Using Queue Time Predictions for Processor Allocation

Allen B. Downey

University of California, Berkeley CA 94720

Abstract. When a moldable job is submitted to a space-sharing parallel computer, it must choose whether to begin execution on a small, available cluster or wait in queue for more processors to become available. To make this decision, it must predict how long it will have to wait for the larger cluster. We propose statistical techniques for predicting these queue times, and develop an allocation strategy that uses these predictions. We present a workload model based on observed workloads at the San Diego Supercomputer Center and the Cornell Theory Center, and use this model to drive simulations of various allocation strategies. We find that prediction-based allocation not only improves the turnaround time of individual jobs; it also improves the utilization of the system as a whole.

1 Introduction

Like many shared resources, parallel computers are susceptible to a tragedy of the commons – individuals acting in their own interests tend to overuse and degrade the resource. Specifically, users trying to minimize run times for their jobs might allocate more processors than they can use efficiently. This indulgence lowers the utilization of the system and increases queue times.

A partial solution to this problem is a programming model that supports *adaptive* jobs, that is, jobs that can be configured to run on clusters of various sizes. These jobs improve system utilization by using fewer processors when system load is high, thereby running more efficiently and increasing the number of jobs in the system simultaneously.

But adaptive jobs are not a sufficient solution to the tragedy of the commons, because users have no direct incentive to restrict the cluster sizes of their jobs. Furthermore, even altruistic users might not have the information they need to make the best decisions.

One approach to this problem is a *system-centric* scheduler that chooses cluster sizes automatically, trying to optimize (usually heuristically) a system-wide performance metric like utilization or average turnaround time. We present several problems with this approach, and suggest an alternative, *job-centric* scheduling, in which users (or a system agent acting on their behalf) make scheduling decisions on a job-by-job basis in order to satisfy user-specified goals.

In order to make these decisions, users need to be able to predict the queue time until a given cluster size is available, and the run time of the job as a function of its cluster size. Toward this end we have developed statistical techniques for predicting queue times on space-sharing parallel computers, and a model for using historical information to predict the run times of parallel jobs. The goal of this paper is to evaluate the usefulness of these predictions for processor allocation.

1.1 Adaptive Jobs

Feitelson and Rudolph [8] propose the following classification of adaptive parallel jobs: *rigid* jobs can run only on a fixed cluster size; *moldable* jobs can be configured to run on a range of cluster sizes, but once they begin execution, they cannot change cluster size. *Evolving* jobs change cluster size as they execute; these changes are initiated by the job, and usually correspond to program phases. *Malleable* jobs can also change size dynamically, but unlike evolving jobs, they can respond to system-initiated reconfiguration requests.

Many of the SPMD parallel languages used for scientific computing generate static jobs (rigid or moldable), but few generate dynamic jobs (evolving or malleable). Thus, workloads in current supercomputing environments are made up almost entirely of static jobs. There is some evidence that the fraction of moldable jobs is significant [6], and it is likely to increase as users shift to higher-level programming models. This paper addresses scheduling strategies for moldable jobs.

1.2 System-centric Scheduling

Many simulation and analytic studies have examined the performance of system-centric allocation strategies, that is, strategies designed to maximize an aggregate performance metric without regard for individual jobs. In most cases, this metric is average turnaround time [10] [17] [16] [9] [1] [15], although some studies also consider throughput [13]. Rosti, Smirni et al. use *power*, which is the ratio of throughput to mean response time [14]. In prior work we used a parallel extension of *slowdown*, which is the ratio of the actual response time of a job to the time it would have taken on a dedicated machine [3]. Feitelson and Rudolph use a different formulation of slowdown [7].

There are several common problems with system-centric schedulers:

Starvation: For many system-centric schedulers there is an identifiable class of jobs that receives unacceptable service. For example, utilization-maximizing schedulers tend to starve jobs with low parallel efficiency and odd-sized jobs that cause fragmentation. Turnaround-minimizing schedulers tend to starve large jobs. Although it may be desirable to give different quality of service to different classes of jobs, it is not acceptable for a real system to allow jobs to starve.

One metric fits all: Another problem is that system-centric schedulers are usually based on a single performance metric. In real systems there are often classes of jobs with different performance requirements. For example, users sometimes submit short batch jobs and wait for the results; for these jobs, turnaround time is critical. On the other hand, many users submit jobs before lunch or before leaving for the day and have no interest in turnaround time; in this case the performance goal is to complete before a given deadline. Other scheduling goals include minimizing the completion time of a set of jobs and minimizing the accounting cost of a job.

Perverse incentives: System-centric schedulers often force users to accept decisions that are good for the system as a whole, but contrary to their immediate interests. For example, if there is a job in queue and one idle processor, a utilization-maximizing system might require the job to run, whereas the job might obtain a shorter turnaround time by waiting for more processors. If such strategies are implemented, users will be unsatisfied with the system, and some of them will take steps to subvert it. Since these systems often rely on job information provided by users, it is not hard for a disgruntled user to manipulate the system for his own benefit. In anecdotal reports from supercomputer centers, this sort of behavior is common, and not restricted to malevolent users; rather, it is understood that users will take advantage of loopholes in system policies.

In summary, system-centric schedulers often provide unacceptable service for some jobs, force other jobs to pay for quality of service they do not require, and create incentives for users to subvert the system.

1.3 Job-centric Scheduling

A possible solution to these problems is *job-centric* scheduling, in which the user (or a system agent acting on the user's behalf) makes scheduling decisions on a job-by-job basis in order to satisfy user-specified goals. Some examples are:

1. A user might minimize the cost of a calculation by choosing the smallest cluster size that allows the job to fit in memory.
2. A user might choose a cluster size that yields an acceptable probability that the job will complete before a deadline.
3. A user might choose the cluster size that minimizes the turnaround time of a job (the sum of its queue time and run time).

The strategy we propose in this paper tries to minimize the turnaround time of each job (the third example), but the techniques we develop can be extended to address the other two goals. Thus, job-centric scheduling can solve the one-metric problem.

Also, since the scheduling strategy we propose is based on a FIFO queue, it has no problems with starvation. Compared with some system-centric schedulers, it tends to improve the performance of large, highly-parallel jobs at the expense of smaller jobs, but in supercomputing environments this discrimination is acceptable, if not desirable.

