

See discussions, stats, and author profiles for this publication at: <http://www.researchgate.net/publication/222659044>

Using Moldability to Improve the Performance of Supercomputer Jobs

ARTICLE *in* JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING · OCTOBER 2002

Impact Factor: 1.18 · DOI: 10.1006/jpdc.2002.1869 · Source: DBLP

CITATIONS

48

READS

28

2 AUTHORS, INCLUDING:



Walfredo Cirne

Google Inc.

117 PUBLICATIONS 2,501 CITATIONS

SEE PROFILE

Using Moldability to Improve the Performance of Supercomputer Jobs

PhD Thesis

Walfredo Cirne

A Deus, por tudo.

À Zane, pelo amor.

À Helena, pelo futuro.

À Poema, pelo amor e doçura.

A Papai e Mamãe, pelo amor e dedicação.

To Fran, for not letting me get away with sloppy thinking.

À Capes e UFPB, pelo apoio.

Abstract

Distributed-memory parallel supercomputers are an important platform for the execution of high-performance parallel jobs. In order to submit a job for execution in most supercomputers, one has to specify the number of processors to be allocated to the job. However, most parallel jobs in production today are moldable. A job is moldable when the number of processors it needs to execute can vary, although such a number has to be fixed before the job starts executing. Consequently, users have to decide how many processors to request whenever they submit a moldable job.

In this thesis, we show that **the request that submits a moldable job can be automatically selected in a way that often reduces the job's turn-around time**. The turn-around time of a job is the time elapsed between the job's submission and its completion.

More precisely, we will introduce and evaluate **SA**, an application scheduler that chooses which request to use to submit a moldable job on behalf of the user. The user provides **SA** with a set of possible requests that can be used to submit a given moldable job. **SA** estimates the turn-around time of each request based on the current state of the supercomputer, and then forwards to the supercomputer the request with the smallest expected turn-around time.

Users are thus relieved by **SA** of a task unrelated with their final goals, namely that of selecting which request to use. Moreover and more importantly, **SA** often improves the turn-around time of the job under a variety of conditions. The conditions under which **SA** was studied cover variations on the characteristics of the job, the state of the supercomputer, and the information available to **SA**. The emergent behavior generated by having most jobs using **SA** to craft their requests was also investigated.

Table of Contents

1. INTRODUCTION	1
1.1. PARALLEL SUPERCOMPUTERS	1
1.2. JOB MOLDABILITY	3
1.3. THESIS SUMMARY	4
1.4. THESIS OUTLINE	6
2. SA: THE SUPERCOMPUTER APPLES	7
2.1. GENERIC SA	8
2.2. SA OVER CONSERVATIVE BACKFILLING	9
2.2.1. CONSERVATIVE BACKFILLING	9
2.2.2. SA OVER CONSERVATIVE BACKFILLING	11
2.3. EQUIVALENCE BETWEEN THE TWO VERSIONS OF SA	13
3. WORKLOAD MODELS	15
3.1. COLLECTING INFORMATION	15
3.2. MODELS OVERVIEW	17
3.3. RIGID WORKLOAD MODEL	18
3.3.1. INSTANT OF ARRIVAL	18
3.3.2. CANCELLED JOBS	23
3.3.3. PARTITION SIZE	26
3.3.4. EXECUTION TIME AND REQUESTED TIME	32
3.4. MOLDABILITY MODEL	38
3.4.1. PARTITION SIZES	39
3.4.2. ACCURACY	42

3.4.3. REQUEST TIME	43
3.5. MODELS SUMMARY	50
<u>4. PERFORMANCE METRICS FOR SA</u>	<u>55</u>
4.1. GAUGING JOB PERFORMANCE	55
4.2. AGGREGATING EXPERIMENTS	55
<u>5. THE PERFORMANCE OF SA</u>	<u>59</u>
5.1. EXPERIMENTAL SET-UP	59
5.2. OVERALL PERFORMANCE	61
5.3. FACTORS THAT INFLUENCE SA	63
5.3.1. JOB CHARACTERISTICS	64
5.3.2. INFORMATION AVAILABLE TO SA	75
5.3.3. THE STATE OF THE SUPERCOMPUTER	77
5.4. VALIDATING THE RESULTS	82
<u>6. EMERGENT BEHAVIOR OF MULTIPLE SAS</u>	<u>86</u>
6.1. PERFORMANCE IMPACT OF EMERGENT BEHAVIOR	87
6.2. INCREASING THE OFFERED LOAD	95
6.2.1. DIRECT INCREASE OF THE OFFERED LOAD	96
6.2.2. LARGE- σ WORKLOADS	98
<u>7. RELATED AND FUTURE WORK</u>	<u>101</u>
7.1. RELATED WORK	101
7.1.1. SUPERCOMPUTER SCHEDULING	101
7.1.2. APPLICATION SCHEDULING	104
7.1.3. EMERGENT BEHAVIOR	110
7.2. FUTURE WORK	112

<u>8. SUMMARY</u>	<u>115</u>
<u>ACKNOWLEDGMENTS</u>	<u>119</u>
<u>A. SURVEY'S QUESTIONNAIRE</u>	<u>120</u>
<u>B. SURVEY'S RESULTS</u>	<u>124</u>
<u>C. EMERGENT BEHAVIOR RESULTS</u>	<u>131</u>
<u>D. MOLDABLE BACKFILLING RESULTS</u>	<u>135</u>
<u>REFERENCES</u>	<u>139</u>

List of Figures

Figure 1 – Users selecting requests to submit their jobs.....	3
Figure 2 – Request selection by SA.....	5
Figure 3 – Allocation list after the submission of five requests.....	10
Figure 4 – Allocation list after the backfilling initiated by A finishing at time 2.....	10
Figure 5 – SAcB pseudo-code	11
Figure 6 – SAcB example: availability list.....	12
Figure 7 – SAcB example: schedule A	12
Figure 8 – SAcB example: schedule B	13
Figure 9 – SAcB example: schedule C	13
Figure 10 – Histograms of arrival hour for our reference workloads	19
Figure 11 – Observed data and fitted model for arrival rate.....	23
Figure 12 – Histograms of cancellation lag	25
Figure 13 – Cancellation Lag CDF	26
Figure 14 – Uniform-log fit for partition size.....	27
Figure 15 – Histogram of partition sizes.....	28
Figure 16 – Survey results for the constraints jobs face regarding partition size	29
Figure 17 - Observed data and statistical model for partition size (ANL workload).....	30
Figure 18 - Observed data and statistical model for partition size (CTC workload).....	30
Figure 19 - Observed data and statistical model for partition size (KTH workload).....	31
Figure 20 - Observed data and statistical model for partition size (SDSC workload)....	31
Figure 21 – Accuracy \times Execution Time	33
Figure 22 – Accuracy \times Requested Time.....	34
Figure 23 – Distribution of the accuracy a for the completed jobs.....	35

Figure 24 – Observed data and statistical model for accuracy	36
Figure 25 – Observed data and statistical model for requested time	38
Figure 26 – Survey results for minimum partition size	40
Figure 27 – Survey results and model for the minimum partition size c_{min}	40
Figure 28 – Survey results for number of requests used to submit a job	41
Figure 29 – Survey results and statistic model for $c_u > 1$	42
Figure 30 – Downey’s speed-up function $S(n, A, \sigma)$ for different values of A	44
Figure 31 – Downey’s speed-up function $S(n, A, \sigma)$ for different values of σ	44
Figure 32 – Survey results for efficient partition size s_{effic}	46
Figure 33 – Distribution of s_{effic} for the responses with s_{min} in the $[11,30]$ range	47
Figure 34 – Joint CDF for s_{effic} and s_{min}	48
Figure 35 – Model CDF for the joint distribution of c_{min} and A	48
Figure 36 – Distribution of the survey-based σ estimates	49
Figure 37 – CDF for the survey based estimate of σ and the corresponding model	50
Figure 38 – Distribution of relative turn-around time	63
Figure 39 – Turn-around time by the sequential execution time L	65
Figure 40 – Execution time by the sequential execution time L	67
Figure 41 – Wait time by the sequential execution time L	68
Figure 42 – Turn-around time by average parallelism A	69
Figure 43 – Execution time by average parallelism A	69
Figure 44 – Turn-around time by σ	70
Figure 45 – Turn-around time by minimum partition size c_{min}	71
Figure 46 – Wait time by minimum partition size c_{min}	72
Figure 47 – Turn-around time by maximum partition size c_{max}	73
Figure 48 – Execution time by maximum partition size c_{max}	73
Figure 49 – Turn-around time by kind of partition size c_{kind}	74

Figure 50 – Turn-around time by accuracy a	76
Figure 51 – Turn-around time by number of requests v	76
Figure 52 – Turn-around time by load per processor D	79
Figure 53 – Wait time by load per processor D	80
Figure 54 – Execution time by load per processor D	80
Figure 55 – Requested computation time by load per processor D	81
Figure 56 – Wait time over turn-around time as the load per processor grows	82
Figure 57 – NAS results by kind of benchmark.....	85
Figure 58 – Requested computation time by load per processor D	90
Figure 59 - Distribution of the load per processor D	91
Figure 60 – Turn-around time by load per processor D	92
Figure 61 – Execution time by load per processor D	93
Figure 62 – Wait time by load per processor D	93
Figure 63 – The effect of offered load	97
Figure 64 – Distributions of σ	99
Figure 65 – Offered load as a function of κ_σ	100
Figure 66 – The influence of the distribution of σ on the emergent behavior of SA ...	100
Figure 67 – Different kinds of schedulers found in a computational grid	106
Figure 68 – The structure of an AppLeS.....	107

List of Tables

Table 1 – Workloads used in this research	16
Table 2 – Amount of days eliminated from the experimental data.....	21
Table 3 – Coefficients of the polynomials that model job arrival rate	22
Table 4 – Percentage of completed and cancelled jobs	24
Table 5 – χ_n and ρ_n obtained by fitting partition size to a uniform-log distribution.....	26
Table 6 – Percentage of jobs with a power-of-2 partition size	27
Table 7 – Percentages of different kinds of non-power-of-2 jobs	32
Table 8 – Correlation coefficient between accuracy and execution time $r(a, te)$ and correlation coefficient between accuracy and requested time $r(a, tr)$	32
Table 9 – α_a and β_a obtained by fitting accuracy to a gamma distribution	36
Table 10 – χ_{tr} and ρ_{tr} obtained by fitting requested time to a uniform-log distribution .	37
Table 11 – Summary of the Rigid Workload Model	52
Table 12 – Summary of the Moldability Model	54
Table 13 – Overall results (in seconds).....	61
Table 14 – How the adaptive requests impacted on the turn-around time.....	61
Table 15 – NAS benchmarks used in the validation experiments	83
Table 16 – Overall NAS results (in seconds).....	84
Table 17 – Scenarios simulated in the emergent behavior experiments	88
Table 18 – Overall results with SA scheduling all jobs (in seconds)	89

1. Introduction

Performance is a very important aspect of computer systems. This has been true since the very birth of modern computers, when computers were seen as calculators whose *raison d'être* was their ability to perform arithmetic operations faster than the human being. Today, performance remains a major concern for both industry and academia.

A common approach to improve performance is parallelism. A *parallel job* (or simply *job* throughout this thesis) is composed by many *tasks* that execute simultaneously on multiple processors. By using more than processor, a parallel job can run faster than its sequential counterpart.

1.1. Parallel Supercomputers

Distributed-memory parallel supercomputers (or simply *parallel supercomputers* or even *supercomputers* in this thesis) are high-end machines designed to support the execution of parallel jobs. A parallel supercomputer is composed of many processors, each with its own memory. The processors are interconnected by very fast internal networking. Some supercomputers implement distributed shared-memory schemes (e.g., the SGI Origin 2000). But this is only for the convenience of the application developer. Under the hood, shared-memory is implemented using message-passing (possibly followed by remapping memory pages).

In order to promote the performance of parallel jobs whose tasks frequently communicate and synchronize, parallel supercomputers are typically *space-shared*. That is, jobs receive a dedicated *partition* to run for a pre-established amount of time. Note that having a dedicated partition greatly simplifies work distribution concerns. It reduces work distribution to balancing the load across the processors. Although this often

is a hard task by itself, it is surely easier than work distribution in a dynamically-changing non-dedicated heterogeneous environment.

Supercomputer Scheduling

Since jobs have dedicated access to processors in a spaced-shared supercomputer, an arriving job may not find enough resources to execute immediately. When this happens, the arriving job waits until enough processors become available. More precisely, jobs that cannot start immediately are placed in a *queue*, which is controlled by the supercomputer scheduler. The *supercomputer scheduler* is the entity that receives requests to run jobs. It decides when jobs start and what processors they use. In particular, the supercomputer scheduler decides which job in the wait queue is the next to run. In order to make this decision, it typically requires each job to specify n , the number of processors it needs, and tr , the time requested for execution of the job. In the current practice, the supercomputer scheduler enforces the request time tr . That is, a job is killed if it exceeds its request time tr .

Note that supercomputer scheduling is an *on-line scheduling problem* [44]. An *on-line scheduler* deals with jobs that continually arrive to the system. In contrast, an *off-line scheduler* assumes that all jobs are available from the outset. Off-line scheduling is more amenable to analytical solutions and there is a great deal of research in the area [35]. However, the results of these investigations often cannot be applied to the on-line problem.

There are a handful of supercomputer schedulers currently in production. These include the Easy [66] [82], PBS [58], Maui [69], and LSF [74] schedulers. Unfortunately, these schedulers can radically change their behavior depending on how they have been configured for a given system, which makes characterizing them a very complex task. There are also numerous simulation-based studies on queue disciplines for supercomputer schedulers. For a nice survey on the area, we refer the reader to [40].

1.2. Job Moldability

As we shall see in detail on Chapter 3, most parallel jobs in production today seem to be moldable. A *moldable job* can run on multiple partition sizes [40]. However, supercomputer schedulers accept only *static requests*. A *static request* can be characterized by the partition size n and the request time tr . In particular, the partition size n determines unequivocally how many processors are allocated to the job being submitted. Since moldable jobs can use multiple partition sizes, there are multiple different requests that can be used to submit a given moldable job. In current practice, the users choose which request to use at the submission of their jobs, as illustrated by Figure 1.

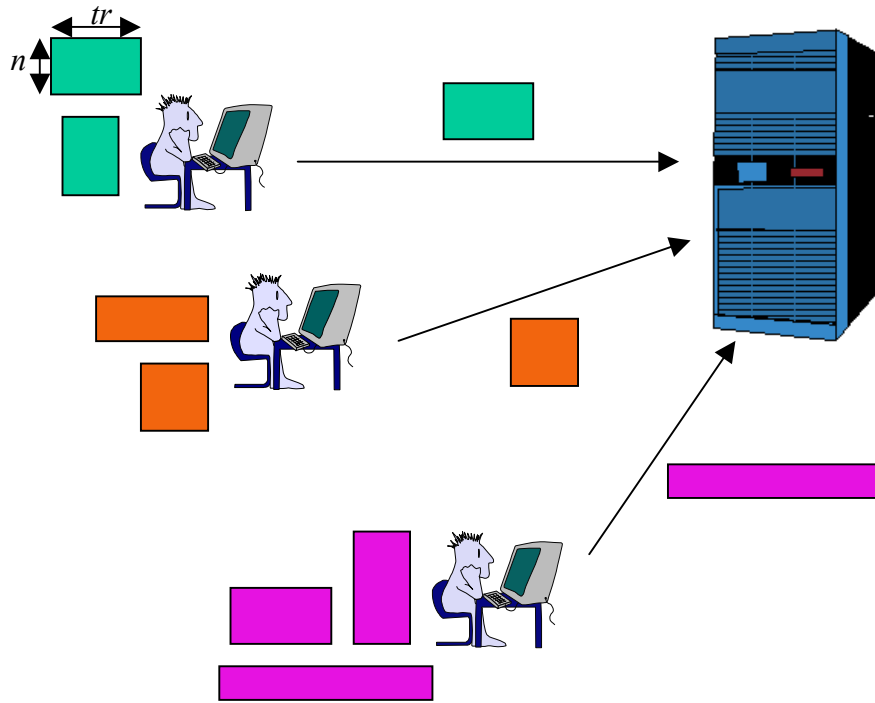


Figure 1 – Users selecting requests to submit their jobs

The decision made by the user of which request to use is important because it affects the job's turn-around time. The *turn-around time* of a job is the time elapsed between the job's submission and its completion. The turn-around time is a natural metric for the job performance because it captures the user's view of how long a job takes to complete.

In order to understand how the request used to submit a moldable job j affects its turn-around time, note that the turn-around time can be decomposed into *wait time* tw and *execution time* te . More precisely, $tt = tw + te$. Since jobs run in dedicated partitions, it is feasible for the user to evaluate the effect of the partition size n on the execution time te (by benchmarking the job, for example). The execution time typically diminishes as n grows (up to some point, at least).

However, the user cannot in general estimate the wait time tw because it depends on n , tr , the supercomputer scheduler, and the current load of the system. Indeed, research efforts that aimed to forecast the supercomputer wait time found it difficult to obtain good predictions [31] [55] [84] [85]. And, with an estimate for the execution time te alone, the user is not able to identify which request will minimize job j 's turn-around time tt .

1.3. Thesis Summary

We show in this thesis that **the request that submits a moldable job can be automatically selected in a way that often reduces the job's turn-around time**. More precisely, we will introduce and evaluate **SA**, a scheduler that chooses the request used to submit a moldable job on behalf of the user. The user provides **SA** with a set of possible requests that can be used to submit a given moldable job j . **SA** estimates the turn-around time of each request based on the current state of the supercomputer, and then forwards to the supercomputer the request with the smallest expected turn-around time. Figure 2 illustrates the role of **SA** in the job submission process.

SA stands for Supercomputer AppLeS. AppLeS (Applications-Level Schedulers) are application schedulers developed by Fran Berman's group at UCSD and Rich Wolski at University of Tennessee [9] [10] [83] [88] [89]. *Application schedulers* perform scheduling decisions for individual applications but do not control resources. They obtain access to resources by submitting requests to the appropriate resource schedulers. *Resource schedulers* do control the resources they schedule on. A supercomputer scheduler, for example, is a resource scheduler. One salient characteristic of resource

schedulers is that they receive requests from multiple users, and thus have to arbitrate among such users. Application schedulers, on the other hand, do not have to arbitrate among different users. Their goal is solely to improve the performance of the applications they serve. They can even limit themselves to schedule a single application (or a class of similar applications). By targeting a single application, application schedulers can rely on the application's structure and characteristics to produce good schedules.

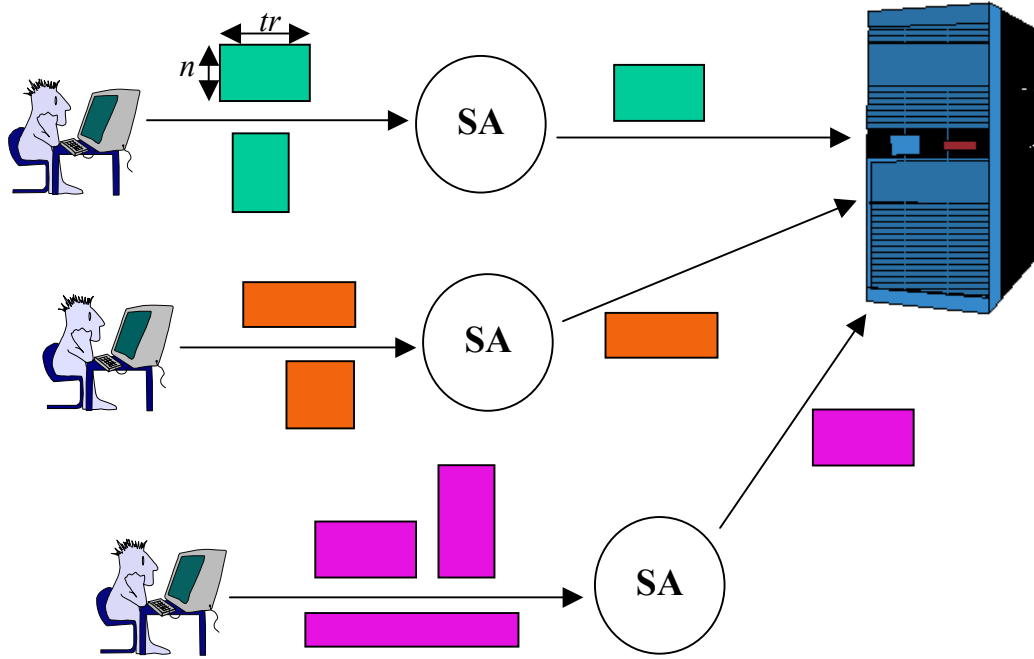


Figure 2 – Request selection by SA

Users are relieved by SA of a task unrelated with their final goals, namely that of selecting which request to use. Moreover and more importantly, SA often improves the turn-around time of the job under a variety of conditions. As we shall see in Chapters 5 and 6, the conditions under which SA was studied cover variations on the characteristics of the job, the state of the supercomputer, and the quality of the information available to SA.

We also investigate the emergent behavior created by having multiple instances of SA in the system. This is indeed a very important issue because there is theoretical evidence that systems in which resource allocation is performed by many independent entities can exhibit performance degradation [71] and even chaotic behavior [59]. As

we shall see in Chapter 6, one emergent behavior resultant of using **SA** with many jobs is that the system as a whole becomes more competitive, making it harder for each instance of **SA** to improve the performance of the job it schedules. On the other hand, the emergent behavior generated by **SA** also seems to reduce (i) the occurrence of very high load conditions, and (ii) the wait time of jobs that arrive when the supercomputer is experiencing moderate to high load. In light load conditions, the increased competition caused by other instances of **SA** make the performance improvement obtained by an individual instance of **SA** to be smaller than when a single **SA** is present in the system. In moderate to high loads, however, the reduction in the wait times and in the occurrence of very high load scenarios seem to overcome the performance degradation caused by the increased competition for resources produced by the other instances of **SA**.

1.4. Thesis Outline

This thesis is organized in five parts. First, this chapter provides the introduction and presents our research scenario, setting the stage for the rest of the thesis. The second part of the thesis consists of Chapter 2, which describes **SA**, our application scheduler for supercomputers. The third part discusses how to evaluate **SA** in order to determine its efficacy in improving jobs' performance: Chapter 3 describes the workloads used for performance evaluations, and Chapter 4 discusses performance metrics. The fourth part of the thesis contains the results of such an evaluation: Chapter 5 focus on the performance of **SA** under current workload conditions, while Chapter 6 investigates the emergent behavior caused by multiple instances of **SA** and its impact on performance. The fifth and last part concludes the thesis: Chapter 7 reviews the literature for related research and also delineates directions for future work. Chapter 8 summarizes our contributions.

2. SA: The Supercomputer AppLeS

This chapter describes **SA**, the Supercomputer AppLeS. **SA** is an application scheduler that adaptively selects the request that submits a moldable job to the supercomputer. A *moldable job* is one that can run on partitions of different sizes, although it cannot change the size of partition (i.e., gain and/or lose processors) during the execution [40]. Most parallel jobs in production today seem to be moldable (see Chapter 3).

Note that current supercomputers schedulers typically accept static requests. A *supercomputer request* (or simply *request*) contains the job's partition size n and request time tr . The semantics of a request is that the job executes over exactly n processors for no longer than tr time units. Since moldable jobs can run on multiple partition sizes, they can also be submitted using multiple distinct requests. Therefore, one has to choose which request to use when submitting a moldable job to a supercomputer. Nowadays, the user is the one whose chooses which request to use (as represented in Figure 1).

SA acts on behalf of the user and selects the request that submits a moldable job j (as depicted in Figure 2). Users provide **SA** with a set of requests for job j . That is, the user submits a job j to **SA** by providing a set of requests, each of which can be used to submit job j to the supercomputer scheduler. As we shall see in more detail, **SA** estimates the turn-around time of each request based on the current state of the supercomputer, and then forwards to the supercomputer the request with the smallest expected turn-around time.

Note that **SA** requires no changes in the behavior of the supercomputer scheduler. From the viewpoint of the supercomputer scheduler, the request that comes from **SA** submitting job j is just like any other: it is a pair (n, tr) that specifies the size of the partition to be allocated to job j (n) and establishes an upper-bound for the execution time of j (tr).

SA has two versions: generic **SA** (**SAg**, described in Section 2.1) and **SA** for conservative backfilling supercomputer schedulers (**SACb**, described in Section 2.2). Both versions of **SA** select exactly the same request (as we shall prove in Section 2.3). They differ with respect to their generality and speed. **SAg** makes no assumptions about the underlying supercomputer scheduler. **SAg** is therefore generic. But it can be slow because of the many simulations it needs to perform in order to schedule a job. If the behavior of the supercomputer scheduler is known, it may be possible to speed-up the execution of **SA**. We exemplify this by describing **SACb**, a version of **SA** that assumes the supercomputer scheduler to be conservative backfilling.

Since **SAg** and **SACb** are equivalent, we simply use the term **SA** throughout most of this thesis. The terms **SAg** and **SACb** are only used when it is important to make clear which version of **SA** we are referring to.

2.1. Generic SA

SA receives a set of requests $\mathbf{r} = (r^{[1]}, \dots, r^{[v]})$ that can be used to submit a job j . **SA**'s goal is to improve job j 's turn-around time by selecting the request to be sent to the supercomputer. The generic implementation of **SA**, which we denote by **SAg**, works without knowledge about the underlying supercomputer scheduler. **SAg** simulates the submission of all requests in \mathbf{r} , and then selects the request $r^{[s]}$ that achieves the smallest turn-around time in the simulations. The request $r^{[s]}$ is then used to submit job j to the supercomputer.

The simulation of the submission of job j by a given request $r^{[i]}$ starts from the current state of the supercomputer. **SAg** then provides the simulator of the supercomputer scheduler with all scheduling events until the completion of j . These scheduling events are jobs submissions and completions. Only one submission is provided: $r^{[i]}$, which submits job j . Completions are provided to all jobs in the system (including job j). The completions are calculated assuming that each job execute for their requested time tr . In summary, **SA** drives the simulation of request $r^{[i]}$ by (i) assuming no future job arrivals, and (ii) making $te = tr$ for all jobs.

In reality, however, new jobs do arrive in the system after j . Besides, most jobs execute for less time than they request (as we will see in Section 3.3.4). Therefore, there are no guarantees that **SA** will select the request that will deliver the shortest turn-around time. However, as shown in Chapters 5 and 6, the requests selected by **SA** significantly improve the turn-around time over the requests selected by the user.

The appeal of **SAg** is that it makes no assumptions on the behavior of the supercomputer scheduler. The supercomputer scheduler is treated as a black box to which **SAg** sends events representing arrivals and completions of jobs. Whereas this approach makes for a generic formulation of **SA**, it may also be a somewhat slow solution since each request $r^{[i]}$ must be simulated until the completion of j . If we know the characteristics of the underlying supercomputer scheduler, we may be able to make **SAg** run faster, as exemplified in the following section.

2.2. SA over Conservative Backfilling

When the supercomputer scheduler is known, it may be possible to optimize **SA** by avoiding the simulation of each request $r^{[i]}$. This section describes **SAcb**, a version of **SA** that assumes the supercomputer scheduler to be *conservative backfilling* [43]. Conservative backfilling was chosen as the target for a faster version of **SA** because it can be viewed as an idealized representative of today's supercomputer schedulers. In practice the behavior of supercomputer schedulers varies from machine to machine. Even when the same scheduling software is used (e.g., Easy [66] [82], PBS [58], Maui [69], and LSF [74]), each site establishes its own policies, causing the behavior of their schedulers to differ. However, almost everywhere backfilling is used to reduce unnecessary idle time, making conservative backfilling a good representative of current practice in supercomputer scheduling.

2.2.1. Conservative Backfilling

Conservative backfilling uses an *allocation list* that maintains, for any given time, which processors are assigned to which jobs [43]. The allocation list can be im-

plemented as a linked list whose nodes represent time periods in which all processors in system are allocated in the same way. Arriving jobs are processed using the *first fit* strategy, i.e. they are put in the first slot they fit. For example, Figure 3 depicts the submission of five requests in the following order: *A*, *B*, *C*, *D*, and *E*. Note that *C* is placed before *B* because, at time 1, the available resources cannot fulfill *B*, but they are enough for *C*.

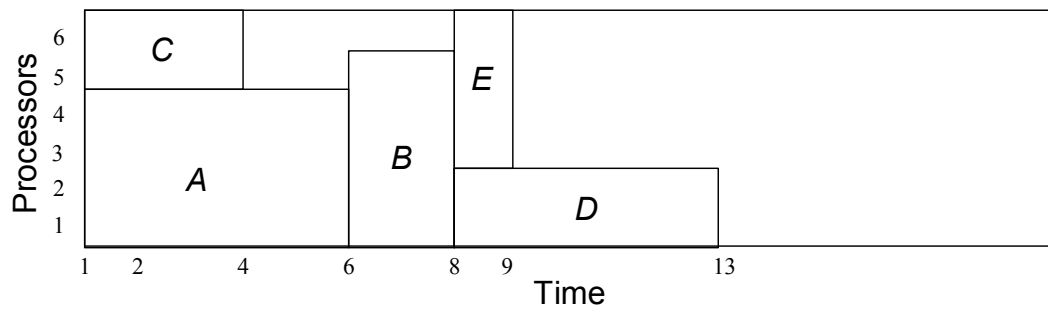


Figure 3 – Allocation list after the submission of five requests

Whenever a job finishes using less time than it required, conservative backfilling traverses the queue (in submission order) and “promotes” the first job that fits in the just-made-available slot. Of course, this may create another available slot. Such a slot is backfilled in the same way. The process stops only when no more backfilling can be done. For example, Figure 4 shows what happens when *A* finishes at time 2: *B* is back-filled to start immediately after *C*, *D* “follows” *B*, but *E* can finish before *B* starts and thus is backfilled all the way to start running immediately.

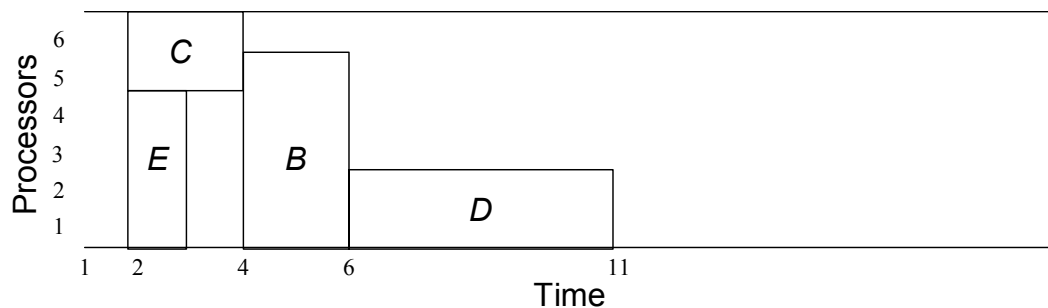


Figure 4 – Allocation list after the backfilling initiated by *A* finishing at time 2

2.2.2. SA over Conservative Backfilling

The state of a supercomputer controlled by conservative backfilling can be summarized by an *availability list* that contains the number of free processors per time period. For example, the availability list that describes the supercomputer state depicted in Figure 3 is [(from 1, to 4, 0 processors), (from 4, to 6, 2 processors), (from 6, to 8, 1 processor), (from 8, to 9, 0 processors), (from 9, to 13, 4 processors), (from 13, to ∞ , 6 processors)].

SAcb uses the availability list to select which request to use without simulating each request (the procedure used by **SAg**). **SAcb** traverses the availability list searching for a slot big enough to accommodate one of the requests it can use to submit job j . When such a slot is found, **SAcb** determines the request with sooner completion time among the requests that fit in the slot. Such a request (and its completion time) is memorized. After traversing the availability list, **SAcb** compares the memorized requests, selecting the request with the smallest completion time. The algorithm is:

```

1.  # SAcb pseudo-code
2.
3.  for each time period  $f$  in the availability list
4.
5.      let  $f = (s, e, n)$ , where  $s$  is the start of the time period,
                                 $e$  is its end, and
                                 $n$  is the number of processors available
6.
7.      # walk through the availability list to determine  $d$ , the last
      # instant at which we can allocate  $n$  processors starting from  $s$ 
8.      let  $d = e$ 
9.      let  $g = (s_g, e_g, n_g)$  be the time period succeeding  $f$ 
10.     while  $n_g \geq n$ 
11.         let  $d = e_g$ 
12.         let  $g = (s_g, e_g, n_g)$  be the time period succeeding  $g$ 
13.
14.     if the job can run on  $n$  processors in time  $(d - s)$ 
15.         let  $r$  be the request with smallest execution time among
         those that can run in time  $(d - s)$ 
16.         memorize request  $r$  and its expected completion time
17.
18. choose the memorized request with the least completion time

```

Figure 5 – SAcb pseudo-code

For example, assume that **SA**_{cb} is scheduling a job j that can run over 10, 20, or 30 processors. Job j needs 5 time units when using 10 processors, 3 time units with 20 processors, and 2 time units with 30 processors. Assume that the availability list is [(from 0, to 1, 5 processors), (from 1, to 5, 10 processors), (from 5, to 6, 0 processors), (from 6, to 7, 10 processors), (from 7, to 11, 20 processors), (from 11, to ∞ , 40 processors)], as graphically shown by Figure 6. In this case, **SA** finds three candidate requests: (A) 10 processors starting at 6 and finishing at 11 (Figure 7), (B) 20 processors starting at 7 and finishing at 10 (Figure 8), and (C) 30 processors, starting at 12 and finishing at 14 (Figure 9). Since B is expected to finish earlier, **SA** submits job j to the supercomputer by requesting 20 processors and 3 time units.

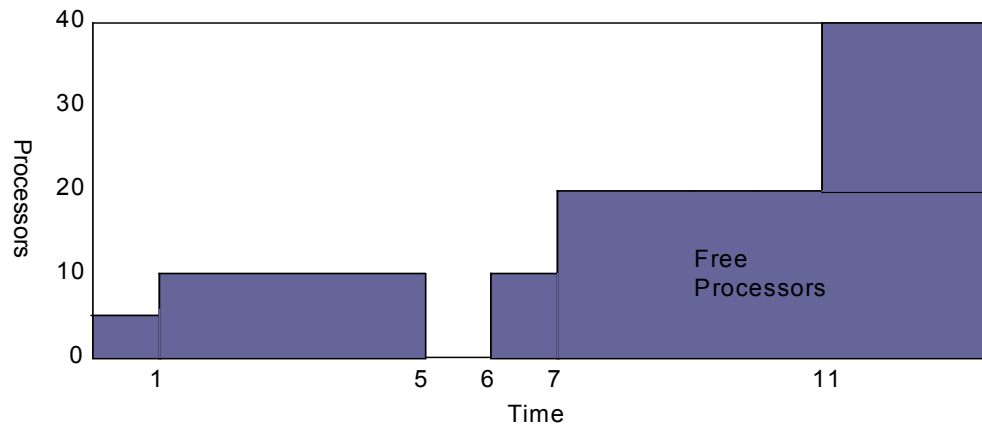


Figure 6 – **SA**_{cb} example: availability list

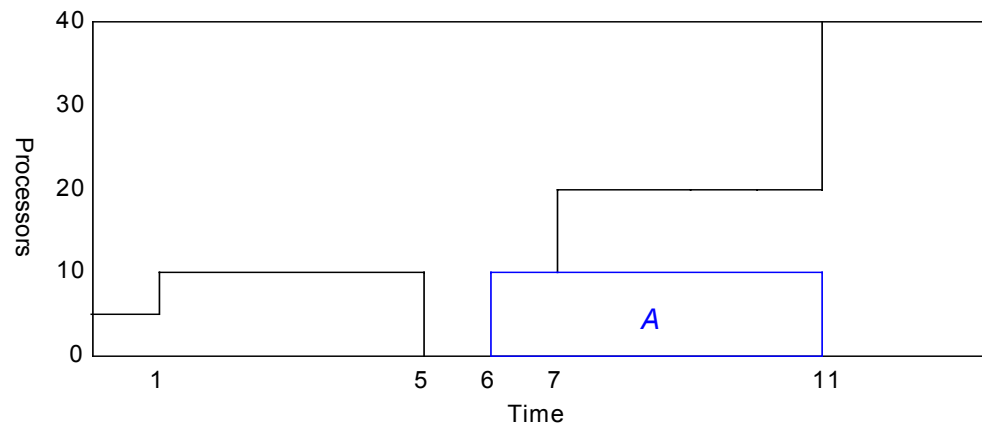


Figure 7 – **SA**_{cb} example: schedule A

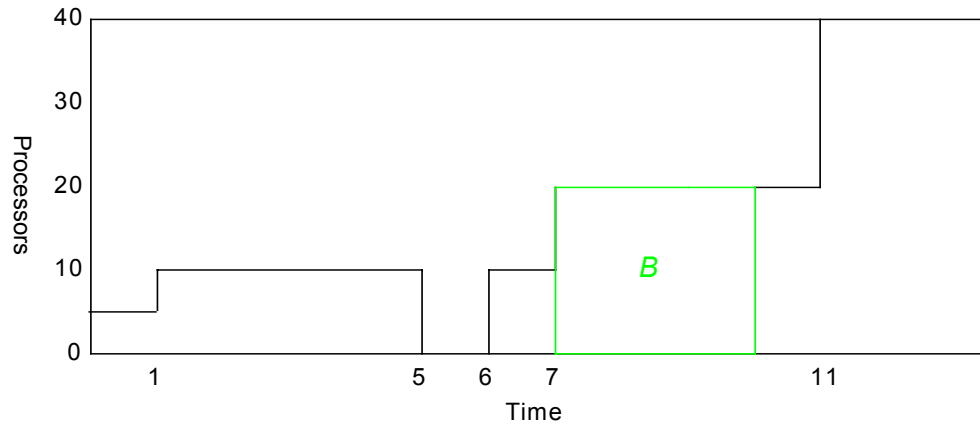


Figure 8 – SACb example: schedule *B*

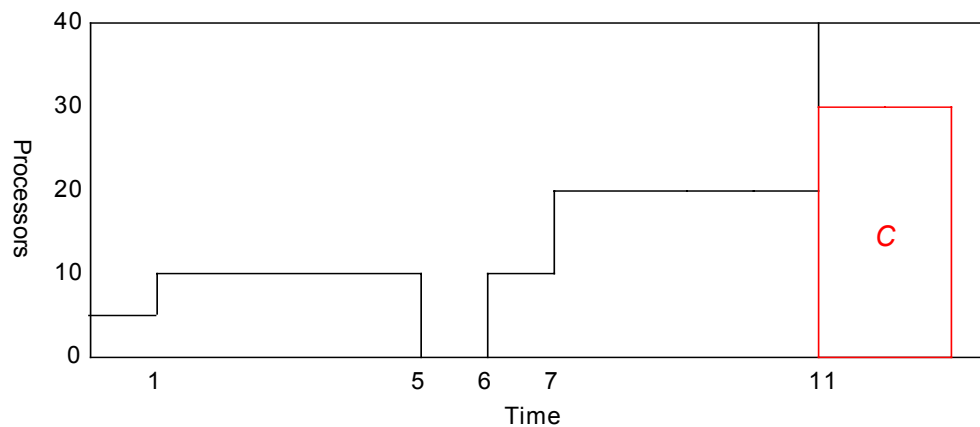


Figure 9 – SACb example: schedule *C*

Note that **SA** does not use any scheme to estimate the actual execution time of the jobs that are in the queue. It schedules as if all the jobs in the queue would take all the time they have requested. Nevertheless, it can obtain better turn-around times than traditional user-chosen requests (as we show in Chapters 5 and 6).

2.3. Equivalence between the two Versions of SA

Here we show that **SAg** and **SACb** always select the same request r to submit the target job j . Of course, we assume the supercomputer scheduler to be conservative back-filling (otherwise **SACb** would not work). The proof follows.

Theorem 1: Assuming that the supercomputer scheduler is conservative backfilling, **S**Ag and **S**Acb always select the same request.

Proof: Note that there is no backfilling in the simulations of the conservative backfilling scheduler conducted by **S**Ag. That is because **S**Ag makes $te = tr$ for all jobs in the system, and thus no job finishes before its requested time. This implies that conservative backfilling limits to first fit the arriving requests in the allocation list. Since **S**Ag chooses the request with smallest completion time among all simulations, **S**Ag over conservative backfilling selects the request r whose *first fit* in the allocation list results in the smallest completion time.

SAcb, on the other hand, selects the request s whose *fit* in the allocation list results in the smallest completion time. The availability list represents the empty slots in conservative backfilling's allocation list. For each slot, **S**Acb considers the request (if any) that finishes sooner if started in that slot (see Figure 5, line 15). All slots are considered and, in the end, **S**Acb picks the request that completes soonest (line 18).

Note that the fit that results in the smallest completion time must be a first fit. Otherwise, the same request s could start sooner (in its first fit) and thus complete sooner. Therefore s is the request whose first fit in the allocation list produces the shortest completion time. Consequently, r and s are the same request. \square

3. Workload Models

The performance of a scheduling solution is influenced by the workload submitted to the system [2] [44] [67] [90]. *Realistic* workloads are crucial to establish how scheduling solutions perform *in practice*. Therefore, in order to evaluate how SA is going to perform in practice, we need to determine the mix of moldable jobs that is likely to compose a supercomputer workload in real life.

