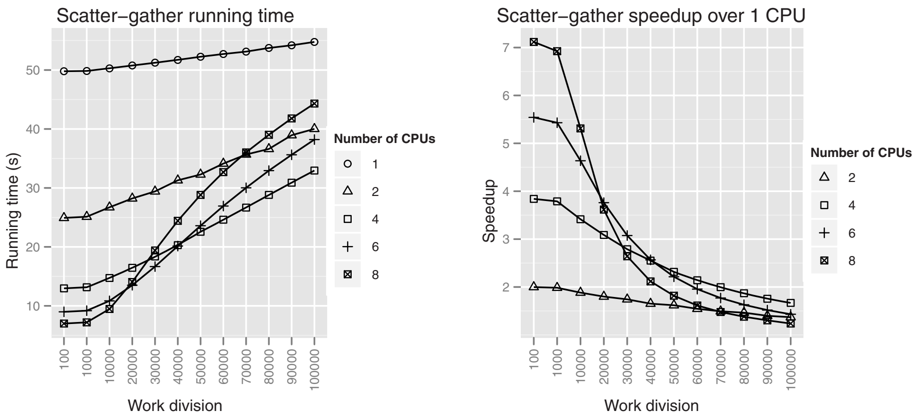


Fig. 6. Performance of the scatter/gather benchmark under WS scheduler. Note that the scale on the x-axis is non-linear and non-uniform.

Figure 6 shows the running time of the **scatter/gather** benchmark for the WS policy on 1 – 8 CPUs. The maximum speedup (7.1 on 8 CPUs) is achieved for $d = 100$ and drops sharply for $d > 1000$; at $d = 20000$ the speedup on 8 CPUs is nearly halved. Thus, we can say that WS performs well as long as each process uses at least $100\mu s$ of CPU time per message, i.e., as long as the computation-to-communication ratio is $\geq\sim 75$. When this ratio is smaller, communication and scheduling overheads overtake, and WS suffers drastic performance degradation.

Figure 7 uses the box-and-whiskers plot[6] to show the distribution of running times and number of repartitionings achieved for all benchmarked τ values of GP policy. The plots show the running time and the number of repartitions for the **RND-B** benchmark on 8 CPUs and $d = 1000$. From the graphs, we can observe several facts:

- WS has undetectable variation in running time (the single line at $\tau = 0$), whereas GP has large variation.
- The number of repartitionings is inversely-proportional with τ, but it has no clear correlation with either variance or median running times.
- When $\tau \geq 72$, the *minimal* achieved running times under the GP policy show rather small variation.

We have thus investigated the relative performance of GP and WS policies, but now considering the *minimal* real running amongst all GP experiments for all values of τ. The graphs (not shown) are very similar to those of figure 5, except that GP speedup is slightly ($< 2\%$ on 8 CPUs) larger.

[6] This is a standard way to show the distribution of a data set. The box's span is from the lower to the upper quartile, with the middle bar denoting the median. Whiskers extend from the box to the lowest and highest measurements that are not outliers. Points denote *outliers*, i.e., measurements that are more than 1.5 times the box's height below the lower or above the upper quartile.