Finally, because job-centric scheduling makes decisions on behalf of individual jobs, it does not create incentives for users to subvert its decisions. As a result this strategy is robust in the presence of self-interested users.

1.4 Why FIFO?

In order to minimize the average turnaround time of a set of jobs, it is optimal to schedule the shortest job first. In supercomputing environments, though, the system seldom knows the run times of jobs *a priori*. Nevertheless, the system can often use information about queued jobs (executable names, user names, queue names) to identify and give priority to short jobs. Such non-FIFO strategies have been shown to improve overall system performance [9][1].

One problem with such strategies is that they tend to starve large jobs; that is, jobs that request a large cluster size might wait in queue indefinitely while smaller jobs run. This problem is not hypothetical, but has been observed frequently at supercomputing sites like the San Diego Supercomputer Center. This situation is particularly problematic because supercomputer centers have a mandate to run large, highly-parallel jobs that cannot run anywhere else. Thus, these sites have been forced to adopt ad hoc measures to expedite large jobs. In some cases, an operator has to override the system's scheduler to rescue starving jobs. It is clear that this is not an appropriate long-term solution.

Another problem with non-FIFO strategies is that they make the system less predictable. Predictability is a useful property because it allows users to decide what jobs (and what problem sizes) to run, when to run them and, in a distributed system, where to run them.

Because of these problems, we have chosen to focus on FIFO queueing strategies. However, there is a natural extension of FIFO scheduling, called backfilling, that has the potential to increase system performance without causing starvation. In a FIFO system, when a large job is waiting at the head of the queue, there may be smaller jobs in queue that could run. Backfilling is the process of allowing these jobs to run, on the condition that they not delay the large job. The EASY scheduler uses this strategy [18] for rigid jobs. In future work we plan to add backfilling to our strategy for moldable jobs.

1.5 Outline

Section 2 presents speedup model we use in Sect. 3 to develop an abstract workload model. This workload is based on observations from the San Diego Supercomputer Center and the Cornell Theory Center. Section 4 describes the statistical techniques we use to predict queue times. Section 5 describes the simulator we use to evaluate various allocation strategies. Section 6 presents our evaluation of these strategies from the job's point of view, and Sect. 7 discusses the effect these strategies have on the system as a whole.

2 Job Model

In order to evaluate the proposed allocation strategies, we will use a simulation based on an *abstract workload model.* On existing systems, we often collect statistics about actual (concrete) workloads; for example, we might know the duration and cluster size of each job. The workloads we observe are the result of interactions between the job mix, the properties of the hardware, and the behavior of the allocation strategy. Thus, it may not be correct to use a concrete workload from one system to simulate and evaluate another. Our goal is to create an abstract workload that separates the characteristics of the job mix from the effect of the system.

Previously [2], we proposed a model of moldable jobs that characterizes each job by three parameters: L, the sequential lifetime of the job, A, the average parallelism, and σ, which measures the job's variance in parallelism. Using this model we can calculate the speedup and run time of a job on any number of processors. This section summarizes our *job model.*

Once we have a model of individual jobs, we can construct a *workload model* that describes the system load, the arrival process, and the distribution of job parameters. Section 3 presents this *abstract workload model.*

2.1 A Model of Moldable Jobs

Our model of parallel speedup is based on a family of curves parameterized by average parallelism, A, and variance in parallelism, V. For given values of these parameters, we construct a hypothetical *parallelism profile*[1] with those values, and use the profile to derive a speedup curve. We use two families of profiles, one for programs with low V, the other for programs with high V. In previous work we showed that this family of speedup profiles captures, at least approximately, the behavior of a variety of parallel scientific applications on a variety of architectures [2].

2.2 Low-variance Model, $\sigma \leq 1$

Figure 1a shows a hypothetical parallelism profile for a program with low variance in parallelism. The degree of parallelism is A for all but some fraction σ of the duration ($0 \leq \sigma \leq 1$). The remaining time is divided between a sequential component and a high-parallelism component. The average parallelism of this profile is A; the variance is $V = \sigma(A-1)^2$.

A program with this profile would have the following speedup as a function of cluster size:

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

[1] Sevcik defines the parallelism profile as the distribution of potential parallelism during the execution of a program[17].

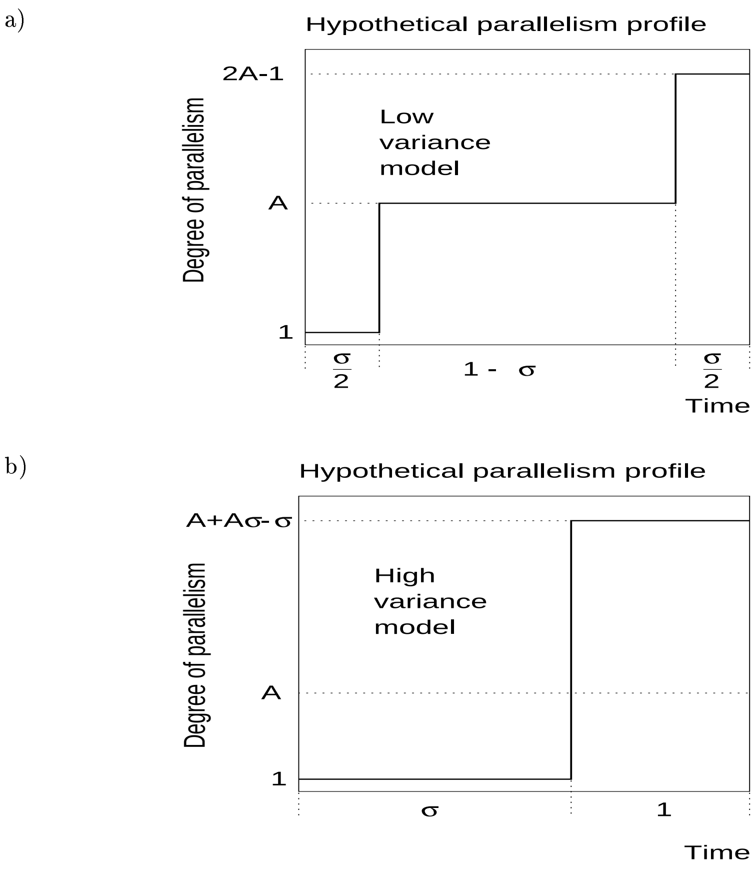


Fig. 1. The hypothetical parallelism profiles we use to derive our speedup model.