Workload logs can be obtained by recording all scheduling events that happen in a system. The logs can then be used to drive simulations that gauge the performance of competing scheduling solutions. Such logs capture the production use of a system and thus are undoubtedly realistic.

Alas, supercomputer workload logs currently available contain only one request per job (namely, the request actually used by the user to submit the job). There is no information about the jobs' moldability, which is essential for SA. Furthermore, we cannot easily vary characteristics of the workload log (e.g. the offered load) to investigate how such a particular characteristic impacts on scheduling solutions.

This chapter describes how we have dealt with this difficulty. Here we introduce our workload model for moldable jobs. Such a model was derived from statistical observations of four workload logs, and from the results of a survey we conducted among supercomputer users. The use of real-life data as the foundation for our model leads us to believe that it is likely to produce realistic workloads.

3.1. Collecting Information

Since a key goal here is to produce a realistic model, we need information on the workloads experienced by production supercomputers. Although there are a handful of submission logs available, such logs contain only one request per job. We ran a survey

among supercomputer users to complement the logs and provide us some insight on the characteristics a moldable workload would have in practice.

Rigid Workload Logs

We considered workload logs from different sources (from fellow researchers to supercomputer centers to the web [45] [56]). Our criteria for using a log as *reference* for our model were that (i) the log should come from a supercomputer that could receive arbitrary requests (not only requests for power-of-2 partitions), and (ii) the log should contain a minimum of information to be useful in building our model (i.e., submission time, partition size, requested time, and execution time). We were able to find four workloads that meet these criteria. Such workloads are summarized in Table 1.

Name	Machine	Processors	Jobs	Period
ANL	Argonne National Laboratory SP2	120	7995	Oct 1996 Dec 1996
CTC	Cornell Theory Center SP2	430	79279	Jul 1996 May 1997
KTH	Swedish Royal Institute of Technology SP2	100	28479	Sep 1996 Aug 1997
SDSC	San Diego Super-computer Center SP2	128	16376	Jan 1999 May 1999

Table 1 – Workloads used in this research

All four reference logs come from IBM SP2 machines. This is because they were the only ones that meet our criteria. In particular, many of the available logs miss the job request times. We believe that using SP2 logs does not bias our results in any way, as SP2s are typical representatives of the machines we target, namely distributed-memory spaced-shared parallel supercomputers.

User Survey

A moldable job is, by definition, a parallel job that can use partitions of various sizes to run. Note, however, that this definition does not mean that a moldable job can run over partitions of arbitrary size. There may be a minimum and a maximum on the

partition size that can be used. There may also be some algorithmic restriction on the size of the partition (e.g., some parallel algorithms require a power-of-2 partition). Moreover, the user might provide only a subset of all potential partition sizes a job could possibly use. Beyond restrictions in partition sizes, the reduction of execution time as the partition size grows varies across different jobs (i.e., different jobs exhibit different speed-up behaviors).

We designed a user survey to understand how the aforementioned characteristics of moldable jobs are distributed in practice. The survey's questionnaire can be found in Appendix A, and a summary of the responses is provided in Appendix B. The survey consisted of 12 multiple-choice questions, and was conducted on-line via email and the Web between 17 April and 31 May 2000. Electronic questionnaires were distributed among supercomputer users at NASA, NCSA, NERSC, NPACI, and elsewhere. Answering the survey was of course voluntary, which renders it a self-selected sampling. Multiple-choice questions were used because (i) they raise the number of responses when self-selected sampling is used, and (ii) they ease the analysis of the results [8]. We received 214 responses to our survey.

3.2. Models Overview

Our *moldable workload model* has two independent parts, namely the *rigid workload model* and the *moldability model*. The rigid workload model produces a stream of jobs, each with one known request. The moldability model generates alternative requests for a given job j , for which only one request is known. A moldable workload is obtained by using the rigid workload model to produce a stream of jobs, and then applying the moldability model to each of these jobs. The result is a stream of moldable jobs. Note that the rigid workload model and the moldability model can be used independently. In particular, the moldability model can be used over a workload log to provide alternative requests for the jobs in the log.

Our reference workload logs were the basis from which we statistically derived the rigid workload model (although most jobs in the logs are probably moldable, there

is only one request available for each job, making the jobs appear rigid). The moldability model was derived from the survey’s results.

We should also point out that we tried to correlate the parameters that describe the workload logs, as well as the questions that compose the survey. We systematically calculated the correlation coefficient [29] of all pairs of workload parameters and all pairs of survey questions. All values above 0.5 (or below -0.5) were carefully investigated, and the resulting findings are described below. The goal was to reproduce significant correlations among the parameters of the model, making the model more realistic.

3.3. Rigid Workload Model

A rigid workload is composed of a stream of jobs, each with one request. Each job j is characterized by its instant of arrival ia , partition size n , requested time tr , and execution time te . For the jobs that are cancelled by the user, we also want to know their instant of cancellation $ic > ia$.

3.3.1. Instant of Arrival

The pattern of job submission is affected by the work cycles of the supercomputer’s users [32] [39] [44]. For example, typically more jobs are submitted during the day than during the night, as seen in Figure 10 (which indicates how many jobs arrived by hour of the day for the reference workloads). For a workload model to better capture the dynamics of the system, such behavior must be represented.

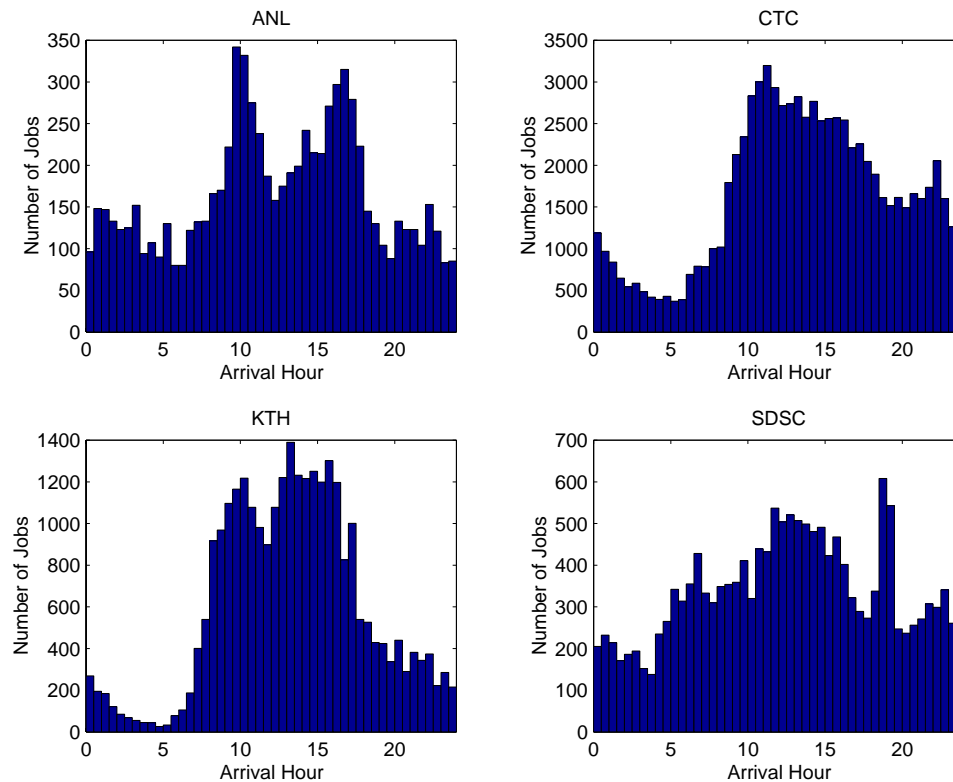


Figure 10 – Histograms of arrival hour for our reference workloads

Methodology

As Figure 10 suggests, it is very hard to model the job arrival time through “common” distributions. In this work, we apply the methodology proposed by Calzarossa and Serazzi [17] to numerically fit a polynomial to the job arrival rate λ_a . Although the methodology was conceived to model the process arrival for a uniprocessor time-shared system, we found it to be applicable in our scenario, namely job arrivals for parallel supercomputers.

In order to fit a polynomial to the arrival rate found in our reference workload logs, we must derive the arrival rate from the arrival instants (which are in the log). Following the methodology of Calzarossa and Serazzi [17], we smoothed out the arrival rate by using a moving average estimator. More precisely, for a given time m (in minutes), the arrival rate $\lambda_a(m)$ is estimated using all arrivals in the 10 minute interval cen-

tered on m . Also, in order to avoid numerical instabilities while fitting the data, we scale a given minute of the arrival m to the range $[-0.5, +0.5]$ as follows:

$$m_s = \frac{m - \frac{m_{max} - m_{min}}{2}}{m_{max} - m_{min}}$$

The polynomial fitting itself is done using the least square error estimator [29]. The degree d of the polynomial is chosen by incrementing d until the square error doesn't decrease significantly over two successive increments [17].

Eliminating Outliers

The analysis of the data from the reference logs reveals that a few days strongly deviate from the normal submission pattern. For example, Figure 10 shows the arrival of jobs at SDSC to have a spike between 18:30 and 19:30, a phenomenon that is not present in any of the other workloads. It turns out that, on 7 February 1999, 592 jobs (of which 579 were from the same user) were submitted to the supercomputer between 18:30 and 19:30. By not considering that single day, the spike in the graph disappears.

Such “uncommon” days appear in all workloads. In effect, we fitted a polynomial *per day* and ran a cluster analysis technique to classify the days in two groups. For all supercomputers, the clustering technique segregated *a single* day in one of the two groups. Since we want to model the common usage of a supercomputer, we eliminated the uncommon days that were clustered alone.

More precisely, we ran a Z score [29] over the coefficients of the polynomials to make the magnitude of such coefficients unimportant, and regarded the results as a point in \mathbf{R}^d . We then applied standard hierarchical cluster analysis (using the euclidian distance in \mathbf{R}^d) to separate the days in two groups. If a group consists of a single day (i.e., all other days are closer to each other than to this day), it is considered an outlier and excluded from the data. The procedure is repeated until no day is clustered alone. Table 2 shows how many days were excluded from each supercomputer log.

Workload	Days in the log	Uncommon Days Excluded
ANL	78	4
CTC	339	3
KTH	745	2
SDSC	150	2

Table 2 – Amount of days eliminated from the experimental data

We also used cluster analysis to look for other patterns of day-to-day variations. In particular, we tried to establish a statistically valid differentiation between weekdays and weekends. However, we were not able to find a way to cluster the days into disjoint groups that could be explored to enhance the model.

Fitting Arrival Rate

The polynomial fitting per se was a straightforward task. ANL required a degree 12 polynomial, CTC a degree 8 polynomial, KTH a degree 13 polynomial, and SDSC a degree 10 polynomial. Table 3 shows the coefficients of such polynomials. We suspect that ANL and KTH required higher-degreed polynomials to better model the small decrease in the job arrival rate verified around 12:00 (see Figure 10). The CTC and SDSC reference workloads do not present such a decrease.

The fitted polynomials are very different from one another. This suggests that there is no single model for the arrival time that works well across different sites. This is in agreement with the original work of Calzarossa and Serazzi, who warn against using the polynomial they found for alternative contexts [17]. Rather they highlight the importance of their work as a methodology.

Term	ANL	CTC	KTH	SDSC
1	7166.5	254.04	-32683	-5499.6
m_s	64174	-25.820	-61444	1527.0
m_s^2	866.10	-258.51	32553	3015.9
m_s^3	-45317	8.4442	49406	-951.95
m_s^4	-2922.9	81.612	-12611	-573.08
m_s^5	11761	-3.6628	-15113	199.32
m_s^6	895.15	-9.6309	2380.9	48.583
m_s^7	-1349.0	0.76455	2154.0	-15.611
m_s^8	-93.266	0.56501	-221.65	-2.5879
m_s^9	62.485		-134.82	0.36675
m_s^{10}	2.2601		8.3039	0.21262
m_s^{11}	-0.66152		1.9175	
m_s^{12}	0.17191		0.022406	
m_s^{13}			0.097106	

Table 3 – Coefficients of the polynomials that model job arrival rate

Figure 11 shows the observed arrival rate for each workload under consideration, as well as the polynomials that fit them. It is important to point out that filtering out the outliers made it possible to model the workloads with simpler, lower-degree polynomials that avoided the uncommon behavior of a few days. More notably, the SDSC model avoids the one-day spike around 19:00 discussed above.

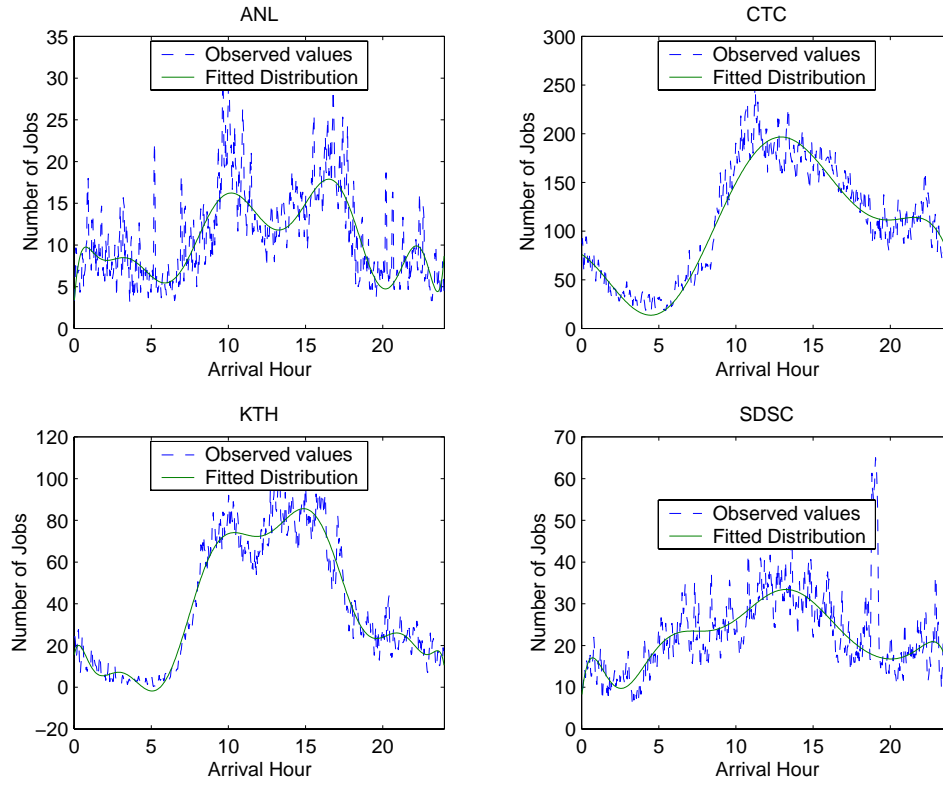


Figure 11 – Observed data and fitted model for arrival rate

We should also mention that we tried to correlate arrival time with other parameters. For example, we expected large jobs to be submitted at night. However, this was not supported by the reference workload logs. The correlation coefficient between these two parameters was low (in the range $[-0.1, 0.1]$) for all reference workloads. More generally, we didn't detect strong correlation between arrival instants and any other parameter.

3.3.2. Cancelled Jobs

Not all supercomputer jobs complete their execution. Some of them are *cancelled* by the user. Cancelled jobs may affect the course of action of the scheduler, even when they are cancelled before they start. Cancellation may also make the load offered by a given workload to change depending on the scheduler being used. In fact, consider a scheduler s that finishes a job j before its cancellation arrives, and a scheduler r for

which the cancellation of j arrives before the job starts. Everything else being the same, scheduler s has to deal with a greater load than scheduler r . Consequently, due to its impacts on scheduling, cancellation should be taken into account when modeling super-computer workloads.

We model cancelled jobs by providing a probability pc that a given job will be cancelled. Moreover, for each cancelled job, we also need to know the instant of cancellation ic , or alternatively the *cancellation lag* $lc = ic - ia$ (where ia is the instant of the arrival of the job in the system).

We have information on cancellations only for CTC and SDSC. The ANL and KTH logs do not discriminate between completed and cancelled jobs. Table 4 displays the percentage of completed and cancelled for the CTC and SDSC reference workloads. Such values can be used to provide realistic estimates of the probability of cancellation pc .

Workload	Cancelled Jobs
CTC	12.22%
SDSC	23.31%

Table 4 – Percentage of completed and cancelled jobs

A visual inspection of the cancellation lag histograms for both CTC and SDSC show a fat-tailed distribution (see Figure 12). Although the cancellation lag can be as large as 10 days for the SDSC workload and over 2 months for the CTC workload, most cancellations happen shortly after the job's submission. In fact, 54.65% of SDSC's and 24.73% of CTC's cancellations happen less than 10 minutes after the job submission.

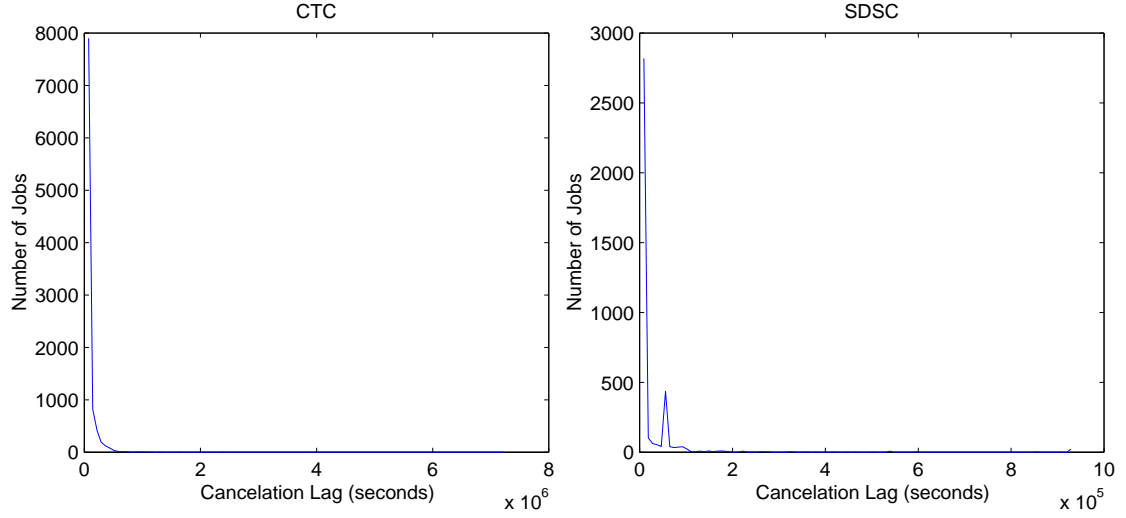


Figure 12 – Histograms of cancellation lag

To build a realistic workload model, it is important to represent both the agglomeration of short cancellation lags lc and the fat tail of the distribution. This behavior can be effectively modeled using a uniform log distribution. In such a distribution, *the logarithms* of the values are uniformly distributed. More precisely, a *uniform-log distribution* is characterized by parameters χ and ρ , and has cumulative distribution function $cdf(x) = \chi \cdot \log_2(x) + \rho$ [31]. Since we deal with many distributions in this thesis, we index the distribution parameters with the variable they are modeling in order to avoid ambiguity. For example, the uniform-log parameters that model the cancellation lag lc are written as χ_{lc} and ρ_{lc} .

Using the least-square linear regression technique, we fitted SDSC's cancellation lags obtaining $\chi_{lc} = 0.06442$ and $\rho_{lc} = -0.1498$. For CTC, we obtained $\chi_{lc} = 0.06421$ and $\rho_{lc} = -0.3180$. The SDSC and CTC values for χ_{lc} and ρ_{lc} are similar, suggesting that the cancellation lag may not change significantly across workloads (unlike arrival rate). Figure 13 shows the data as well as the fitted uniform-log distributions for both CTC and SDSC.

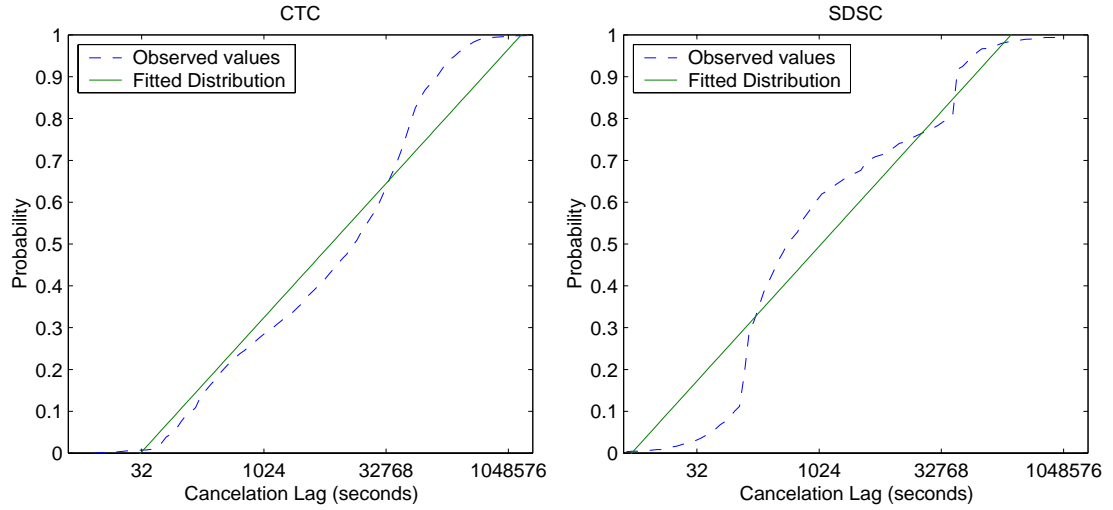


Figure 13 – Cancellation Lag CDF

3.3.3. Partition Size

As first noticed by Downey [32] [33], the uniform-log distribution also provides a good fit for the partition sizes in a supercomputer workload log. This was the case for our four reference workloads. Table 5 shows the parameters obtained by fitting the observed partition sizes to a uniform-log distribution via the method of least squared error. Figure 14 plots the observed partition sizes and the uniform-log distributions fitted to them.

Workload	χ_n	ρ_n
ANL	0.1381	0.2163
CTC	0.0709	0.5283
KTH	0.0893	0.4764
SDSC	0.1150	0.2537

Table 5 – χ_n and ρ_n obtained by fitting partition size to a uniform-log distribution

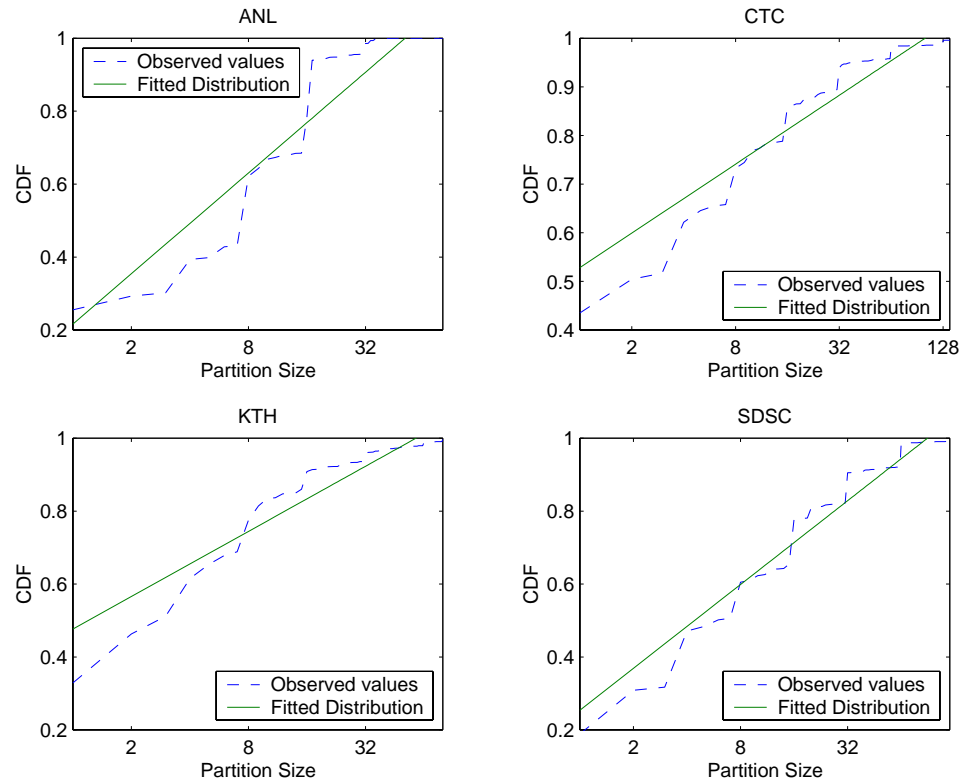


Figure 14 – Uniform-log fit for partition size

However, the uniform-log distribution alone does not capture a salient characteristic of how partition sizes are distributed. As Table 6 and Figure 15 show, the distribution of partition sizes seems to be dominated by power-of-2 values. This is a relevant characteristic because the fraction of power-of-2 jobs in the workload strongly influences the performance of the system [67]. In general, the greater the fraction of power-of-2 jobs in the workload, the better the performance of many scheduling solutions [67]. We independently confirmed this result.

Workload	Power-of-2 Jobs
ANL	69.87%
CTC	83.16%
KTH	73.45%
SDSC	83.95%

Table 6 – Percentage of jobs with a power-of-2 partition size

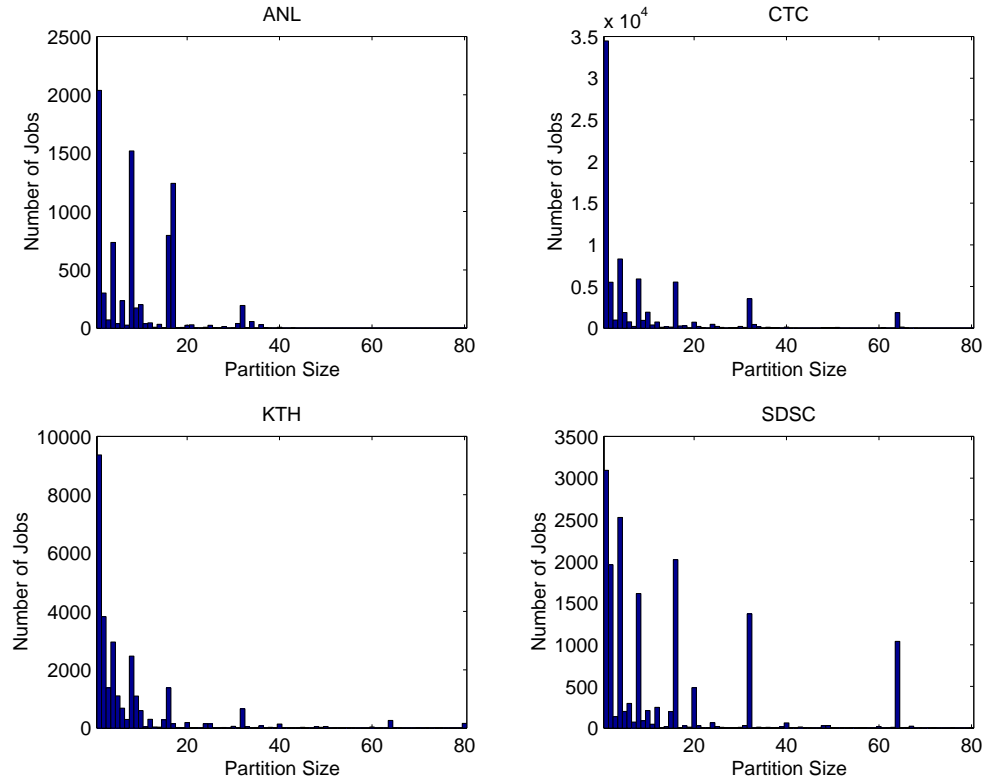


Figure 15 – Histogram of partition sizes

There has been some controversy on whether to incorporate the dominance of power-of-2 partitions into a workload model. Some researchers have accounted for the high incidence of power-of-2 jobs and modeled the partition size accordingly [42]. Others, however, believe this is mainly due to old habits (the first parallel supercomputers *required* power-of-2 partition sizes) and the design of some submission interfaces (which “suggest” the submission of power-of-2 jobs) [32]. Based solely on the workload logs, it is impossible to decide whether the prevalence of power-of-2 jobs is due to the nature of the parallel jobs, or is an artifact of behavioral inertia and interface design.

In the survey, we inquired about the constraints jobs have regarding partition size (question 7, see Appendix A). Figure 16 summarizes the responses. To our surprise, the majority of the answers (69.4% of them, excluding “do not know”) described jobs that have no partition size restriction. This result suggests that the high incidence of

power-of-2 requests in the workload logs is *not* an intrinsic characteristic of the jobs. It might indeed be that the popularity of power-of-two partitions is due to behavioral inertia and interface design. Moreover, we believe that the fact that the selection of partition size is made by humans also contributes for the prevalence of power-of-two partitions. When no constraint exists, humans tend to pick “round” numbers, and powers of two are many people’s idea of “round number” when they deal with computers.

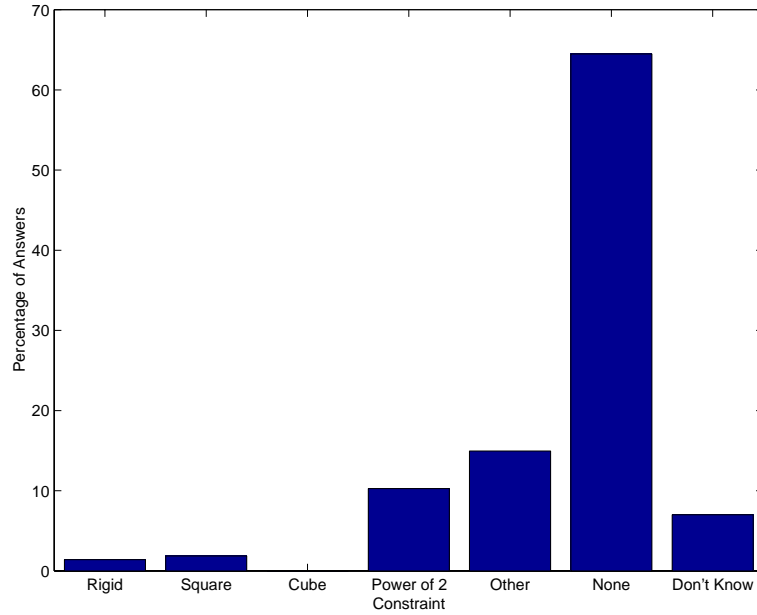


Figure 16 – Survey results for the constraints jobs face regarding partition size

In short, it seems that supercomputers workloads *do not have* to be dominated by power-of-2 jobs, but in practice they *are*. We believe that the high percentage of power-of-2 jobs is related to fact that the user is the one who chooses the partition sizes. Since we expect the user to keep playing this role, our model for partition size has a bias in favor of power-of-two partitions.

More precisely, we define the probability pb of a job been a power-of-2 partition size. Table 6 shows values pb assumed for our reference workloads. We start by using a uniform-log distribution to generate the partition size. The resulting partition size then has a probability pb of being changed to its closest power-of-2 value. Figure 17 through

Figure 20 display the observed values for partition side by side with the fitted model for our reference workloads.

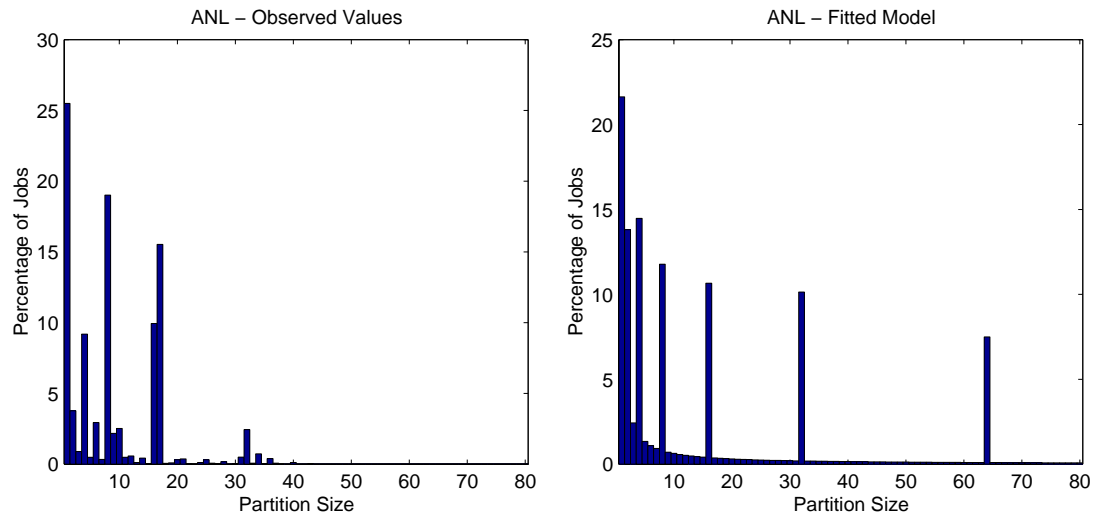


Figure 17 - Observed data and statistical model for partition size (ANL workload)

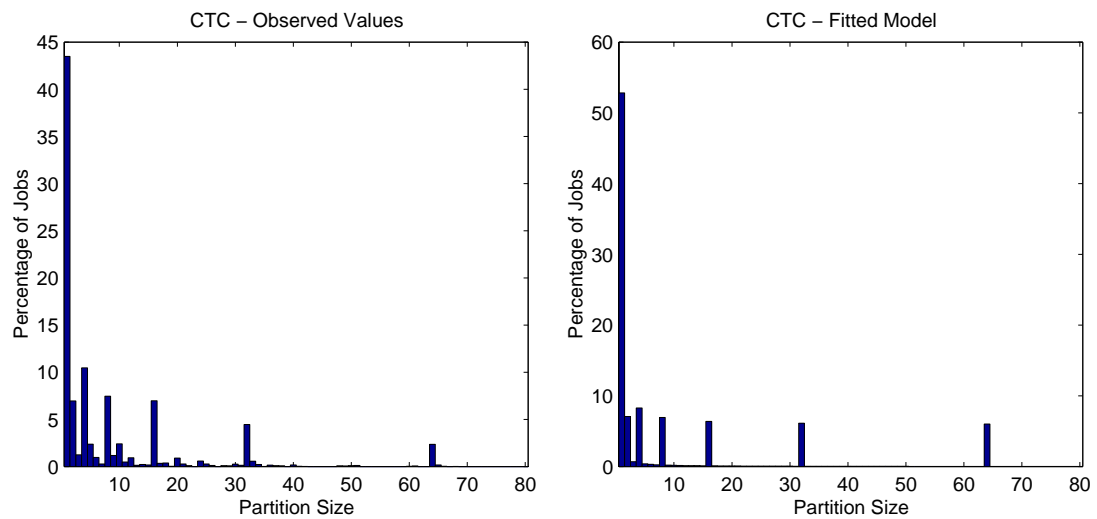


Figure 18 - Observed data and statistical model for partition size (CTC workload)

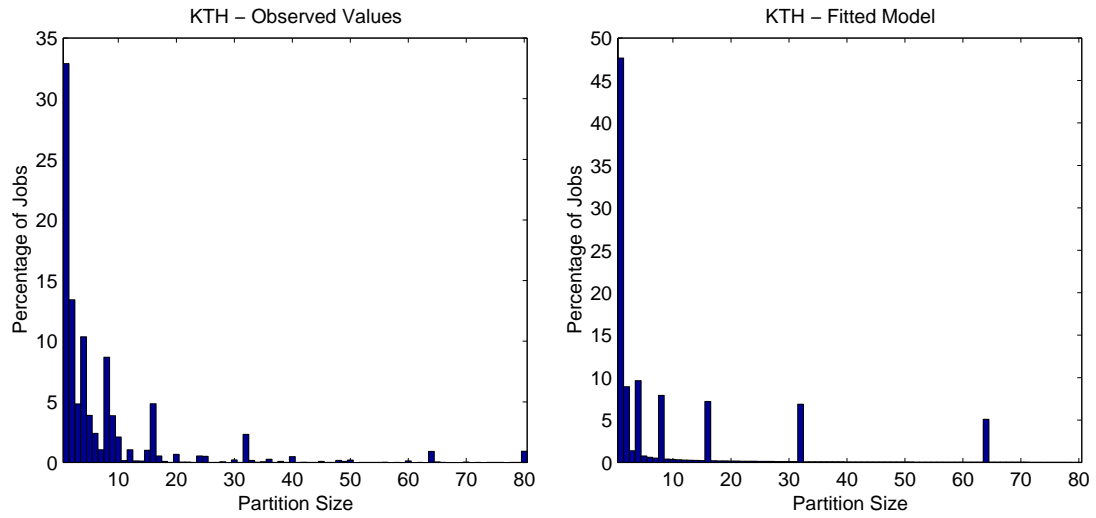


Figure 19 - Observed data and statistical model for partition size (KTH workload)

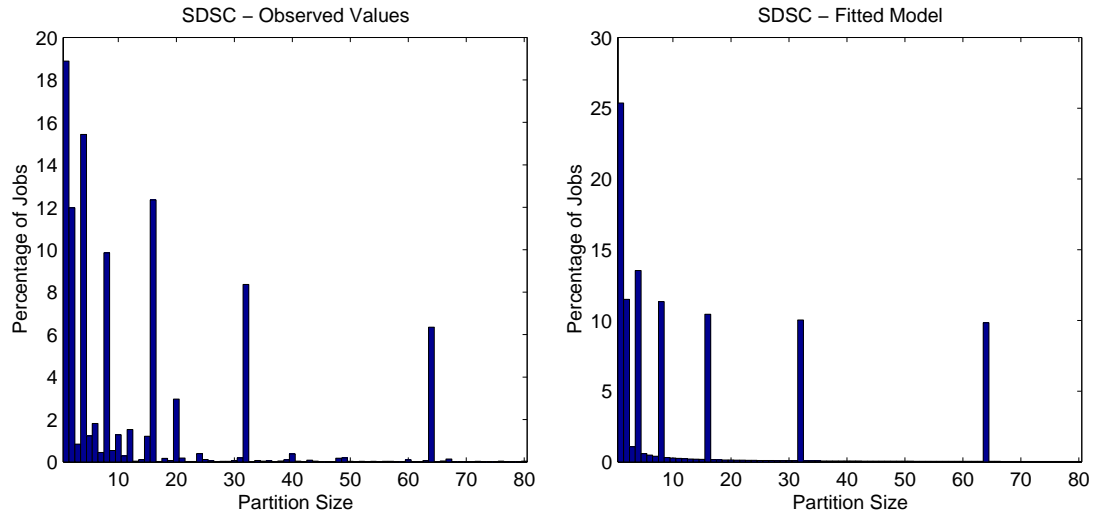


Figure 20 - Observed data and statistical model for partition size (SDSC workload)

We also checked for high incidence of square, even, multiple-of-4, and multiple-of-10 jobs among the jobs that are not power-of-2. As can be seen in Table 7, our four reference workloads do not exhibit a concentration of such jobs that is large enough to be explored in the model.

Workload	Square	Even	Multiple-of-4	Multiple-of-10
ANL	9.55%	28.27%	5.56%	9.92%
CTC	10.35%	49.53%	19.14%	24.05%
KTH	17.88%	36.93%	15.84%	16.91%
SDSC	6.09%	61.49%	37.67%	30.75%

Table 7 – Percentages of different kinds of non-power-of-2 jobs

3.3.4. Execution Time and Requested Time

Execution time te and requested time tr are obviously related ($te \leq tr$). In order to capture the relationship between te and tr in the model, we define the *request accuracy* a as the fraction of the requested time that was indeed used by a job. That is, $a = te / tr$. Since jobs cannot run longer than the amount of time they request, a is always a number between 0 and 1.

Note that we only need to model two parameters out of te , tr , and a . The third parameter can be derived from the others (by using the relation $a = te / tr$). We would prefer to use two parameters that are not strongly related. This simplifies the model because it eliminates the need to account for any correlation: Two parameters that are not related can be modeled *independently*. As mentioned above, te and tr are related. This leaves us with a and either te or tr as the parameters to model.

The analysis of the reference workloads reveals accuracy to have a much greater correlation with execution time than with requested time. For all workloads, the correlation coefficient between accuracy and execution time is considerably greater than the correlation coefficient between accuracy and requested time, as shown in Table 8.

Workload	$r(a, te)$	$r(a, tr)$
ANL	0.5273	0.0772
CTC	0.7024	0.2651
KTH	0.5010	0.2098
SDSC	0.7398	0.3985

Table 8 – Correlation coefficient between accuracy and execution time $r(a, te)$ and correlation coefficient between accuracy and requested time $r(a, tr)$

Figure 21 shows the accuracy a as a function of the execution time te . The figure groups the completed jobs in 100 equally-sized “buckets” according to their execution time. The jobs in a bucket have their accuracies averaged to produce the values plotted in Figure 21. Of course there is high variance in the execution time of the jobs grouped in a given bucket, but the average shows a trend for accuracy to grow with execution time. On the other hand, plotting accuracy a as a function of requested time tr (see Figure 22) doesn’t reveal much correlation between these two parameters.