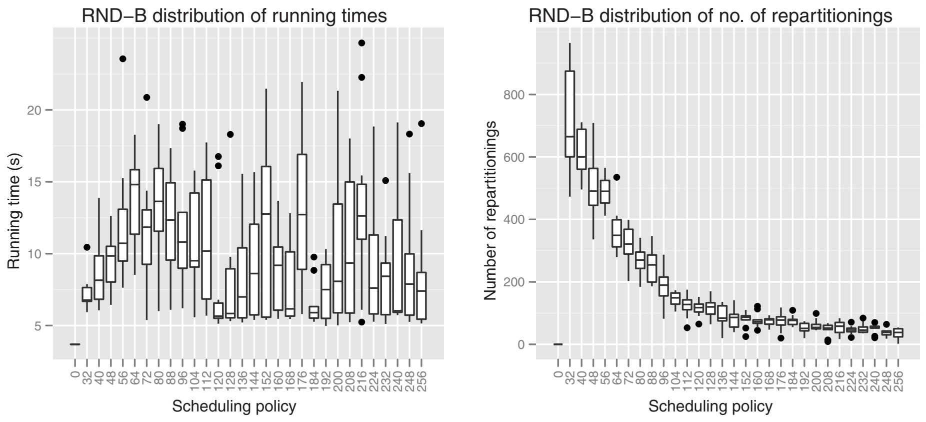


Fig. 7. Illustration of GP variance in running time for RND-B on 8 CPUs and $d = 1000$, which is one step past the WS peak speedup. x-axis is the value of the idle time parameter τ for the GP policy. $\tau = 0$ shows the results for WS.

4 Discussion

In the previous section we have analyzed performance of applications running under graph-partitioning and work-stealing schedulers. WS has been *formally proven* to be optimal *only* for the restricted class of fully-strict computations, but it nevertheless gives best performance also on our benchmark programs, none of which is fully-strict. We can summarize our findings as follows:

– WS gives best performance, with speedup almost linearly proportional with the number of CPUs, *provided* that 1) there is enough parallelism in the network, and 2) the computation to communication ratio, which directly influences scheduling overheads, is at least ~ 75.
– GP and WS show similar patterns in running time, but GP never achieves the same peak speedup as WS.
– There exists an optimal work division d at which the largest speedup is achieved; this granularity is different for WS and GP.
– Increasing d beyond peak speedup leads to a sharp increase in message throughput. This increase quickly degrades performance because message-passing and context-switch overheads dominate the running time.
– GP has large variance in running time; neither the median running time nor its variance is correlated with the idle time parameter τ.
– GP achieves a greater proportion of local traffic than WS, and this ratio falls very slightly with the number of CPUs. The proportion of local traffic under WS falls proportionally with the number of CPUs.
– We have not found any apparent correlation between locality and running time or speedup.

Regarding GP, there are two main obstacles that must be overcome before it can be considered as a viable approach for scheduling general-purpose workloads on multi-processor machines:

- Unpredictable performance, as demonstrated by figure 7.
- The difficulty of automatically choosing the right moment for rebalancing.

As our experiments have shown, monitoring the accumulated idle time over all CPUs and triggering rebalancing after the idle time has grown over a threshold is not a good strategy. Thus, work-stealing should be the algorithm of choice for scheduling general-purpose workloads. Graph partitioning should be reserved for specialized applications using fine-grained parallelism, that in addition have enough knowledge about their workload patterns so that they can "manually" trigger rebalancing.

Saha et al. [14] emphasize that fine-grained parallelism is important in large-scale CMP design, but they have not attempted to *quantify* parallelism granularity. By using a cycle-accurate simulator, they investigated the scalability of work-stealing on a CPU having up to 32 cores, where each core executes 4 hardware threads round-robin. The main finding is that contention over run-queues generated by WS can limit, or even worsen, application performance as new cores are added. In such cases, applications perform better with static partitioning of load with stealing turned off. The authors did not describe how did they partition the load across cores for experiments with work-stealing disabled.

We deem that their results do not give a full picture about WS performance, because contention depends on three additional factors, neither of which is discussed in their paper, and all of which can be used to reduce contention. Contention can be reduced by 1) overdecomposing an application, i.e., increasing the total *number of processes* in the system proportionally with the number of CPUs; by 2) decreasing the number of CPUs to match the *average parallelism* available in the application, which is its intrinsic property; or 3) by increasing the *amount of work* a process performs before it blocks again. The first two factors decrease the probability that a core will find its run-queue empty, and the third factor increases the proportion of useful work performed by a core, during which it does not attempt to engage in stealing.

Indeed, our H.264 benchmark shows that even when the average parallelism is low (only 2.8), the WS running time on 6 and 8 cores does not increase relative to that on 4 CPUs, thanks to overdecomposition. If there were any scalability problems due to contention, the H.264 benchmark would exhibit slowdown similar to that of the ring benchmark.