2.3 High-variance Model, $\sigma \geq 1$

In the low variance model, σ cannot exceed 1, and thus the variance cannot exceed $V = (A-1)^2$. In this section, we propose an extended model in which σ can exceed 1 and the variance is unbounded. The two models can be combined naturally because (1) when the parameter $\sigma = 1$, the two models are identical, and (2) for both models the variance is $\sigma(A-1)^2$.

From the latter property we derive the semantic content of the parameter σ – it is approximately the square of the coefficient of variation of parallelism, CV^2. This approximation follows from the definition of coefficient of variation, $CV = \sqrt{V}/A$. Thus, CV^2 is $\sigma(A-1)^2/A^2$, which for large A is approximately σ.

Figure 1b shows a hypothetical parallelism profile for a program with high variance in parallelism. A program with this profile would have the following speedup as a function of cluster size:

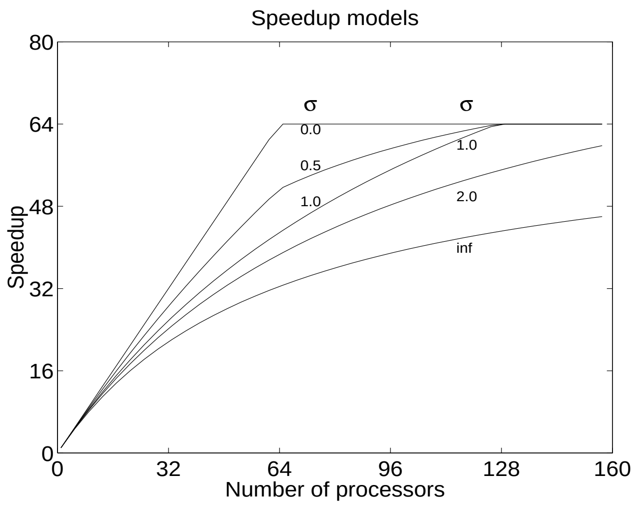


Fig. 2. Speedup curves for a range of values of σ.

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

Figure 2 shows speedup curves for a range of values of σ (with $A = 64$). When $\sigma = 0$ the curve matches the theoretical upper bound for speedup – bound at first by the "hardware limit" (linear speedup) and then by the "software limit" (the average parallelism A). As σ approaches infinity, the curve approaches the theoretical lower bound on speedup derived by Eager et al. [5]: $S_{min}(n) = An/(A + n - 1)$.

Of course, for many jobs there will be ranges of n where this model is inapplicable. For example, a job with large memory requirements will run poorly (or not at all) when n is small. Also, when n is large, speedup may decrease as communication overhead overwhelms computational speedup. Finally, there are some applications that require cluster sizes with specific characteristics; e.g. powers of two and perfect squares. Thus we qualify our job model with the understanding that for each job there may be a limited range of viable cluster sizes.

3 Workload Model

3.1 Distribution of Lifetimes

Ideally, we would like to know the distribution of L, the *sequential lifetime*, for a real workload. Sequential lifetime is the time a job would take on a single processor, so if we knew L, A and σ, we could calculate the speedup, $S(n, A, \sigma)$,

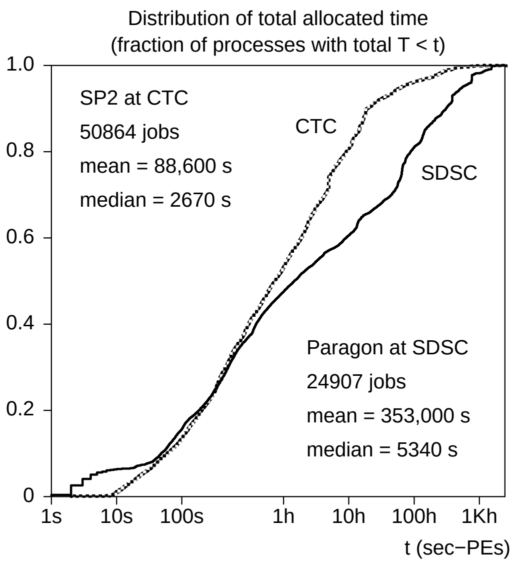


Fig. 3. Distribution of total allocated time for jobs at SDSC and CTC.

on n processors and the run time, L/S. But for most jobs we do not know L; often it is not even defined, because memory requirements prevent some jobs from running on a single processor. On the other hand, we do know the *total allocated time*, T, which is the product of wall clock lifetime and cluster size. For programs with linear speedup, T equals L, but for programs with sublinear speedups, T can be much larger than L.

Figure 3 shows the distribution of total allocated time for jobs from the Intel Paragon at SDSC and the IBM SP2 at CTC. On both machines, the distribution is approximately uniform (linear) in log space, or a *uniform-log distribution.* Thus the cumulative distribution function (cdf) of T has the form:

$$cdf_T(t) = Pr\{T \leq t\} = \beta_0 + \beta_1 \ln t \tag{3}$$

where $t_{min} \leq t \leq t_{max}$, and β_0 and β_1 are the intercept and slope of the observed line. The upper and lower bounds of this distribution are $t_{min} = e^{-\beta_0/\beta_1}$ and $t_{max} = e^{(1.0-\beta_0)/\beta_1}$.

We know of no theoretical reason that the distribution should have this shape, but we believe that it is pervasive among batch workloads, since we have observed similar distributions on the Cray C90 at SDSC, and other authors have reported similar distributions on other systems [6][20].