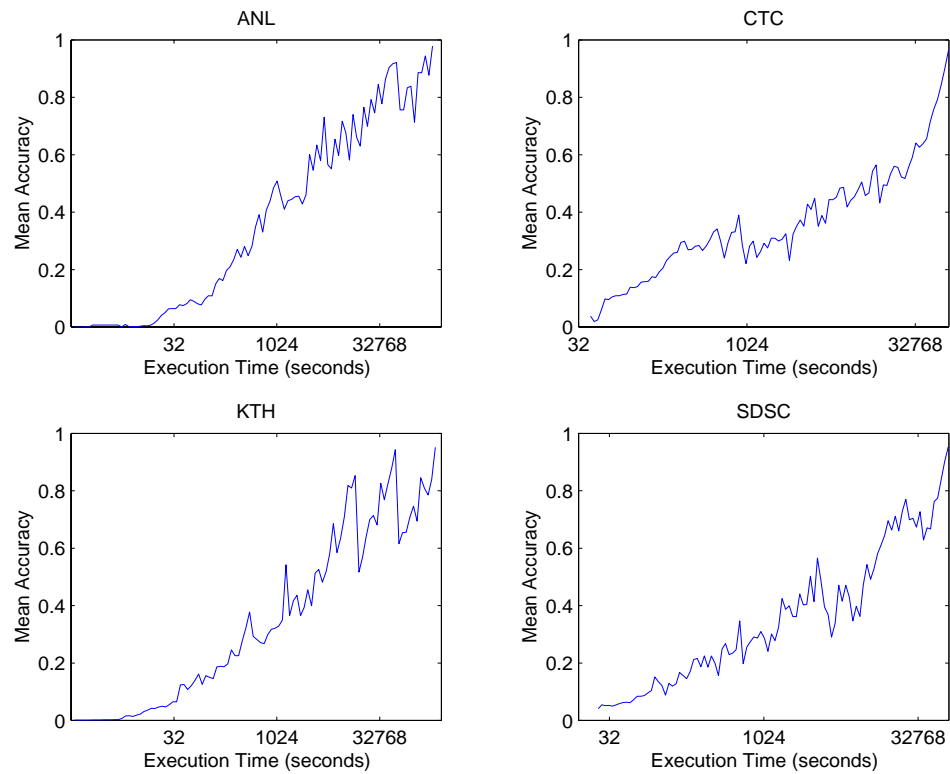


Figure 21 – Accuracy \times Execution Time

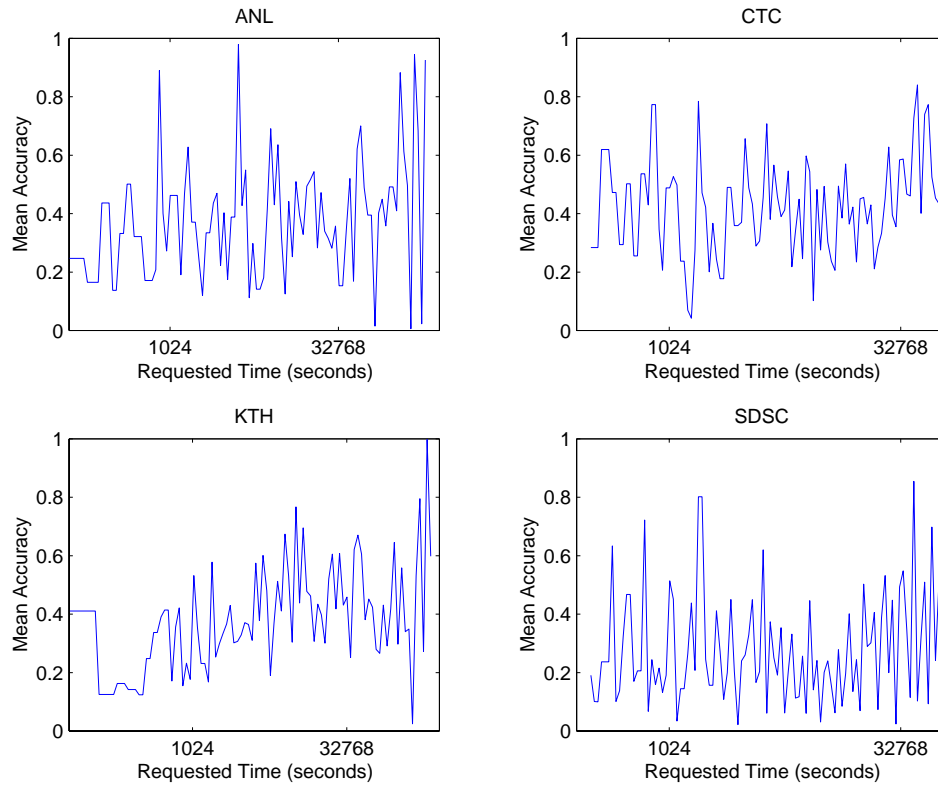


Figure 22 – Accuracy × Requested Time

We believe that the correlation between execution time and accuracy is related to the fact that *jobs often fail, and thus execute for much less than the user expected*. In other words, we believe failed jobs to have in general poorer accuracy a than successful jobs. Unfortunately the logs do not contain information on whether a completed job failed or succeeded. But it is reasonable to imagine the longer the execution time, the more likely it is that the job succeeds. In fact, many failures happen at the beginning of the execution, while the job is setting up its environment. For example, many jobs open files in the beginning of their execution, and a misspelled filename could cause a failure. This argument would also help to explain the large number of jobs with very low accuracy (as we soon shall see, there are many jobs with poor accuracy in all reference workloads).

Since requested time shows little correlation to accuracy, we model these parameters independently. Execution time is then derived by using $te = tr \cdot a$. As a side

note, we should also mention that requested time was easier to model (i.e., it yielded a better fit) than execution time. Although this was not our primary reason, it provided additional support to our choice of deriving execution time from the explicitly modeled accuracy and requested time.

Accuracy

Figure 23 shows how accuracy is distributed in the workload logs. Only completed jobs are considered. The figure groups the completed jobs in 100 equally-sized “buckets” according to their accuracy, and plots the fraction of jobs in each bucket. It is somewhat surprising to see how bad accuracy can be. The ANL workload, for example, has 27.82% of the requests with accuracy below 0.01 (i.e., these jobs used less than 1% of their requested time).

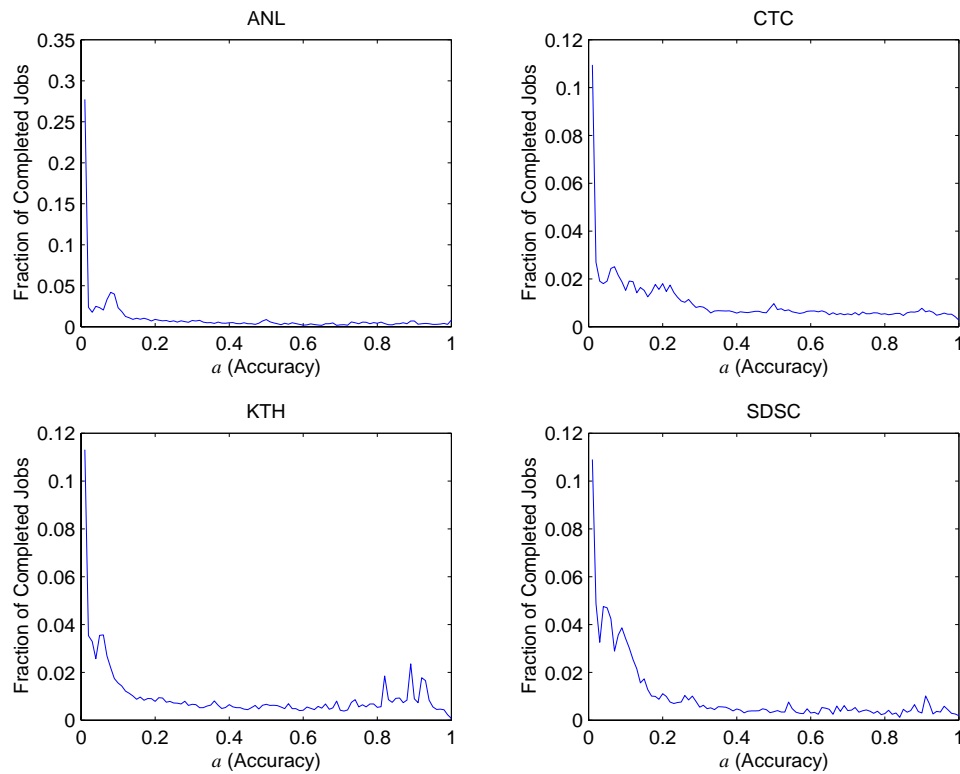


Figure 23 – Distribution of the accuracy a for the completed jobs

The number of jobs with low accuracy decreases as accuracy a grows (leveling off at around $a = 0.3$). This suggests the use of a distribution that favors small values and quickly decreases. We use the gamma distribution [29] to model such behavior. Table 9 shows the parameters obtained by fitting the observed accuracy to a gamma distribution through the method of maximum likelihood [29]. Figure 24 presents the observed distribution of accuracy and the corresponding model for all four reference workloads.

Workload	α_a	β_a
ANL	0.3779	1.0599
CTC	0.6743	0.7808
KTH	0.6153	1.0635
SDSC	0.5898	0.5793

Table 9 – α_a and β_a obtained by fitting accuracy to a gamma distribution

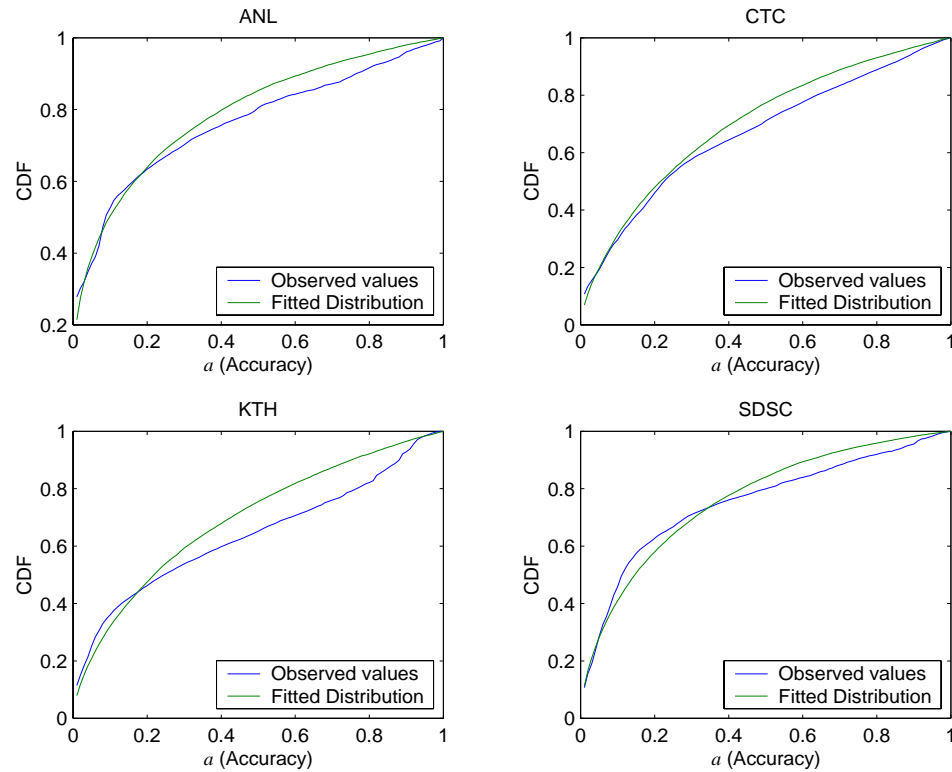


Figure 24 – Observed data and statistical model for accuracy

The gamma distribution parameters obtained by fitting the accuracy to the four reference workloads are somewhat similar to one another. ANL exhibits the smallest α value. This is because ANL presents a much higher number of jobs with very poor accuracy. Such a fact indicates that the distribution of accuracy can vary somewhat from site to site in practice.

Requested Time

For all reference workloads, the uniform-log distribution provides a very good fit for request times, and thus it is the distribution used in our model. We use least-square linear regression to find χ_{tr} and ρ_{tr} for the different reference workloads. Table 10 displays such values and Figure 25 plots the observed requested times against the model. The similarity of the values obtained for χ_{tr} and ρ_{tr} across all four reference workloads suggests that the requested time doesn't vary widely across workloads.

Workload	χ_{tr}	ρ_{tr}
ANL	0.1146	-0.8579
CTC	0.0941	-0.7256
KTH	0.1035	-0.7313
SDSC	0.1032	-0.7027

Table 10 – χ_{tr} and ρ_{tr} obtained by fitting requested time to a uniform-log distribution

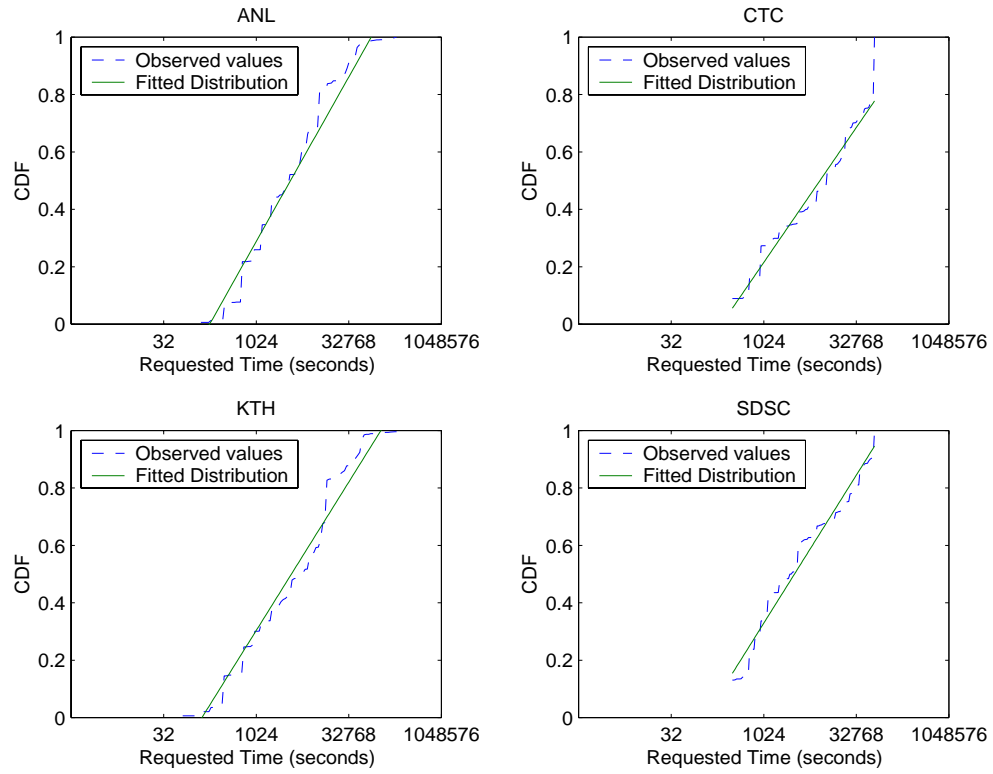


Figure 25 – Observed data and statistical model for requested time

3.4. Moldability Model

In our model, moldable jobs are an extension of rigid jobs. Recall that a job j is said to be *moldable* when it can run on partitions of different sizes. Therefore the input of the moldability model is a job j for which one request (partition size n , requested time tr , accuracy a) is known. Let v be number of partition sizes on which j can execute. The moldability model uses n , tr , and te to produce the v requests that can be used to submit job j . That is, the moldability model produces the v -tuples $\mathbf{n} = (n^{[1]}, \dots, n^{[v]})$, $\mathbf{tr} = (tr^{[1]}, \dots, tr^{[v]})$, $\mathbf{a} = (a^{[1]}, \dots, a^{[v]})$ that describe v requests for job j . The question is then how to generate *realistic* values for such v -tuples.

3.4.1. Partition Sizes

Some moldable jobs cannot run over a partition of arbitrary size. Factors such as memory requirements, amount of parallelism, and algorithmic constraints restrict the partition size that can be used by a given moldable job j . For example, memory requirements can establish a minimum partition size on which a job can run, a factor we model as c_{min} . Similarly, the amount of parallelism determines the maximum partition size a job can use, a factor we model as c_{max} . Some parallel algorithms also have constraints on the set of partition sizes they can use. We don't model algorithmic constraints directly because the user choices for partition size seem to provide strong restriction than the algorithm constraints themselves, as discussed in Section 3.3.3.

Regarding user behavior, we cannot expect that the user will in general craft *all* possible requests, one for each possible partition size that can possibly be used by the user's job. We define c_u to be the number of choices that the user is willing to provide. That is, c_u establishes an upper bound on how many alternative requests can be used to run job j . The c_u partition sizes are uniformly chosen between c_{min} and c_{max} . However, a chosen partition size is turned into its closest power-of-2 with probability pb . This is to mimic how users choose partition sizes (see Section 3.3.3).

In order to provide a realistic model, we must determine how c_{min} and c_u are distributed in practice. c_{max} can be uniquely determined by other parameters of our model (as we shall see shortly) and hence does not need to be modeled directly. Since the distribution of c_{min} and c_u cannot be derived from the reference workload logs, we rely on the survey for establishing realistic models for these distributions.

Minimum Partition Size (c_{min})

Figure 26 displays the results for the survey question that asked for the minimum partition size that can be used to run the respondent's job (question 4). Note that most jobs can run sequentially, but some really need larger partitions. These characteristics suggest the use of the uniform-log distribution to model c_{min} . Using the method of the least squared error, we obtained $\chi_{cmin} = 0.06920$ and $\rho_{cmin} = 0.6279$, and a very good fit, as shown in Figure 27.

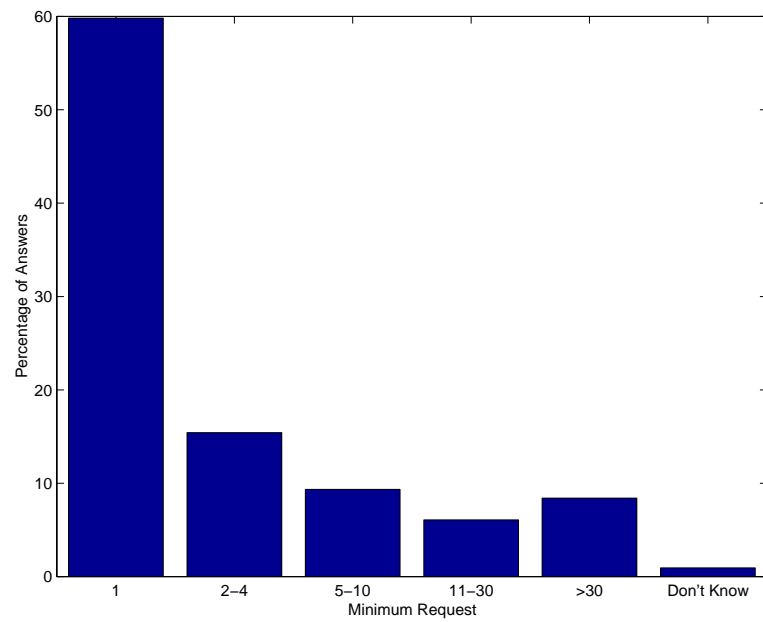


Figure 26 – Survey results for minimum partition size

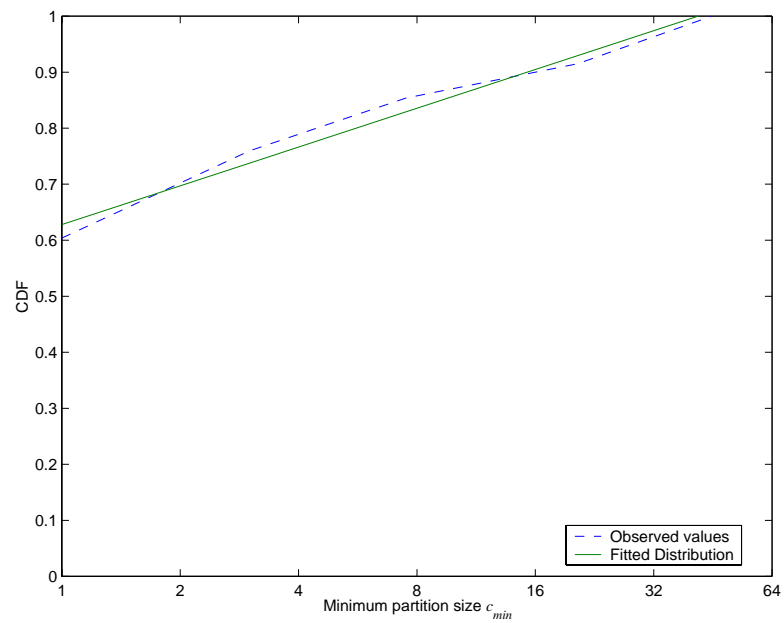


Figure 27 – Survey results and model for the minimum partition size c_{min}

Number of Requests Provided by the User (c_u)

Our model for c_u is also based on our survey. Figure 28 shows how many different partition sizes users have requested for their jobs (question 8). Such results indicate how many requests the users can currently use to submit their jobs, and thus are the natural estimate for c_u . It may be that such values increase when schedulers like SA make it more advantageous for the user to determine more alternative requests, but we take a conservative approach and model c_u after the current practice.

In order to better match the survey results, we use a two-stage model, segregating the probability that $c_u = 1$ from the probability that $c_u > 1$. The survey indicates that around 5% of the jobs have $c_u = 1$. We thus make $\Pr[c_u = 1] = 0.05$ and $\Pr[c_u > 1] = 0.95$. Note that by making $c_u = 1$ for 5% of the jobs, we also address the fact that part of the workload is formed by rigid jobs (about 2% of the jobs seem to be rigid, see Figure 16). Here again, the user behavior in choosing the partition size seems to exhibit stronger restrictions than the algorithmic constraints exhibited by some jobs.

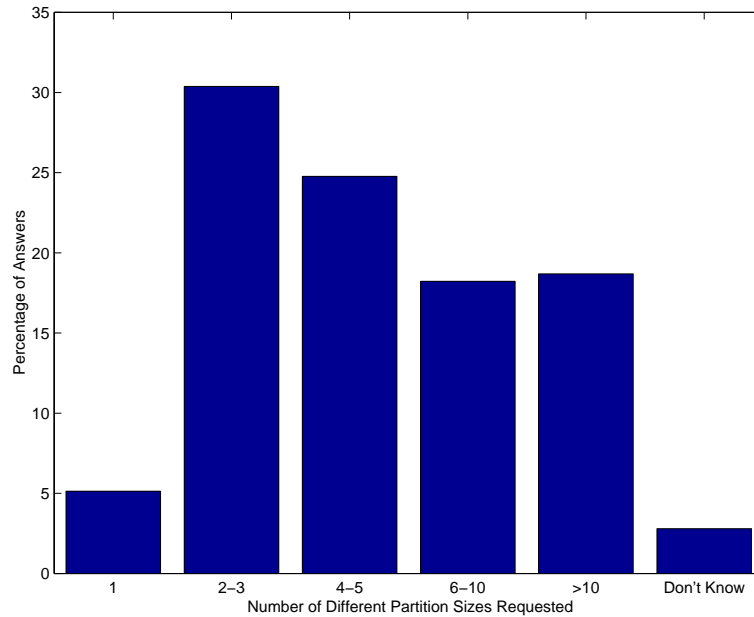


Figure 28 – Survey results for number of requests used to submit a job

For $c_u > 1$, we use a uniform-log distribution to determine which value c_u assumes. By fitting the survey results through the least squared error method, we determined the parameters $\chi_{cu} = 0.1918$ and $\rho_{cu} = 0.1876$. Figure 29 shows the model of the distribution when $c_u > 1$, as well as the plot of the corresponding survey results.

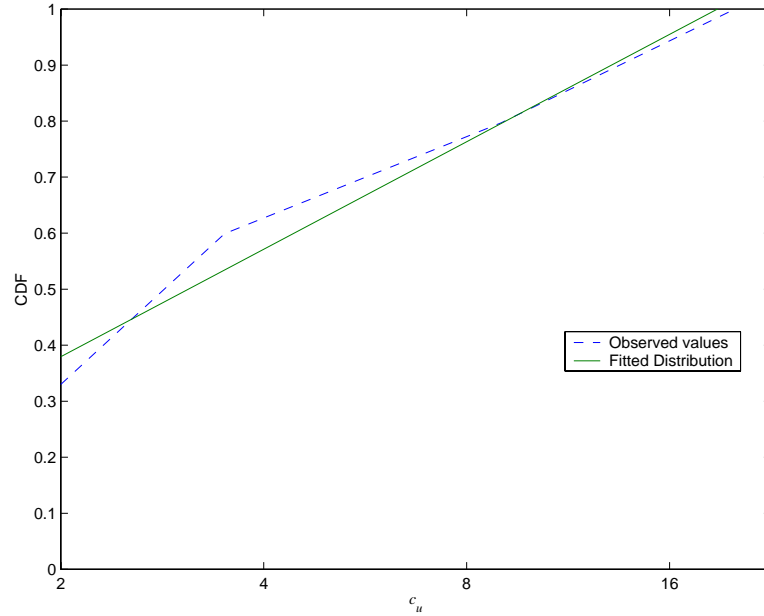


Figure 29 – Survey results and statistic model for $c_u > 1$

3.4.2. Accuracy

Because today's supercomputers receive a static request, there is no data on the accuracy of different choices for a given job. However, the accuracy a measures how well a user is able to estimate the execution time of the job. Therefore, we hypothesize the accuracies of the multiple requests for a given moldable job to be similar. Following this rationale, we assume the accuracy a to remain the same for all choices of a given job j . That is, $a^{[1]} = \dots = a^{[v]} = a$. Recall that the accuracy a is chosen using the statistical model described in Section 3.3.4.

3.4.3. Request Time

We use Downey's model of the speedup of parallel jobs [30] to derive the requested times of the choices $\mathbf{tr} = (tr^{[1]}, \dots, tr^{[v]})$. Speed-up measures how much faster a job j that uses n processors executes in comparison to j 's execution using only one processor. Symbolically: $S(n) = te(1) / te(n)$. Downey's speedup model uses two parameters: A (the *average parallelism*) and σ (an approximation of the *coefficient of variance in parallelism*). The speed-up of a job is then given by:

$$S(n, A, \sigma) = \begin{cases} \frac{An}{A + \sigma(n-1)/2} & (\sigma \leq 1) \wedge (1 \leq n \leq A) \\ \frac{An}{\sigma(A-1/2) + n(1-\sigma/2)} & (\sigma \leq 1) \wedge (A \leq n \leq 2A-1) \\ A & (\sigma \leq 1) \wedge (n \geq 2A-1) \\ \frac{nA(\sigma+1)}{\sigma(n+A-1)+A} & (\sigma \geq 1) \wedge (1 \leq n \leq A+A\sigma-\sigma) \\ A & (\sigma \geq 1) \wedge (n \geq A+A\sigma-\sigma) \end{cases}$$

Intuitively speaking, A establishes the maximum speedup a job can achieve. The larger the value of A , the greater the speedup a job can achieve. Figure 30 exemplifies how A affects the speed-up of a job. It fixes $\sigma = 1$ and shows speed-up curves for different values of A .

σ , on the other hand, determines how fast a job achieves its maximum speed-up (A). That is, σ determines how close to linear the speed-up is. The smaller the σ , the faster the job reaches its maximum speedup, and hence the closer to linear the speed-up curve is. Figure 31 explores the effect of σ on the speed-up behavior. It fixes $A = 60$ and displays speed-up curves for different values of σ .

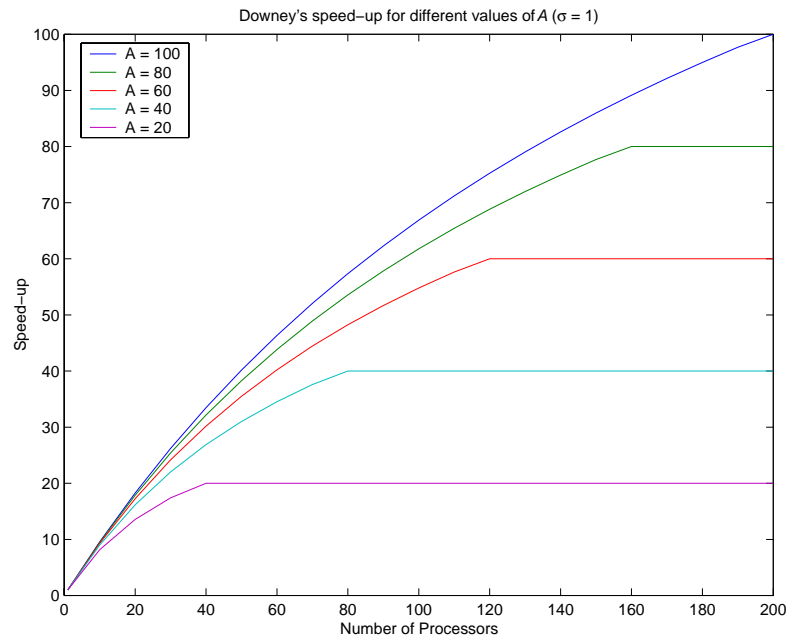


Figure 30 – Downey's speed-up function $S(n, A, \sigma)$ for different values of A

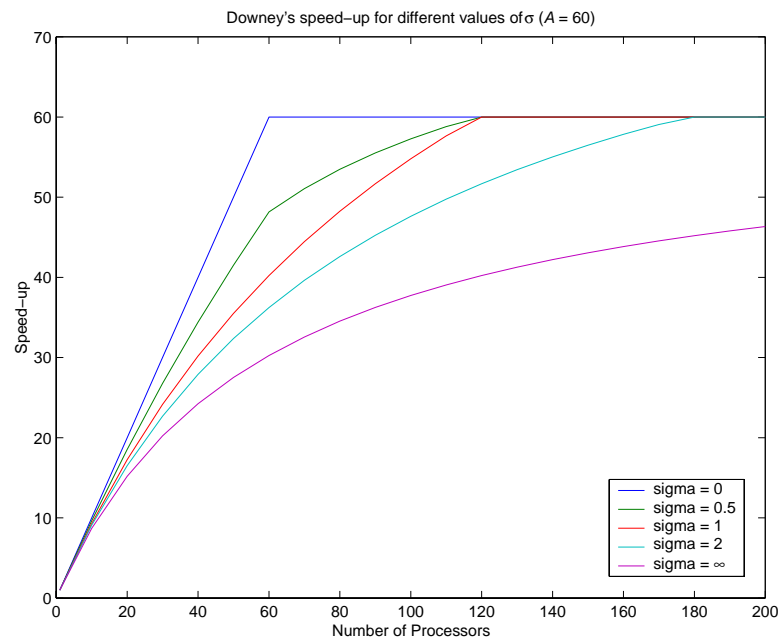


Figure 31 – Downey's speed-up function $S(n, A, \sigma)$ for different values of σ

Using the model for rigid jobs described in Section 3.3, we generate n , tr and a and then derive $te = tr \cdot a$. Note that tr and te are respectively the requested time and execution time for the job j running over n processors. But A , σ , n , and te uniquely determine the sequential execution time L of the job: $L = te(1) = te \cdot S(n, A, \sigma)$. L represents how “large” a job is. The greater the L , the more processing is required to complete the job.

With A , σ and L , we can determine the execution time of the job running over an arbitrary partition size n' by $te(n') = \frac{L}{S(n', A, \sigma)}$. In particular, we generate $\mathbf{te} = (te^{[1]}, \dots, te^{[v]})$ by evaluating $te(n')$ at the partition sizes $\mathbf{n} = (n^{[1]}, \dots, n^{[v]})$. From \mathbf{te} and a , we calculate $\mathbf{tr} = (tr^{[1]}, \dots, tr^{[v]}) = (te^{[1]} / a, \dots, te^{[v]} / a)$.

In order to complete the moldability model, we need to establish how A and σ are distributed. Unfortunately A and σ cannot be directly modeled from the survey. We felt that asking a direct question about the average in parallelism (A) or its coefficient of variance (a close approximation to σ) would be too technical for most users. Instead we indirectly inferred A and σ based on the survey’s questions about the minimum, efficient and maximum partition sizes (questions 4, 6, and 5, respectively).

Modeling A

We use the survey’s *efficient partition size* s_{effic} as an estimate for the average parallelism A . The efficient partition size s_{effic} was defined in the survey as “the partition size beyond which additional processors do not reduce the application’s execution time enough to make it worth requesting them” (see question 6 at Appendix A). The intuition is that s_{effic} represents the “knee” in the speed-up curve, i.e. the point that maximizes the benefit/cost ratio (benefit meaning “lower execution time” and cost meaning “use of more processors”). This concept is in consonance with a more formal analysis of speed-up behavior by Eager et al [36]. Eager et al found that (i) the knee k of the speed-up curve must satisfy $\frac{A}{2} \leq k \leq 2A - 1$, and (ii) adding more processors when the partition size is smaller than A has much greater impact on the execution time than when the par-

tion size is greater than A [36]. These results provide the rationale for modeling A after the efficient partition size s_{effic} .

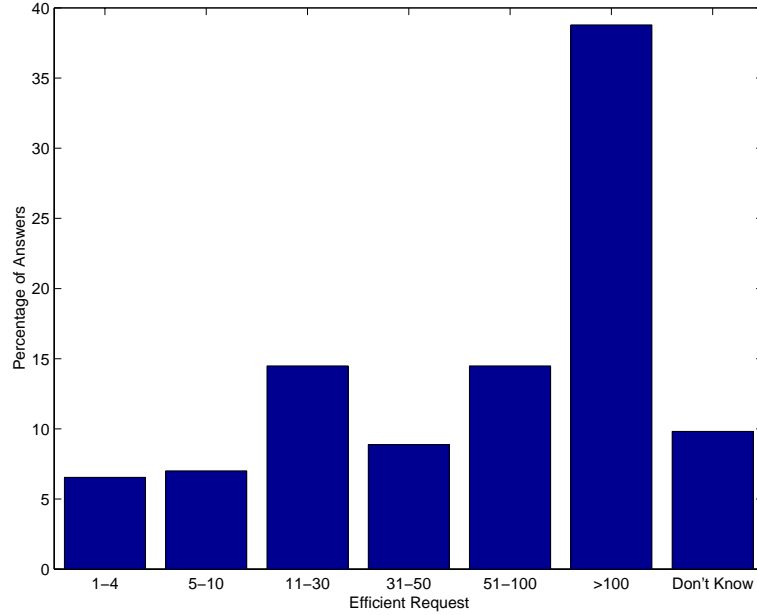


Figure 32 – Survey results for efficient partition size s_{effic}

Figure 32 displays how the efficient partition size s_{effic} was distributed among the survey respondents. Alas modeling A directly after s_{effic} can introduce a bias in the model. This is because s_{effic} and the minimum partition size s_{min} are correlated. Since $s_{min} \leq s_{effic}$, the distribution of s_{effic} skews towards larger values as s_{min} grows. For example, Figure 33 presents the distribution of s_{effic} for the survey responses that had s_{min} in the $[11, 30]$ range. Since we are already using s_{min} to model c_{min} , we should take this correlation into consideration when modeling A .

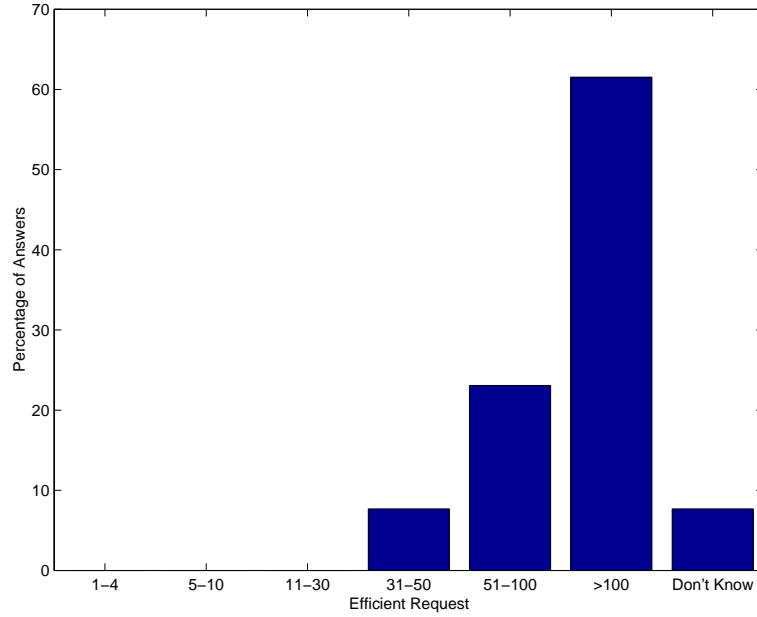


Figure 33 – Distribution of s_{effic} for the responses with s_{min} in the [11,30] range

To understand how s_{min} and s_{effic} interact, consider their joint distribution of probability [29], whose CDF is shown in Figure 34. Note that the Figure's axes are in log scale, which suggests that a generalization of the uniform-log distribution might provide an adequate fit for this joint distribution. Note also that the CDF slope is more accentuated for large values of s_{min} (compared with small values of s_{min}). That is because when s_{min} is large, s_{effic} must also be large, as exemplified in Figure 33.

We are able to capture this behavior by using a *joint uniform-log distribution*, which is a generalization of the uniform-log distribution. The joint uniform-log distribution is determined by parameters ϕ , γ , η and ρ , and has cumulative distribution function $cdf(x, y) = \phi \cdot \log_2(x) \cdot \log_2(y) + \gamma \cdot \log_2(x) + \eta \cdot \log_2(y) + \rho$. Making $x = s_{min}$ and $y = s_{effic}$, we found $\phi_A = 0.009548$, $\gamma_A = -0.01877$, $\eta_A = 0.07468$, and $\rho_A = -0.009198$ via least squares fit. The fit was very good, with correlation coefficient of 0.974. The resulting joint uniform-log distribution for A and c_{min} can be seen in Figure 35.

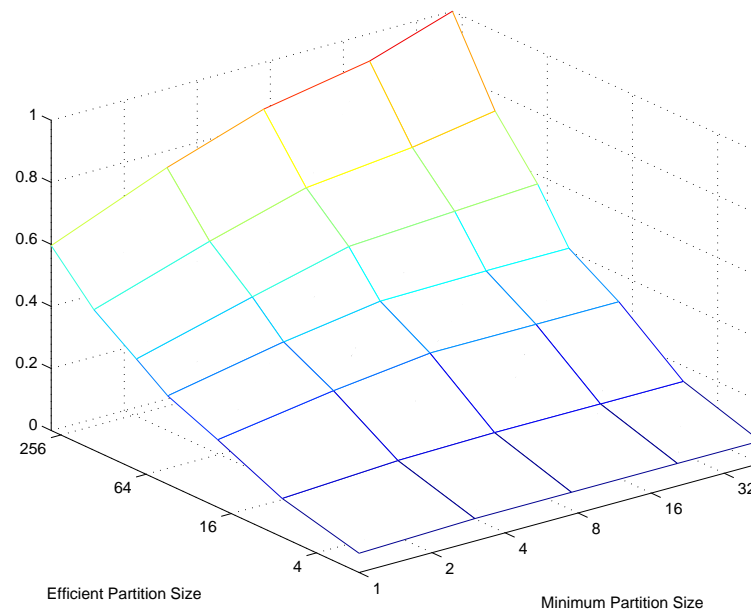


Figure 34 – Joint CDF for s_{effic} and s_{min}

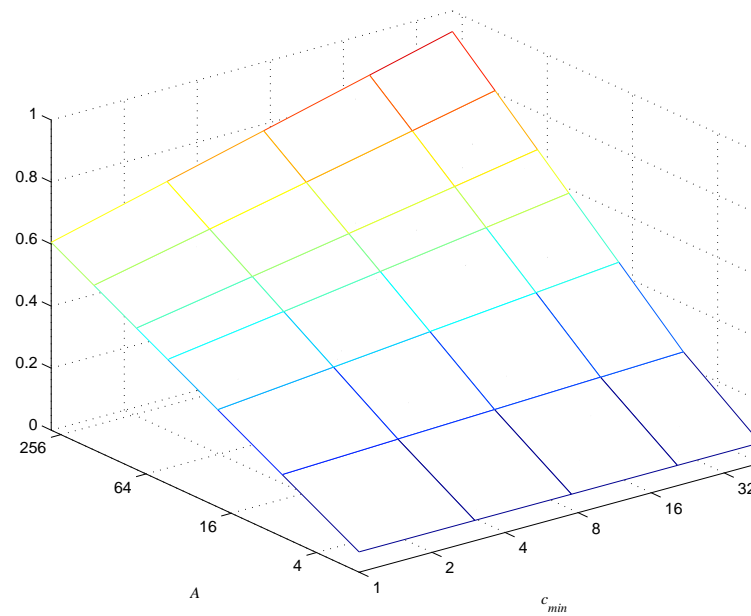


Figure 35 – Model CDF for the joint distribution of c_{min} and A

Modeling σ

As discussed above, σ cannot be directly modeled after a question asked in the survey. Instead we indirectly infer σ based on the relation between the efficient partition size s_{effic} and the maximum partition size s_{max} . The idea is that when the s_{effic} and s_{max} are close, the speedup is close to linear, and thus the job has small σ . Conversely, when s_{effic} and s_{max} are far apart, the speedup should be strongly sublinear, and hence the job has large σ . Figure 31 contains a more graphical representation of this phenomenon.