5 Conclusion and Future Work

In this paper, we have experimentally evaluated performance of two load-balancing algorithms: work-stealing and an algorithm by Devine et al., which is based on graph-partitioning. We have used as the workload a set of synthetic message-passing applications described as directed graphs. Our experimental results confirm the previous results [13, 14] which have reported that WS leads to almost linear increase in performance given enough parallelism, and expand those results by identifying limitations of WS. We have and experimentally found the lower threshold of computation to communication ratio (~ 75), below which the WS performance drops sharply. GP suffers the same performance degradation, but the overall application performance may (depending on the exact workload) be slightly better in this case than under WS. The presented numbers are specific to our implementation; we would expect the threshold of computation to communication ratio to increase under less-efficient implementations of work-stealing, and vice-versa.

GP never achieved the same peak speedup as WS, but this is not its biggest flaw. The main problem with GP is its instability – running times exhibit a great variance, and the ratio of worst and best running time can be more than 4, as can be seen in figure 7. In our research group, we are currently investigating alternative approaches to load-balancing, which would yield results that are more stable than those obtained with today's methods based on graph-partitioning.

As opposed to the large variance of running time under GP, the proportion of local traffic in the total traffic volume is stable and shows only a very mild decrease as the number of CPUs increases. On 8 cores, the proportion of local traffic was *at least* 70%. On the other hand, proportion of local traffic under WS decreases proportionally with the increase in the number of CPUs. On 8 cores, the proportion was *at most* 18%. For most work division factors, GP had ~ 8 times larger proportion of local traffic than GP. Contrary to our expectations, the strong locality of applications running under GP does not have a big impact on the running time on conventional shared-memory architectures. One of possible reasons for this is that our workloads are CPU-bound, but not memory-bound, so GP effectively helps in reducing only the amount of inter-CPU synchronization. However, the overhead of inter-CPU synchronization on our test machine is very low, so the benefits of this reduction become annihilated by the generally worse load-balancing properties of GP.

Nevertheless, we believe that this increase in locality would would lead to significant savings in running time on distributed systems and CPUs with more complex topologies, such as Cell, where inter-CPU communication and process migration are much more expensive than on our test machine. However, practical application of GP load-balancing in these scenarios requires that certain technical problems regarding existing graph partitioning algorithms, described in appendix, be solved.

Ongoing and future activities include evaluations on larger machines with more processors, possibly also on Cell, and looking at scheduling across machines in a distributed setting. Both scenarios have different topologies and interconnection latencies. In a distributed scenario, we envision a two-level scheduling approach where GP will be used to distribute processes across nodes, while WS will be used for load-balancing within a single node.

We have also found weaknesses in an existing, simulation-based results about WS scalability [14]. Based on the the combined insight from theirs and our results, we have identified a new, orthogonal dimension in which we would further like to study WS performance characteristics: the relation between the number of processes in the system and the number of CPUs. Ultimately, we would like to develop an analytical model of WS performance characteristics, which would take into consideration the number of processes in the system, work granularity (which is inversely proportional with the amount of time a CPU core spends on useful work), as well as the system's interconnection topology.

Acknowledgments

We thank to Paul Beskow for fruitful discussions and proofreading the paper, and to the anonymous reviewers for their very helpful comments.

References

1. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI), Berkeley, CA, USA, p. 10. USENIX Association (2004)
3. Valvag, S.V., Johansen, D.: Oivos: Simple and efficient distributed data processing. In: 10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC 2008, September 2008, pp. 113–122 (2008)
4. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 59–72. ACM, New York (2007)
5. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
6. de Kruijf, M., Sankaralingam, K.: MapReduce for the Cell BE Architecture. University of Wisconsin Computer Sciences Technical Report CS-TR-2007 1625 (2007)
7. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: PACT 2008: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pp. 260–269. ACM, New York (2008)
8. Vrba, Ž., Halvorsen, P., Griwodz, C.: Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors. In: Proceedings of the International Workshop on Multi-Core Computing Systems, MuCoCoS (2009)
9. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA), pp. 119–129. ACM, New York (1998)
10. Catalyurek, U., Boman, E., Devine, K., Bozdag, D., Heaphy, R., Riesen, L.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE, Los Alamitos (2007); Best Algorithms Paper Award
11. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing 74 (1974)

12. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA (1996)
13. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). SIGMETRICS Perform. Eval. Rev. 26(1), 266–267 (1998)
14. Saha, B., Adl-Tabatabai, A.R., Ghuloum, A., Rajagopalan, M., Hudson, R.L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling scalability and performance in a large scale cmp environment. SIGOPS Oper. Syst. Rev. 41(3), 73–86 (2007)
15. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Quebec, Canada, June 1998, pp. 212–223 (1998); Proceedings published ACM SIGPLAN Notices, Vol. 33(5) (May 1998)
16. Catalyurek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems 10(7), 673–693 (1999)
17. Richardson, I.E.G.: H.264/mpeg-4 part 10 white paper, `http://www.vcodex.com/files/h264_overview_orig.pdf`
18. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics (1947)
19. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Comput. 34(6-8), 318–331 (2008)

A Selecting the Best GP Result Set

To evaluate the GP policy, we have varied the idle time parameter τ over a range of values for each given work division d. A single result set consists of 10 measurements for a given combination of d and τ. We cannot choose the best τ for a given d by simply selecting the τ having the lowest average (or median) running time. The reason is that consecutive runs of an experiment with the *same* combination of d and τ can have high variance.

To choose the best τ for a given d, we compare all result sets against each other and select the set $s(\tau)$ which compares smaller against the largest number of other sets. Formally, for a fixed d, we choose τ as follows:

$$\tau = \min(\arg\max_{\tau \in T} |\{\tau' \in T : (\tau' \neq \tau) \wedge (s(\tau) < s(\tau'))\}|)$$

where $|X|$ denotes cardinality of set X, $T = \{8, 16, \ldots, 256\}$ is the set of τ values over which we evaluated the GP policy, and $s(\tau)$ is the result set obtained for the given τ. To compare two result sets, we have used the one-sided Mann-Whitney U-test [18][7] with 95% confidence level; whenever the test for $s(\tau)$ against $s(\tau')$ reported a p-value less than 0.05, we considered that the result set $s(\tau)$ comes from a distribution with stochastically smaller [18] running time.

[7] Usually, the Student's t-test is used. However, it makes two assumptions, for which we do not know whether they are satisfied: 1) that the two samples come from a normal distribution 2) having the same variance.

B Migration Cost

α is just a scale factor expressing the relative costs of communication and migration, so it can be fixed to 1, and t_{mig} scaled appropriately; in our experiments we have used $\alpha = 1$ and $t_{mig} = 16$. This is an arbitrary low value reflecting the fact that our workloads have low migration cost because they have very light memory footprint. Since load-imbalance is the primary reason for bad performance of GP, another value for t_{mig} would not significantly influence the results because graph partitioning *first* establishes load-balance and *then* tries to reduce the cut cost. Nevertheless, even such a low value succeeds in preventing that a process with low communication volume and CPU usage is needlessly migrated to another CPU.

Workloads with a heavy memory footprint could benefit if the weight of their migration edges is a *decreasing* function $c(t_b)$ of the amount of time t_b a process has been blocked. As t_b increases, the probability that CPU caches will still contain relevant data for the given process decreases, and the cost of migrating this process becomes lower.

C NUMA Effects and Distributed Process Networks

We have realized that the graph-partitioning model described in section 2 does not always adequately model application behavior on NUMA architectures because it assumes that processes migrate to a new node together with their data. However, NUMA allows that processes and their data reside on separate nodes, which is also the case in our implementation. Nevertheless, the model describes well applications that use NUMA APIs to physically migrate their data to a new node. Furthermore, the graph-partitioning algorithm assumes that the communication cost between two processes is constant, regardless of the CPUs to which they are assigned. This is not true in general: for example, the cost of communication will be $\sim 10\%$ bigger when the processes are placed on CPUs 0 and 7 than when placed on CPUs 0 and 2.