Depending on the allocation policy, the distribution of T could differ from the distribution of L. For all practical policies, though, the two distributions have the same shape, with different parameters. Thus, in our simulations, we assume that the distribution of L is uniform-log. For the scheduling policies we

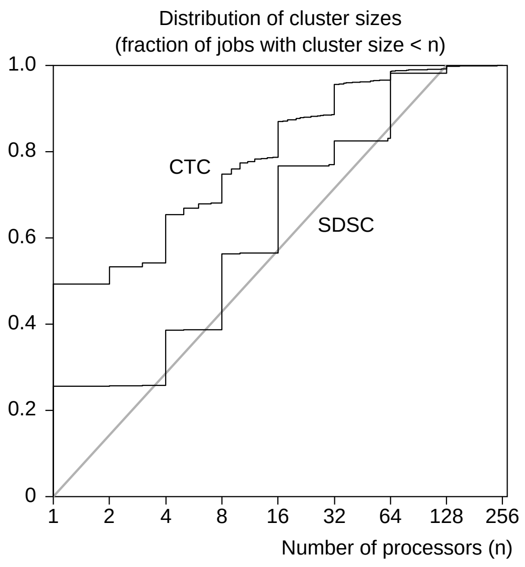


Fig. 4. Distribution of cluster sizes for jobs at SDSC and CTC.

consider, the resulting distribution of T is also uniform-log, excepting a few of the longest and shortest jobs.

In our simulations, L is distributed between e^2 and e^{12} seconds (approximately 7 seconds to 45 hours). The median of this distribution is 18 minutes; the mean is 271 minutes.

3.2 Distribution of Average Parallelism

For our workload model, we would like to know the parallelism profile of the jobs in the workload. But the parallelism profile reflects *potential* parallelism, as if there were an unbounded number of processors available, and in general it is not possible to derive this information by observing the execution of the program.

In the accounting data we have from SDSC and CTC, we do not have information about the average parallelism of jobs. On the other hand, we do know the cluster size the user chose for each job, and we hypothesize that these cluster sizes, in the aggregate, reflect the parallelism of the workload.

Figure 4 shows this distribution for the workloads from SDSC and CTC. In both cases, most jobs have cluster sizes that are powers of two. Neither the Intel Paragon nor the IBM SP2 require power-of-two cluster sizes, but in both cases the interface to the queueing system suggests powers of two and few users have an incentive to resist the combination of suggestion and habit. We believe that the step-wise pattern in the distribution of cluster sizes reflects this habit and not the true distribution of A. Thus for our workload model, we use a uniform-log distribution with parameters $A_{min} = 1$ and $A_{max} = N$, where N is the number of processors in the system. The gray line in the figure shows this model.

Our model fits the SDSC distribution well, but the CTC distribution contains significantly more sequential jobs than the model. This excess is likely due to the fact that the SP2 at CTC has more memory on each node than most workstations, and provides some software that is not available on workstations. Thus, many users submit sequential jobs to the SP2 that they would ordinarily run on workstations. Our workload does not model this behavior because it is not typical of supercomputer sites.

3.3 Distribution of Variance (σ)

In general there is no way to measure the variance in potential parallelism of existing codes explicitly. In previous work, we proposed a way to infer this value from observed speedup curves [2]. To test this technique, we collected speedup curves for a variety of scientific applications running on a variety of parallel computers. We found that the parameter σ, which approximates the coefficient of variance of parallelism, was typically in the range 0–2, with occasional higher values.

Although these observations provide a range of values for σ, they do not tell us its distribution in a real workload. For this study, we use a uniform distribution between 0 and 2.

4 Predicting Queue Times

In previous work we presented statistical techniques for predicting the remaining queue time for a job at the head of the queue [4]. Since we use these predictions in Sect. 6.3, we summarize the techniques here.

We describe the state of the machine at the time of an arrival as follows: there are p jobs running, with ages a_i and cluster sizes n_i (in other words, the ith job has been running on n_i processors for a_i seconds). We would like to predict $Q(n')$, the time until n' additional processors become available, where $n' = n - n_{free}$, n is the number of processors requested, and n_{free} is the number of processors already available. In the next two sections we present ways to estimate the median and mean of $Q(n')$.

4.1 Median Predictor

We can calculate the median of $Q(n')$ exactly by enumerating all possible outcomes (which jobs complete and which are still running), and calculating the probability that the request will be satisfied before a given time t. Then we set this probability to 0.5 and solve for the median queue time. This approach is not feasible when there are many jobs in the system, but it leads to an approximation that is fast to compute and almost as accurate.

We represent each outcome by a bit vector, b, where for each bit, $b_i = 0$ indicates that the ith job is still running, and $b_i = 1$ indicates that the ith job has completed before time t. Since we assume independence between jobs in the system, the probability of a given outcome is the product of the probabilities of each event (the completion or non-completion of a job). The probability of each event comes from the conditional distribution of lifetimes. For a uniform-log distribution of lifetimes, the conditional distribution $cdf_{L|a}$ is

$$\begin{aligned} 1 - cdf_{L|a}(t) &= Pr\{L > t | L > a\} \\ &= \frac{1 - cdf_L(t)}{1 - cdf_L(a)} \\ &= \frac{1 - \beta_0 - \beta_1 \ln t}{1 - \beta_0 - \beta_1 \ln a} \end{aligned} \tag{4}$$

where $t_{min} \leq a \leq t \leq t_{max}$. Thus, the probability of a given outcome is

$$Pr\{b\} = \prod_{i|b_i=0} cdf_{L|a_i}(t) \cdot \prod_{i|b_i=1} \left(1 - cdf_{L|a_i}(t)\right) \tag{5}$$

For a given outcome, the number of free processors is the sum of the processors freed by each job that completes:

$$F(b) = \sum_i b_i \cdot n_i \tag{6}$$

Thus at time t, the probability that the number of free processors is at least the requested cluster size is the sum of the probabilities of all the outcomes that satisfy the request:

$$Pr\{F \geq n'\} = \sum_{b|F(b) \geq n'} Pr\{b\} \tag{7}$$

Finally, we find the median value of $Q(n')$ by setting $Pr\{F > n'\} = 0.5$ and solving for t.

Of course, the number of possible outcomes (and thus the time for this calculation) increases exponentially with p, the number of running jobs. Thus this is not a feasible approach when there are many running jobs. But when the number of additional processors required (n') is small, it is often the case that there are several jobs running in the system that will single-handedly satisfy the request when they complete. In this case, the probability that the request will be satisfied by time t is dominated by the probability that one of these benefactors will complete before time t.