More specifically, assuming $s_{effic} = A$, $s_{max} = c_{max}$, and using the equations that define the Downey model, we have that (i) $s_{effic} = s_{max} \Leftrightarrow \sigma = 0$, and (ii)

$$\sigma \geq 1 \Rightarrow \sigma = \frac{s_{max} - s_{effic}}{s_{effic} - 1}. \text{ Since } \sigma = \frac{s_{max} - s_{effic}}{s_{effic} - 1} \text{ is a generalization of both equations (i)}$$

and (ii), we use it as the estimate for σ . Figure 36 displays how these estimates for σ are distributed in the survey's results. Note that the discontinuities in the graph are due to the fact that the survey used multiple-choice questions, therefore creating an artificial discretization for s_{effic} and s_{max} .

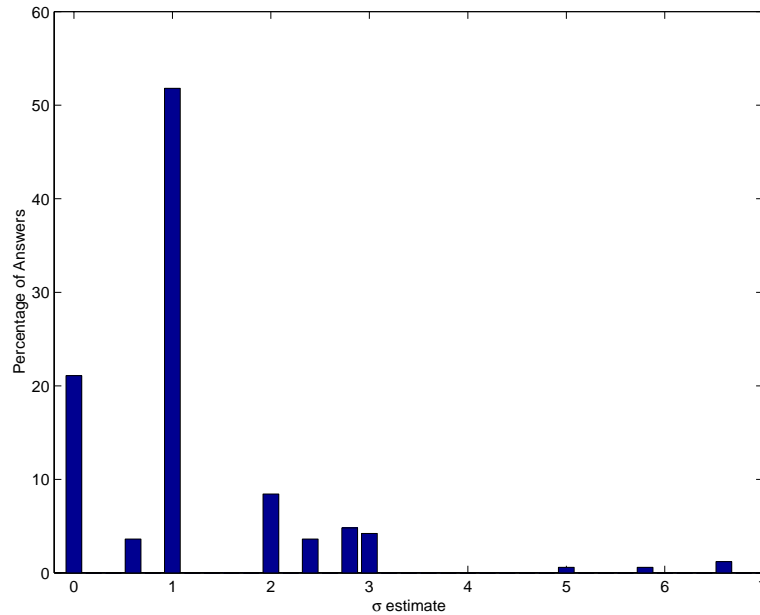


Figure 36 – Distribution of the survey-based σ estimates

We use a normal distribution to model σ . It provides a good fit for the σ estimates derived from the survey, especially when we consider that the discontinuities in the distribution of such estimates are an artifact of the survey. Using maximum likelihood fitting, we obtained parameters $\mu_\sigma = 1.209$ and $\sigma_\sigma = 1.132$. Figure 37 shows the CDF of the model and the observed estimates.

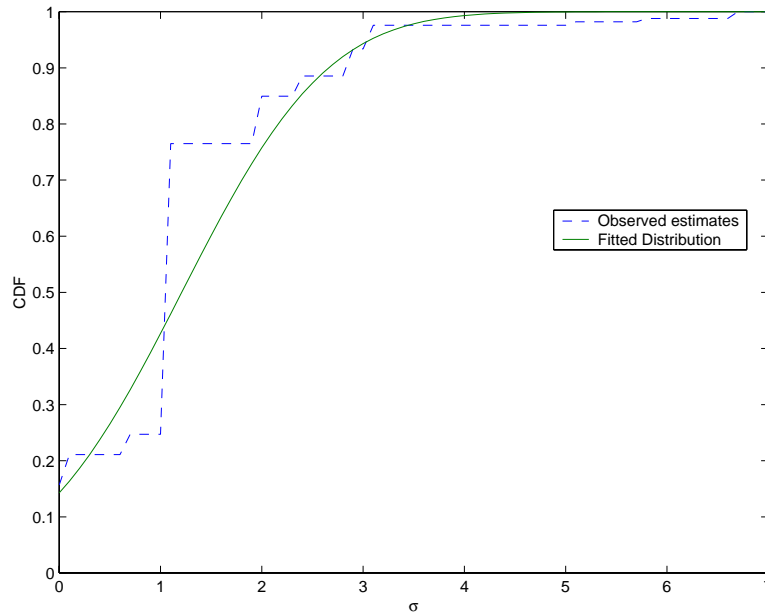


Figure 37 – CDF for the survey based estimate of σ and the corresponding model

3.5. Models Summary

As discussed in this chapter, we generate synthetic moldable workloads by combining two models: the rigid workload model and the moldability model. The rigid workload model produces a stream of jobs, each with one known request. The moldability model generates alternative requests for a given job j , for which only one request is known. Synthetic moldable workloads are obtained by using the rigid workload model to produce a stream of jobs, and then applying the moldability model to the jobs that are to be moldable. Such synthetic workloads are used for performance evaluation throughout this thesis.

This section has three purposes. First, it summarizes the rigid workload model and the moldability model, providing a concise reference that the reader might find useful. Second, it reviews the values we use for each parameter in the models. Third, it delineates how the synthetic workloads are used for performance evaluation.

Rigid Workload Model

A synthetic workload consists of a sequence of jobs. Each job j arrives at the system at a given instant ia . In our model, the jobs' arrival times vary according to the hour of the day. Fitting jobs' arrival times revealed little commonality among the arrival patterns of our four reference workloads (see Section 3.3.1). This leads us to believe that no single model is appropriate to capture the different arrival patterns found in real life. In our model, we employ all four polynomials that fit the arrival rate of our reference workloads (see Table 3) as representatives of the arrivals patterns one might find in practice. Every time a synthetic workload is generated, one of these four polynomials is randomly chosen to provide the arrival rate.

Note that the selected polynomial must be multiplied by P / O , where P is the number of processors of the supercomputer to which the synthetic workload is going to be submitted, and O is the number of processors in the supercomputer the originated the polynomial ($O = 120$ for ANL, $O = 430$ for CTC, $O = 100$ for KTH, and $O = 128$ for SDSC, as shown in Table 1). This normalizes the arrival pattern to the size of the target supercomputer. Not doing so would result in lighter-than-reality loads when $P > O$ (unchanged arrival rate applied to a larger supercomputer), and in heavier-than-reality loads when $P < O$ (unchanged arrival rate applied to a smaller supercomputer).

Each job in the synthetic workload has a probability of being cancelled $pc = 0.15$ (see Section 3.3.2). If a job j is cancelled, the cancellation lag lc determines the time elapsed between the arrival of j and its cancellation. Cancellation lags are uniform log distributed with parameters $\chi_{lc} = 0.065$ and $\rho_{lc} = -0.32$ (see Section 3.3.2). A cancelled job is removed from the system, whether it is running or waiting in the queue.

Each job in the rigid workload is characterized by its partition size n , request time tr , and accuracy a . The partition size n is initially drawn from a uniform log distri-

bution with parameters $\chi_n = 0.12$ and $\rho_n = 0.20$, but with probability $pb = 0.75$ the resulting partition size is changed to the closest power-of-2 value (see Section 3.3.3). The requested time tr is uniform log distributed with parameters $\chi_{tr} = 0.10$ and $\rho_{tr} = -0.75$ (see Section 3.3.4). The accuracy a is gamma distributed with parameters $\alpha_a = 0.6$ and $\beta_a = 0.6$ (see Section 3.3.4). Note that the execution time te is determined by $te = tr \cdot a$.

Table 11 summarizes the rigid workload model, listing the distributions used and their parameters.

Characteristic	Model	Parameters
Arrival Time ia	Polynomial $(P/O) \cdot \lambda_a(m_s)$ describes the arrival rate within a day	The coefficients for the four different $\lambda_a(m_s)$ are displayed in Table 3
Cancellation	Probability pc	$pc = 0.15$
Cancellation Lag lc	Uniform-log distributed	$\chi_{lc} = 0.065$; $\rho_{lc} = -0.32$
Partition Size n	Uniform-log distributed	$\chi_n = 0.12$; $\rho_n = 0.20$
Power-of-2 Partition Size	Probability pb	$pb = 0.75$
Request Time tr	Uniform-log distributed	$\chi_{tr} = 0.10$; $\rho_{tr} = -0.75$
Accuracy a	Gamma distributed	$\alpha_a = 0.6$; $\beta_a = 0.6$

Table 11 – Summary of the Rigid Workload Model

We also define a *load multiplier* κ_L that multiplies the arrival rate *and* the request time generated by the model. The load multiplier κ_L allows us to alter the load offered to the supercomputer, and thus investigate how scheduling solutions behave under different load conditions. By multiplying both the arrival rate and the request time we keep the characteristics of the workload. In particular, this approach does not shorten the daylong arrival cycle, as does compressing the arrival time [34] [44].

Moldability Model

The moldability model generates v requests for a job j with one known request (partition size n , request time tr , and accuracy a). That is, the moldability model produces the v -tuples $\mathbf{n} = (n^{[1]}, \dots, n^{[v]})$, $\mathbf{tr} = (tr^{[1]}, \dots, tr^{[v]})$, $\mathbf{a} = (a^{[1]}, \dots, a^{[v]})$ that describe v

requests for job j . The modeling of \mathbf{a} is straightforward: $a^{[1]}, \dots, a^{[v]} = a$ (see Section 3.4.2).

As described in Section 3.4.1, v is no greater than the number of requests the user is willing to provide c_u . We have that $\Pr[c_u = 1] = 0.05$ and $\Pr[c_u > 1] = 0.95$. If $c_u > 1$, its value is uniform-log distributed with parameters $\chi_{cu} = 0.1918$ and $\rho_{cu} = 0.1876$. Additionally, the partition sizes generated by the moldability model $n^{[1]}, \dots, n^{[v]}$ are in the $[c_{min}, c_{max}]$ range. The minimum partition size c_{min} is uniform-log distributed with parameters $\chi_{cmin} = 0.06920$ and $\rho_{cmin} = 0.6279$. As explained in Section 3.4.3, the

maximum partition size c_{max} is determined using $c_{max} = \begin{cases} 2A - 1 & (\sigma \leq 1) \\ A + A\sigma - \sigma & (\sigma \geq 1) \end{cases}$.

As discussed in Section 3.4.3, the request time for a given partition size n' is calculated by $tr(n') = \frac{L}{S(n', A, \sigma) \cdot a}$. A is jointly uniform-log distributed together with c_{min} , using parameters $\varphi_A = 0.009548$, $\gamma_A = -0.01877$, $\eta_A = 0.07468$, and $\rho_A = -0.009198$. σ is uniformly distributed with parameters $\mu_\sigma = 1.209$ and $\sigma_\sigma = 1.132$.

Table 12 condenses the essential information about the moldability model, listing the distributions used and their parameters.

Characteristic	Model	Parameters
Minimum Partition Size c_{min}	Uniform-log distributed	$\chi_{c_{min}} = 0.06920$; $\rho_{c_{min}} = 0.6279$
Number of Requests Provided by the User c_u	Probability determines whether $c_u = 1$. Uniform-log distributed for $c_u > 1$.	$\Pr[c_u = 1] = 0.05$; $\Pr[c_u > 1] = 0.95$; $\chi_{c_u} = 0.1918$; $\rho_{c_u} = 0.1876$
Downey's A	Jointly uniform-log distributed with the minimum partition size c_{min}	$\phi_A = 0.009548$; $\gamma_A = -0.01877$; $\eta_A = 0.07468$; $\rho_A = -0.009198$
Downey's σ	Normally distributed	$\mu_\sigma = 1.209$; $\sigma_\sigma = 1.132$

Table 12 – Summary of the Moldability Model

Using the Workload Models

The separation of our moldable workload model into two independent parts allows us to investigate the performance of **SA** on *current* workload conditions. This is done by using the moldability model on a *single* job in the rigid workload, and repeating this experiment multiple times to understand the performance of such a moldable job (see Chapter 5).

Of course the performance of **SA** scheduling many jobs is also of interest. We also investigate this scenario by using the moldability model on *all* jobs generated by the rigid workload model. That is the focus of Chapter 6.

4. Performance Metrics for SA

The *metric* used to compare competing solutions is a key aspect of performance evaluation. Since the use of inappropriate metrics can result in misleading conclusions [34] [44], one wants to find a metric that is unbiased and that captures our intuition of good performance for the target scenario.

SA aims to improve the performance of one job. Therefore, our performance metric should capture our intuitive notion of individual job performance. This chapter discusses how to gauge job performance, and how multiple experiments can be statistically aggregated without biasing the results.

4.1. Gauging Job Performance

Job performance should be evaluated from the user's point of view. After all, a job exists to produce results to its user. Turn-around time captures the user's view of how long the system takes to run a job. The *turn-around time*¹ tt of a job j is the time elapsed between j 's submission and its completion. That is: $tt = tw + te$, where tw is the queue wait time, and te is the execution time of the job. We use turn-around time throughout this thesis to measure the performance of a job.

4.2. Aggregating Experiments

As we shall see in Chapters 5 and 6, the evaluation of SA is based on *experiments* on which a target job j is submitted using different requests, such as the traditional static request provided by the user and the request chosen by SA. Since each experiment focuses on one target job j , it is straightforward to use the turn-around time to measure the performance obtained by the different requests that compose an experi-

ment. However, in order to draw statistically valid conclusions, we need to perform experiments in a variety of circumstances. Consequently, we need a way to summarize multiple turn-around times in a single value.

Note that any method of combining multiple turn-around times in a single value can serve as a performance metric for supercomputer scheduling. However, the opposite is not necessary true. Some metrics used to evaluate supercomputer schedulers, such as makespan, throughput, and system utilization, are not directly based on job performance. One can always argue that a system with good performance according to one of these metrics probably provides good service for the jobs that formed the evaluation workload, and conclude that these resource-centric metrics could thus be used to gauge job performance. However, the jobs used to gauge the performance of **SA** do not necessarily come from the same workload. In many of our experiments, the target jobs come from different workloads. Moreover, it has been shown that makespan, system utilization, and throughput are not appropriate metrics for on-line schedulers (such as supercomputer schedulers) because they are strongly influenced by the job arrival and the job requirements, which are out of control of the scheduler [44].

Of course, there are metrics for supercomputer scheduling that are based on job performance. In fact, two of these metrics – *mean turn-around time* and *mean slowdown* – are popular ways to determine the performance of supercomputer schedulers [44]. As we shall see, however, these metrics are not appropriate for our research scenario because they bias towards long jobs [34] [44] and/or reward performance-poor scheduling strategies for moldable jobs. We will argue that the *geometric mean of turn-around times* is an appropriate performance metric for our research scenario.

Mean Turn-Around Time

Since turn-around time provides a good metric for a single job, many researchers have used the *arithmetic mean* $\text{mean}(x_1, \dots, x_n) = \frac{x_1 + \dots + x_n}{n}$ to combine the turn-

¹ Turn-around time is also referred to as *service time* or *response time*.

around times of all jobs in the workload into a metric for the supercomputer scheduler [1] [44] [64]. The mean has the advantage of being easily understood and widely used to combine multiple experiments into a single value. For example, the arithmetic mean is a sensible choice for establishing the “true” value of physical measurements because measurement errors are reduced by repeating the *same* trial and averaging the results.

However, a set of turn-around times of different jobs cannot be considered a number of measurements of the same trial. In typical supercomputer workload, jobs differ widely in execution time (and thus in turn-around time), as seen in Section 3.3.4. The problem this causes is that mean turn-around time can be dominated by long jobs [34] [44]. For example, the mean turn-around time for 100 one-hour jobs and 1 one-week job is 2.7 hours. For another example, improving a job’s turn-around time from 20000 seconds to 18000 seconds (a 10% improvement) reduces the mean turn-around time by $2000 / J$, while improving another job’s turn-around time from 200 seconds to 100 seconds (a 50% improvement) reduces the mean only by $100 / J$, where J is the total number of jobs in the evaluation workload.

The dominance of long jobs on the mean turn-around time is an undesirable property for a performance metric because short jobs are most common in today’s workloads (see Section 3.3 and [44]). Therefore a scheduler can be ranked superior even if it increases the turn-around time of most jobs (the short ones).

Mean Slowdown and its Derivatives

Some authors have addressed this problem by using the *slowdown*² $s = tt / te$ instead of the turn-around time [43] [44] [107]. Slowdown provides a measure that is normalized by the job’s execution time and hence long jobs are not overemphasized in the mean slowdown.

A problem with slowdown is that jobs with extremely short execution time incur very large slowdown. For example, a one-second job that waits 10 minutes in the queue has a slowdown of 600. The standard solution for this problem is to establish a lower

² Slowdown is also referred to as *expansion factor*.

bound for the execution time, typically 10 seconds [43] [44] [107]. More precisely, the performance of the workload is measured by the *mean bounded slowdown*, where bounded slowdown $bs = \frac{tt}{\max(10, te)}$. Returning to the example, a one-second job that waits for 10 minutes to run has bounded slowdown of 60.

However, slowdown and its derivatives are not appropriate for moldable jobs because the execution time of a moldable job depends on the partition size it uses. For moldable jobs, one can often improve the slowdown by increasing the execution time te , which can be accomplished by selecting the smallest possible partition size. Since $s = \frac{tt}{te} = \frac{te + tw}{te}$, increasing te often leads to a small slowdown s . The problem is that such a strategy can (and often does) *increase* the turn-around time.

Geometric Mean of Turn-Around Times

The *geometric mean* $\text{geomean}(x_1, \dots, x_n) = \sqrt[n]{x_1 \cdot \dots \cdot x_n}$ equally rewards the improvement in the turn-around time of any job in the workload. In fact, it is clear from the definition of geometric mean that $\frac{\text{geomean}(x_1, \dots, x_n)}{\text{geomean}(y_1, \dots, y_n)} = \text{geomean}(\frac{x_1}{y_1}, \dots, \frac{x_n}{y_n})$. Therefore, unlike the arithmetic mean, the geometric mean does not favor long jobs. For this very reason, the geometric mean is used to aggregate the execution time of the programs that compose the Spec benchmark [87].

A criticism of the geometric mean is that it doesn't indicate the processing time of the workload [73]. However, we are not using the performance metric for this purpose here. Instead, we use the performance metric to *compare* alternative scheduling solutions. For this goal, the geometric mean is a good way to aggregate multiple turn-around times because it equally considers the improvement in performance of any job. Hence we use the geometric mean of the turn-around times throughout this thesis to evaluate the performance of a set of experiments.

5. The Performance of SA

SA seeks to reduce the turn-around time of a moldable job j by adaptively selecting the request that submits j to the supercomputer. But **SA** does not always select the best request because (i) the execution times of the jobs in the system are not known (request times are used as estimates), and (ii) future arrivals can affect jobs already in the system (see Chapter 2). **This Chapter investigates the performance SA delivers in current real-life scenarios.** More precisely, this chapter addresses three important research questions regarding the effectiveness of **SA**:

- i) What performance improvement can **SA** deliver in real-life scenarios?
- ii) Which factors influence **SA**'s performance?
- iii) What is the maximum performance improvement attainable by adaptively selecting the request that submits a moldable job? How close does **SA** get to such a maximum?

This chapter is organized as follow. Section 5.1 describes the experiments we conducted to answer the research questions stated above. Section 5.2 analyzes the results of the experiments as a whole, establishing the average performance improvement that **SA** is expected to deliver in real-life conditions. Section 5.3 considers how the results of the experiments are influenced by parameters that describe the job, the system, and the information available to **SA**. Finally, validation for the experimental set-up is provided in Section 5.4.

5.1. Experimental Set-up

We conducted 360000 experiments to investigate the performance **SA** delivers in current real-life conditions. Each experiment targets a single job j and establishes (i) the turn-around time of j when it is submitted using the user request, (ii) the turn-around time obtained using **SA** to determine a request for j , and (iii) the best turn-around time

among all requests that were available to **SA**. Each experiment focuses on a single job j because the characteristics of the workload can change when many jobs are scheduled by **SA** (as we shall see in Chapter 6). By using **SA** on a single job, we do not significantly alter current real-life conditions.

The large number of experiments (360000) was necessary because jobs vary widely in many aspects (see Chapter 3), and therefore statistics that express the behavior of a set of jobs converge slowly [44]. This is the case for the geometric mean of turn-around times (the performance metric employed in this thesis, see Chapter 4). For our experiments, the geometric mean of turn-around times stabilized at around 30000 trials. Since we group experiments into deciles to investigate the effect some parameters have on **SA** (as we shall see in Section 5.3), we had to assure that each decile would have more than 30000 experiments.

For each experiment, we generated a 10000-job workload using the workload model described in Section 3.5. The simulated supercomputer for our experiments had 500 processors and was scheduled with conservative backfilling (see Section 2.2.1). The target job j is randomly selected and has v requests created by the moldability model (Section 3.4). This generates $v + 1$ workloads that differ only regarding job j . One workload has j as a moldable job, with v alternative requests. The other v workloads have j as a rigid job: there is one workload with j as a rigid job for each request j can use (including the original user request). Each of these workloads is then simulated, with **SA** being used only for the workload that has j as a moldable job.

Note that the request with smallest turn-around time among all v static requests is the *best request* that could be chosen by any application scheduler that adaptively selects the request the submits job j . The best request thus provides a bound to the performance improvement that can be achieved with **SA**. As we shall see in Sections 5.2 and 5.3, the request selected by **SA** and the best request exhibit similar behavior in most circumstances. This creates the need to often refer to “the **SA** request and the best request” throughout the rest of this thesis. Since the best request can be thought as a per-

fect application scheduler that adaptively selects a request for a moldable job, we define the term *adaptive requests* to mean “the SA request and the best request”.

5.2. Overall Performance

Table 13 shows the overall results for the 360000 experiments described in Section 5.1. The experiments show substantial improvement in adaptively selecting super-computer requests. The turn-around time of the best request is in general about 44% of the turn-around time obtained by the user request. In addition, SA is able to deliver turn-around times that are close to the best request (SA’s turn-around times are only 13% greater than those attained by the best request). In our experiments, SA is able to reduce the turn-around time to about half of that obtained by the user request.

	Best	SA	User
Geometric Mean of the Turn-Around Time	1264	1429	2878

Table 13 – Overall results (in seconds)

Table 13 conveys a notion of “average” performance. It is also interesting to understand the how often SA improves an individual job’s turn-around time. Table 14 partitions the jobs in those whose turn-around times improved, remained the same, and worsen with the adaptive requests. Of course no job has the turn-around time worsen by the best request.

	Best	SA
Jobs with <i>better</i> turn-around time	53.9%	45.8%
Jobs with <i>same</i> turn-around time	46.1%	45.3%
Jobs with <i>worse</i> turn-around time	-----	8.8%

Table 14 – How the adaptive requests impacted on the turn-around time

Note that the distribution of v , i.e. the number of requests offered to **SA** (see Section 3.4.1), defines the percentage of jobs for which an adaptive request is the same as the user request. In fact, for a job with a single request ($v = 1$), it is clear that an adaptive request and the user request are always the same. When $v = 2$, there is a 50% chance that an adaptive request equals the user request, and so on. Therefore, the fraction of jobs for which an adaptive request and the user request are expected to coincide

in our experiments is $\frac{\sum_j 1/v_j}{J} = 0.448$, where v_j is the number of requests available for job j , and J is the number of jobs in the workload.

Figure 38 provides a more detailed view on how jobs had their turn-around times improved, unchanged, and worsen by the adaptive requests. It shows the distribution of the *relative turn-around time* for **SA** and the best request. The *relative turn-around time* of an adaptive request is the ratio of the turn-around achieved by such adaptive request to the turn-around time obtained by the corresponding user request. Relative turn-around time expresses the performance **SA** and the best request achieved as a fraction of the turn-around time of the user request. Therefore, relative turn-around times smaller than 1 imply that the adaptive request delivered a turn-around time smaller than the one obtained by the user request. Conversely, a relative turn-around time greater than 1 denotes that the user request had smaller turn-around time than the corresponding adaptive request. Of course, the relative turn-around time of the best request cannot be greater than 1. In our experiments, this happens for **SA** 8.8% percent of the time, as shown in Table 14.

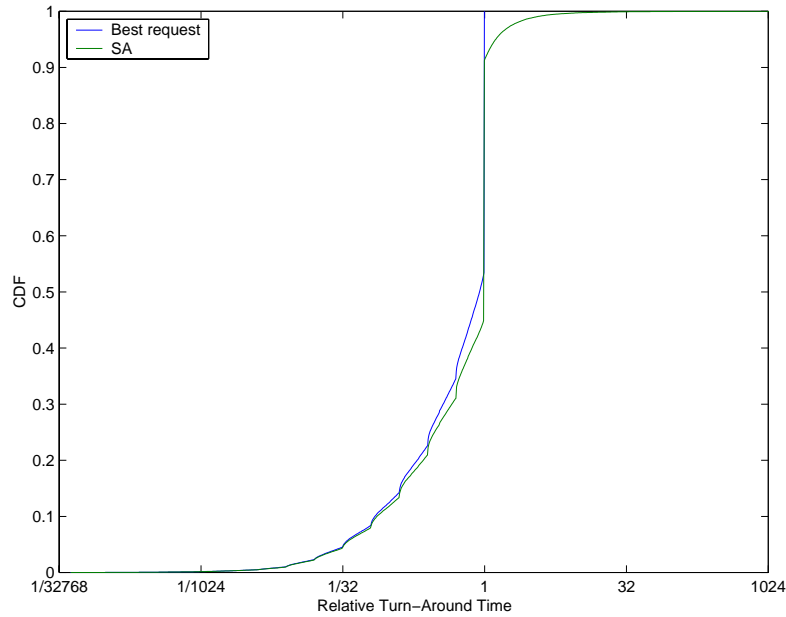


Figure 38 – Distribution of relative turn-around time

5.3. Factors that Influence SA

The performance of **SA** can be influenced by (i) the characteristics of the target job j , (ii) the information available to **SA** about job j , and (iii) the load of the supercomputer at the moment **SA** schedules job j . In this section, we investigate how these factors impact the adaptive requests: the **SA** request and the best request (which corresponds to the maximum possible performance that can be achieved by any of the v possible requests).

Our initial expectation is that the larger and more diverse the set of possible requests, the better the performance **SA** and the best request should attain. The rationale is that a large and diverse set of requests gives more latitude in finding a good request to use. The results confirmed such expectation, and also revealed other characteristics of the behavior of **SA** and the best request, as we will see next.

5.3.1. Job Characteristics

Jobs vary regarding size (i.e., the amount of computation they need to complete), speed-up characteristics, and restrictions on which partition sizes they can use. We use the sequential execution time L as a measure for job size. Note that we cannot employ the *computation time* $ce = te \cdot n$ actually used by a job as a measure for job size because such a variable depends on \mathbf{SA} ³. \mathbf{SA} selects the partition size n that a job uses, therefore affecting $ce = te \cdot n$.

The parameters of the Downey model can be used to characterize the speed-up behavior of a moldable job j . Recall that Downey's parameters are the average parallelism A and an approximation to the coefficient of variance in the parallelism σ . A indicates how many processors j can effectively use, and σ denotes the slope of j 's speed-up (the closer σ is to 0, the closer to linear the speed-up is)⁴.

The partition size constraints are tracked through three parameters: the minimum partition size c_{min} , the maximum partition size c_{max} , and the kind of partition size c_{kind} . The kind of partition size c_{kind} differentiates between power-of-2 and non-power-of-2 jobs. Recall that there is a probability $pb = 0.75$ that the partition size is a power of two. This requirement captures the current practice for partition size selection and seems to be stronger than intrinsic algorithmic constraints (see Section 3.3.3).

Sequential execution time L

Figure 39 shows the results of the experiments described in Section 5.1 as a function of the sequential execution time L . More precisely, Figure 39a displays the results directly as geometric mean of turn-around times, whereas Figure 39b presents the same results using relative turn-around times. Recall that the *relative turn-around time* of an adaptive request is the ratio of the turn-around time of the adaptive request to the turn-around time of the corresponding user request. Relative turn-around time is useful

³ Recall that te is job's execution time and n is its partition size.

⁴ See Section 3.4.3 for a thorough description of the Downey model.

in assessing the impact of the sequential execution time L on the adaptive requests because L and the turn-around time tt are correlated: Larger jobs in general have a greater turn-around time because they take longer to execute. Relative turn-around time normalizes the turn-around time of the adaptive requests with respect to the user request. Consequently, it factors out the correlation between the parameter being studied (in this case, the sequential execution time L) and the turn-around time.

In Figure 39, the experiments are grouped in deciles according to L . Since we conducted 360000 experiments (see Section 5.1), each data point in the graph averages around 36000 experiments. The values of L on the x-axis show the boundaries of the deciles⁵. That is, the values that surround a given data point denote the range averaged by such a point. For example, the first data point represents the jobs with $L \in [0, 92)$, the second data corresponds to the jobs with $L \in [92, 348)$, and so on. Unless stated otherwise, the following graphs use this same convention.

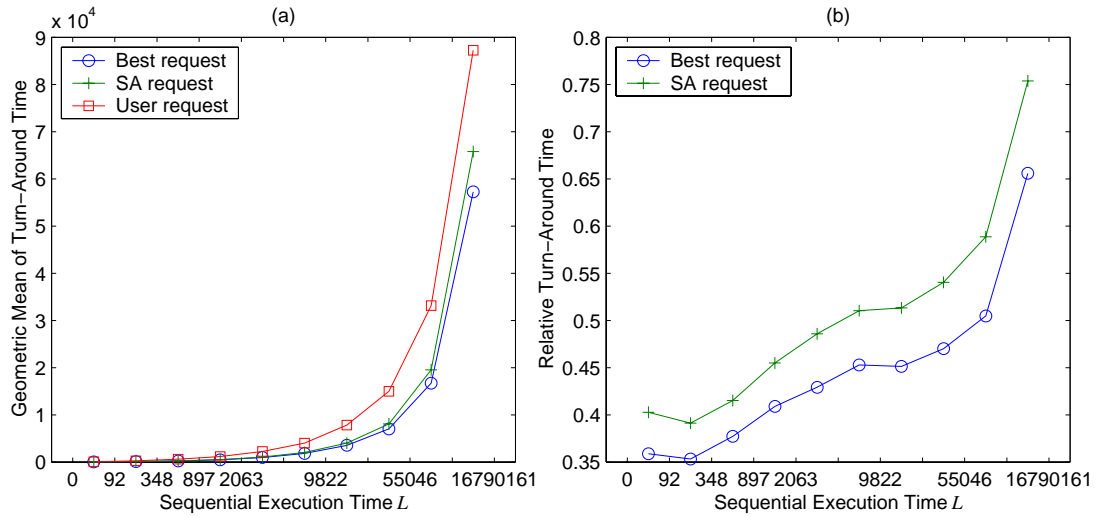


Figure 39 – Turn-around time by the sequential execution time L

⁵ Some values in the x-axis (i.e. deciles boundaries) are omitted due to space constraints.

As can be seen in Figure 39a, the larger the L (i.e., the more computation a job carries), the greater the turn-around time. This is because large jobs naturally have long execution times (see Figure 40a), and a long execution time contributes to an increase in the turn-around time. The increase in turn-around time with the growth of L makes it hard to visualize other patterns in Figure 39a.

The relative turn-around time graph (Figure 39b) provides a more insightful picture. The relative turn-around time increases as L grows, meaning that the improvement in the turn-around time delivered by the adaptive requests decreases with L . The behavior of the best request (which is closely followed by **SA**) indicates that there is less room for performance improvement as L grows.

Decomposing the turn-around time into execution time (Figure 40) and wait time (Figure 41) helps in understanding why large jobs gain less from the adaptive requests. L does not seem to exert a clear influence on the relative execution time (see Figure 40b). On the other hand, the relative wait time clearly grows with L (see Figure 41b). In fact, for large values of L , the adaptive requests incur wait times greater than the ones obtained by the user requests (see Figure 41a). Nevertheless, the turn-around time of the adaptive requests is better than that obtained by the user request even for large jobs (see Figure 39a). This implies that, for large jobs, the reduction in execution time is greater than the increase in wait time. However, for small jobs, adaptive requests are able to simultaneously reduce the execution time (see Figure 40b) *and* the wait time (see Figure 41b), which translates into a greater improvement in the turn-around time.

We believe that the large wait times faced by large jobs when using the adaptive requests are due to the inability of large jobs to use small holes in the supercomputer schedule. Recall that **SA** works by searching for a “good” hole in the supercomputer schedule, picking the hole that gives the job being scheduled the soonest *expected* finish time. Similarly, the best requests can be thought of as an scheduler that *always* finds the hole in the supercomputer schedule that delivers the earliest expected finish time for the job being scheduled. We conjecture that the holes that exist in the beginning of the supercomputer schedule (i.e., the ones that incur small wait time) are small because re-

quests are placed in the supercomputer schedule using first fit (see Section 2.2.1). Since a job with large L can only be placed in large holes, it seems to be more likely for such a job to be placed towards the end of the supercomputer schedule, which results in a large wait time. As we shall see soon, jobs with large values for the minimum partition size c_{min} also seem to experience the same phenomenon, reinforcing our conjecture that the ability to use small holes in the supercomputer schedule is key for achieving small wait times.

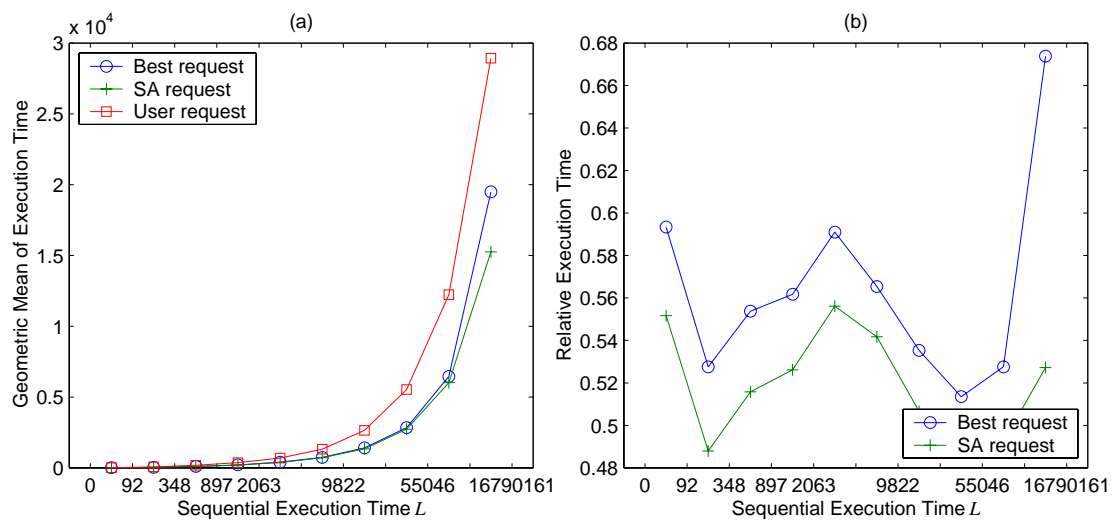


Figure 40 – Execution time by the sequential execution time L

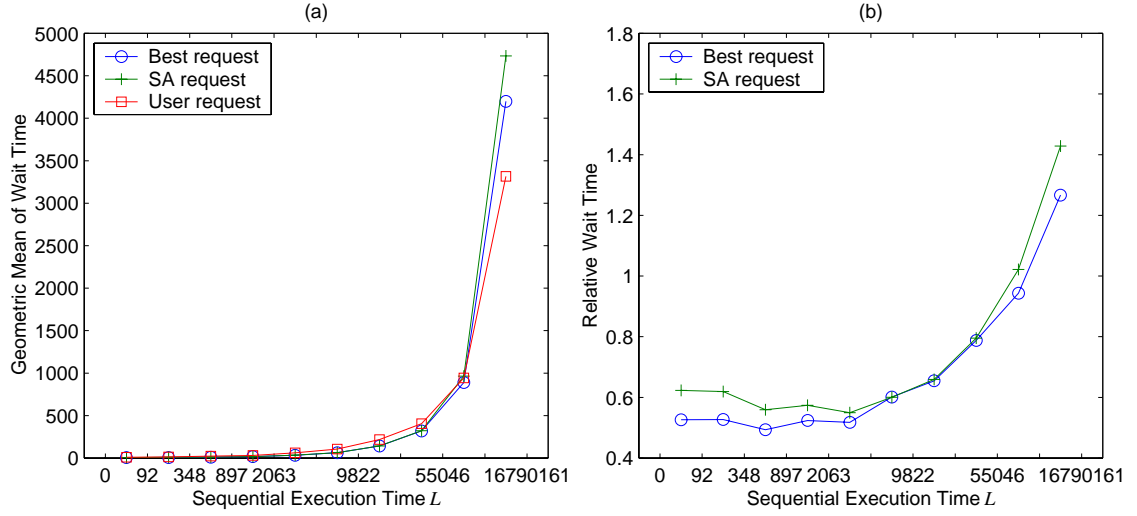


Figure 41 – Wait time by the sequential execution time L

Note that Figure 40 presents the results directly as execution time and also as *relative execution time*. The *relative execution time* of an adaptive request (the SA request or the best request) is the ratio of the execution time of the adaptive request to the execution time of the corresponding user request. Likewise, Figure 41 displays the results directly as wait time and also as *relative wait time*. The *relative wait time* of an adaptive request is the ratio of the wait time experienced by such an adaptive request to the wait time of the user request. As it happens with the turn-around time, both wait time and execution time are correlated with parameters whose effect on the adaptive requests we intend to investigate (in this case, the sequential execution time L). Relative measures for wait time and execution time address this issue because they eliminate the correlation between the parameter being studied and both wait time and execution time.

Average parallelism A

Figure 42 shows the impact of the average parallelism A on the turn-around time of the target job j . The relative turn-around time graph (Figure 42b) indicates that the adaptive requests are less effective in reducing the job's turn-around time for small values of A . Since A determines the maximum speed-up that can be achieved by job j (see Section 3.4.3), the possible requests for jobs with small A do not vary in execution time

as much as the requests for jobs with large A . Requests with similar execution time give less latitude for **SA** and the best request in reducing the job's execution time (see Figure 43b), making it harder for the adaptive requests to improve the job's turn-around time.

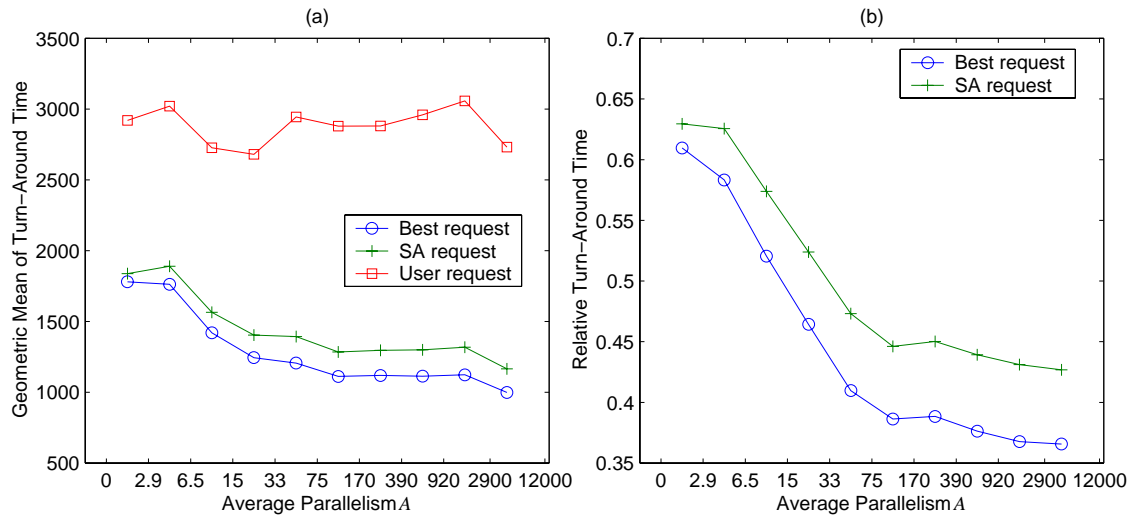


Figure 42 – Turn-around time by average parallelism A

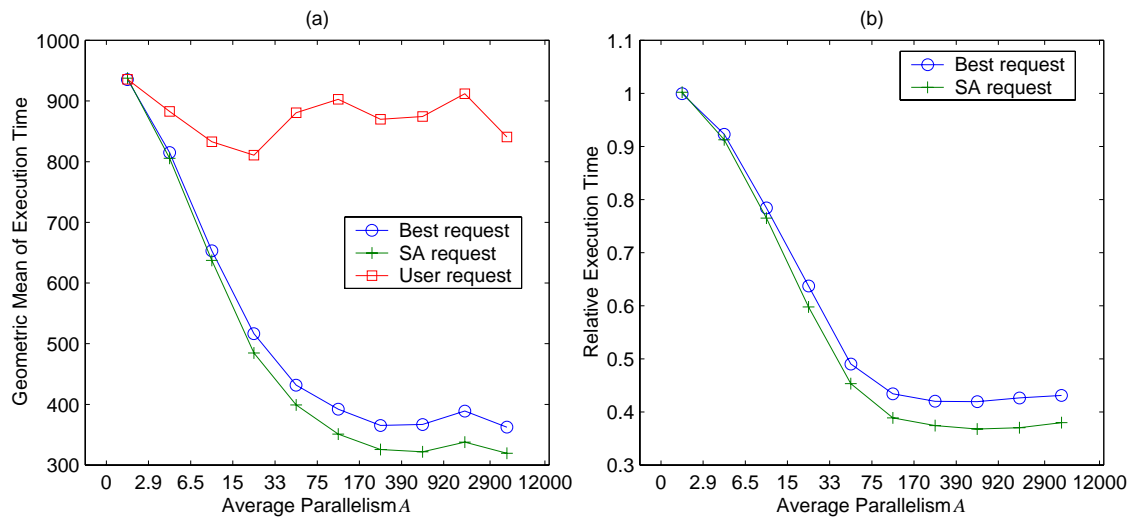


Figure 43 – Execution time by average parallelism A

Note also that the decrease of the relative turn-around time of the adaptive requests appears to taper at around $A = 100$ (Figure 42b). This suggests that a job does not need high average parallelism A in order to fully benefit from SA.