These observations affect very little our findings because of three reasons: 1) the workloads use little memory bandwidth, so their performance is limited by message-passing, context-switch and inter-CPU synchronization overheads, 2) NUMA effects are averaged out by round-robin allocation of physical pages across all 4 nodes, 3) synchronization cost between processes assigned to the same CPU is minimal since contention is impossible in this case.

In a distributed setting, load-balancing based on graph models is relevant because of several significant factors: processes *must* be migrated together with their data, high cost of data migration, and high cost of communication between processes on different machines. Indeed, we have chosen to implement Devine's et.al. algorithm [10] because they have measured improvement in application performance in a distributed setting. The same algorithm is applicable to running other distributed frameworks, such as MapReduce or Dryad.

D Graph Partitioning: Experiences and Issues

Nornir measures CPU consumption in nanoseconds. This quickly generates large numbers which PaToH partitioner [16] used in our experiments cannot handle, so it exits with an error message about detected integer overflow. Dividing all vertex weights by the smallest weight did not work, because it would still happen that the resulting weights are too large. To handle this situation, we had two choices: run the partitioning algorithm more often, or aggressively scale down all vertex weights. The first choice made it impossible to experiment with infrequent repartitionings, so we have used the other option in our experiments: all vertex weights have been transformed by the formula $w' = w/1024 + 1$ before being handed over to the graph partitioner. This loss of precision, however, causes an *a priori* imbalance on input to the partitioner, so the generated partitions have worse balance than would be achievable if PaToH internally worked with 64-bit integers. This rounding error may have contributed to the limited performance of GP load-balancing, but we cannot easily determine to what degree.

As exemplified above, the true weight of an edge between two vertices in the process graph may depend on which CPUs the two processes are mapped to. This issue is addressed by *graph mapping* algorithms implemented in, e.g., the Scotch library [19]. However, SCOTCH does does not support pinning of vertices to given partitions, which is the essential ingredient of Devine's algorithm. On the other hand, PaToH supports pinning of vertices, but does not solve the mapping problem, i.e., it assumes that the target graph is a complete graph with equal weights on all edges. Developing algorithms that support both pinned vertices *and* solve the mapping problem is one possible direction for future research in the area of graph partitioning and mapping algorithms.

Author Index

Bansal, Nikhil 169
Birkenheuer, Georg 80
Bobroff, Norman 190
Brinkmann, André 80
Bunde, David P. 116

Cao, Yangjie 207
Colling, David 37
Coppinger, Richard 190
Corbalan, Julita 59

Espeland, Håvard 280

Fölling, Alexander 16
Fong, Liana 190
Fox, Geoffrey C. 1

Gong, Yili 1
Grimme, Christian 16
Griwodz, Carsten 280
Guim, Francesc 59

Halvorsen, Pål 280
Hildrum, Kirsten 169
Hsu, Wen-Jing 207

Karl, Holger 80
Kaur, Manpreet 157

Lepping, Joachim 16
Leung, Vitus J. 116
Lingrand, Diane 37

Martyniak, Janusz 37
Miguel-Alonso, Jose 138
Montagnat, Johan 37

Navaridas, Javier 138

Papaspyrou, Alexander 16
Parekh, Sujay 169
Pascual, Jose Antonio 138
Pierce, Marlon E. 1

Rajan, Deepak 169
Rodero, Ivan 59

Seelam, Seetharami 190
Sinnen, Oliver 157
Sodan, Angela C. 232, 259
Sun, Hongyang 207

Thebe, Ojaswirajanya 116
To, Andrea 157
Tran Ngoc, Minh 101

Vrba, Željko 280

Wagle, Rohit 169
Wolf, Joel 169
Wolters, Lex 101
Wu, Kun-Lung 169

Xu, Jing 190

Zeng, Xijie 232