In other words, the chance that the queue time for n' processors will exceed time t is approximately equal to the probability that none of the benefactors will complete before t:

$$Pr\{F < n'\} \approx \prod_{i|n_i \geq n'} 1 - cdf_{L|a_i}(t) \tag{8}$$

The running time of this calculation is linear in p. Of course, it is only approximately correct, since it ignores the possibility that several small jobs might complete and satisfy the request. Thus, we expect this predictor to be inaccurate when there are many small jobs running in the system, few of which can single-handedly handle the request. The next section presents an alternative predictor that we expect to be more accurate in this case.

4.2 Mean Predictor

When a job is running, we know that at some time in the future it will complete and free all of its processors. Given the age of the job, we can use the conditional distribution (4) to calculate the probability that it will have completed before time t.

We approximate this behavior by a model in which processors are a continuous (rather than discrete) resource that jobs release gradually as they execute. In this case, we imagine that the conditional cumulative distribution indicates what *fraction* of a job's processors will be available at time t.

For example, a job that has been running for 30 minutes might have a 50% chance of completing in the next hour, releasing all of its processors. As an approximation of this behavior, we predict that the job will (deterministically) release 50% of its processors within the next hour.

Thus we predict that the number of free processors at time t will be the sum of the processors released by each job:

$$F = \sum_i n_i \cdot cdf_{L|a_i}(t) \tag{9}$$

To estimate the mean queue time we set $F = n'$ and solve for t.

4.3 Combining the Predictors

Since we expect the two predictors to do well under different circumstances, it is natural to use each when we expect it to be most accurate. In general, we expect the Median Predictor to do well when there are many jobs in the system that can single-handedly satisfy the request (benefactors). When there are few benefactors, we expect the Mean Predictor to be better (especially since, if there are none, we cannot calculate the Median Predictor at all). Thus, in our simulations, we use the Median Predictor when the number of benefactors is 2 or more, and the Mean Predictor otherwise. The particular value of this threshold does not affect the accuracy of the combined predictor drastically.

5 Simulations

To evaluate the benefit of using predicted queue times for processor allocation, we use the models in the previous section to generate workloads, and use a simulator to construct schedules for each workload according to the proposed allocation strategies. We compare these schedules according to several performance metrics.

Our simulations try to capture the daily work cycle that has been observed in several supercomputing environments (the Intel iPSC/860 at NASA Ames and the Paragon at SDSC [6] [20]):

– In early morning there are few arrivals, utilization is at its lowest, and queue lengths are short.
– During the day, the arrival rate increases and jobs accumulate in queue. Utilization is highest late in the day.
– In the evening, the arrival rate falls but the utilization stays high as the jobs in queue begin execution.

To model these variations, we divide each simulated day into two 12-hour phases: during the daytime, jobs arrive according to a Poisson process and either begin execution or join the queue, depending on the state of the system. During the night, no new jobs arrive, but the existing jobs continue to run until all queued jobs have been scheduled.

We choose the day-time arrival rate in order to achieve a specified offered load, ρ. We define the offered load as the total sequential load divided by the processing capacity of the system: $\rho = \lambda \cdot E[L]/N$, where λ is the arrival rate (in jobs per second), $E[L]$ is the average sequential lifetime (271 minutes in our simulations), and N is the number of processors in the system (128 in our simulations). The number of jobs per day is between 160 (when $\rho = 0.5$) and 320 (when $\rho = 1.0$).

6 Results: Job Point-of-view

In this section, we simulate a commonly-proposed, system-centric scheduling strategy and show that this strategy often makes decisions that are contrary to the interests of users. We examine how users might subvert such a system, and measure the potential benefit of doing so.

Our baseline strategy is AVG, which assigns free processors to queued jobs in FIFO order, giving each job no more than A processors, where A is the average parallelism of the job. Several studies have shown that this strategy performs well for a range of workloads [17] [9] [12] [19] [1] [3].

The problem with this strategy is that it forces users to accept decisions that are contrary to their interests. For example, if there is a large job at the head of the queue, it will be forced to run on any available cluster, even a single processor. From the system's point of view, this decision is expected to yield high utilization; from the job's point of view, though, it would be better to wait for a larger cluster.

To see how often this situation arises, we ran 120 simulated days with the AVG strategy and an offered load, ρ, of 0.75 (30421 jobs). Most jobs (63%) are allocated the maximum cluster size, A processors. So there is no reason for users to intervene on behalf of these jobs.

For each of the remaining jobs, we used oracular prediction to find the optimal cluster size. In other words, we found the value of n that minimized the turnaround time $Q(n) + R(n)$, where $Q(n)$ is the queue time until n processors are available, and $R(n)$ is the run time of the job on n processors. As under AVG, n can be no greater than A. We call this strategy OPT.

For the jobs allocated fewer than A processors, most of the time (62%) the best thing is to accept the decision of the system and begin running immediately. Only 38% of these jobs (14% of all jobs) would benefit by waiting for a larger cluster.

But for those jobs, which we call *rebels*, the benefit can be substantial. For each rebel, we calculated the *time savings*, which is the difference between the job's turnaround time on the system-chosen cluster, and the turnaround time it would have on the optimal cluster size. The median savings per rebel is 15 minutes (the median duration of all jobs is only 3 minutes). The average time savings is 1.6 hours (the average duration of all jobs is 1.3 hours). Thus, although most jobs are well-served by system-centric scheduling, many jobs can significantly improve their performance by subverting the system.

In the following sections, we will consider several strategies users might employ to subvert a system-centric scheduler and improve the performance of their jobs. These strategies are based on the assumption that users have the ability to impose minimum cluster sizes on their jobs. It is probably necessary for a real system to provide such a mechanism, because many jobs cannot run on small clusters due to memory constraints.

The metric we use to compare these strategies is *time savings per job*: the total time savings (for all rebels) divided by the number of jobs (including non-rebels). This metric is more meaningful than time savings per rebel – according to the latter metric, it is optimal to choose only one rebel with the largest time savings. The strategy that users would choose is the one that maximizes time savings per job. Under OPT, the average time savings per job is 13.8 minutes.

6.1 STUB: Stubborn Self-interest

Previously [3], we evaluated a simple strategy, STUB, in which users impose a minimum cluster size on their jobs of fA, where f is some fraction between 0 and 1. We believe that this strategy models user behavior in existing supercomputing environments: users choose a fixed cluster size for their jobs that is roughly proportional to the job's available parallelism, but unrelated to current system load. For values of f greater than 0.5, the performance of this strategy degrades drastically.