σ

Figure 44 presents the effect of σ on the turn-around time of target job j in the experiments. Somewhat surprisingly, variations in σ show very little impact on the performance of either SA or the best request. A large σ implies that the job's speed-up is strongly sublinear. Therefore, the possible requests for a large- σ job vary considerably in their computation time ($ce = n \cdot te$). The results of the experiments indicate that such a variance in the computation time is not very important for the adaptive requests. It seems that having multiple distinct requests to choose from is the key enabling feature to improve the job's turn-around time by adaptively selecting which request to use.

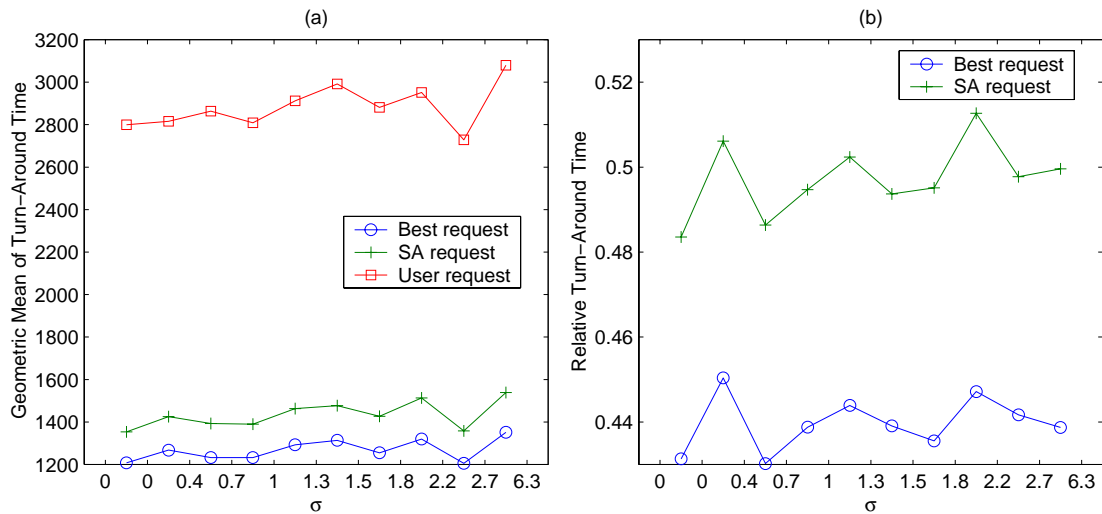


Figure 44 – Turn-around time by σ

However, if *many* jobs with large values of σ have their requests adaptively chosen (not the case addressed in this chapter), it is possible for the system as a whole to exhibit poor emergent behavior. Poor emergent behavior can arise if most of the selected requests are large. Multiple large requests boost the supercomputer load, which

typically increases the turn-around time of most jobs in the system. Section 6.2.2 investigates this issue.

Minimum partition size c_{min}

The impact of the minimum partition size c_{min} on the adaptive requests and the user request can be seen on Figure 45. Note that Figure 45 has only 5 data points, whereas the previous figures have 10 data points. This is because $c_{min} = 1$ for 62.6% of the experiments (see Section 3.4.1 for the distribution of c_{min}). Therefore, the first data point roughly represents the six first deciles of the distribution.

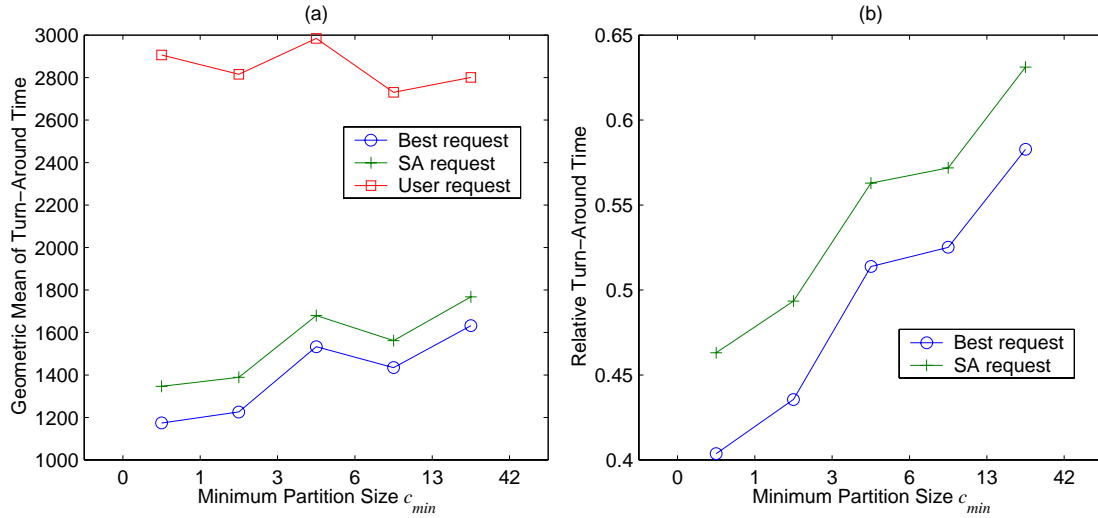


Figure 45 – Turn-around time by minimum partition size c_{min}

As can be seen in Figure 45b, the effectiveness of the adaptive requests decreases as c_{min} increases. A large c_{min} implies that there are no requests with small partition size n available to be selected. Small partition sizes allow the use of small holes in the supercomputer schedule. As discussed in the analysis of the results with respect to the sequential execution time L , it seems that the ability to use small holes in supercomputer schedule is essential to reduce the wait time. In fact, as can be seen in Figure 46, the wait time of the adaptive requests grows with c_{min} , even surpassing the wait time of the user request for $c_{min} > 13$. Such a lack of ability to reduce wait time seems to com-

promise the capacity of the adaptive requests to improve the turn-around time, as observed for jobs with large c_{min} and large L .

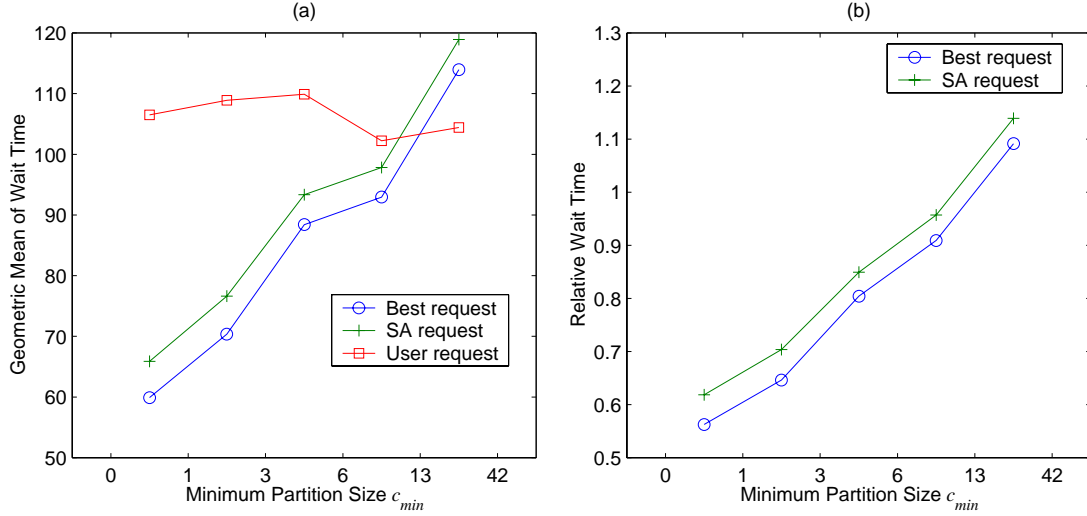


Figure 46 – Wait time by minimum partition size c_{min}

Maximum partition size c_{max}

Figure 47 shows the effect of the maximum partition size c_{max} on the turn-around time. Up to certain point (around $c_{max} = 100$), the performance improvement generated by the adaptive requests grows as c_{max} grows. After that point, the performance improvement practically levels off.

It appears that the restriction introduced by a small c_{max} reduces the capability of the adaptive requests to reduce the job's execution time. Small execution times often require the use of many processors and c_{max} poses an upper bound on how many processors a job can use. That is, jobs with small c_{max} cannot use many processors. This seems to preclude SA and the best request from improving the job's turn-around time by reducing its execution time. The behavior of the execution time as c_{max} varies (shown in Figure 48) supports this explanation.

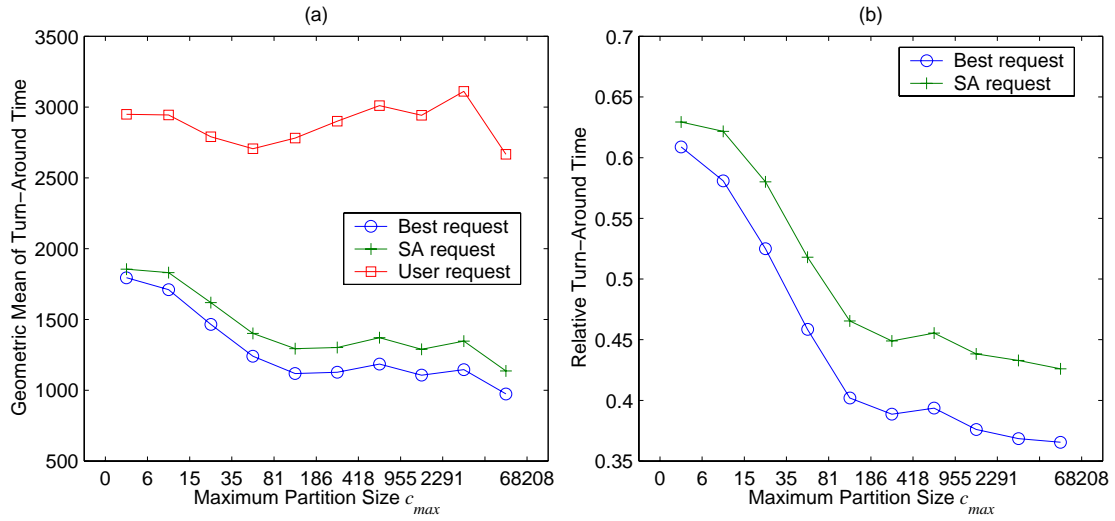


Figure 47 – Turn-around time by maximum partition size c_{max}

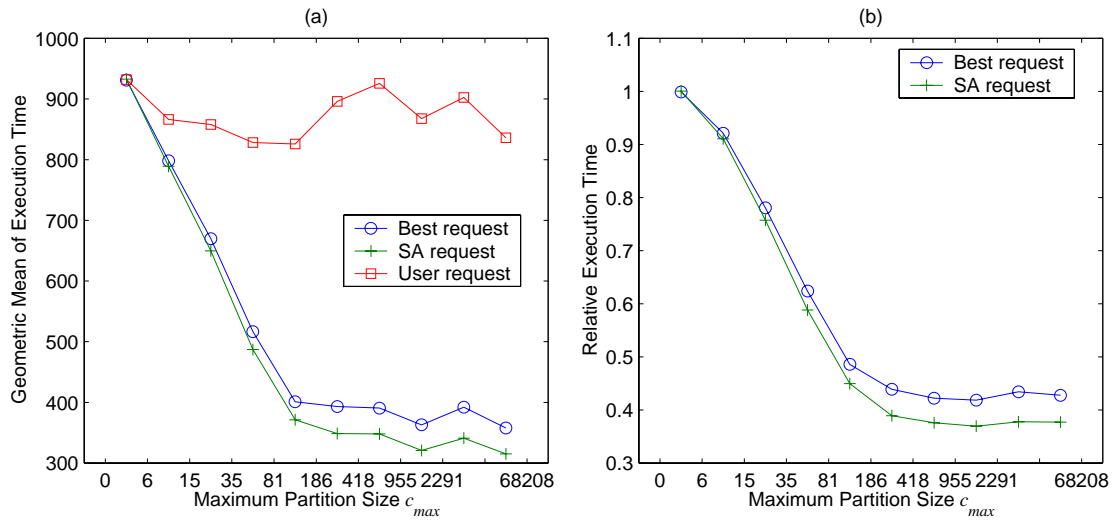


Figure 48 – Execution time by maximum partition size c_{max}

Note that the effect of the maximum partition size c_{max} on the turn-around time is similar to the effect caused by the average parallelism A . This may not be a coincidence. Both c_{max} and A pose restrictions on the speed-up behavior of a job: c_{max} repre-

sents the partition size n after which the speed-up curve levels off, whereas A denotes the speed-up value after which the speed-up curve levels off.

Kind of partition size c_{kind}

In our model, 75% of the jobs use power-of-2 partition sizes. This enables us to capture the current practice for partition size selection, which seems to be stronger than intrinsic algorithmic constraints (see Section 3.3.3). Figure 49 segregates the power-of-2 jobs from the non-power-of-2 jobs. Non-power-of-2 jobs experience greater turn-around time than power-of-2 jobs. Moreover, the performance improvement obtained by SA and the best request is smaller for non-power-of-2 jobs.

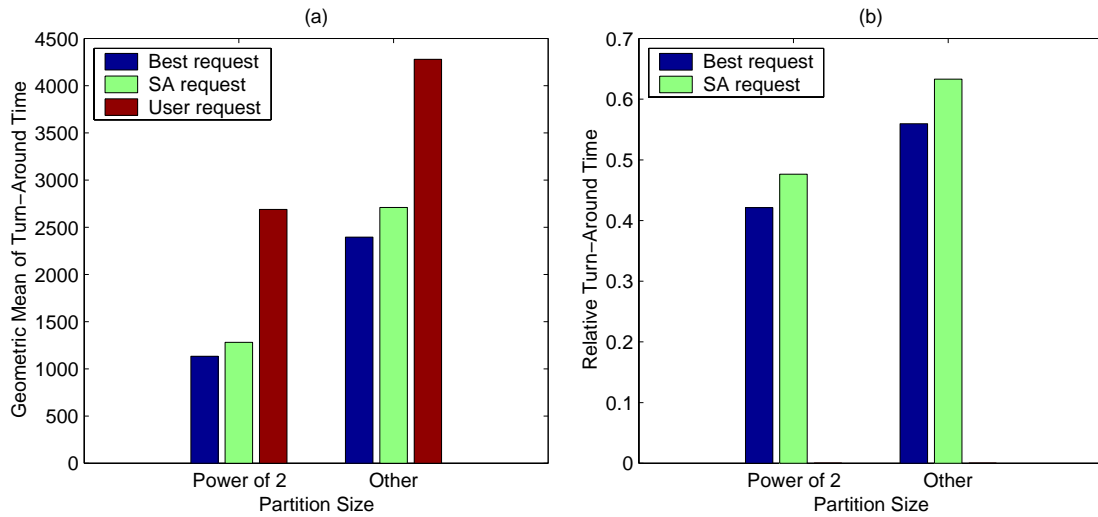


Figure 49 – Turn-around time by kind of partition size c_{kind}

Since the majority of the jobs in the workloads used on the experiments are power-of-2, we believe that it is easier for the resource scheduler to “pack” another power-of-2 job into the schedule than to find an appropriate hole for a job with arbitrary partition size. This phenomenon is in consonance with results showing that workloads with high percentage of power-of-2 jobs exhibit better performance under a variety of supercomputer schedulers [67].

5.3.2. Information Available to SA

The information made available to **SA** varies regarding v , the number of requests that is available to **SA**, and a , the accuracy of such requests. Recall that the accuracy a is defined as $a = te / tr$, and therefore a small value of a implies that the request asked for much more time than the job actually used.

Accuracy a

As can be seen in Figure 50a, the turn-around time tt grows with the accuracy a . However, we believe this is an artifact of the coupling between accuracy and execution time as represented in our model (see Section 3.3.4). The jobs with smaller accuracy tend to run for less time, thus reducing their turn-around time.

The relative turn-around time (see Figure 50b) provides a better evaluation of the impact of accuracy over the adaptive requests. It is interesting to see that the best request delivers a greater performance improvement for low accuracy jobs. **SA**, on the other hand, seems to be almost unaffected by the accuracy of the requests (in consonance with other studies that have found inaccurate user's estimates not to significantly hurt performance [43] [107]).

Another way to phrase this phenomenon is to say that the gap in performance improvement between the best request and **SA** is greater for small values of a (say $a < 0.1$). The best request is identified through the simulation of all possible requests: After all requests are simulated, the one with smallest turn-around time is named the best request. Since the best request is identified *a posteriori*, it is immune to the accuracy of the request. However, **SA** is not omniscient. Poor accuracy makes **SA** request much more cycles than the job is actually going to use. This precludes **SA** from using some holes in the schedule that would actually be enough to fit job j .

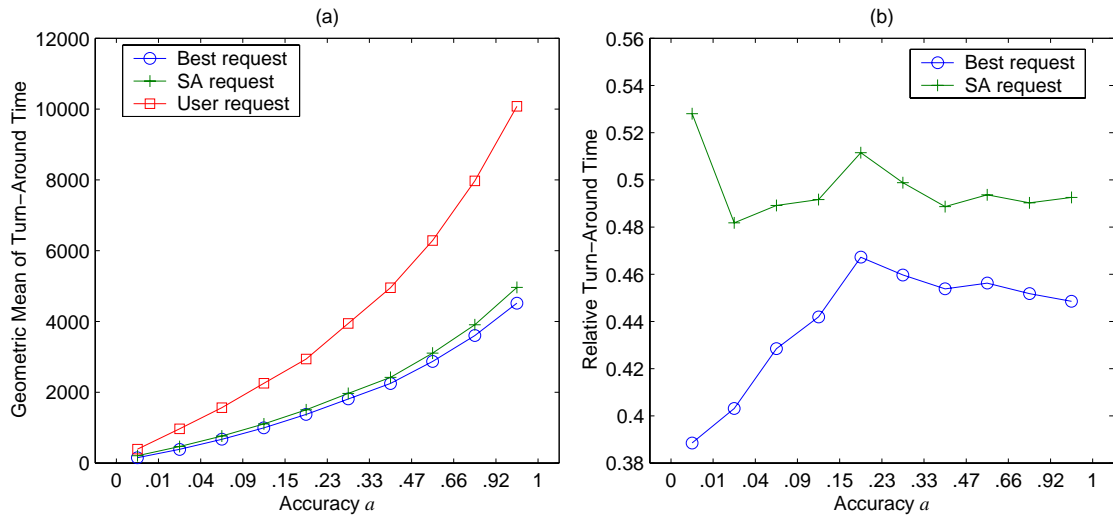


Figure 50 – Turn-around time by accuracy a

Number of requests ν

As shown in Figure 51, the performance improvement achieved by the adaptive requests increases with the number of requests ν . This result seems intuitive. The more requests that are available, the greater the flexibility the adaptive requests have in leveraging the holes in the supercomputer schedule.

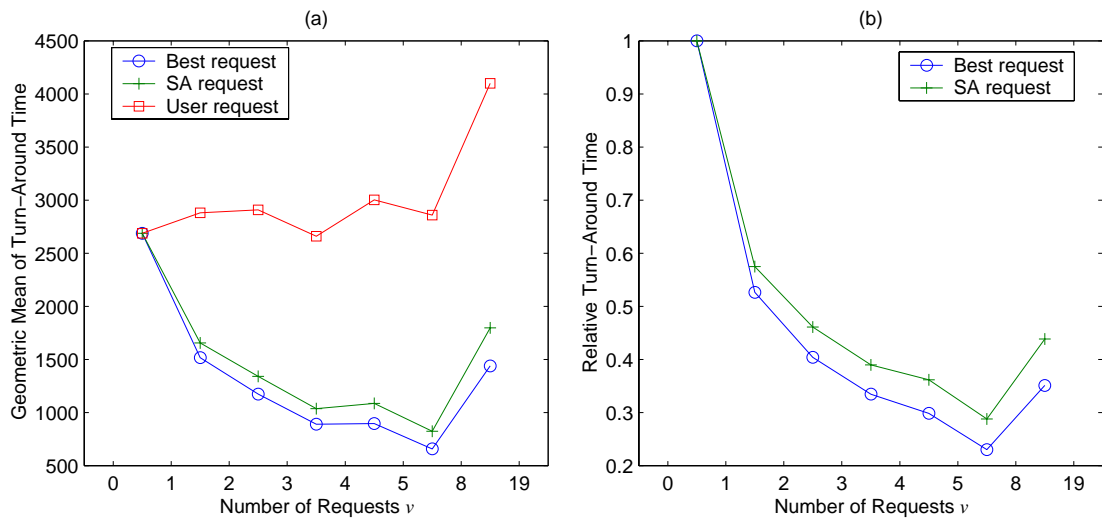


Figure 51 – Turn-around time by number of requests ν

Note that the increase in the turn-around time for $\nu > 8$ is due to the fact that there are no power-of-2 jobs with $\nu > 9$. Recall that the simulated supercomputer has 500 processors and thus power-of-2 jobs cannot have more than 9 requests (1, 2, 4, 8, 16, 32, 64, 128, and 256). In particular, the power-of-2 jobs that would otherwise have more than 9 requests must remain with 9 requests. Therefore, the data point for $\nu > 8$ in Figure 51 contains mainly non-power-of-2 jobs. Since jobs with non-power-of-2 partition sizes exhibit poorer performance (as discussed in the previous section), there is an increase in the turn-around time for $\nu > 8$.

Note also that Figure 51 only has 7 data points (instead of 10, as most of the previous figures). This is because $\nu = 2$ for 37.1% of the experiments (see Section 3.4.1 for the distribution of ν). The second data point therefore represents almost four deciles of the experiments.

5.3.3. The State of the Supercomputer

SA is an application scheduler and, as such, makes decisions based on the state of the supercomputer. In this section, we investigate how the load of the supercomputer at the moment job j arrives in the system influences the performance improvement achieved by SA and the best request. We expect that the more work the system already has, the greater the queue wait time for an incoming job typically will be. Of course, a large queue wait time contributes to a large turn-around time.

We use the *load per processor* D to gauge the load of the supercomputer at the moment SA schedules a job. This measure weights the amount of computation the supercomputer has to perform to finish all jobs currently in the system against the supercomputer size. Since a larger supercomputer will be able to deal with more load and more jobs than a smaller one, consideration of the supercomputer size enables us to compare results across supercomputers of different sizes. More precisely, the *load per processor* D is defined as:

$$D = \frac{\sum_j n_j \cdot (tr_j - (i_{now} - i_j))}{P}$$

where:

n_j is the number of processors requested by job j

tr_j is the execution time requested by job j

i_{now} is the current time instant

i_j is the time instant j started running (if j hasn't started yet, then $i_j = i_{now}$)

P is the number of processors in the supercomputer

Figure 52 shows the effect of the load per processor D on the turn-around time of the **SA** request, the best request, and the user request. As expected, the turn-around time grows with the load per processor (see Figure 52a). This is because, as expected, the more load there is in the system, the longer an arriving job has to wait in the queue (as can be seen in Figure 53a).

The relative turn-around time provides a more useful measure because it factors out the impact of the load per processor D on the turn-around time tt . The relative turn-around graph (see Figure 52b) indicates that the performance improvement achieved by **SA** decreases with the load per processor D until around $D = 200000$, when it seems to level off. Note that **SA** performs better for lightly loaded systems in two distinct ways. First and more obviously, the improvement delivered by **SA** is greater for lightly loaded systems. Second and maybe more interestingly, the gap between **SA** and the best request is smaller for lightly loaded systems.

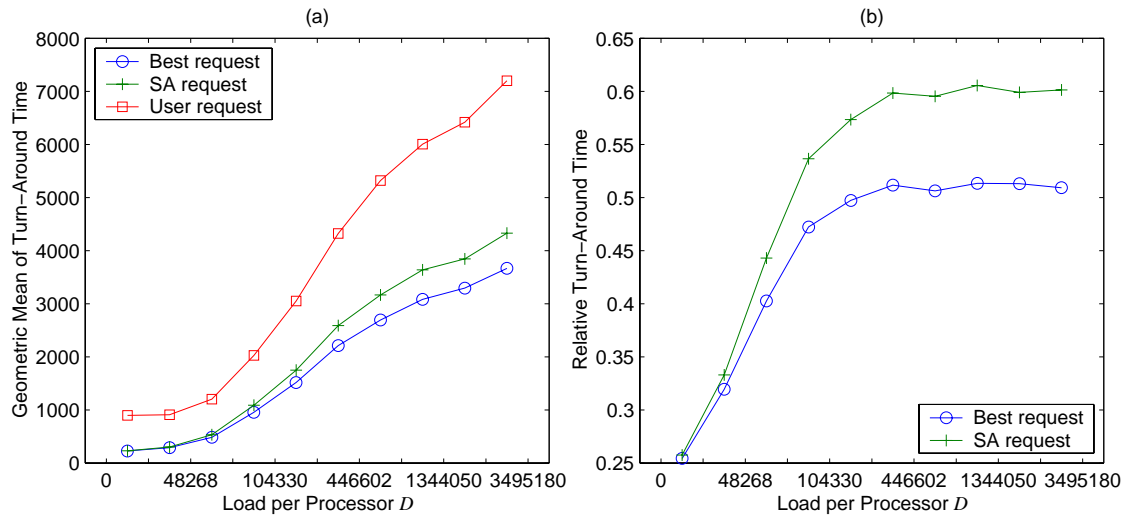


Figure 52 – Turn-around time by load per processor D

The fact that lightly loaded systems provide a more favorable environment for the adaptive requests can be better understood by decomposing the turn-around time into wait time (shown in Figure 53) and execution time (shown in Figure 54). Note that the “strategy” used by the adaptive requests varies depending on the load. For lightly loaded supercomputers, the adaptive requests seem to focus on reducing the execution time (see Figure 54) by selecting large requests (see Figure 55), even if such requests bear a slightly greater wait time than the user request (see Figure 53). This approach produces good results because, for lightly loaded supercomputer, the wait time is often very small anyways, thus having minimum impact on the turn-around time (see Figure 56 for the fraction of the turn-around time that is due to the wait time).

For heavily loaded supercomputers, on the other hand, the wait time corresponds to a sizable fraction of the turn-around time (see Figure 56). The adaptive requests then start to focus on reducing the wait time (see Figure 53), even when this requires selecting smaller requests (see Figure 55) that increase the execution time (see Figure 54). This seems to be a good approach to reduce the turn-around time in heavily loaded systems, although it does not achieve the same kind of performance improvement that is possible in lightly load supercomputers.

As for the increase in the gap between SA and the best request as the load grows, we believe it is due to the lower level of uncertainty SA faces on lightly loaded systems (compared to heavily loaded systems). Recall that SA uses the request time of the jobs already in the system as estimates for their execution time. Fewer jobs in the system thus reduce the overall error associated with these estimates.

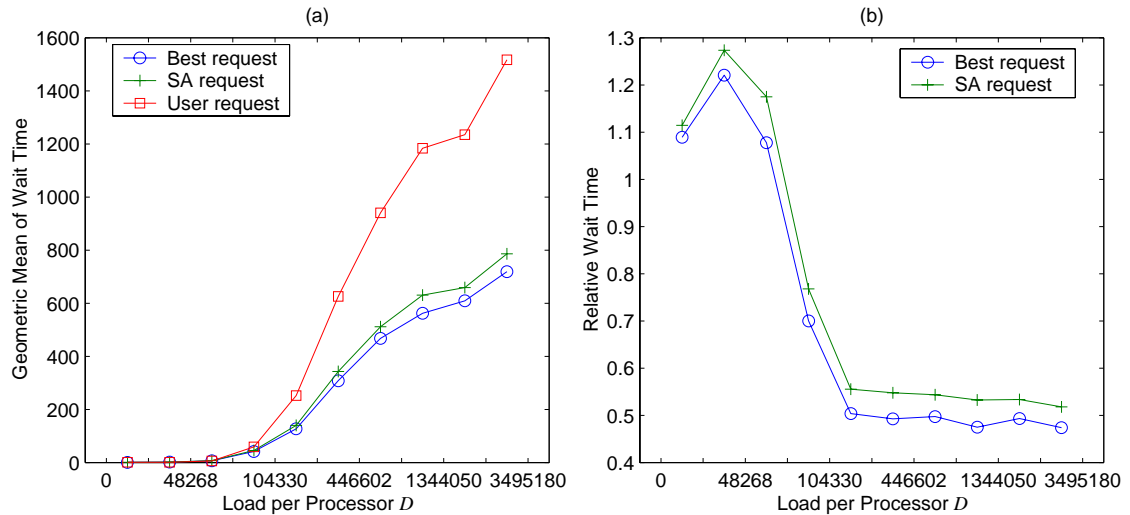


Figure 53 – Wait time by load per processor D

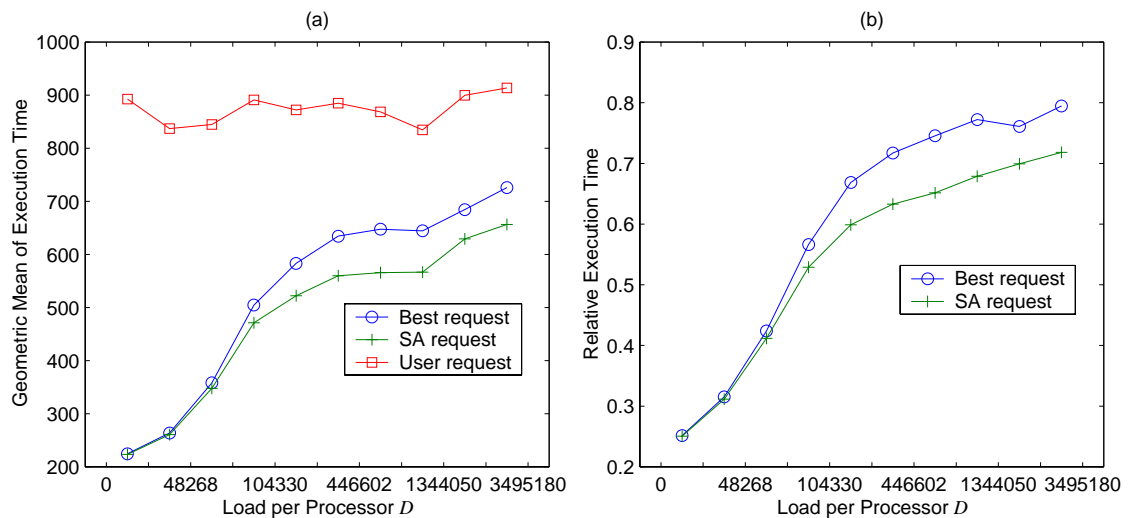


Figure 54 – Execution time by load per processor D

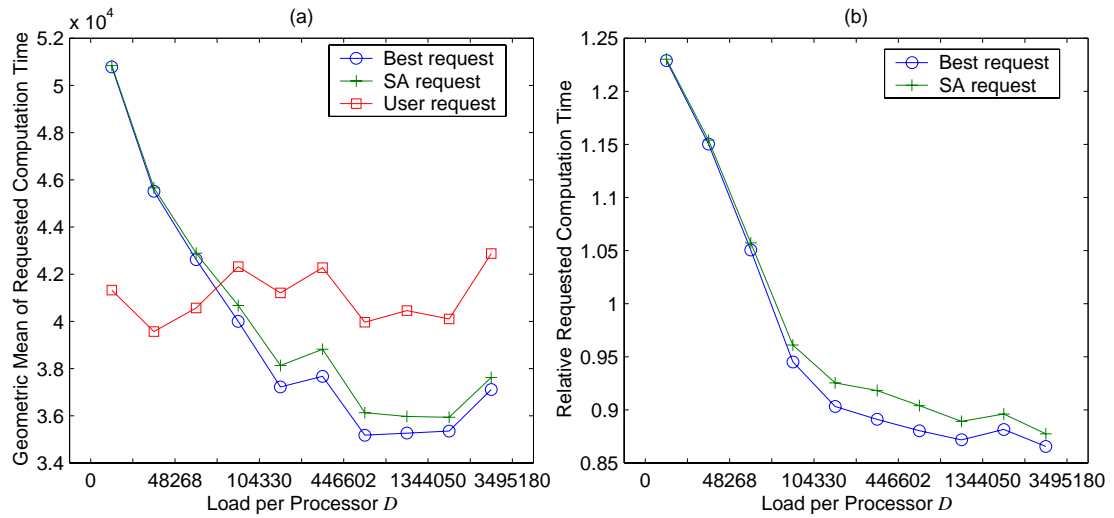


Figure 55 – Requested computation time by load per processor D

Note that Figure 55 measures the size of requests directly as requested computation time $cr = tr \cdot n$ (where tr is job's request time and n is its partition size) and also as *relative requested computation time*. The *relative requested computation time* of an adaptive request (the SA request or the best request) is ratio of the requested computation time of the adaptive request to the requested computation time of the corresponding user request.

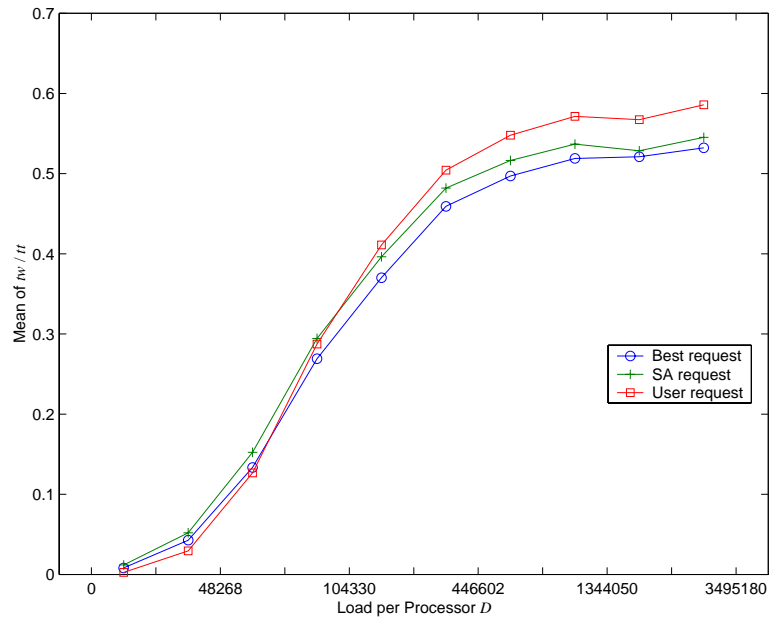


Figure 56 – Wait time over turn-around time as the load per processor grows

5.4. Validating the Results

Simulations are an important research tool [105]. They allow us to explore issues that are not tractable analytically or experimentally. However, they can produce invalid results due to a number of reasons, from poor modeling of reality to undetected bugs in the simulator. Consequently, it is important to double-check the results obtained via simulations.

In this section, we show that the models introduced in Chapter 3 indeed capture real supercomputer scenarios with reasonable accuracy. We do so by running experiments with real workloads and real jobs' speed-ups. Instead of using synthetic workloads as we have done in the rest of this chapter, we use our four reference workloads directly (see Section 3.1 for a description of the reference workloads). Instead of using the moldability model of Section 3.4, we use NAS benchmarks as the jobs to be scheduled by SA (see <http://www.nas.nasa.gov/Software/NPB/> for the speed-up behavior of NAS benchmarks).

We use five NAS benchmarks whose execution times are available for the SP2 on a variety of partition sizes: MG, LU, SP, BT, and EP. MG and LU require a power-of-two partition size and thus are the most constrained jobs. For the SP2, <http://www.nas.nasa.gov/Software/NPB/> contains execution time information for MG and LU over 8, 16, 32, 64, 128, and 256 processors. Consequently, SA has 4 to 6 choices of request for MG and LU, depending on the number of processors of the supercomputer being used (see Table 1). SP and BT require perfect-square partition sizes. There is data on their execution time for 9, 16, 25, 36, 64, 121, and 256 processors; thus providing 5 to 7 requests to SA. There are no restrictions for EP. It can run over any number of processors and thus there are as many requests as processors in the supercomputer (see Table 1). Table 15 summarizes the characteristics of the NAS benchmarks used in our validation experiments.

Benchmark	Partition size constraint	Number of requests
MG	power of 2	4, 5, or 6
LU	power of 2	4, 5, or 6
SP	perfect square	5, 6, or 7
BT	perfect square	5, 6, or 7
EP	unrestricted	100, 120, 128, or 430

Table 15 – NAS benchmarks used in the validation experiments

For each experiment, we randomly select a target job j whose partition size is compatible with the NAS benchmark b we want to introduce into the workload. For example, we look for perfect-square jobs when we introduce the BT benchmark. The target job j is then replaced by the NAS benchmark b . As before, each experiments consists of $\nu + 1$ simulations, where ν is the number of requests that can be used to submit the NAS benchmark b to the supercomputer. One simulation uses SA to select which request submits b . Moreover, there is one simulation for each of the ν requests that can be used for b . We performed 40000 experiments in total: 8000 per NAS benchmark.

Table 16 shows the overall results of the experiments based on NAS benchmarks. NAS benchmarks obtained much smaller turn-around times than synthetic jobs in the experiments based on our workload models (see Table 13). This is an expected outcome because NAS benchmarks are relatively small jobs (the largest sequential execution time L among all NAS benchmarks is 21190 seconds). As shown in Figure 39, the turn-around time for small jobs is much smaller than the overall results of our previous experiments.

	Best	SA	User
Geometric Mean of the Turn-Around Time	429	543	1478

Table 16 – Overall NAS results (in seconds)

Relatively speaking, however, the results found with NAS benchmarks are similar to those found with our workload models (see Table 13). As before, **SA** is close to results obtained by the best request. Furthermore, the turn-around times obtained by **SA** correspond to 37% of those obtained by the user request, a result even better than the one achieved using our workload model.

We attribute this better performance to the fact that **EP** can use *any* partition size (and all these possibilities are provided to **SA**). This seems to create the opportunity for an even greater improvement in performance. Indeed, consider Figure 57, which groups the turn-around times by the restriction on partition size posed by the NAS benchmarks. Note that **SA** delivers impressive performance improvement for **EP**: It reduces the turn-around time for a little more than 15% of the user request. Note also that **SA** is still able remain close to the maximum improvement for **EP**, which suggests that increasing the number of choices does not make it harder for **SA** to find a near optimal request.

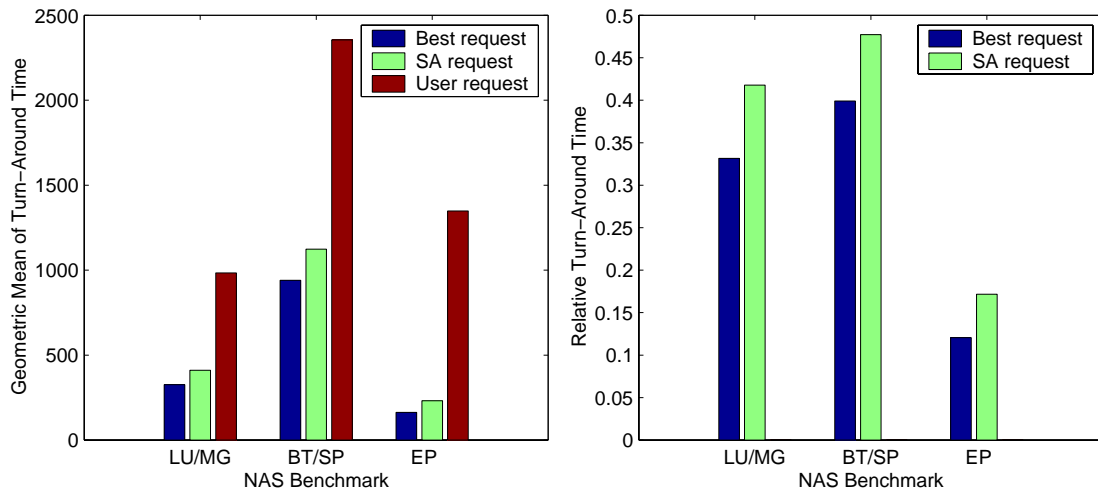


Figure 57 – NAS results by kind of benchmark

In short, the NAS benchmarks differ from our model in two important ways. First, NAS benchmarks are small jobs, whereas our model covers the wide distribution of job sizes found in practice: from small to very large. Second, the EP benchmark offers many more requests to **SA** than any job in our model. Although it is conceivable that embarrassingly parallel jobs will provide **SA** with a multitude of requests in practice, we took a more conservative approach and modeled the number of requests available to **SA** after the current practice, as discussed in Section 3.4.1. Taking into account the differences between NAS benchmarks and moldable jobs generated with our model, the results found with both of them are very similar. We therefore believe that the results based on NAS benchmarks validate the set-up used in our simulations (described in Section 5.1) and the conclusions based on such results (presented in Sections 5.2 and 5.3).

This chapter has investigated the performance of **SA** scheduling one job in current workload conditions. However, such conditions may change as a result of having **SA** scheduling many jobs in the workload, as we shall see in the next chapter.

6. Emergent Behavior of Multiple SAs

The previous chapter examined how **SA** performs in current real-life conditions. However, the widespread use of **SA** may change such current conditions. When many **SAs** are scheduling jobs on one supercomputer, the decision made by one **SA** affects the state of the system, therefore impacting other instances of **SA**. The global behavior of the system thus comes from the *emergent behavior* of all **SAs**.