For this paper, we examine this strategy from the point of view of individual jobs. Testing a range of values of f, we find that the time savings per job peaks at 8.8 minutes, with $f = 0.4$. Under this strategy, 37% of the jobs rebel, of whom 34% end up worse off – their turnaround times would have been shorter if they had not waited. Nevertheless, from the point of view of individual users, STUB is an acceptable if not optimal strategy. Most jobs that hold out for more processors improve their turnaround times by doing so, and the average time savings are significant.

6.2 HEUR: Using Job Characteristics

Using additional information about jobs, we expect that we can identify more successfully the jobs that will benefit by rebelling. Assuming that users know, approximately, the run times of their jobs, we construct a heuristic policy, HEUR, that allows only long jobs with high parallelism to rebel.

Specifically, any job with sequential lifetime greater than L_{thresh} and parallelism greater than A_{thresh} will wait for at least some fraction, f, of its maximum cluster size, A. The parameters A_{thresh}, L_{thresh}, and f must be tuned according to system and workload characteristics.

If we know the run times of jobs, we can make a further improvement to this strategy, which is to cut the losses of a rebel that is waiting too long in queue. To do this, we calculate its potential time savings, $t_{save} = R(n_{free}) - R(fA)$, where n_{free} is the number of free processors, fA is the number of processors the job is waiting for, and $R(n)$ is the job's run time on n processors.

Based on the time savings, we calculate a trial period the rebel is willing to wait, kt_{save}, where k is a free parameter. If this period elapses before the rebel begins execution, the rebel runs on the available processors.

Searching the space of feasible parameters, we find that the following values are best: $L_{thresh} = 0$, $A_{thresh} = 1$, $f = 1.0$ and $k = 0.2$. Thus, contrary to our intuition, job characteristics are not useful for choosing which jobs should rebel; rather, they are most useful for deciding how long a rebel should wait before giving up.

Using these parameters, the average time savings per job is 12.8 minutes, which is 45% higher than under STUB. As under STUB, 37% of the jobs rebel, but twice as many of them (68%) end up worse off. Although the majority of rebels suffer, the overall performance of HEUR is good because the losers lose small and the winners win big. Thus, self-interested users might adopt this strategy, if they are not too averse to risk.

6.3 PRED: Using Predicted Queue Times

In this section we evaluate a strategy, called PRED, that uses the queue time predictors described in Sect. 4. PRED chooses an optimal cluster size for each job in the same way as OPT, except that instead of using deterministic queue times, PRED estimates $Q(n)$ based on the current state of the system.

Under PRED, a rebel may reconsider its decision after some time and, based on a new set of predictions, decide to start running. Because recomputing predictions incurs overhead, it is not clear how often jobs should be prompted to reconsider. In our system, jobs reconsider whenever a job completes or a new job arrives in queue, and whenever the predicted queue time elapses.

The average time savings per job under PRED is 12.8 minutes. Thus, from the point of view of individual jobs, PRED is no better than HEUR. The difference is that PRED is more deft in its selection of rebels. Only 8% of all jobs rebel, and the vast majority of them end up with shorter turnaround times (92%). For the losers the time lost is small (3.7 minutes on average), but for the majority, the benefit is substantial (the median time savings is 55 minutes; the average is 2.7 hours). Thus, risk-averse users would prefer PRED over HEUR.

Another advantage of PRED over HEUR is that it has no free parameters. In a real system, it may be difficult to tune HEUR's four parameters; their values will depend on both system and workload characteristics.

6.4 BIAS: Bias-corrected Prediction

Each time a simulated job uses a prediction to make an allocation decision, we record the prediction and the outcome. Figure 5a shows a scatterplot of these predicted and actual queue times. We measure the quality of the predictions by two metrics, accuracy and bias. Accuracy is the tendency of the predictions and outcomes to be correlated; the coefficient of correlation (CC) of the values in Fig. 5a is 0.48 (calculated under a logarithmic transformation).

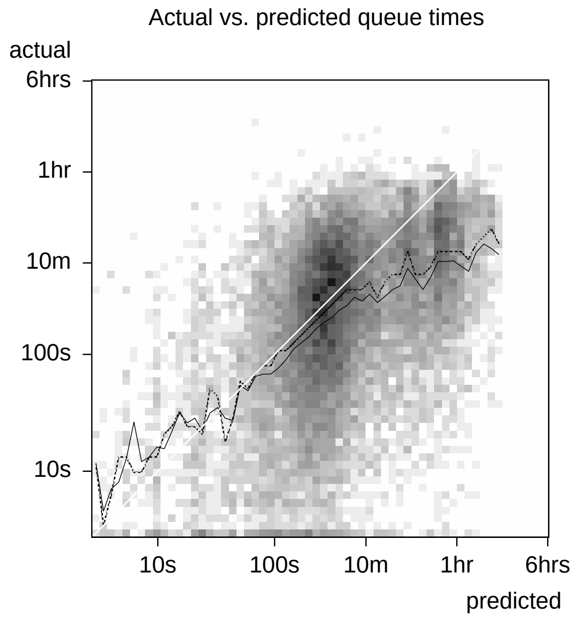
Bias is the tendency of the predictions to be consistently too high or too low. The lines in the figure, which track the mean and median of each column, show that short predictions (under ten minutes) are unbiased, but that longer predictions have a strong tendency to be too high. We can quantify this bias by fitting a least-squares line to the scatterplot. For a perfect predictor, the slope of this line would be 1 and the intercept 0; for our predictors the slope is 0.6 and the intercept 1.7.

Fortunately, if we know that a predictor is biased, we can use previous predictions to estimate the parameters of the bias, and apply a corrective transformation to the calculated values. In this case, we estimate the intercept (β_0) and slope (β_1) of the trend line, and apply the transformation $q_{corr} = q * \beta_1 + \beta_0$, where q is the calculated prediction and q_{corr} is the bias-corrected prediction. Figure 5b shows the effect of running the simulator again using this transformation. The slope of the new trend line is 1.01 and the intercept is -0.01, indicating that we have almost completely eliminated the bias.

Although we expected to be able to correct bias, we did not expect this transformation to improve the accuracy of the predictions; the coefficient of correlation should be invariant under an affine transformation. Surprisingly, bias correction raises CC from 0.48 to 0.59. This effect is possible because past predictions influence system state, which influences future predictions; thus the two scatterplots do not represent the same set of predictions. But we do not know why unbiased predictions in the past lead to more accurate predictions in the future.

The improvement in bias and accuracy is reflected in greater time savings. Under BIAS (PRED with bias-corrected prediction) the average time savings per job increases from 12.8 minutes to 13.5 minutes, within 3% of optimal. In practice, the disadvantage of BIAS is that it requires us to record the result of past predictions and estimate the parameters β_0 and β_1 dynamically.

a) Raw predictors



b) Predictors with bias correction

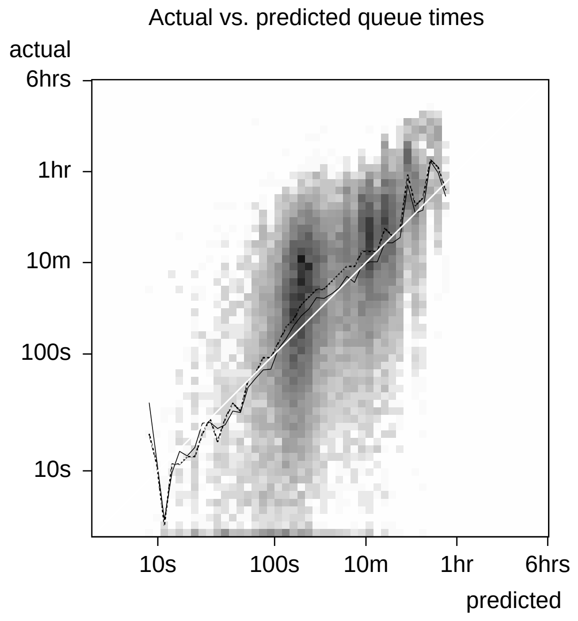


Fig. 5. Scatterplot of predicted and actual queue times (log scale). The white lines show the identity function; i.e. a perfect predictor. The solid lines show the average of the actual queue times in each column; the broken lines show the median.

6.5 Summary of Allocation Strategies

Table 1. Comparison of the strategies (job point-of-view).

	Information used	Average time savings per job (minutes)	Fraction of jobs that rebel	Average time savings per rebel (minutes)	Fraction of rebels that lose
STUB	A	8.8	37%	23.8	34%
HEUR	A,$R(n)$	12.8	37%	34.6	68%
PRED	A,$R(n)$,$E[Q(n)]$	12.8	8%	160	8%
BIAS	A,$R(n)$,$E[Q(n)]$,β_0,β_1	13.5	10%	135	8%
OPT	A,$R(n)$,$Q(n)$	13.8	14%	98.6	0%

Table 1 summarizes the performance of the various allocation strategies. Not surprisingly, the strategies that use more information generally yield better performance.

PRED and BIAS are more conservative than OPT; that is, they choose fewer rebellious jobs. PRED's conservativism is clearly a consequence of the tendency of our predictions to be too long. By overestimating queue times, we discourage jobs from rebelling. But it is not as clear why BIAS, which does not overestimate, is more conservative than OPT. In any case, both prediction-based strategies do a good job of selecting successful rebels; only 8% of rebels ended up spending more time in queue than they save in run time.

7 Results: System Point-of-view

Until now, we have been considering the effect of allocation strategies on individual jobs. Thus in our simulations we have not allowed jobs to effect their allocation decisions; we have only measured what would happen if they had. Furthermore, when we tuned these strategies, we chose parameters that were best for individual jobs.

In this section we modify our simulations to implement the proposed strategies and evaluate their effect on the performance of the system as a whole. We use two metrics of system performance: average turnaround time and utilization. We define utilization as the average of efficiency over time and processors, where efficiency is the ratio of speedup to cluster size, $S(n)/n$. The efficiency of an idle processor is defined to be 0. In our simulations, we can calculate efficiencies because we know the speedup curves for each job. In real systems this information is not usually available.

Table 2. Performance from the system's point-of-view.

	Average utilization (120 days)	Average turnaround time in minutes (30421 jobs)
AVG	.557	79.9
STUB	.523	113
HEUR	.526	109
PRED	.570	77.5
BIAS	.561	84.1

Table 2 shows the results for each allocation strategy, using the same workload as in the previous section. In the presence of self-interested users, the performance of AVG degrades severely. If users choose cluster sizes naively (STUB) the utilization of the system drops by 6% and turnaround times increase by 41%. The situation is only slightly better if users take steps to reduce long delays (HEUR).

PRED performs slightly better than AVG, which performs slightly better than BIAS. It may seem odd that PRED does better than BIAS, since BIAS is based on more accurate predictions. The reason is that PRED's predictions are consistently conservative, which has the effect of discouraging some borderline rebels. This conservativism reduces queue times and increases utilization. In practice, though, users might eventually notice that predicted queue times are too high and apply bias correction on their own behalf. Thus, in the presence of self-interested users, we expect PRED to yield performance similar to BIAS. Fortunately, this degradation is not nearly as severe as under AVG; the utilization of the system drops slightly (1.6%) and turnaround times increase by 8.5%.

One surprising result is that the predictive strategies yield higher utilization than AVG. Because these strategies often leave processors idle (which decreases utilization) and allocate larger clusters (which decreases efficiency), we expected these strategies to *decrease* overall utilization.

The reason they do not is that these strategies are better able to avoid L-shaped schedules. Figure 6 shows two schedules for the same pair of jobs. Under AVG, the second arrival would be forced to run immediately on the small cluster, which improves utilization in the short term by reducing the number of idle processors. But after the first job quits, many processors are left idle until the next arrival. Our predictive strategies allow the second job to wait for a larger cluster, which not only reduces the turnaround time of the second job; it also increases the average utilization of the system.