This is a very important issue because there is theoretical evidence that systems in which resource allocation is performed by many independent entities can exhibit performance degradation [71] and even chaotic behavior [59]. There are two basic concerns about a system in which many entities make decisions independently. First: Is the system as a whole stable, or does it oscillate in some thrashing cycle? Second: What is the impact of multiple **SAs** on the performance attained by each of them?

In our environment, the stability of the system is *not* a problem. Stability is a problem for systems formed by multiple independent entities when such entities can keep prompting each other to make decisions in an endless feedback cycle [59]. When this happens, the system as whole never stabilizes. If the entities in an unstable system are making scheduling decisions, the overhead of carrying on the constant flow of decisions is likely to preclude the system from performing much useful work. We thus say that such a system is in a thrashing cycle. Since **SA** makes only one decision, there is no chance for a feedback behavior to occur. A supercomputer on which multiple instances of **SA** schedule jobs is always stable.

On the other hand, having multiple **SAs** in the system can have a performance impact on the system as whole and on each instance of **SA** in particular. A way to think about **SA** is that it leverages inefficiencies of the supercomputer schedule. By considering multiple requests, **SA** searches for a “good” hole in the supercomputer schedule, picking the hole on which the job being scheduled is expected to finish earlier. We therefore expect the performance improvement obtained by an instance of **SA** to be

smaller when most jobs use **SA**. Section 6.1 investigates this hypothesis. In this chapter, we also investigate how the increase in the total load submitted to the supercomputer affects **SA** (see Section 6.2).

6.1. Performance Impact of Emergent Behavior

SA selects the request that is expect to deliver the smallest turn-around time by searching the supercomputer schedule for holes that fit the possible requests (see Chapter 2). Having many **SA**s searching for holes in the supercomputer schedule is likely to make the supercomputer schedule more compact, with less “big” holes. While a more compact schedule makes the system as a whole more efficient, it also makes harder for each instance of **SA** to find a hole that delivers a large performance improvement. In short, we expect the competition for resources to become tougher with multiple **SA**s, and this tough competition to reflect on the performance improvement attained by each **SA** individually.

Experimental Set-up

We conducted a number of experiments to investigate the emergent behavior of **SA**. The experimental set-up used here is similar to the one used in the previous chapter (see Section 5.1). As before, we conducted 360000 experiments. A 10000-job synthetic workload is generated for each experiment, which targets a randomly chosen job j . The simulated supercomputer has 500 processors, and the supercomputer scheduler is conservative backfilling. This time, however, the moldability model is applied to all jobs. This enables us to use **SA** with multiple jobs, therefore creating the emergent behavior we want to examine.

Each experiment consists of four simulations. All four simulations use the same workload. They vary in whether (i) the target job j uses **SA** or the user request, and (ii) all the other jobs use **SA** or the user request. When most jobs in the workload use **SA**, we say that the workload is *adaptive*. When most jobs are submitted through the user request, we say that we have a *static* workload. Therefore, when all jobs use **SA**, the turn-around time of job j represents the performance **SA** achieves within adaptive work-

loads. When job j is scheduled by SA but all the other jobs use the user request, the simulation determines the performance of SA in static workloads. When job j is submitted through the user request and all other jobs use SA, we have the performance the user request achieves in adaptive workloads. When all jobs are submitted through the user request, job j represents the performance of the user request in static workloads (the current supercomputer usage scenario). Table 17 summarizes the four scenarios simulated in the experiments.

Performance of	Target job j uses	Other jobs use
SA in adaptive workloads	SA request	SA request
SA in static workloads	SA request	User request
User request in adaptive workloads	User request	SA request
User request in static workloads	User request	User request

Table 17 – Scenarios simulated in the emergent behavior experiments

Note that the notion of *best request* is not well-defined for an environment with multiple instances of SA. Since the turn-around time of a job j can be affected by a job g that arrives after j , we cannot determine the best request for job j without knowing which request job g is going to use. And, of course, we cannot determine the best request for g without knowing which request j is going to use (because the request of j contributes to determine the state of system encountered by g).

Overall Results

Table 18 displays the results obtained from the experiments just described, i.e., Table 18 shows how SA and the user request behave within static and adaptive workloads. Recall that the turn-around times obtained in the 360000 experiments are summarized by the geometric mean statistic (as explained in Chapter 4).

Request used for job j	SA		User	
Workload	Static	Adaptive	Static	Adaptive

Geometric mean of the turn-around time	1429	1357	2878	2721
---	------	------	------	------

Table 18 – Overall results with SA scheduling all jobs (in seconds)

To our surprise, both **SA** and the user request have slightly smaller turn-around times when in adaptive workloads than when in static workloads. As we shall see in more detail, it turns out that the emergent behavior of many instances of **SA** reduces (i) the occurrence of very high load conditions, and (ii) the wait time of jobs submitted to systems experiencing moderate to high load. In moderate to high load situations, these reductions in load and wait time seem to overcome any performance degradation potentially caused by the increased competition for resources from other instances of **SA**. In low load scenarios, however, both **SA** and the user request achieve smaller turn-around times when in a static workload, as we initially expected.

Reduction of Very High Load Conditions

Recall from Section 5.3.3 that, when within static workloads, **SA** uses small requests under heavy load conditions. This approach reduces the queue wait time, although it in general increases the execution time. Since heavily loaded systems generate large wait times, such an approach is effective in reducing the job's turn-around time, the final goal of **SA**. As shown in Figure 58, the tendency of **SA** to reduce the size of the request as the load grows remains the same when it is in adaptive workloads. This is the expected behavior because **SA** only considers the job it schedules, without worrying about the performance of other jobs.

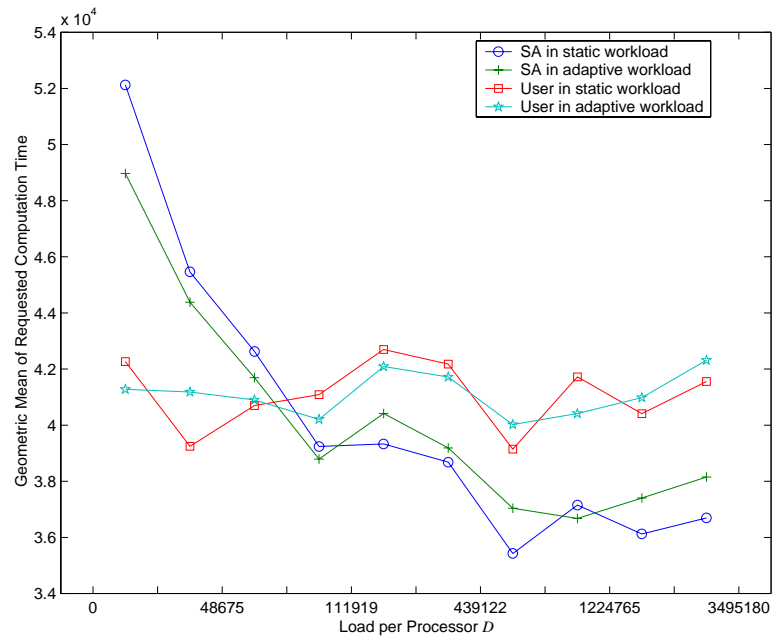


Figure 58 – Requested computation time by load per processor D

Since **SA** tends to favor small requests as the load grows, the emergent behavior generated by having many **SAs** in the system reduces the occurrence of very high load conditions. Consider Figure 59 for the distribution of the load per processor D at the moment the target job j arrives in the system. For approximately 60% of the experiments, job j does not experience much difference in the system load whether the previous jobs used **SA** (adaptive workload) or the users requests (static workload). These are the 60% of the experiments on which job j found the supercomputer to be less loaded ($D < 500000$). For the other 40% of the experiments, job j found smaller load in the supercomputer when the previous jobs used **SA** (i.e., the adaptive workload scenario). These 40% of the experiments are the ones on which the supercomputer was more loaded at the moment of the arrival of job j ($D > 500000$).

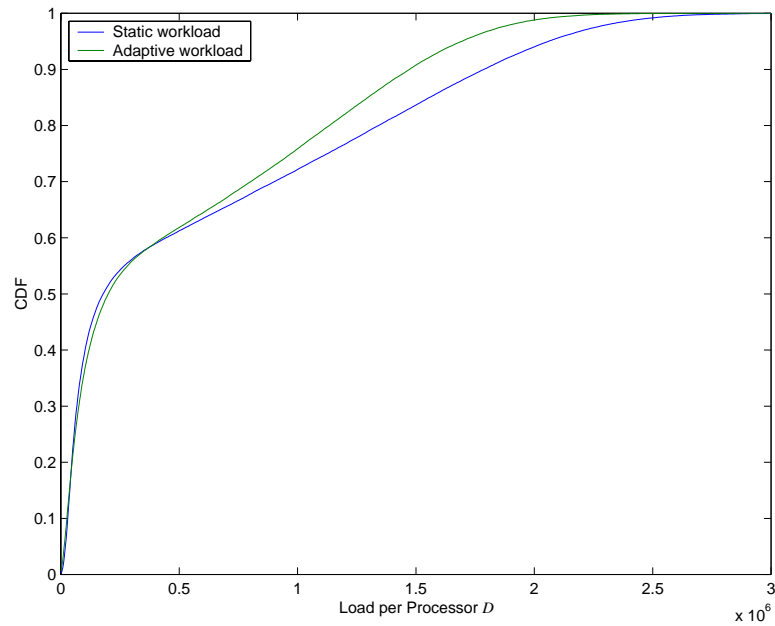


Figure 59 - Distribution of the load per processor D

Reduction of Wait Time in Moderate to High Load Conditions

Besides reducing the chance that the system experiences heavy loads, the emergent behavior of **SA** also appears to reduce the job's wait time in moderate to heavy load conditions. See Figure 60 for the influence of the load per processor D on the turn-around time of **SA** and the user request within both adaptive and static workloads. For $D > 110000$, both **SA** and the user request experience smaller turn-around times when they are in adaptive workloads.

It is interesting to note that the execution times of jobs using the user request do not seem to be affected by whether the workload is adaptive or static (see Figure 61). On the other hand, for jobs using **SA**, the execution time follows a pattern similar to the one experienced by the turn-around time: In lightly loaded conditions, jobs using **SA** have better execution time in static workloads. From moderate to high load, the execution time of jobs using **SA** is better in adaptive workloads.

The wait time (shown in Figure 62) shows the same trend for jobs using the user request and **SA**. The wait time is small under light loads. For moderate to high loads, jobs using both **SA** and the user request have smaller wait times in adaptive workloads.

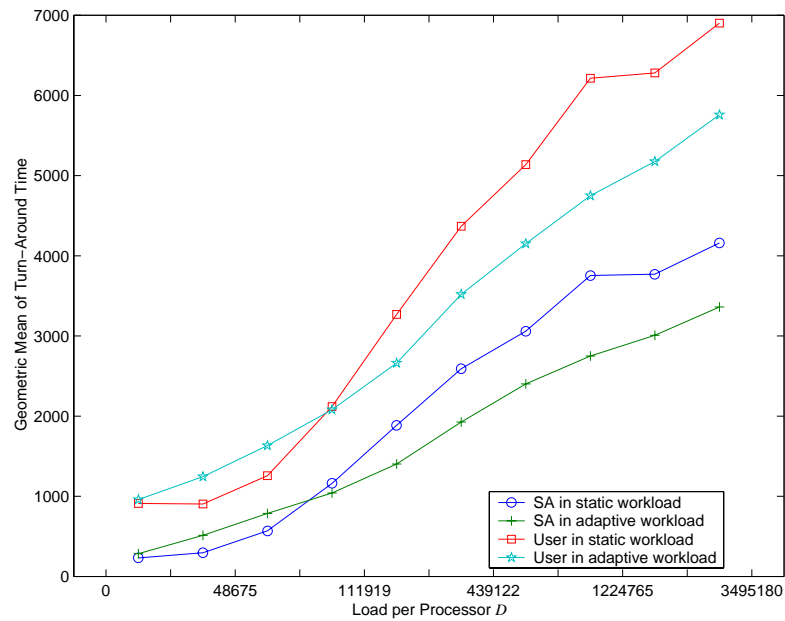


Figure 60 – Turn-around time by load per processor D

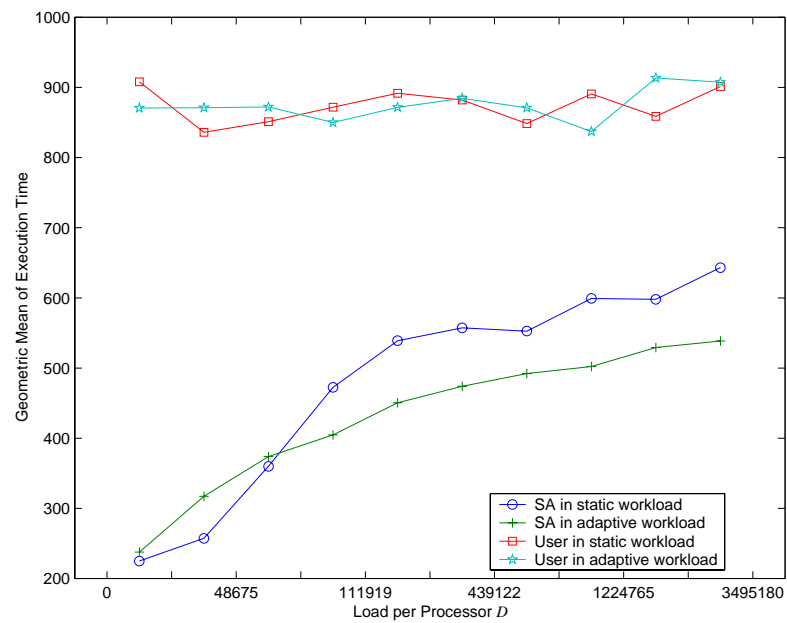


Figure 61 – Execution time by load per processor D

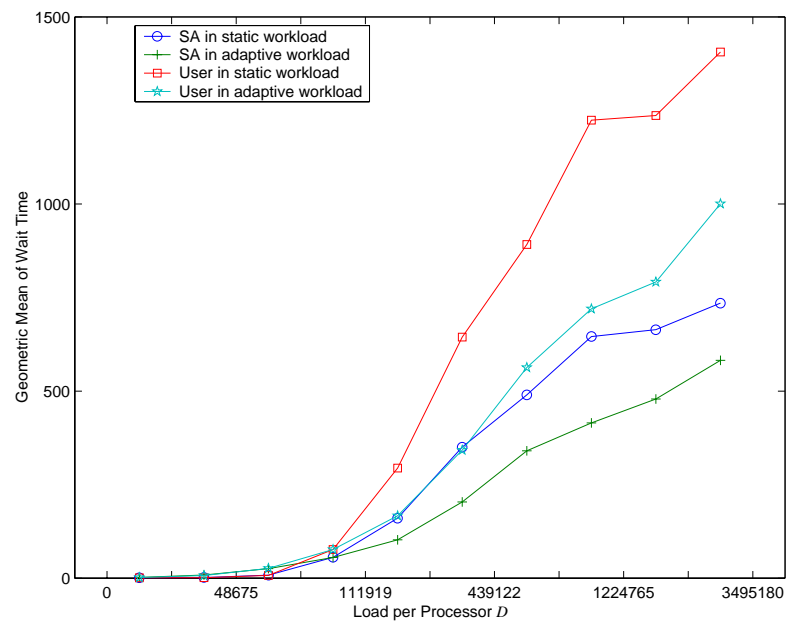


Figure 62 – Wait time by load per processor D

We conjecture that the smaller wait time jobs experience when $D > 110000$ and the workload is adaptive happens because the emergent behavior of **SA** causes better packing in the supercomputer schedule. **SA** exploits existing holes in the supercomputer schedule to improve the job's performance. The overall effect of having all jobs using **SA** therefore appears to be that the supercomputer schedule becomes more compact, with fewer big holes. The fact that **SA** prefers small requests as the load grows provides extra support for this conjecture because smaller requests are easier to pack.

When the user request is used in adaptive workloads, a better packing in the supercomputer schedule translates in general into a smaller wait time. Since the user request is static, the wait time experienced by the job is determined mainly by the state of the supercomputer schedule. Therefore, everything else being the same, a job using the user request will in general experience a smaller wait time when encountering a better packed supercomputer schedule.

When **SA** is used in adaptive workloads, it is less clear that a better packing is responsible for a smaller wait time. After all, **SA** leverages holes in the supercomputer schedule to improve better performance. However, recall from Section 5.3.3 that **SA** achieves smaller improvement in the turn-around time under high-load conditions. The fact that the system as a whole is more efficient (i.e., the supercomputer schedule is more compact) seems to be more beneficial than the small improvements **SA** would achieve on average with a less compact schedule. Moreover, with a more compact schedule, **SA** does not have to sacrifice the execution time so severely in order to reduce the wait time, as it does in a static workload (see Figure 61).

Increased Competition in Lightly Loaded Systems

As can be seen in Figure 60, **SA** in static workloads performs better than **SA** in adaptive workloads when the supercomputer is lightly loaded. This suggests that adaptive workload indeed increases the competition for resources, making it harder for **SA** to find a good request to use, as we initially expected.

Note that this explanation is consistent with the fact that, under moderate to high loads (in our experiments, for $D > 110000$), the performance of **SA** in adaptive work-

loads exceeds the performance of **SA** in static workloads. We credit the better performance **SA** achieves within adaptive workloads under moderate to high load to emergent behaviors that reduce the occurrence of high loads conditions and the wait time of the job. Since such emergent behaviors appear to come from the tendency of individual **SA** to select small requests as the load grows, they are negligible for light load conditions.

Other Factors

Whether the workload **SA** is static or adaptive seem to have little impact on the behavior of **SA** as a function of the parameters that describe the job's characteristics (sequential execution time L , average parallelism A , σ , minimum partition size c_{min} , maximum partition size c_{max} , and kind of partition size c_{kind}) and the information available to **SA** (accuracy a , and number of requests v). These results can be found in Appendix C.

This comes as no surprise. Emergent behaviors cannot affect the characteristics of the job being scheduled by **SA** or the information offered to **SA** about such a job. Multiple instances of **SA** can only affect the state of the supercomputer, and they do so, generating the emergent behaviors we have described herein.

6.2. Increasing the Offered Load

Up to this point, we have used workloads that reflect either current real-life conditions or conditions we expect to see as a result of the widespread utilization of **SA**. However, it is also of interest to understand how **SA** behaves under other conditions. In particular, how schedulers behave as the *offered load* increases is of great theoretical and practical interest. The *offered load* is the aggregated computation time needed to process an input workload. More precisely, the *load offered by a workload W* (or simply *offered load*) is defined as $\sum_{j \in W} ce_j$, where j is a job in the workload W , $ce_j = n_j \cdot te_j$ is the computation time used by job j , n_j is the number of processors allocated to job j , and te_j is the execution time of job j .

Note that the concepts of *offered load* and *the load when SA schedules a job* are distinct. The load of the system when SA makes a scheduling decision is a snapshot of the system condition. Such a concept has been measured in this thesis through the load per processor D . On the other hand, the offered load corresponds to the total amount of computation carried by the workload as a whole. That is, the offered load is a measure for the workload as a whole, whereas the system load when SA makes a decision varies from job to job in the same workload. Of course, as the offered load increases, SA will more likely schedule a job in heavy load conditions.

The load offered to the supercomputer can increase for several reasons. More jobs and/or larger jobs increase the load offered to a system (assuming everything else is kept constant). We investigate this *direct* increase on the offered load via the *load multiplier* κ_L . Recall from Section 3.5 that the load multiplier κ_L multiplies both the arrival rate and the request time generated by our workload model.

There is also the possibility that the offered load changes due to the decisions made by SA. This can happen because SA selects which request to use out of a set of requests that typically vary regarding the computation time ce needed to complete the job. For example, SA tends to use small requests in heavy load conditions, as seen in the previous section. We examine here the impact of the variation on the computation time ce needed by the possible requests of a job. More precisely, we increase the distribution of σ . A workload formed by jobs with large σ can generate a greater load to the system if large requests are often selected by SA.

6.2.1. Direct increase of the offered load

The load multiplier κ_L changes the total workload load by multiplying both the arrival rate and the request time generated by our workload model (see Section 3.5). For example, $\kappa_L = 2$ implies that the supercomputer receives a load 4 times greater (twice the arrival rate and twice the request time) than loads currently found in practice. Of course, $\kappa_L = 1$ represents the load conditions currently found in practice.

By varying κ_L , we can investigate how **SA** behaves as the offered load grows. Figure 63 shows the overall performance of **SA** scheduling all jobs in the workload for $\kappa_L = 0.5, 1.0, 1.5, \dots, 4.0$. Each data point in the graph aggregates the results of 400000 jobs. More precisely, 40 10000-job experiments were run for each data point. The results of all jobs are considered (i.e., these experiments do not focus on a single target job). Each 10000-job workload was simulated twice, one time using **SA** and the other time using the user request, in both cases for all jobs in the workload.

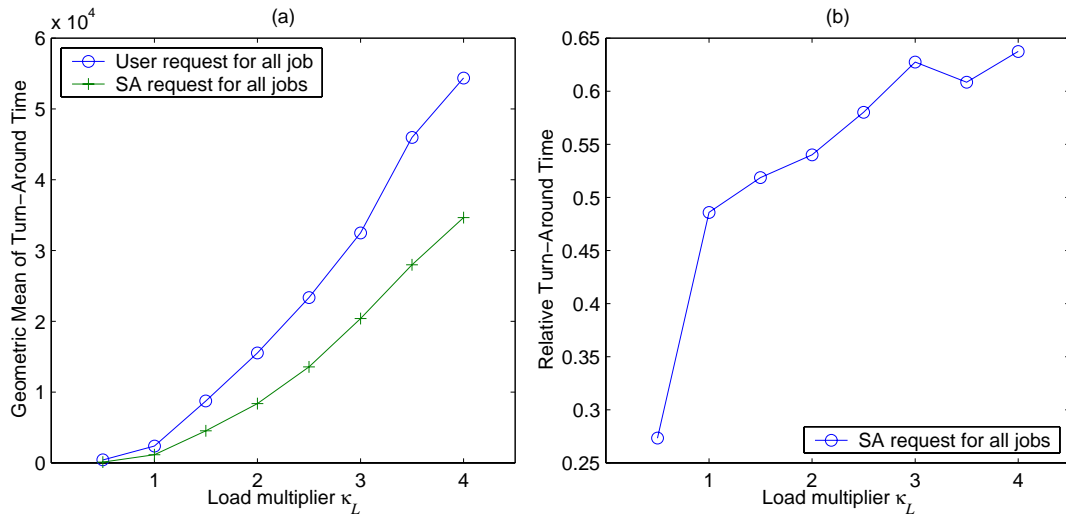


Figure 63 – The effect of offered load

As expected, the performance of the system as a whole degrades with the increase in the load (see Figure 63a). More interesting here is how the load impacts on **SA**, an issue better tracked by the relative turn-around time (Figure 63b). Notice that **SA** performs extremely well (relative turn-around time = 0.27) when very little load is offered to the supercomputer ($\kappa_L = 0.5$). This is in consonance with the finding the **SA** works better in lightly loaded conditions (see Section 5.3.3). Possibly more important is the observation that the performance improvement achieved by **SA** degrades slowly as the load increases. Even for extremely high loads ($\kappa_L = 4.0$), **SA** still achieves considerably smaller turn-around time than the user request.

6.2.2. Large- σ workloads

The value of σ (which determine how close to linear the job speed-up is) seems to have little effect on the performance of **SA** (see Section 5.3.1). However, the distribution of σ may influence the emergent behavior of a system with multiple instances of **SA**. If large values of σ are common, it might happen that **SA** increases the load offered to the supercomputer, therefore reducing the overall performance of the system. This is because the requests **SA** choose from normally vary with respect to the computation time ce needed to complete the job. Large values of σ accentuate such a variance, creating the potential for multiple **SAs** to select requests that demand large amounts of computation to complete the jobs, increasing therefore the load submitted to the supercomputer. The focus of this section is to establish whether such possible emergent behavior indeed appears, and, if so, to determine the extent of its impact on the performance achieved by **SA**.

In order to investigate our hypothesis, we artificially change the distribution of σ to make the occurrence of large values more common. Recall that σ is modeled via a normal distribution (see Section 3.4.3). We multiply the mean of such a distribution μ_σ by a constant κ_σ . This allows us to raise the values of σ without changing the shape of the distribution. Figure 64 shows distributions of σ for κ_σ varying from 1 to 4. Note that $\kappa_\sigma = 1$ is the original distribution, which we believe represents the values found in current workloads (see Section 3.4.3).

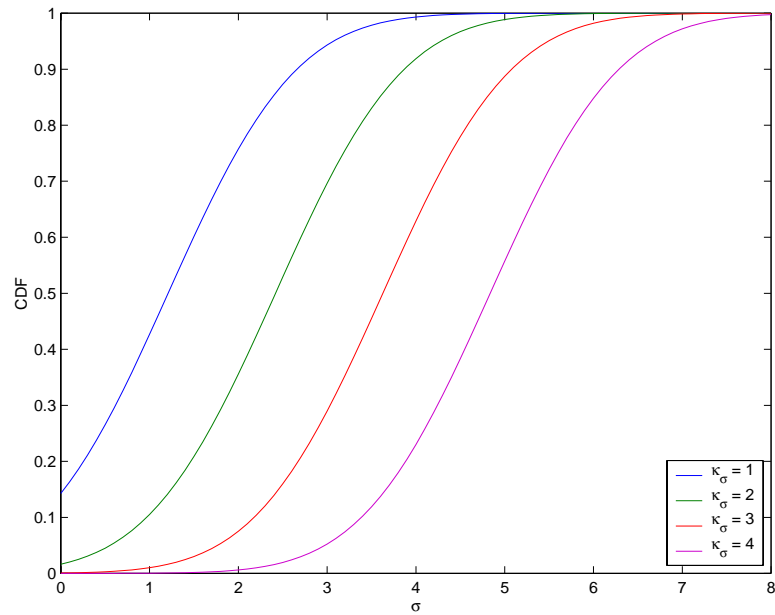


Figure 64 – Distributions of σ

We conducted experiments that are almost identical to the ones discussed in the pervious section (Section 6.2.1). The difference is that here we varied κ_σ from 0.5 to 4.0 in 0.5 steps (instead of varying κ_L , as before). As conjectured, the offered load increases as the distribution of σ grows when **SA** is used (see Figure 65). As a result, the turn-around time of **SA** grows as the distribution of σ increases (see Figure 66a). Since the user request is not affected by σ in our model (see Figure 66a), the performance improvement delivered by **SA** decreases as the distribution of σ grows (see Figure 66b). However, **SA** is still able to improve over the user request even for workloads with very high values of σ ($\kappa_\sigma = 4$).

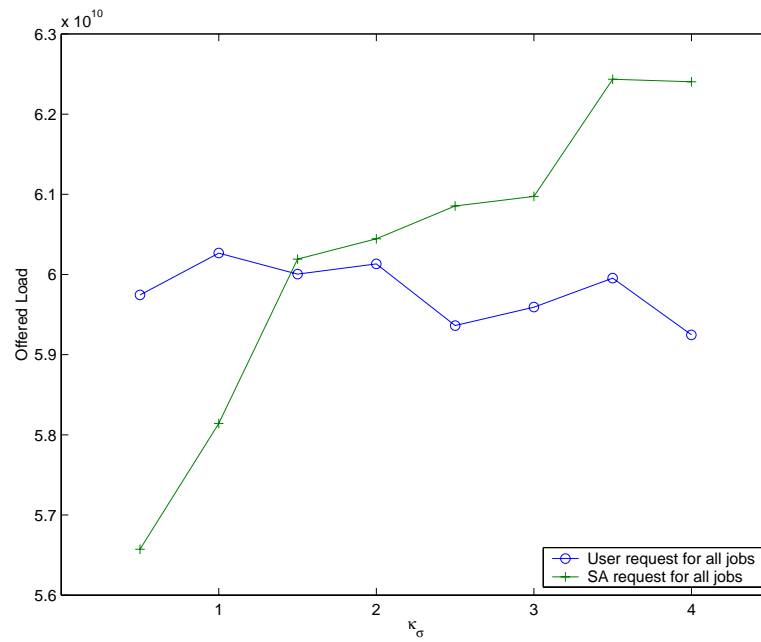


Figure 65 – Offered load as a function of κ_σ

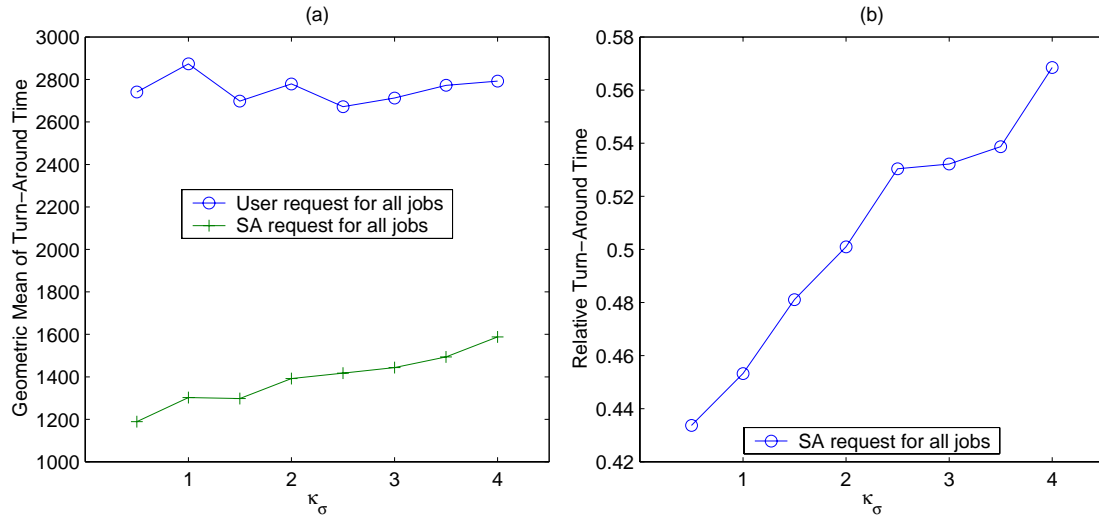


Figure 66 – The influence of the distribution of σ on the emergent behavior of SA

7. Related and Future Work

No research happens in a vacuum. Research simultaneously draws from previous efforts and opens new possibilities to be explored. This chapter covers these connections between our own research and other efforts. We start by describing related work available in the literature. We then move on to sketch research ideas that are the natural “next steps” to the work described herein.

7.1. Related Work

This section surveys the state-of-art in the three areas that are most relevant for our research: supercomputer scheduling, application scheduling, and the emergent behavior of systems on which resource allocation is performed by multiple independent entities.

7.1.1. Supercomputer Scheduling

Scheduling distributed-memory parallel supercomputers is an instance of the more general problem of scheduling multiprocessor computers. The features that particularize scheduling parallel supercomputers are (i) the continuous arrival of jobs to the system, and (ii) the high cost of task migration.

Other Multiprocessor Scheduling Problems

A scheduler that deals with jobs that continually arrive to the system is called an *on-line scheduler* [44]. In contrast, an *off-line scheduler* assumes that all jobs are available when the scheduler starts. Off-line scheduling is more amenable to analytical solutions and there is a great deal of research done in this area [35]. However, the results of these investigations often cannot be applied to the on-line problem.

Likewise, much of the work in scheduling shared-memory multiprocessors does not apply directly to distributed-memory parallel computers. In principle, on-line

schedulers designed for shared-memory machines [5] [13] [15] [86] can execute on distributed-memory supercomputers. However, migrating a task is much cheaper in shared-memory multiprocessors than on distributed-memory machines. For shared-memory machines, the cost of restarting a task is roughly independent to the processor assigned to the task. In a distributed-memory supercomputer, on the other hand, restarting a task on different processor involves transferring (at least part of) the address space of the task, an expensive operation. Since most scheduling solutions for shared-memory do take advantage of the relative low cost of task migration to improve performance, these solutions incur prohibitively high overhead when used on distributed-memory machines.

Preemption Capabilities

Assumptions about the supercomputer vary mainly regarding preemption capabilities⁶. Supercomputers can support (i) no preemption capabilities (i.e., jobs run to completion or until a time limit), (ii) local preemption (i.e., preemption within the threads in a processor), and (iii) preemption over parallel jobs as a whole (a.k.a. *gang scheduling*). Most supercomputers currently run jobs until completion. This makes non-preemptive policies of great interest for researchers in supercomputing scheduling [1] [2] [24] [32] [43] [66] [67] [69] [82] [90]. Local preemption seems to be of little value for supercomputer scheduling due to their negative impact on the performance of tightly coupled parallel jobs [38]. Gang scheduling has been shown to improve performance under a variety of scenarios [42] [54] [79] [81] [106]. However, the interaction between gang scheduling and job's characteristics such as I/O patterns [65] and memory consumption [6] make gang scheduling somewhat complex in practice.

Characteristics of the Job

Assumptions about the job vary regarding job flexibility and the knowledge the supercomputer scheduler has about jobs⁶. Job flexibility describes how many processors a job will be able to use throughout its execution. A *rigid job* uses a fixed and pre-

⁶ This classification is based on the work of Feitelson et al [40].

established number of processors. A *moldable job* uses a fixed number of processors during its execution, but that number can be chosen from a set of possible partition sizes before the job starts. A *malleable job* can change its partition size during the execution. In current practice, supercomputer schedulers accept rigid jobs [58] [66] [69] [74] [82] and thus much of the research available in the literature assume jobs to be rigid [1] [2] [43] [55] [67] [78] [85] [90]. However, there is evidence that performance can be improved by allowing jobs to be moldable [24] [32], as in this thesis, or even malleable [76].

Despite such evidence, we do not know of any supercomputer in production that supports non-rigid jobs. We believe that this is due to the difficulty in proving that a given scheduling solution for moldable or malleable jobs will work in practice. Critics question the degree to which production jobs are moldable or malleable. The very lack of a production system that supports non-rigid jobs makes it very hard to answer this question. We have addressed this issue (regarding moldable jobs, at least) in this thesis. The results of the survey we conducted among supercomputer users (see Chapter 3 and Appendices A and B) lead us to believe that (i) most jobs current in production are moldable, and (ii) we can model the characteristics of their moldability, as discussed in Section 3.4.

The knowledge the supercomputer scheduler has about jobs' execution time can be (i) none, (ii) user-provided, (iii) statistical, and (iv) complete. Since Lifka et al showed that user-provided information can improve the supercomputer utilization and decrease the jobs' turn-around time by reducing fragmentation [66] [82], little effort has been dedicated to supercomputer schedulers that know nothing about the jobs. Schedulers based on user provided information now constitute standard practice [66] [69] [74] [82], and therefore are a natural target for many research endeavors [24] [43] [107]. In recent years, there has been considerable work on statistically deriving information from the past behavior of jobs [31] [84] and applying such information (sometimes combined with user provided information) to improve supercomputer scheduling [32] [55] [85]. Although complete knowledge is assumed by some researchers [1] [2] [67] [90], perfect knowledge cannot be achieved in practice for a general purpose system.

However, schedulers that assume perfect knowledge are often useful in providing an upper bound for scheduling performance (as “the best request” used in Chapter 5, for example).

Workload Models

Since the performance of a computer system depends on the workload to which such a system is submitted [18] [21] [44] [67], workload modeling plays a vital role in performance evaluation. A workload model enables the researcher to explore the performance of the system in a multitude of scenarios. As with any model, a key issue with supercomputer workload models is how well they represent reality. This motivates the derivation of models from workload logs.

Parallel supercomputers are relatively recent artifacts, and thus good workload logs are just appearing [39] [45] [56] [61]. The availability of such logs has prompted great activity in supercomputer workload modeling in recent years [21] [30] [33] [34] [40] [44] [60]. However, modeling workloads is challenging [34]. In particular, there are many important aspects of the supercomputer workloads that have not been modeled, or that have been modeled only incipiently [34].

This situation had a strong impact on our research because **SA** targets moldable jobs, and moldability is one of the aspects of the supercomputer workload that has not been sufficiently studied in the literature. Such a situation impelled us towards an extensive modeling effort (as described in Chapter 3).

Other Factors in Scheduling

Much of the research available in the literature focuses on scheduling processors. In practice, however, other resources (e.g., disk, memory, communication infrastructure) may also need to be considered. There is some research in multi-resource scheduling [6] [15] [70], but it still remains a largely unexplored issue.

7.1.2. Application Scheduling

Application schedulers are an essential component of scheduling solutions for computational grids. *Computational grids* are platforms for the execution of parallel

jobs that are composed by geographically dispersed resources that may be under the control of multiple entities [53]. Since grid resources are geographically dispersed and often in different administrative domains, it is not feasible for a single scheduler to oversee the entire system.

In a computational grid, groups of resources are independently controlled by different *resource schedulers*. *Resource schedulers* control the resources they schedule. One salient characteristic of resource schedulers is that they receive requests from multiple users, and thus must arbitrate among such users. Therefore, in order to use resources controlled by multiple resource schedulers, one has to (i) select the resources to use, (ii) partition the work across the selected resources, and (iii) submit requests to the appropriated resource schedulers to have the selected resources carry out the work assigned to them. This is the task of the *application scheduler*. Application schedulers do not control the resources they use. They obtain access to resources by submitting requests to the appropriate resource schedulers. Figure 67 illustrates the relationship between the different kinds of schedulers in a grid.

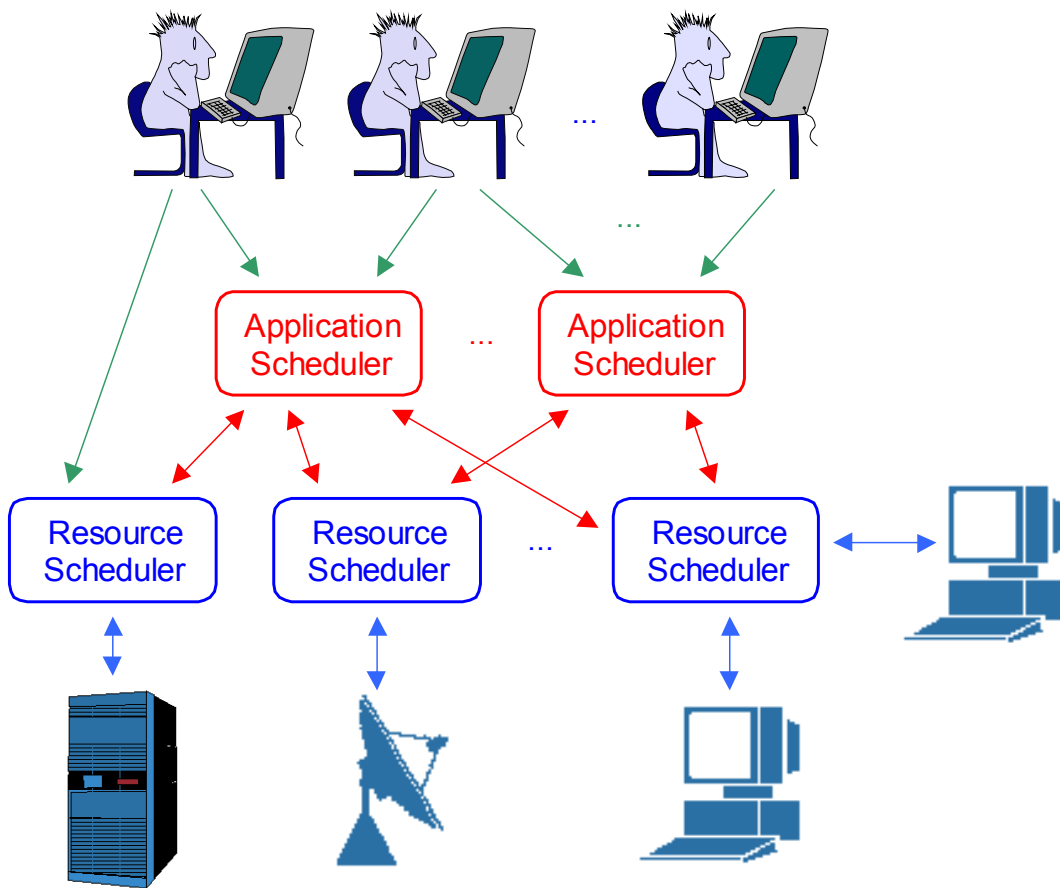


Figure 67 – Different kinds of schedulers found in a computational grid

AppLeS: An Example of an Application Scheduler

As an example of application schedulers, let us look at AppLeS (Application-Level Schedulers) [9]. AppLeS are application schedulers developed by Fran Berman's group at UCSD and Rich Wolski at the University of Tennessee [9] [10] [27] [83] [88] [89]. Figure 68 shows the general structure of an AppLeS. A typical AppLeS is part of the application it schedules. It starts by obtaining information about the environment. After this, the AppLeS uses heuristics to select feasible sets of resources to be evaluated. It then generates a schedule for each of these sets of resources often using dynamic predictions of resource availability provided by the NWS to parameterize a performance model. The schedule with best expected performance is chosen and then deployed over grid resources.