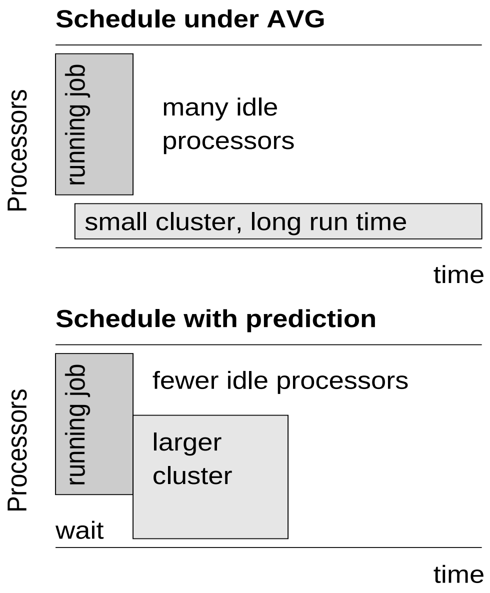


Fig. 6. Sample schedules showing how longer queue times and larger cluster sizes can, paradoxically, improve system utilization. Queue time prediction makes it possible to avoid L-shaped schedules and thereby reduce the number of idle processors.

7.1 Job-centric vs. System-centric

What, then, is the performance advantage of job-centric scheduling over system-centric scheduling? It depends on how aggressively users subvert the system. If users are docile, and do not interfere with the system, the difference is small: PRED saves about 3%, or 140 seconds per job, over AVG (95% confidence interval 1.7% to 4.4%).

But in the presence of self-interested users, the difference is much larger: compared to HEUR, BIAS saves 30%, or almost half an hour per job (95% confidence interval 29.1% to 30.6%).

8 Conclusions

We have proposed a job-centric allocation policy with the following properties:

– Because it is based on a FIFO queueing system, jobs never starve.
– Because it makes decisions on behalf of individual jobs, it does not create incentives for users to subvert the system. As a result, we show that it is robust in the presence of self-interested users.
– The overall performance of the system under this strategy is between 3% and 30% better than under a comparable system-centric policy.

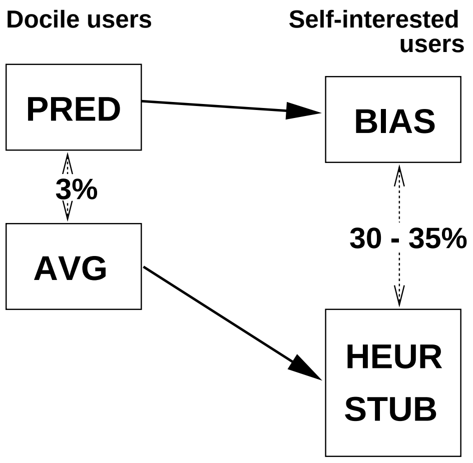


Fig. 7. The performance of many scheduling strategies, like AVG, degrades in the presence of self-interested users. The performance of our job-centric scheduler, PRED, does not degrade as severely.

Also, we show that the prediction techniques we propose are sufficiently accurate for making allocation decisions. From the point of view of individual jobs, our predictive strategy is within 3% of an optimal strategy (with perfect prediction).

8.1 Future Work

In this paper we have considered a single system size (128 processors), distribution of job characteristics (see Sect. 3), and load ($\rho = 0.75$). We would like to evaluate the effect of each of these parameters on our results.

Also, we have modeled an environment in which users provide no information to the system about the run times of their jobs. As a result, our queue time predictions are not very accurate. In the real systems we have examined, the information provided by users significantly improves the quality of the predictions [4]. We would like to investigate the effect of this improvement on our results.

As part of the DOCT project [11] we are in the process of implementing system agents that provide predicted queue times on space-sharing parallel machines. Users can take advantage of this information to choose what jobs to run, when to run them, and how many processors to allocate for each. We expect that this information will improve user satisfaction with these systems, and hope that, as in our simulations, it will lead to improvement in the overall performance of the system.

Acknowledgements

I would like to thank George Kremenek at SDSC and Steven Hotovy at CTC for providing workload data; Reagan Moore at SDSC, Jenny Schopf and Rich

Wolski at the University of California at San Diego, Ken Sevcik at the University of Toronto and the workshop reviewers for comments that greatly improved the quality and clarity of this paper.

References

1. Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1994.
2. Allen B. Downey. A model for speedup of parallel programs. Technical Report CSD-97-933, University of California at Berkeley, 1997.
3. Allen B. Downey. A parallel workload model and its implications for processor allocation. In *The Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, 1997. To appear. Also available as University of California technical report number CSD-96-922.
4. Allen B. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.
5. Derek L. Eager, John Zahorjan, and Edward L. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.
6. Dror G. 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, Springer-Verlag LNCS Vol 949*, pages 337–360, April 1995.
7. Dror G. Feitelson and Larry Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35:18–34, 1996.
8. Dror G. Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 1162*, pages 1–26, April 1996.
9. Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.
10. Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 104–113, 1988.
11. Reagan Moore and Richard Klobuchar. DOCT (distributed-object computation testbed) home page `http://www.sdsc.edu/doct`. San Diego Supercomputer Center, 1996.
12. Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Supercomputing '93 Conference Proceedings*, pages 824–833, March 1993.
13. Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 57–67, May 1996.
14. Emilia Rosti, Evgenia Smirni, Lawrence W. Dowdy, Giuseppe Serazzi, and Brian M. Carlson. Robust partitioning policies of multiprocessor systems. *Performance Evaluation*, 19(2-3):141–165, Mar 1994.

15. Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 949*, pages 165–181, April 1995.
16. Sanjeev K. Setia and Satish K. Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the Internationsal Workshop on Modeling and Simulation of Computer and Telecommunications Systems (MASCOTS)*, January 1993.
17. Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, 17(1):171–180, May 1989.
18. Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY – LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 1162*, pages 41–47, April 1996.
19. Evgenia Smirni, Emilia Rosti, Lawrence W. Dowdy, and Giuseppe Serazzi. Evaluation of multiprocessor allocation policies. Technical report, Vanderbilt University, 1993.
20. Kurt Windisch, Virginia Lo, Dror Feitelson, Bill Nitzberg, and Reagan Moore. A comparison of workload traces from two production parallel machines. In *6th Symposium on the Frontiers of Massively Parallel Computation*, 1996.

This article was processed using the LaTeX macro package with LLNCS style