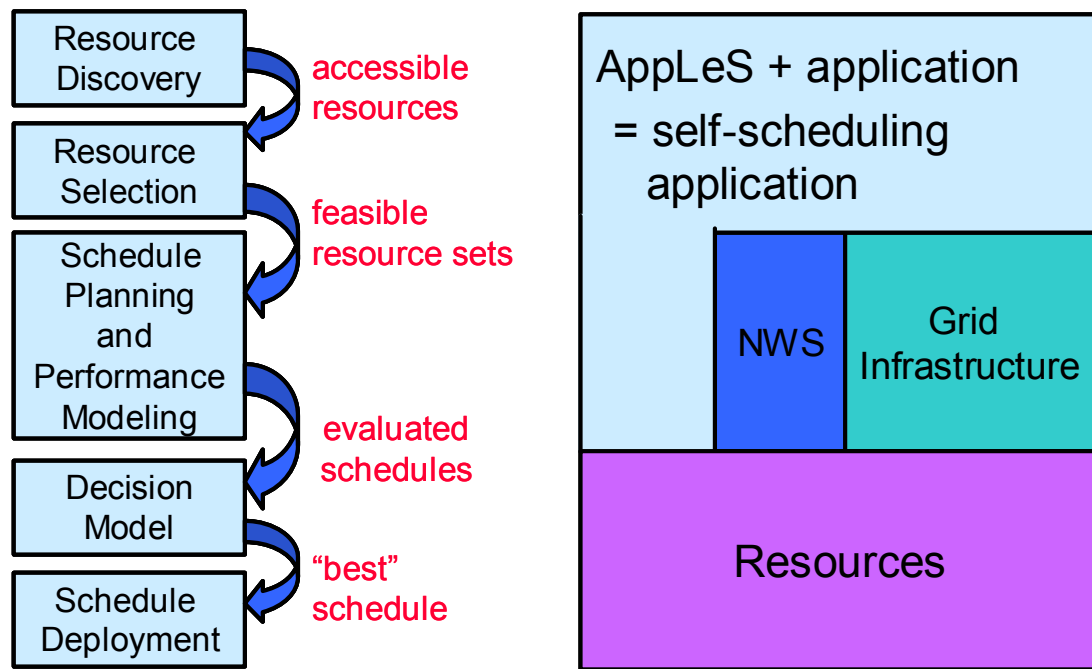


Figure 68 – The structure of an AppLeS

State of Art in Application Scheduling

There has been a great deal of interest in application scheduling in recent years [11]. As with any evolving area, there seems to be reasonable agreement on some aspects of application scheduling, but not on others. Perhaps the point that is most commonly agreed upon is that application scheduling improves when good information about the system is available. In particular, using the last measured value of the system state doesn't seem to be enough to accurately predict its future behavior [9] [19] [68] [77] [80] [88] [89] [100] [104]. This observation has motivated the appearance of systems that probe grid resources and forecast their availability, as with NWS [101] [102], Komodo [77], and Remos [68].

Although sometimes it is possible to formulate a scheduling problem in a way that can be solved in polynomial time [3], most instances of scheduling are NP-Hard. Therefore, most application schedulers use heuristics to navigate through the space of possible schedules. The main components of such heuristics are (i) how two schedules are compared, and (ii) how the space of schedules is traversed.

Most application schedulers use a performance model to compare two possible schedules [3] [4] [9] [19] [77] [80] [89] [99]. Others have devised a mechanism to rank schedules without actually estimating the application's performance [68] [104]. Since application schedulers that rely on performance models provide an estimate on the application's execution time, they can be more easily used as a component by another application scheduler. This ability makes for compositional and scalable solutions and hence is important for large systems [100]. On the other hand, performance models are hard to build. It might be that ranking-based application schedulers are easier to deploy because they do not require detailed knowledge about the application structure.

There are situations in which the space of possible schedules is small. For example, selecting the best server from among a small number of possibilities. In these cases, application schedulers can simply perform an exhaustive search [4] [19] [77] [89] [104]. When the number of possible schedules is non-trivial, heuristics can be used to search such space. Sometimes a polynomial-time optimal solution for part of the problem is used as a component of the heuristic. For example, there are application schedulers that heuristically select the resources to be used, and then perform the work distribution via time balancing [9] [27] [80] [99].

Resources Used by Application Schedulers

Most application schedulers developed so far target time-shared resources [4] [9] [19] [80] [88] [89] [99]. This is a very reasonable starting point because time-shared resources are the most common ones today. However, there are very interesting applications that demand access to other kinds of resources, such as instruments or large distributed-memory supercomputers [83].

Unfortunately it is not trivial to extend an application scheduler to deal with a new kind of resource scheduler. In order for an application scheduler to make good decisions, it needs to know how long a resource scheduler is going to take to process a given request. The problem is that obtaining such information currently requires detailed knowledge about the underlying resource schedulers. For example, application schedulers that use time-shared resources estimate the execution time of a given request

by combining resources' forecasted availability with applications' benchmarks [4] [9] [19] [80] [88] [89] [99].

Consider now a parallel supercomputer. Availability cannot be directly applied to such machines. In most of them, requests have dedicated CPU when they are executing, but might have to wait idle until the requested resources are available. Therefore, an application scheduler that uses a parallel supercomputer cannot use the same technique as the ones that target time-shared computers. Sadly, efforts to predict the supercomputer's queue wait time have not delivered techniques that are accurate enough for application scheduling [31] [55] [84] [85].

This open problem is addressed by this thesis. In Chapter 2, we described **SA**, the Supercomputer AppLeS. In its more general formulation, **SA** performs application scheduling for supercomputer jobs by simulating multiple possible requests, and then submitting the one that is expected to yield the smallest turn-around time. As seen in Chapters 5 and 6, such a simple strategy results in considerable performance gains in a variety of circumstances.

Predictability in Resource Scheduling

As discussed above, good information about the underlying resource schedulers is important for application scheduling. Currently most application schedulers obtain such information from systems that monitor resources and forecast their availability. An alternative approach would be to make resource schedulers predictable by design.

This might be a promising approach because there has been considerable effort to make resource scheduling more predictable within the operating systems community [48] [91] [95] [96]. Many of these efforts primarily aim to provide a fine level of control on how the resources are shared among their users. Predictability comes as a result of this fine level of control.

To a lesser degree, some researchers have started exploring the predictability of resource scheduling as a way to enable multiple schedulers to coexist [20] [23] [57]. These results are naturally more oriented to grid computing. The focus here is on where to draw the line dividing the responsibility of resource and application schedulers, and

what interface should one export to the other. Predictability appears as a requisite for good application scheduling.

More closely related with our research, there has been considerable interest in enhancing supercomputer schedulers to provide *advance reservations* [52]. These efforts address the need for resource schedulers to be predictable in the grid computing environment. However, reservation is not a complete solution. One also needs information that empowers application schedulers in discovering which reservation to ask for. A trial-and-error strategy (as suggested in [52]) would likely result in slow and poor application scheduling.

7.1.3. Emergent Behavior

In order to enable the wide deployment of application scheduling technology, a fundamental issue that needs to be addressed is the determination of the emergent behavior caused by multiple application schedulers in the same system, an issue also known as the *Bushel of AppLeS problem* [10]. This is indeed a very important matter because there is theoretical evidence that systems in which resource allocation is performed by many independent entities can exhibit performance degradation [71] and even chaotic behavior [59].

It has been very difficult to investigate the Bushel of AppLeS question under realistic scenarios because of the difficulty in building experimental testbeds. In addition, there has not been that many application schedulers in production use for any emergent behavior to have appeared in current systems. However, the Bushel of AppLeS problem is a particular instance of the more general problem of determining the emergent behavior of systems with multiple decision makers. In a Bushel of AppLes, it just happens that the decisions are limited to be scheduling decisions. This suggests that we might find useful results from Economics, an area in which systems with co-existing independent decision-makers are commonplace. For example, economic regulations can be seen as mechanisms to control systemic problems by reducing the freedom of the decision-makers [14].

As a matter of fact, there has been research on how economic principles can be used to provide innovative solutions for computer science problems [22] [25] [57] [72] [92] [93] [94]. Of particular interest to us are those papers that, assuming the agents to implement a particular strategy, address emergent characteristics of the system as a whole, such as convergence and stability [97] [98]. Computational markets seem to be a natural scenario to explore the stability and performance of systems with multiple independent decision-makers.

Although small, there is some literature on the Bushel of AppLeS problem per se. Some fundamental work has already been done by Hogg and Huberman [59] and Mitzenmacher [71]. They highlight the importance of diversity for the stability and performance of systems with independent decision-makers. Intuitively, when all decision-makers employ very similar strategies, there is a greater the chance for “herd behavior” to happen. Diverse systems are in general more robust, a fact that is starting to be explored in Computer Science (e.g., in security [49]).

More strongly related to our research, Downey found that choosing the partition size that minimizes the expected turn-around time of each job leads to better global performance than many proposed system-centric strategies [32]. This result did not employ application schedulers per se and was obtained under conditions and assumptions that are different from our own research. However, it suggests that multiple application schedulers might create a positive emergent behavior.

In fact, we also found the emergent behavior caused by multiple instances of SA to be beneficial. As discussed in Chapter 6, the emergent behavior generated by SA seems to reduce (i) the occurrence of very high-load conditions, and (ii) the wait time of jobs that arrive in systems experiencing moderate to high load. This is a remarkable result and one of the first attempts to characterize the performance impact the Bushel of AppLeS problem has on application scheduling.

7.2. Future Work

Research is learning without a textbook. And, as with any learning activity, research has no end. In particular, the research presented in this thesis leaves many open questions to be explored. Two of these open questions seem to be the natural “next steps” for this thesis: (i) deploying and following up **SA**, and (ii) extending **SA** to deal with multiple supercomputers.

Deploying **SA** in Real Systems

The results presented in this thesis were obtained using rigorous scientific methodologies. Care was taken to model the intricacies of real-life supercomputer usage, and we are confident **SA** will improve the job’s turn-around time in practice. But there is more to learn. Deploying **SA** in a real production system will give us feedback that we could not obtain otherwise. In particular, deploying **SA** will enable us to refine our moldability model and possibly the scheduling strategy itself.

Multiple Supercomputers

Since **SA** is an application scheduler, there is no intrinsic reason to restrict the job submission to one supercomputer. After all, an early motivation for our work was to better integrate supercomputers into the computational grid environment. By targeting multiple supercomputers, **SA** in principle would have more opportunities to improve the performance of a job. Moreover, it is conceivable that multiple instances of **SA** would provide load balancing among multiple supercomputers by avoiding the most loaded machines.

However, extending **SA** to deal with multiple supercomputers raises some non-trivial questions. First of all, does **SA** simply submit the job to all possible supercomputers (and cancel all submissions but the first one to start running), or does it select the supercomputer with best expected performance? The former approach will likely give a better performance improvement from the viewpoint of one job, but might also generate negative emergent behavior due to the large increase in cancellations throughout the entire system. If negative emergent behavior really arises, an interesting research goal is

devising a supercomputer scheduling policy (e.g. assigning cost to submissions) that dissuades this strategy of replicating a job to all possible supercomputers.

Also, should SA try to split the job across multiple supercomputers? This is clearly harder than running a job on one supercomputer because of the need to consider (i) the supercomputer-to-supercomputer network performance, and (ii) the transfer of input from and output to the “home” supercomputer. Moreover, network performance and input/output sizes have to be carefully modeled to enable the proper evaluation of such coallocation schemes. On the other hand, the performance improvement can potentially be much greater because unprecedented levels of parallelism can be achieved.

Another important issue regarding coallocation across multiple supercomputers is whether the supercomputer scheduler can provide some additional service to ease coallocation. Advance reservations and availability lists make it possible for an application scheduler to devise the requests to be sent to multiple supercomputers in order to coallocate a set of resources. However, the backfilling of these requests would happen independently from one another, probably breaking the coallocation.

There is the straightforward solution of marking requests as non-backfillable, but not benefiting from backfilling would probably hurt the job’s turn-around time. Such a simple solution might render coallocation uncompetitive compared with using a single supercomputer (where backfilling can be used with no problem).

Perhaps there is a way to enhance the interface provided by the supercomputer scheduler to better support coallocation. An idea would be to make backfilling *conditional* (i.e., you do not lose your original time slot). The idea behind conditional backfilling is that a job gets two (or maybe more) allocation slots, but you can use only one. The application scheduler would have to release the slots that the job is not going to use.

The sketch of these two research topics is not meant to suggest that these are the only research efforts that would refine the results presented herein. Deploying SA and extending it to multiple supercomputers are just natural next steps in our research. Beyond them, there is much more to explore in leveraging moldability to improve the per-

formance of supercomputer jobs and better integrating distributed-memory supercomputers in the computational grid.

8. Summary

Our thesis statement is that **the request that submits a moldable job can be automatically selected in a way that often reduces the job's turn-around time**. We support this claim by (i) describing **SA**, the Supercomputer AppLeS, an application scheduler that automatically selects the request that submits a moldable job, (ii) introducing metrics and models that allow us to evaluate **SA**, and finally (iii) using such metrics and models to show that **SA** indeed improves the job's turn-around time over the user-selected request on a large variety of scenarios.

The main contribution of this thesis is the demonstration that application schedulers can use job moldability to improve the performance of supercomputer jobs, i.e. that our thesis statement holds true. But this is not our only contribution. This thesis also contains two other important contributions to the areas of application scheduling and supercomputer scheduling. First, we conducted the first study that we are aware of on the emergent behavior of application schedulers in real-life scenarios. Second, our workload model is novel and provides a solid basis for evaluating scheduling strategies that leverage moldability to improve performance.

Exploiting Moldability to Improve Turn-Around Time

With the impressive performance of commodity processors, parallelism has become a key approach to achieve superior performance. Parallel supercomputers provide an important platform for parallel jobs, especially for those jobs that demand intensive communication and synchronization. Unfortunately, such supercomputers are expensive and thus have to be shared among multiple users.

The problem is that sharing the supercomputer results in queue delays that can jeopardize the performance gains of parallelism. Exploiting moldability to reduce a job's turn-around time addresses this problem. We introduce **SA** to show how job moldability can be used to reduce a job's turn-around time. As explained in Chapter 2,

SA receives from the user a set of requests that can be used to submit the user's job. **SA** analysis the supercomputer schedule and selects one of the requests to submit the job.

Using the evaluation criteria established in Chapters 3 and 4, we show in Chapters 5 and 6 that moldable jobs that use the request selected by **SA** often achieve smaller turn-around times than those jobs submitted through the user selected request. Moreover, **SA** is shown to be close (within 10%) to the performance achieved by the best request among those it can chose from. In practice, no application scheduler can always select the best request because this would require perfect knowledge about the jobs' execution times and future arrivals.

We also investigate the impact on **SA** of parameters that gauge the characteristics of the job, the information available to **SA**, and the state of the supercomputer. Finally, the emergent behavior of a system on which many jobs have their requests selected by **SA** is also studied (more on that below).

Note also that most jobs are already moldable (see Chapter 3). This makes solutions that explore moldability to improve performance (such as **SA**) immediately applicable in practice. This immediate applicability of **SA** is reinforced by the fact that **SA** is an application scheduler and thus can be deployed without changes in the current software infrastructure that control supercomputers (e.g., operating system, scheduler, account manager).

Emergent Behavior

Application scheduling is a key technology for scheduling in grid computing environments. However, there is concern that undesirable global behavior can emerge from systems with multiple application schedulers [10]. Such an issue has not yet been properly addressed, mainly because application scheduling is a new approach and so systems with multiple application schedulers are not yet common in practice.

In this thesis, we do address this question for **SA**, an application scheduler that selects which request submits a moldable job to a supercomputer. To the best of our knowledge, this is the first study of the emergent behavior of real-life application schedulers. As explained in Chapter 6, there are at least three different emergent behav-

iors that arise from using SA with all jobs in the system. More precisely, the emergent behavior generated by SA seems to (i) increase the competition for resources, making it somewhat harder for each instance of SA to improve performance, (ii) reduce the occurrence of very high load conditions, and (iii) reduce the wait time of jobs that arrive in systems experiencing moderate to high load. Overall, the emergent behaviors (ii) and (iii) seem to overcome the emergent behavior (i), making the performance of SA in emergent behavior conditions to be slightly better than when a single SA is present in the system. Although we cannot generalize our results to the emergent behavior of application schedulers as a whole, it is certainly encouraging that the first in-depth investigation of this issue brought positive results.

Workload Model

Researchers in supercomputer scheduling have long debated whether performance could be improved by allowing more flexibility in the job model than a fixed partition size [32] [40] [44]. The proponents of less strict job models have maintained that the scheduler would gain flexibility in resource allocation, and that this flexibility would translate into better performance. The critics, on the other hand, have suggested that parallel jobs are actually rigid, and hence strategies that rely on non-rigid jobs would not be effective in practice. Furthermore, critics have raised doubts on some scheduling solutions for non-rigid jobs because of non-realistic models used in their evaluation.

A key result of the survey we conducted among supercomputer users was the evidence that most jobs are already moldable (see Chapter 3 and Appendices A and B). Although moldability is not the most flexible job model, it can be exploited to substantially improve the scheduling of supercomputer jobs, as exemplified by SA.

Furthermore, our moldable job model makes it possible to evaluate with good accuracy scheduling solutions that assume jobs to be moldable. The model captures important characteristics that moldable jobs display in practice. These include memory constraints, maximum parallelism, algorithmic constraints on partition size, user behavior in generating requests, and job speedup behavior.

Our workload model is not only novel with respect to moldability, it also captures two other important aspects of a supercomputer workload: job cancellation and request accuracy. Job cancellation is valuable for realistic modeling because cancellations impact the scheduler behavior. Request accuracy may be even more critical for a good modeling. Practically all current supercomputer schedulers use backfilling to reclaim time that was requested but not used. However, part of the literature does not model request time and assumes the scheduler to have perfect knowledge of the job's execution times [1] [2] [67] [90]. This is unfortunate because very often the user's estimates are not good, making the ability of the scheduler in dealing with unused allocations a vital feature.

Final Remarks

We have shown that moldability can be used to improve the turn-around time of supercomputer jobs by enabling an application scheduler to select the request that submits a moldable job to a supercomputer. The need to provide solid basis for the evaluation of our solution led us to develop a novel moldable workload model. We also investigated the emergent behavior that arises when multiple jobs use our solution, in one the first efforts to characterize the emergent behavior of application schedulers. The results within this thesis contribute substantively towards more performance-efficient and environment-sensitive supercomputer scheduling and form an important building block in the pursuit of performance for supercomputer job.

Acknowledgments

We also want to express our gratitude to Alan Su and Jim Hayes for always being available for to give us a second opinion. Many thanks to Victor Hazlewood, Dror Feitelson and the people who made their workloads available at the Parallel Workloads Archive [45]. Thanks also to the members of the AppLeS group, Keith Marzullo, Rich Wolski, Allen Downey, and Warren Smith for the insightful discussions and valuable suggestions.

We thank our supporters: CAPES grant DBE2428/95-4, DoD Modernization contract 9720733-00, NPACI/NSF award ASC-9619020, and NSF grant ASC-9701333.

A. Survey's Questionnaire

1 - Which organization's parallel supercomputers do you use? Mark all that apply.

- NASA
- NCSA
- NERSC
- NPACI
- Other

2 - How many runs do you perform per month?

- 1 - 10
- 11 - 30
- 31 - 50
- 51 - 100
- More than 100

3 - How many processors do you usually request?

- 1 - 4
- 5 - 10
- 11 - 30
- 31 - 50
- 51 - 100
- More than 100
- Do not know

4 - What is the minimum number of processors your application needs to run?

- 1
 - 2 - 4
 - 5 - 10
 - 11 - 30
 - More than 30
 - Do not know
-

5 - What is the maximum number of processors your application can benefit from? Such a maximum is the number after which adding more processors doesn't reduce the application's execution time.

- 1 - 10
- 11 - 30
- 31 - 50
- 51 - 100
- 101 - 200
- More than 200
- Do not know

6 - How many processors can your application efficiently use? This number represents a trade-off. Beyond it, additional processors do not reduce the application's execution time enough to make it worth requesting them.

- 1 - 4
- 5 - 10
- 11 - 30
- 31 - 50
- 51 - 100
- More than 100
- Do not know

7 - Besides a minimum and maximum, does your application require:

- A particular number of processors (i.e., it cannot run with a different number of processors)
 - A perfect-square number of processors
 - A perfect-cube number of processors
 - A power-of-two number of processors
 - Has some other constraint on the number of processors
 - Poses no particular restriction on the number of processors it uses
 - Do not know
-

8 - How many different numbers of processors have you requested for your application? For example, if you have run your application with 8, 16, and 32 processors, the answer would be 3.

- 1
- 2 - 3
- 4 - 5
- 6 - 10
- More than 10
- Do not know

9 - What is the combined size of your input and output?

- Less than 100KB
- 100KB - 1MB
- 1MB - 10MB
- 10MB - 100MB
- More than 100MB
- Do not know

10 - For your application, is the interprocess communication?

- Low; the processes run almost independently
- Moderate
- Heavy; it has noticeable impact on the performance of the application
- Do not know

11 - How often does your application "fail" (i.e., you have to resubmit it)? The failure "cause" is not important here. It can be due to a bug, an invalid input, a system shut-down, or anything else.

- 0 - 20%
 - 21 - 40%
 - 41 - 60%
 - 61 - 80%
 - 81 - 100%
 - Do not know
-

12 - What priority do you normally use when submitting you application?

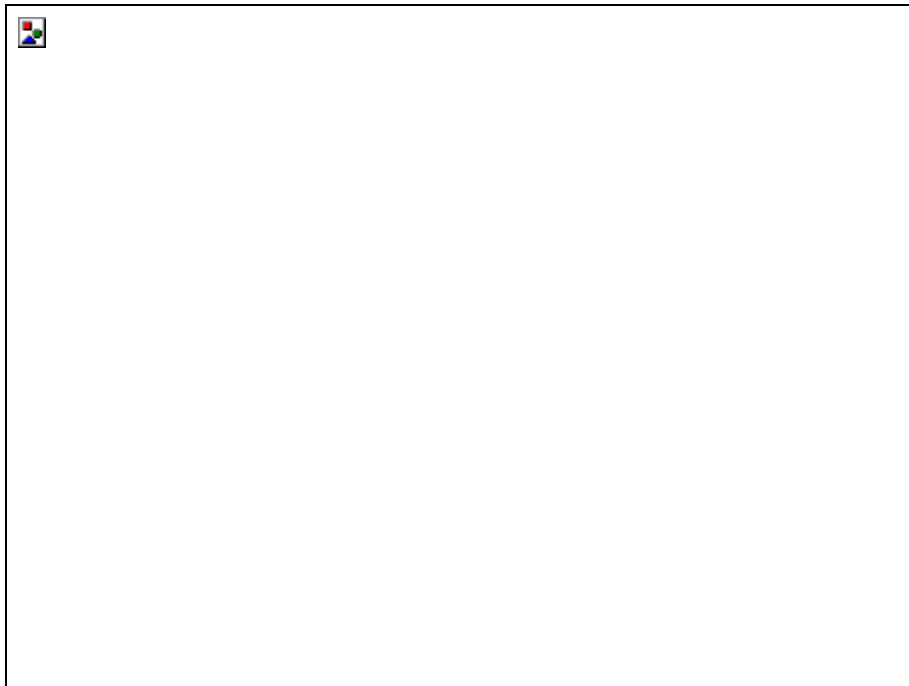
- Low
 - Normal
 - High
 - Do not know
-

B. Survey's Results

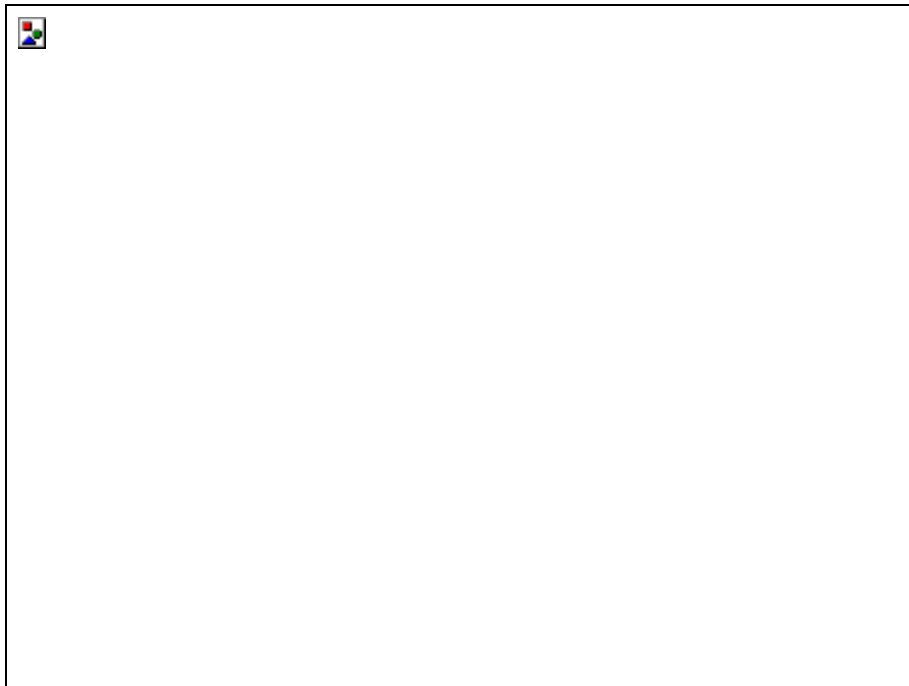
1 - Which organization's parallel supercomputers do you use? Multiple answers were allowed here.

A large empty rectangular box with a black border, intended for a chart or image. In the top-left corner of the box, there is a small icon consisting of four colored squares (red, green, blue, and yellow) arranged in a 2x2 grid.

2 - How many runs do you perform per month?

A large empty rectangular box with a thin black border, intended for a drawing or response. In the top-left corner of the box, there is a small icon consisting of four colored squares (red, green, blue, and yellow) arranged in a 2x2 grid.

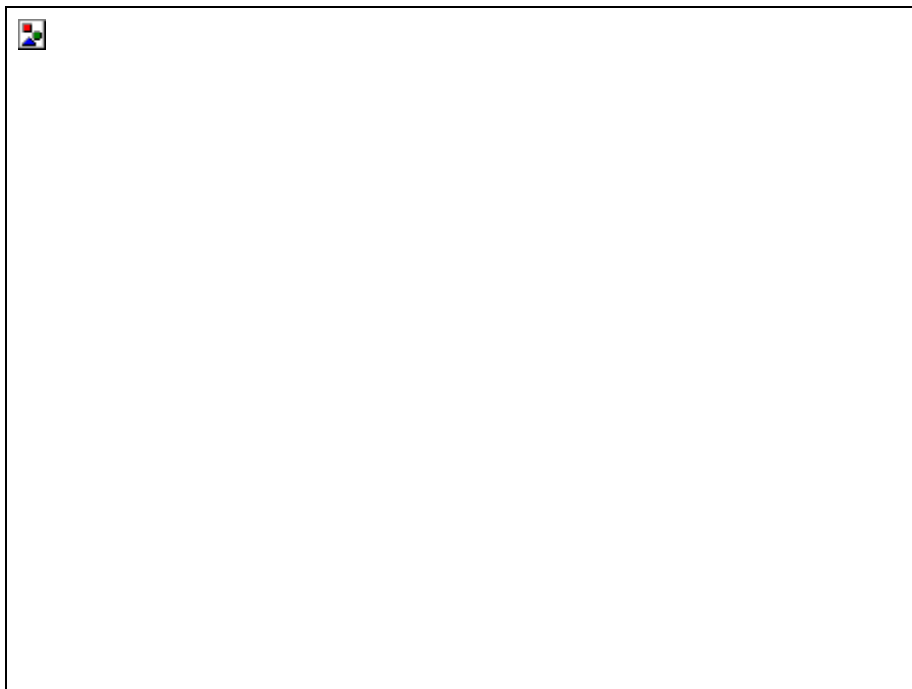
3 - How many processors do you usually request?

A large empty rectangular box with a thin black border, intended for a drawing or response. In the top-left corner of the box, there is a small icon consisting of four colored squares (red, green, blue, and yellow) arranged in a 2x2 grid.

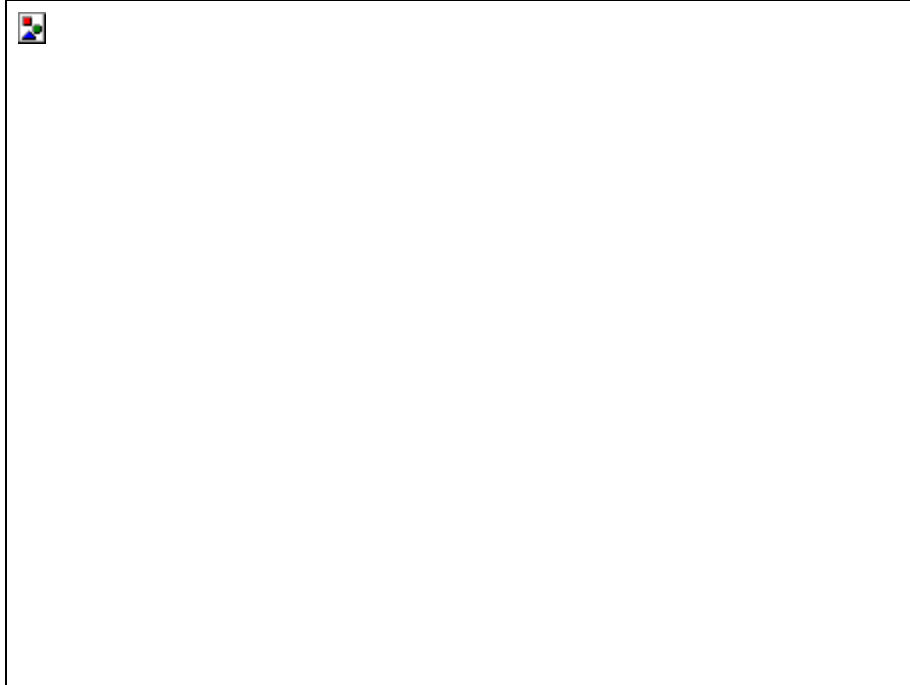
4 - What is the *minimum* number of processors your application needs to run?



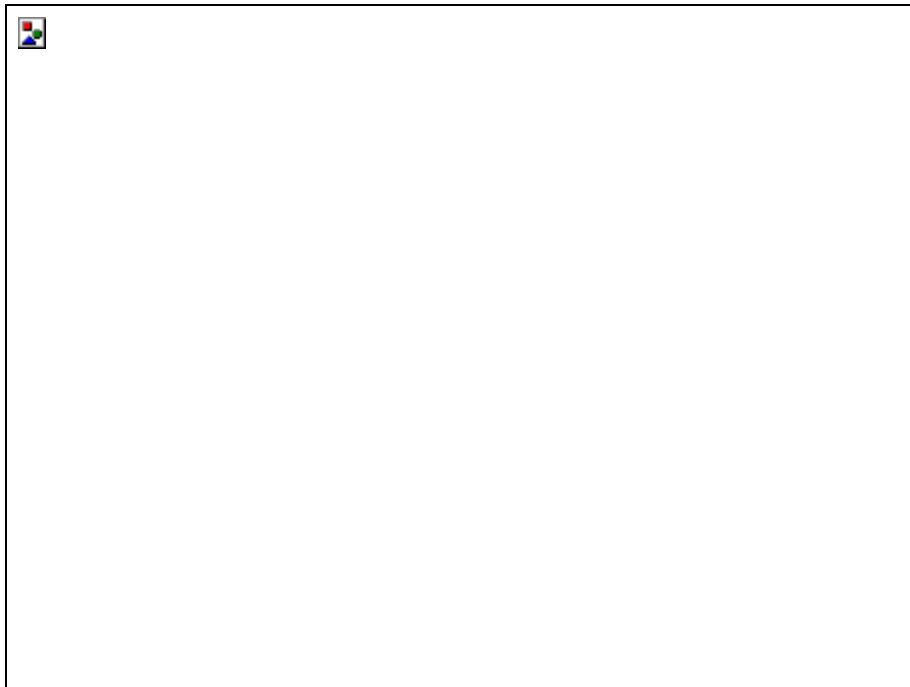
5 - What is the *maximum* number of processors your application can benefit from? Such a maximum is the number after which adding more processors doesn't reduce the application's execution time.



6 - How many processors can your application *efficiently* use? This number represents a trade-off. Beyond it, additional processors don't reduce the application's execution time enough to make it worth requesting them.



7 - Besides a minimum and maximum, does your application require a partition size that is:



8 - How many *different* numbers of processors have you requested for your application? For example, if you have run your application with 8, 16, and 32 processors, the answer would be 3.



9 - What is the combined size of your input and output?



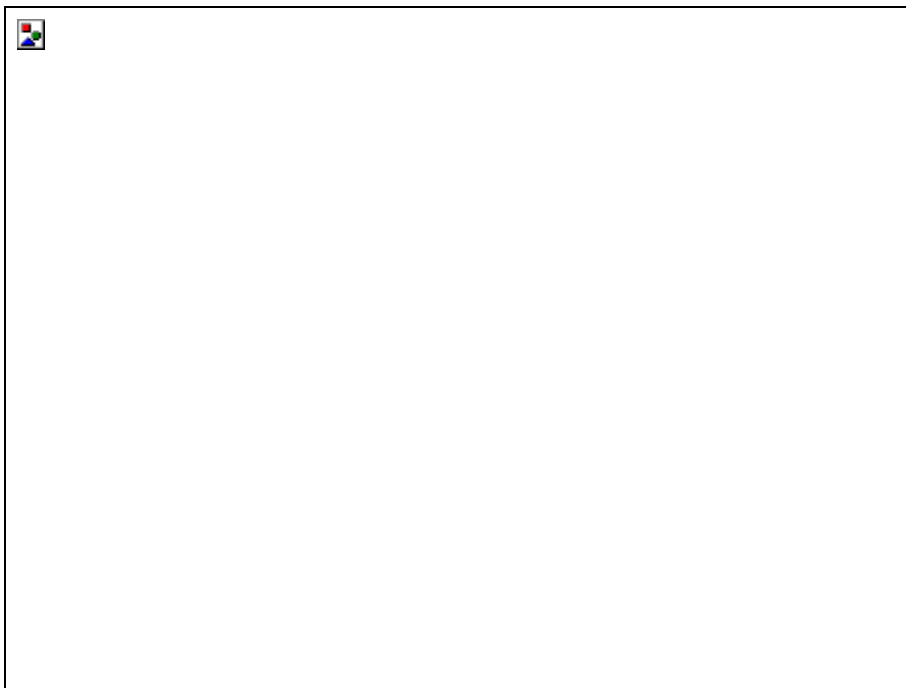
11 - For your application, is the interprocess communication? Low communication means that the processes run almost independently. Heavy communication has noticeable impact on the performance of the application.



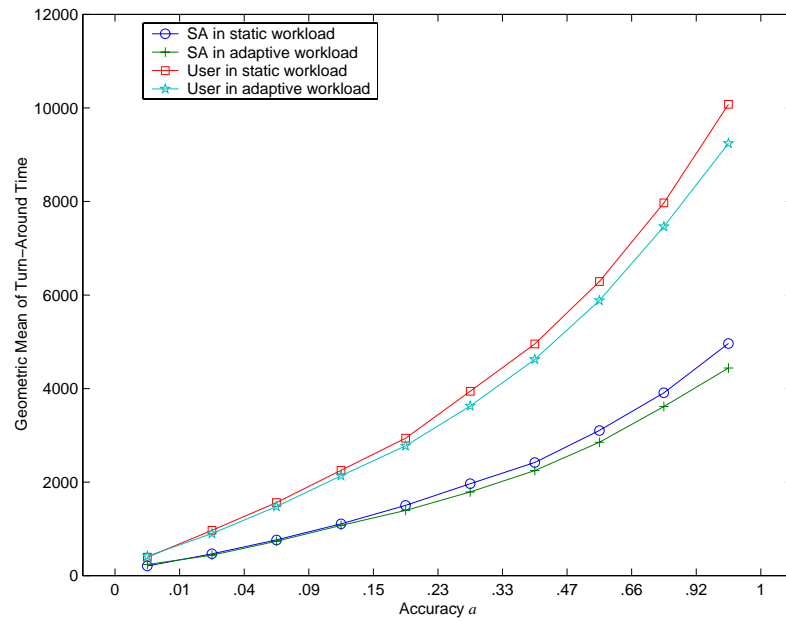
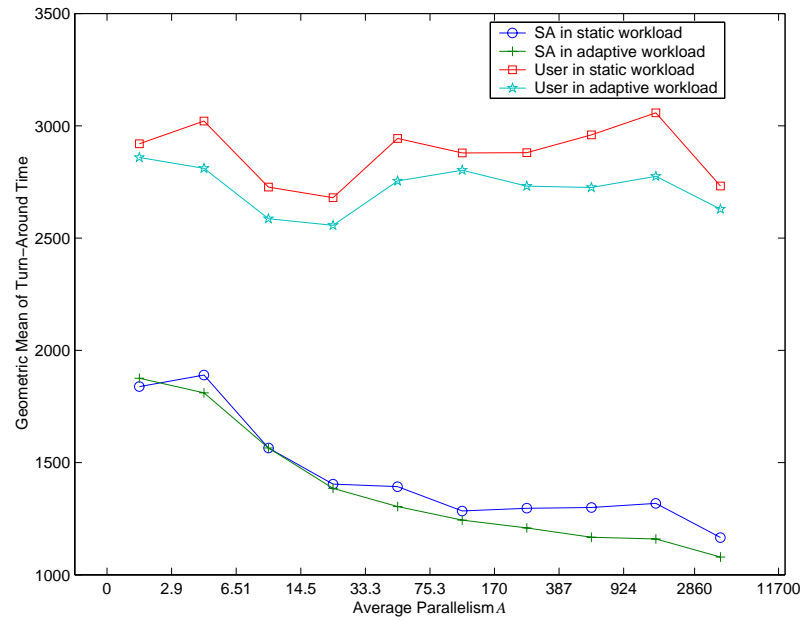
11 - How often does your application "fail" (i.e., you have to resubmit it)? The failure "cause" is not important here. It can be due to a bug, an invalid input, a system shut-down, or anything else.

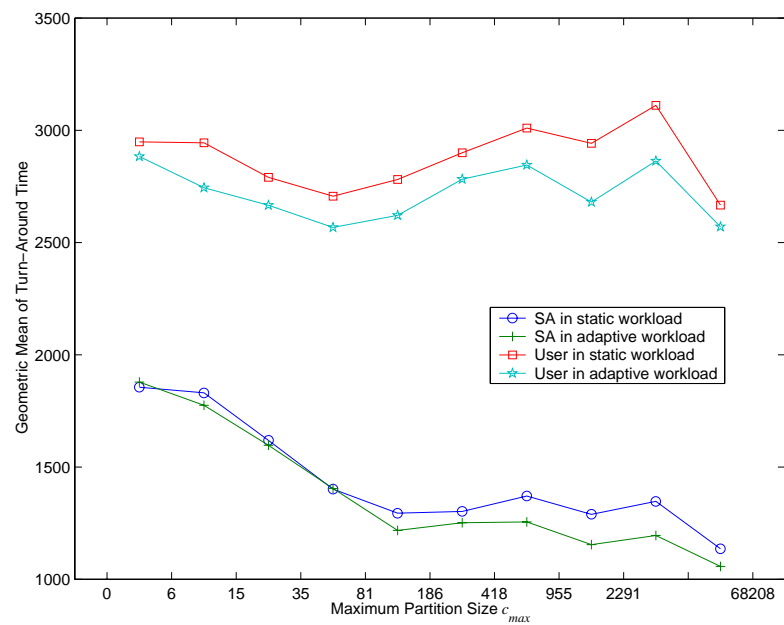
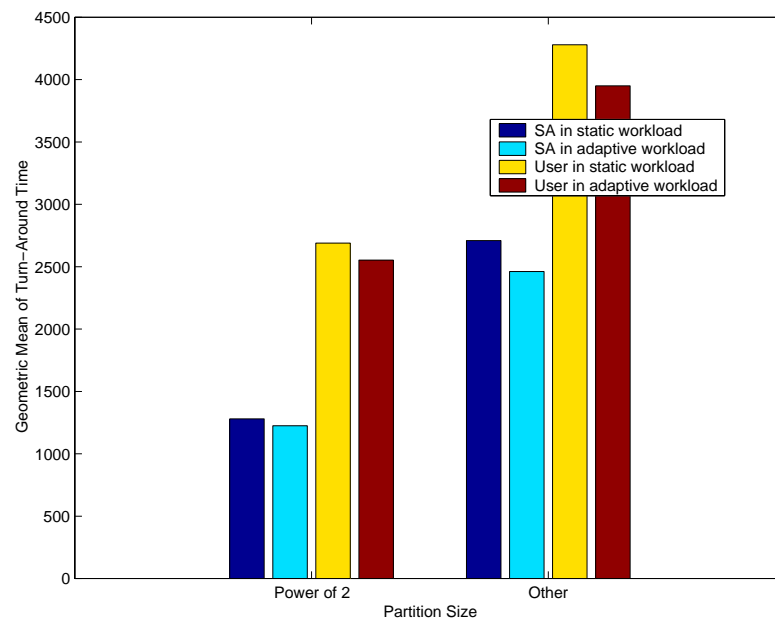


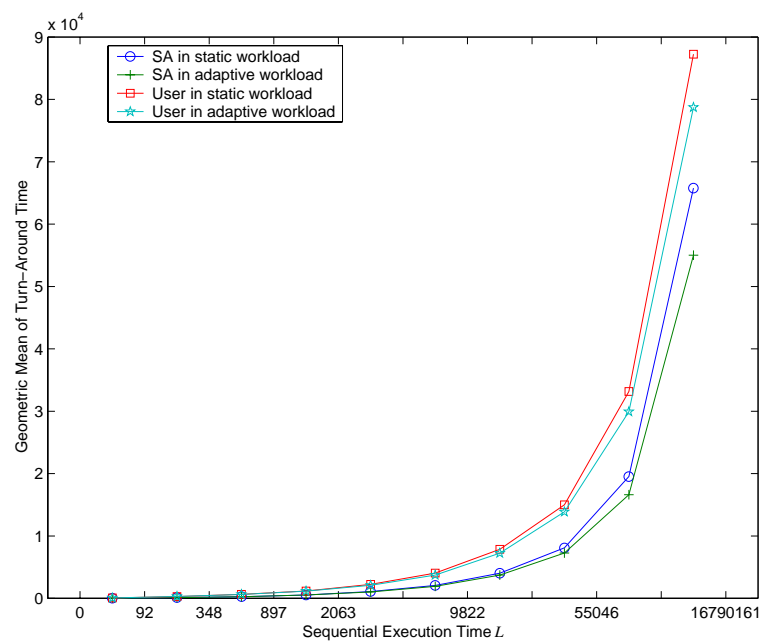
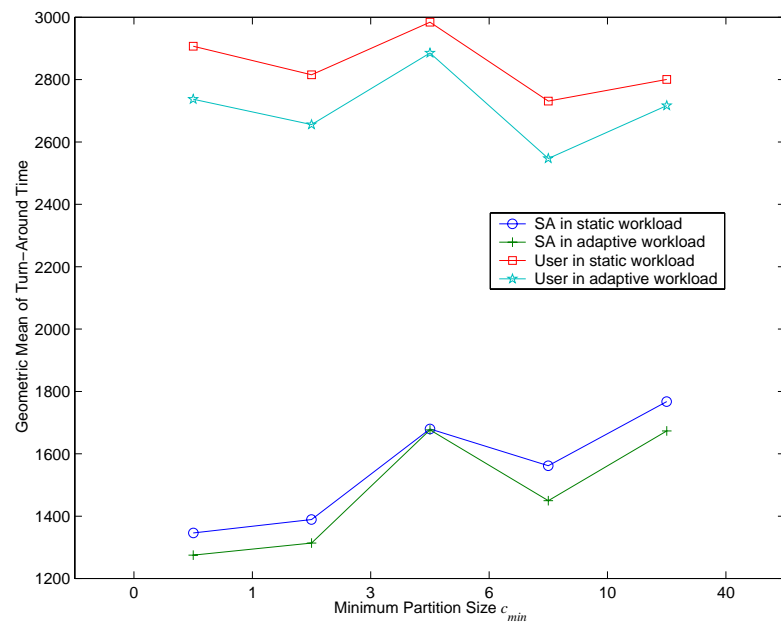
12 - What priority do you normally use when submitting you application?

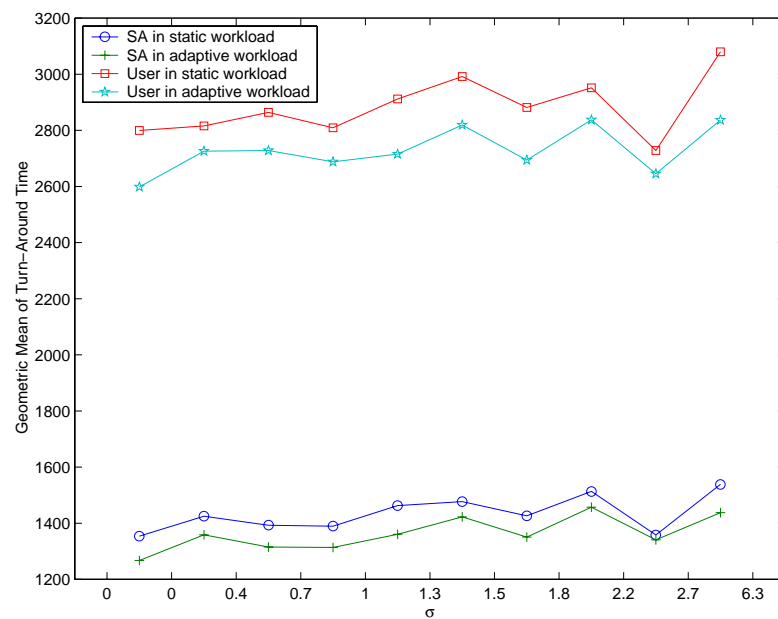
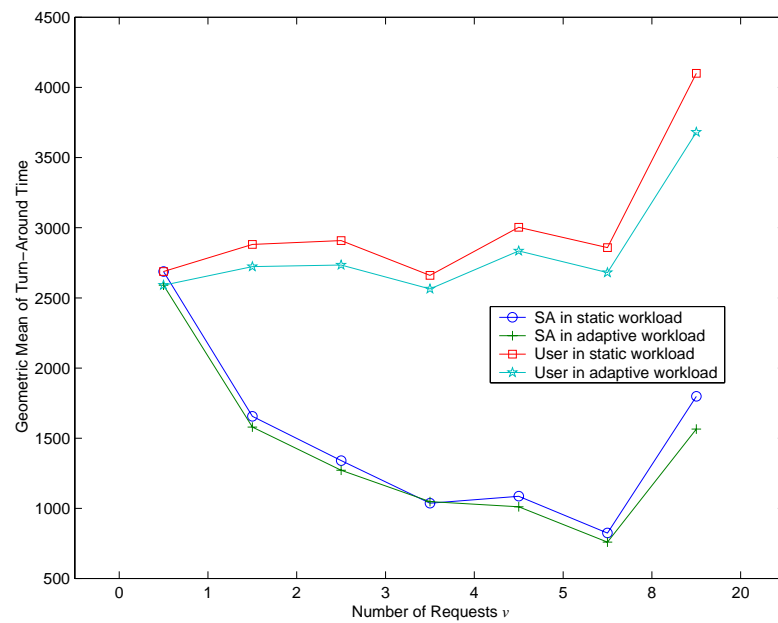
A large empty rectangular box with a black border, intended for a drawing or response. In the top-left corner of the box, there is a small icon consisting of four colored squares (red, green, blue, and yellow) arranged in a 2x2 grid.

C. Emergent Behavior Results

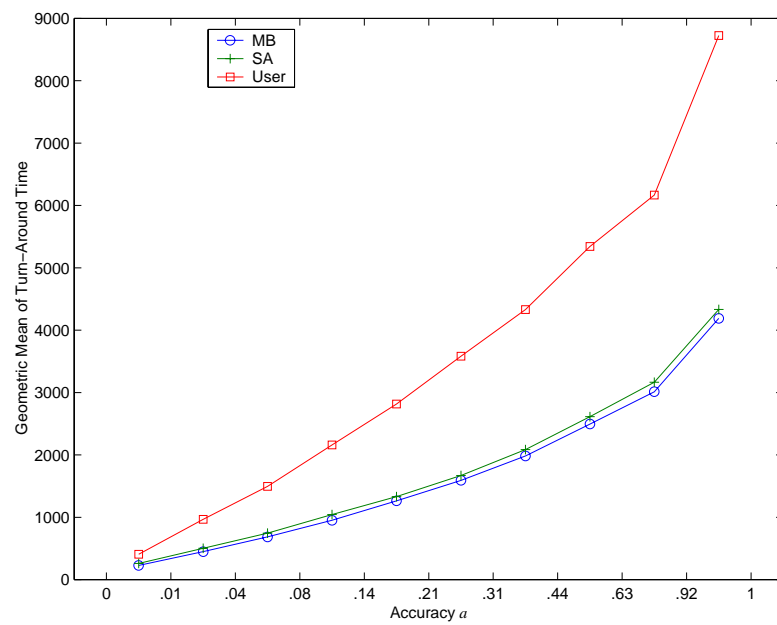
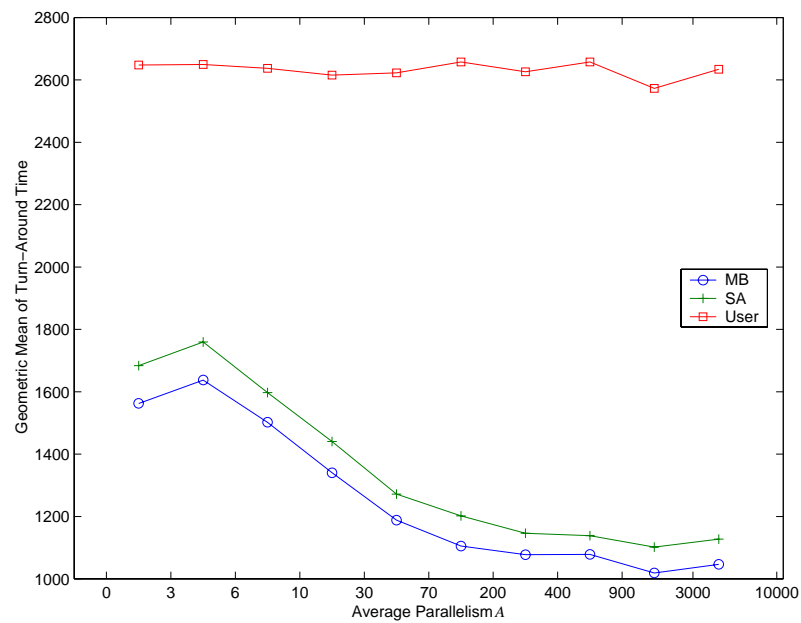


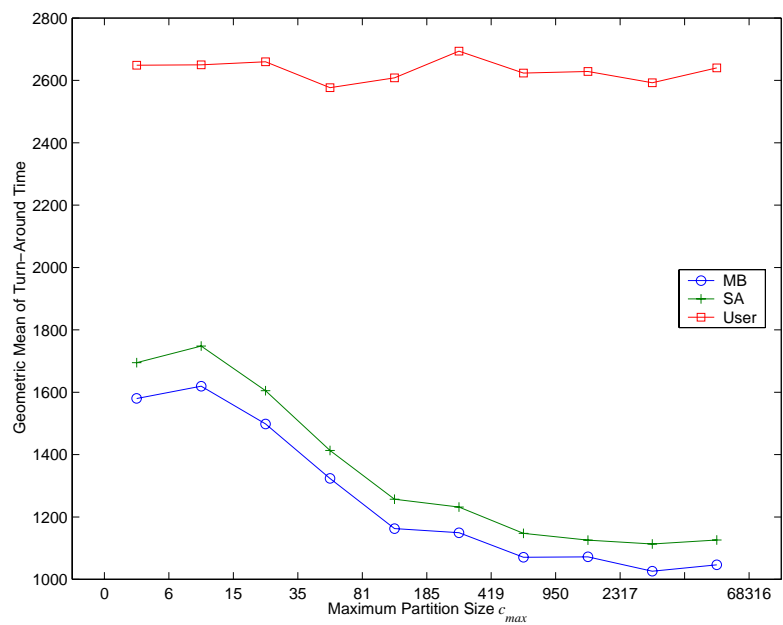
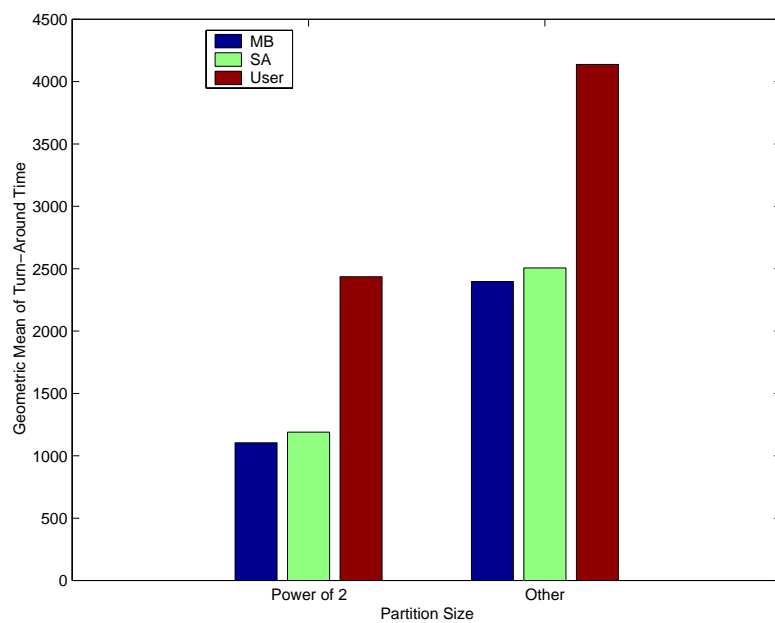


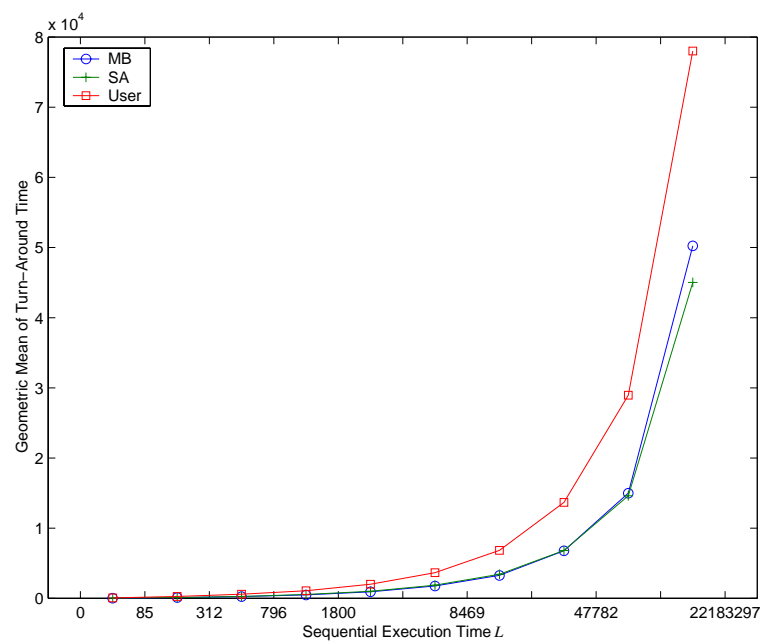
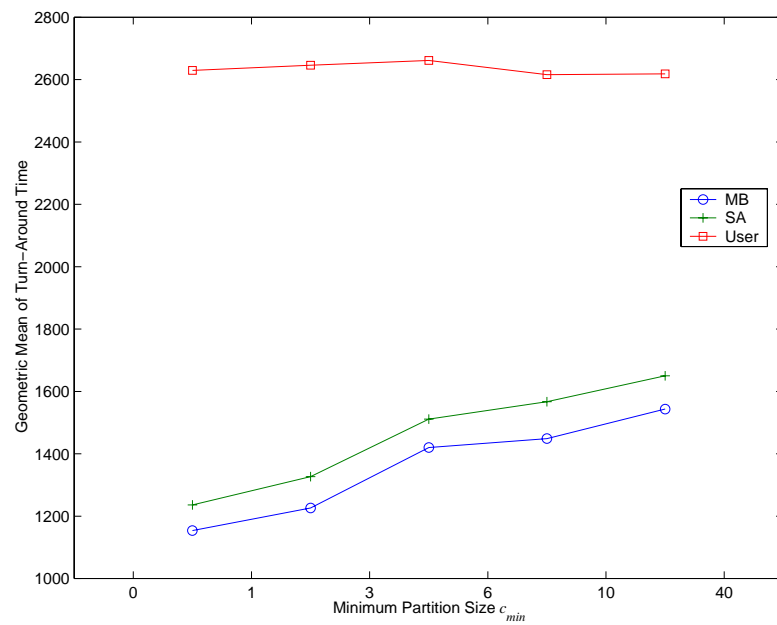


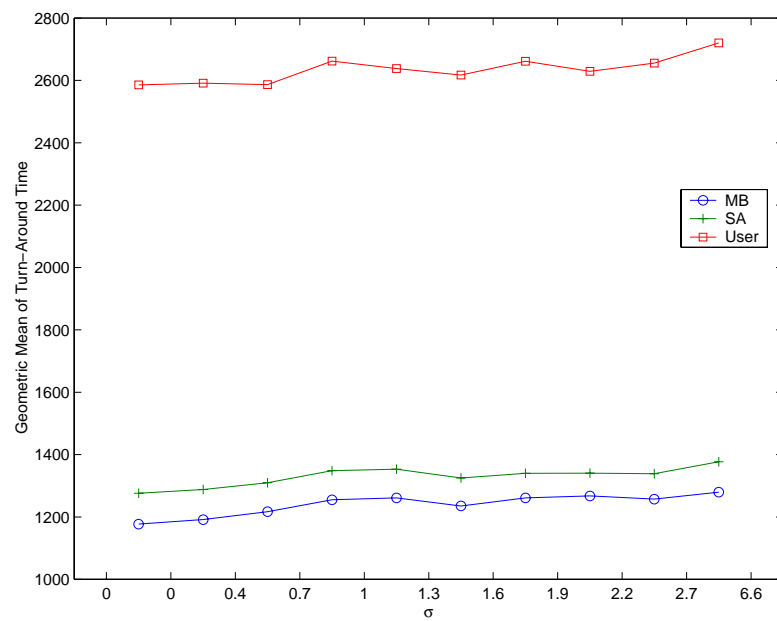
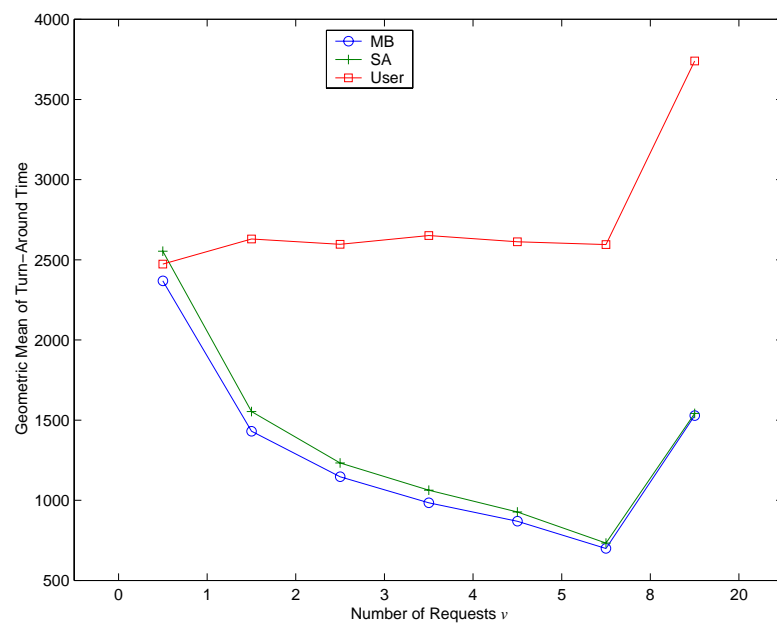


D. Moldable Backfilling Results









References

- [1] Kento Aida, Hironori Kasahara, and Seinosuke Narita. *Job Scheduling Scheme for Pure Space Sharing among Rigid Jobs*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1459, 1988.
 - [2] Kento Aida. *Effect of Job Size Characteristics on Job Scheduling Performance*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science vol. 1911, 2000.
 - [3] Alessandro Amoroso, Keith Marzullo, and Aleta Ricciardi. *Wide-Area Nile: A Case Study of a Wide-Area Data-Parallel Application*. ICDCS'98 – International Conference on Distributed Computing Systems. May 1998.
 - [4] D. Andresen, Tao Yang, O. Ibarra, and O. Egecioglu. *Adaptive partitioning and scheduling for enhancing WWW application performance*. Journal of Parallel and Distributed Computing, vol.49, (no.1), Academic Press, 25 Feb. 1998. p.57-85.
 - [5] Nimar Arora, Robert Blumofe, and C. Greg Plaxton. *Thread Scheduling for Multiprogrammed Multiprocessors*. In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 28 - July 2, 1998.
<http://www.cs.utexas.edu/users/rdb/papers.html>
 - [6] Anat Batat and Dror Feitelson. *Gang Scheduling with Memory Considerations*. IPDPS'2000, International Parallel and Distributed Processing Symposium, pp. 109-114, May 2000.
<http://www.cs.huji.ac.il/~feit/pub.html>
 - [7] O. Babaoglu, A. Bartoli, G. Dini. *Enriched View Synchrony: a Programming Paradigm for Partitionable Asynchronous Distributed Systems*. IEEE Transactions on Computers, vol.46, (no.6), IEEE, June 1997.
 - [8] Earl Babbie. *Survey Research Methods*. Wadsworth Publishing Company, 2nd edition, 1990.
 - [9] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. *Application-Level Scheduling on Distributed Heterogeneous Networks*. Supercomputing'96.
<http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html>
-

-
- [10] Fran Berman and Rich Wolski. *The AppLeS Project: A Status Report*. In Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html>
 - [11] Fran Berman. *High-Performance Schedulers*. In [53]. 1999.
 - [12] Prashant Bhat, Viktor Prasanna, and C. Raghavendra. *Adaptive Communication Algorithms for Distributed Heterogeneous Systems*. Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC'98). Chicago, Illinois, USA. July 1998.
 - [13] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. *Cilk: An Efficient Multithreaded Runtime System*. In The Journal of Parallel and Distributed Computing, 37(1), pages 55-69, August, 1996.
<http://www.cs.utexas.edu/users/rdb/papers.html>
 - [14] Dieter Bos. *Pricing and Price Regulation: An Economic Theory for Public Enterprise and Public Utilities*. Elsevier Science, 1994.
 - [15] Timothy Brecht. *An Experimental Evaluation of Processor Pool-Based Scheduling for Shared-Memory NUMA Multiprocessors*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1291, Dror Feitelson and Larry Rudolph (eds.), 1997.
 - [16] R. B. Bunt. *Scheduling Techniques for Operating Systems*. Computer. October 1976.
 - [17] M. Calzarossa and G. Serazzi. *A characterization of the variation in time of workload arrival patterns*. IEEE Transactions on Computers, vol. C-34, (no.2), Feb. 1985. p.156-62.
 - [18] M. Calzarossa and G. Serazzi. *Workload Characterization: A Survey*. Proceedings of the IEEE, vol. 81, no. 8, pp. 1136-1150, August 1993.
 - [19] Henri Casanova and Jack Dongarra. *NetSolve: A Network Server for Solving Computational Science Problems*. Supercomputing'96, 1996.
<http://www.netlib.org/utk/people/JackDongarra/papers.html>
 - [20] Steve Chapin. *Distributed Scheduling Support in the Presence of Autonomy*. Proceedings of the 4th Heterogeneous Computing Workshop, pp. 22-29, Santa Barbara, CA, April 1995.
<http://www.cs.virginia.edu/~chapin/papers/hcw95.ps>
 - [21] Steve Chapin, Walfredo Cirne, Dror Feitelson, James Jones, Scott Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. *Benchmarks and Standards for the Evaluation of Parallel Job Schedulers*. In Job Scheduling Strategies
-

-
- for Parallel Processing, D. Feitelson and Larry Rudolph (Eds.) Springer-Verlag, Lecture Notes in Computer Science, vol. 1659, pp. 66-89, 1999.
<http://www-cse.ucsd.edu/users/walfredo/resume.html#publications>
- [22] John Cheng and Michael Wellman. *The WALRAS algorithm: A Convergent Distributed Implementation of General Equilibrium Outcomes*. Computational Economics, 12:1-24, 1998.
<http://ai.eecs.umich.edu/people/wellman/Publications.html#MOP>
- [23] Walfredo Cirne and Keith Marzullo. *The Computational Co-op: Gathering Clusters into a Metacomputer*. In Proceeding of IPPS/SPDP'99, April 1999.
<http://www-cse.ucsd.edu/users/walfredo/resume.html#publications>
- [24] Walfredo Cirne and Francine Berman. *Adaptive Selection of Partition Size for Supercomputer Requests*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science vol. 1911, 2000.
<http://www-cse.ucsd.edu/users/walfredo/resume.html#publications>
- [25] Scott Clearwater (editor). *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific. 1996.
- [26] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke. *A Resource Management Architecture for Metacomputing Systems*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science vol. 1459, 1998.
<http://www-fp.globus.org/documentation/papers.html>
- [27] Holly Dail, Graziano Obertelli, Francine Berman, Rich Wolski, and Andrew Grimshaw. *Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem*. Proceedings of the 9th Heterogeneous Computing Workshop, May 2000.
<http://apples.ucsd.edu/hetpubs.html>
- [28] Peter Dinda, D. O'Hallaron. *An Evaluation of Linear Models for Host Load Prediction*. Proceedings of the 8th IEEE Symposium on High-Performance Distributed Computing (HPDC-8), Redondo Beach, CA, August 1999.
<http://www.cs.cmu.edu/afs/cs/usr/pdinda/html/papers.html>
- [29] Jay Devore. *Probability and Statistics for Engineering and the Sciences*. Fourth Edition, Wadsworth Publishing Company, 1995.
- [30] Allen Downey. *A model for speedup of parallel programs*. U.C. Berkeley Technical Report CSD-97-933, January 1997.
-

<http://www.sdsc.edu/~downey/model/>

- [31] Allen Downey. *Predicting queue times on space-sharing parallel computers*. 11th International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland, April 1997.
<http://www.sdsc.edu/~downey/predicting/>
 - [32] Allen Downey. *Using Queue Time Predictions for Processor Allocation*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1291, Dror Feitelson and Larry Rudolph (eds.), 1997.
<http://www.sdsc.edu/~downey/predalloc/>
 - [33] Allen Downey. *A parallel workload model and its implications for processor allocation*. 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97), August 1997.
<http://www.sdsc.edu/~downey/allocation/>
 - [34] Allen Downey and Dror Feitelson. *The elusive goal of workload characterization*. Perf. Eval. Rev. 26(4), pp. 14-29, March 1999.
<http://www.cs.huji.ac.il/~feit/pub.html>
 - [35] M. Drozdowski. *Scheduling Multiprocessor Tasks: An Overview*. European Journal of Operational Research 94, pp. 215-230, 1996.
 - [36] Derek Eager, John Zahorjan, and Edward Lazowska. *Speedup Versus Efficiency in Parallel Systems*. IEEE Transactions on Computers, vol. 38, no. 3, March 1989.
 - [37] Graham Fagg, Keith Moore, Jack Dongarra, and Al Geist. *Scalable Networked Information Processing Environment (SNIPE)*. Supercomputing '97. San Jose, CA, USA. 1997.
<http://www.supercomp.org/sc97/proceedings/TECH/MOORE/INDEX.HTM>
 - [38] Dror Feitelson and Larry Rudolph. *Gang Scheduling Performance Benefits for Fine-Grain Synchronization*. Journal of Parallel and Distributed Computing (16) 306-318, 1992.
 - [39] Dror Feitelson and Bill Nitzberg. *Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.
<http://www.cs.huji.ac.il/~feit/pub.html>
 - [40] Dror Feitelson, Larry Rudolph, Uwe Schweigelshohn, Kenneth Sevcik, and Park-son Wong. *Theory and Practice in Parallel Job Scheduling*. 3rd Workshop on Job
-

-
- Scheduling Strategies for Parallel Processing, Springer-Verlag Lecture Notes in Computer Science, vol. 1291, pp. 1-34, April 1997.
<http://www.cs.huji.ac.il/~feit/parsched/parsched97.html>
- [41] Dror Feitelson. *Packing schemes for gang scheduling*. In Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science vol. 1162, Dror Feitelson and Larry Rudolph (eds.), pp. 89-110, Springer-Verlag, 1996.
<http://www.cs.huji.ac.il/~feit/pub.html>
- [42] Dror Feitelson and Morris Jette. *Improved Utilization and Responsiveness with Gang Scheduling*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), pp. 238-261, Lecture Notes in Computer Science Vol. 1291, Springer-Verlag, 1997.
<http://www.cs.huji.ac.il/~feit/pub.html>
- [43] Dror Feitelson and A. Mu'alem Weil. *Utilization and predictability in scheduling the IBM SP2 with backfilling*. In 12th Intl. Parallel Processing Symp., pp. 542-546, Apr 1998.
<http://www.cs.huji.ac.il/~feit/pub.html>
- [44] Dror Feitelson and Larry Rudolph. *Metrics and Benchmarking for Parallel Job Scheduling*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), pp. 1-24, Springer-Verlag, Lecture Notes in Computer Science vol. 1459, 1998.
- [45] Dror Feitelson. *The Parallel Workloads Archive*.
<http://www.cs.huji.ac.il/labs/parallel/workload/>
- [46] Adam Ferrari et al. *A Flexible Security System for Metacomputing Environments*. Technical Report CS-98-36, Department of Computer Science, University of Virginia.
<http://www.cs.virginia.edu/~legion/papers.html>
- [47] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke. *A Directory Service for Configuring High-Performance Distributed Computations*. 6th IEEE Symposium on High-Performance Distributed Computing, pg. 365-375, 1997.
<http://www-fp.globus.org/documentation/papers.html>
- [48] Liana Fong and Mark Squillante. *Time-Function Scheduling: A General Approach to Controllable Resource Management*. Technical Report RC 20155 (89194), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, August 1995.
-

-
- <http://domino.watson.ibm.com/library/cyberdig.nsf/a3807c5b4823c53f85256561006324be/05a54a208d1f0f6a852565930072576a?OpenDocument>
- [49] Stephanie Forrest et al. *Building Diverse Computer Systems*. Proceeding of the 6th Workshop on Hot Topics in Operating Systems, pp. 67-72. 1997.
<ftp://ftp.cs.unm.edu/pub/forrest/hotos-97.ps>
- [50] I. Foster, J. Geisler, C. Kesselman, S. Tuecke. *Managing Multiple Communication Methods in High-Performance Networked Computing Systems*. Journal of Parallel and Distributed Computing, 40:35-48, 1997.
<http://www-fp.globus.org/documentation/papers.html>
- [51] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. *A Security Architecture for Computational Grids*. Proc. 5th ACM Conference on Computer and Communications Security Conference, pg. 83-92, 1998.
<http://www-fp.globus.org/documentation/papers.html>
- [52] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation*. International Workshop on Quality of Service, 1999.
<http://www-fp.globus.org/documentation/papers.html>
- [53] Ian Foster and Carl Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers. July 1998.
- [54] Gaurav Ghare and Scott Leutenegger. *The Effect of Correlating Quantum Allocation and Job Size for Gang Scheduling*. In Job Scheduling Strategies for Parallel Processing, D. Feitelson and Larry Rudolph (Eds.) Springer-Verlag, Lecture Notes in Computer Science, vol. 1659, pp. 66-89, 1999.
- [55] Richard Gibbons. *A Historical Application Profiler for Use by Parallel Schedulers*. Lecture Notes in Computer Science, vol. 1297, 58-75, Springer-Verlag, 1997.
- [56] Victor Hazlewood. *NPACI JobLog Repository*.
<http://joblog.npaci.edu/>
- [57] Kieran Harty and David Cheriton. *A Market Approach to Operating System Memory Allocation*. In [25], 1996.
<ftp://ftp.dsg.stanford.edu/pub/papers/memmarket.ps.Z>
- [58] Robert Henderson. *Job Scheduling Under the Portable Batch System*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.
-

-
- [59] Tad Hogg and Bernardo Huberman. *Controlling Chaos in Distributed Systems*. IEEE Transactions on Systems, Man, and Cybernetics, vol. 21, no. 6, November/December 1991.
- [60] Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riordan. *Modeling of Workload in MPPs*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1291, Dror Feitelson and Larry Rudolph (eds.), 1997.
- [61] James Jones and Bill Nitzberg. *Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lectures Notes in Computer Science vol. 1659, 1999.
- [62] John Karpovich. *Support for Object Placement in Wide Area Heterogeneous Distributed Systems*. UVa CS Technical Report CS-96-03. January 1996.
<http://www.cs.virginia.edu/~legion/papers.html>
- [63] Nirav Kapadia, José Fortes, and Carla Brodley. *Predictive Application-Performance Modeling in a Computational Grid Environment*. Eighth IEEE Symposium on High-Performance Distributed Computing, July 1999.
- [64] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. *On the Design and Evaluation of Job Scheduling Algorithms*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lectures Notes in Computer Science vol. 1659, 1999.
- [65] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. *Implications of I/O for Gang Scheduled Workloads*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1291, Dror Feitelson and Larry Rudolph (eds.), 1997.
- [66] David Lifka. *The ANL/IBM SP Scheduling System*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science Vol. 949, 1995.
<http://www.tc.cornell.edu/UserDoc/SP/Batch/what.html>
- [67] V. Lo, J. Mache, and K. Windisch. *A comparative study of real workload traces and synthetic workload models for parallel job scheduling*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (eds.), pp. 25-46, Springer Verlag, Lect. Notes Comput. Sci. vol. 1459, 1998.
-

-
- [68] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. *A Resource Query Interface for Network-Aware Applications*. Seventh IEEE Symposium on High-Performance Distributed Computing, July 1998.
<http://www.cs.cmu.edu/~cmcl/remulac/papers.html>
- [69] Maui High Performance Computing Center. *The Maui Scheduler Web Page*.
<http://wailea.mhpcc.edu/maui/>
- [70] C. McCann and J. Zahorjan. *Scheduling Memory Constrained Jobs on Distributed-Memory Parallel Computers*. In SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 208-291, May 1995.
- [71] Michael Mitzenmacher. *How Useful is Old Information?* PODC 97. 1997.
<http://www.research.digital.com/SRC/personal/michaelm/WORK/papers.html>
- [72] Tracy Mullen and Michael Wellman. *Market-based negotiation for digital library services*. Second USENIX Workshop on Electronic Commerce. November 1996.
<ftp://ftp.eecs.umich.edu/people/wellman/usenix96.ps.Z>
- [73] David Patterson, John Hennessy, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Second Edition, Morgan Kaufmann Publishing, 1996.
- [74] Platform Computing Corp. *Load Sharing Facility Web Page*.
<http://www.platform.com/platform/platform.nsf/webpage/LSF?OpenDocument>
- [75] Fabio Previato, Michael Ogg, and Aleta Ricciardi. *Experience with Distributed Replicated Objects: The Nile Project*. European Research Seminar in Advanced Distributed Systems Zinal, Switzerland, 17-21 March 1997.
<http://www.nile.utexas.edu/Nile/conferences/papers/>
- [76] J. Pruyn and M. Livny. *Parallel Processing on Dynamic Resources with Carmi*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.
- [77] M. Ranganathan, A. Acharya, and J. Saltz. *Distributed Resource Monitor for Mobile Objects*. IWOOOS'96.
<http://www.cs.umd.edu/users/acha/publications.html>
- [78] Kostadis Roussos, Nawaf Bitar, and Robert English. *Deterministic Batch Scheduling without Static Partitioning*. In Job Scheduling Strategies for Parallel Processing, D. Feitelson and Larry Rudolph (Eds.) Springer-Verlag, Lecture Notes in Computer Science, vol. 1659, pp. 66-89, 1999.
- [79] Uwe Schewiegelshohn and Ramin Yahyapour. *Improving First-Come-First-Serve Job Scheduling by Gang Scheduling*. In Job Scheduling Strategies for Parallel
-

-
- Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science vol. 1459, 1998.
- [80] Gary Shao, Rich Wolski, and Fran Berman. *Predicting the Cost of Redistribution in Scheduling*. In Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html>
- [81] Fabrício Silva and Isaac Scherson. *Improving Parallel Job Scheduling Using Runtime Measurements*. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science vol. 1911, 2000.
<http://www.cs.huji.ac.il/~feit/parsched/parsched00.html>
- [82] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. *The EASY-LoadLeveler API project*. 2nd Workshop on Job Scheduling Strategies for Parallel Processing, Springer-Verlag Lecture Notes in Computer Science, vol. 1162, pp. 41-47, April 1996.
<http://www.tc.cornell.edu/UserDoc/SP/Batch/what.html>
- [83] Shava Smallen, Walfredo Cirne, Jaime Frey, Francine Berman, Rich Wolski, Mei-Hui Su, Carl Kesselman, Steve Young, and Mark Ellisman. *Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience*. Proceedings of HCW'2000 – Heterogeneous Computing Workshop, May 2000.
<http://www-cse.ucsd.edu/users/walfredo/resume.html#publications>
- [84] W. Smith, I. Foster, and V. Taylor. *Predicting Application Run Times Using Historical Information*. Lecture Notes in Computer Science, 1459:122-142, Springer-Verlag, 1998.
<http://www-fp.mcs.anl.gov/~wsmith/papers.html>
- [85] W. Smith, V. Taylor, and I. Foster. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*. In Proceedings of the IPPS/SPDP'99 Workshop on Job Scheduling Strategies for Parallel Processing, 1999.
<http://www-fp.mcs.anl.gov/~wsmith/papers.html>
- [86] Allan Snaveley and Dean Tullsen. *Symbiotic Jobscheduling for a Simultaneous Multithreading Architecture*. In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, November, 2000
<http://www-cse.ucsd.edu/users/tullsen/research.html>
-

-
- [87] Standard Performance Evaluation Corporation. *The SPEC web page*.
<http://www.spec.org/>
- [88] Neil Spring and Rich Wolski. *Application Level Scheduling of Gene Sequence Comparison on Metacomputers*. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July, 1998.
<http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html>
- [89] Alan Su, Francine Berman, Richard Wolski, and Michelle Strout. *Using AppLeS to Schedule Simple SARA on the Computational Grid*. In International Journal of High Performance Computing Applications, vol. 13, 1999.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html>
- [90] Jaspal Subhlok, Thomas Gross, and Takashi Suzuoka. *Impact of Job Mix on Optimizations for Space Sharing Schedulers*. Supercomputing'96, 1996.
<http://www.supercomp.org/sc96/proceedings/SC96PROC/TTITLES.HTM>
- [91] Ion Stoica et al. *A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems*. 17th Real-Time Systems Symposium, Washington DC, p. 288-299, December 1996.
<http://www.cs.odu.edu/~stoica/pubs.html>
- [92] Michael Stonebraker et al. *Mariposa: A Wide-Area Distributed Database System*. VLDB Journal 5, 1, p. 48-63, January 1996.
<http://epoch.cs.berkeley.edu:8000/mariposa/papers.html>
- [93] Paul Tucker and Fran Berman. *On Market Mechanisms as a Software Technique*. UCSD Technical Report #CS96-513.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#Miscellaneous>
- [94] Carl Waldspurger et al. *Spawn: A Distributed Computational Economy*. IEEE Transactions on Software Engineering, vol. 18, no. 2, pp. 103-17. February 1992.
<http://www.research.digital.com/SRC/personal/caw/papers.html>
- [95] Carl Waldspurger and William Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management*. In Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94), pages 1-11, Monterey, California, November 1994.
<http://www.research.digital.com/SRC/personal/caw/papers.html>
- [96] Carl Waldspurger and William Weihl. *Stride Scheduling: Deterministic Proportional-Share Resource Management*. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.
<http://www.research.digital.com/SRC/personal/caw/papers.html>
-

-
- [97] William Walsh and Michael Wellman. *A market protocol for decentralized task allocation*. Extended version of a paper in Proceedings of the Third International Conference on Multiagent Systems, July 1998.
<http://ai.eecs.umich.edu/people/wellman/Publications.html>
- [98] William Walsh and Michael Wellman. *Efficiency and equilibrium in task allocation economies with hierarchical dependencies*. In Sixteenth International Joint Conference on Artificial Intelligence, pages 520-526, August 1999.
<http://ai.eecs.umich.edu/people/wellman/Publications.html>
- [99] Jon Weissman and Andrew Grimshaw. *A Framework for Partitioning Parallel Computations in Heterogeneous Environments*. Concurrency: Practice and Experience, Vol. 7, No. 5, August 1995.
<http://ringer.cs.utsa.edu/faculty/weissman.html/pub.html>
- [100] Jon Weissman. *Gallop: The Benefits of Wide-Area Computing for Parallel Processing*. Journal of Parallel and Distributed Computing, Vol. 54(2), November 1998.
<http://ringer.cs.utsa.edu/faculty/weissman.html/pub.html>
- [101] Rich Wolski. *Dynamically Forecasting Network Performance Using the Network Weather Service*. In Journal of Cluster Computing, 1998.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#NWS>
- [102] Rich Wolski, Neil Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. To appear in the Journal of Future Generation Computing Systems, 1999.
<http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#NWS>
- [103] Rich Wolski, Neil Spring and Jim Hayes. *Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid*. 8th International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, USA, 3-6 Aug 1999.
<http://www-cse.ucsd.edu/~rich/publications.html>
- [104] Huican Zhu et al. *Adaptive Load Sharing for Clustered Digital Library Servers*. Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, Illinois, July, 1998.
<http://www.alexandria.ucsb.edu/~zheng/publications.html>
- [105] Bernard Zeigler, Herbert Praehofer, and Tao Gon Kim. *Theory of Modeling and Simulation*. Second Edition. Academic Press. 2000.
-

- [106] Yanyong Zhang, Anand Sivasubramaniam, Hubertus Franke, and José Moreira. *Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques*. IPDPS'2000, International Parallel and Distributed Processing Symposium, Cancun, Mexico, May 1-5, 2000.
 - [107] D. Zotkin and P. Keleher. *Job-Length Estimation and Performance in Backfilling Schedulers*. 8th International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, USA, 3-6 Aug 1999.
-