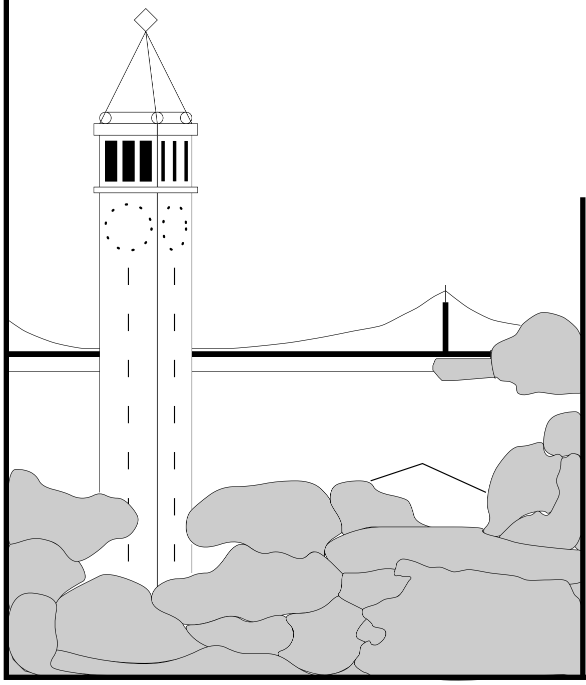
Using Queue Time Predictions for Processor Allocation

Allen B. Downey



Report No. UCB/CSD-97-929

January 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Using Queue Time Predictions for Processor Allocation

Allen B. Downey *

January 1997

Abstract

When a malleable job is submitted to a space-sharing parallel computer, it must choose often 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 the environment we have observed at the San Diego Supercomputer Center, and use this model to drive simulations of various allocation strategies. We conclude that prediction-based allocation not only improves the average turnaround time for the jobs; it also improves the utilization of the system as a whole.

Keywords: parallel, space-sharing, partitioning, scheduling, allocation, malleable, multiprocessor, prediction, queue.

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 runtimes for their jobs might allocate more processors than they can use efficiently. This overindulgence lowers the effective use of the system and increases the queue times of all users' jobs.

A partial solution to this problem is a programming model that supports *malleable* jobs, *i.e.*, jobs that can be configured to run on clusters of various sizes. In existing systems, these jobs cannot change their cluster sizes dynamically; that is, once they begin execution, they cannot add or yield processors.

Malleable 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 malleable 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 solution 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

*EECS — Computer Science Division, University of California, Berkeley, CA 94720 and San Diego Supercomputer Center, P.O. Box 85608, San Diego, CA 92186. Supported by NSF grant ASC-89-02825 and by Advanced Research Projects Agency/ITO, Distributed Object Computation Testbed, ARPA order No. D570, Issued by ESC/ENS under contract #F19628-96-C-0020. email: downey@sdsc.edu, http://www.sdsc.edu/~ downey

or average turnaround time. The problem is that such systems 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, there are two possible outcomes: at best, users will be unsatisfied with the system; at worst, they will take steps to subvert it. Since these systems often rely on application 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 user behavior is common, and not restricted to malevolent users; rather, it is an understanding in these environments that users will take advantage of loopholes in system policies.

Given that this is true, an important property for a scheduling strategy is robustness in the presence of self-interested users. As we will show in Section 5, many commonly-proposed allocation strategies do not have this property; their overall performance degrades severely if users try, naively, to improve the performance of individual jobs.

Thus our goal is to find a scheduling strategy that does not make decisions that are contrary to the interests of the users. We propose an *application-centric* scheduler that uses application information (run times on various cluster sizes) and system state (predicted queue times) to choose the cluster size with the shortest expected turnaround time for each job.

This strategy optimizes the performance of individual jobs, so users have no incentive to subvert its decisions. The question, though, is what effect these local optimizations will have on the performance of the system as a whole. Using simulations based on a stochastic workload model, we show that the performance of one such strategy *exceeds* that of the best system-centric scheduler, improving both system utilization and average turnaround time.

1.1 Queueing strategy

A scheduling strategy consists of a *queueing strategy*, which chooses which job in queue begins execution, and an *allocation strategy*, which chooses how many processors are allocated to each job.

In this paper, we will be using only first-come-first-served (FCFS) queueing strategies. Thus, once a job arrives at the head of the queue, its remaining queue time does not depend on the other jobs in queue or on future arrivals. Furthermore, if the job at the head of the queue decides to hold out — to leave processors idle until a larger cluster is available — the other jobs in queue are not permitted to pass it by.

In real systems, there are often several queues with different priorities. The queueing strategy within each queue is FCFS, but jobs in different queues do not necessarily run in the order they arrive. The techniques we present here can be extended to model this environment, although we expect it to be more difficult to predict the behavior of low-priority queues, since they are affected by higher-priority arrivals.

Some prior studies have proposed more general non-FCFS queueing strategies [11][4]. These strategies have the potential to reduce turnaround times by identifying short jobs (one way or another) and giving them priority. Furthermore, a non-FCFS strategy would discourage jobs from holding out for more processors, since a stubborn job would risk starvation. Thus the immediate effect might be to reduce the number of idle processors. Despite these advantages, it is not clear that non-FCFS strategies will improve system performance. If jobs in queue are forced to compete for idle processors, we expect the result to be similar to the ASP strategy we examined in [5], in which idle processors are divided evenly among the jobs in queue. In that study, we found that

ASP was significantly worse than the FCFS strategies. FCFS strategies have one other advantage, which is that they are more predictable. This property has a direct impact on user satisfaction, and also lends itself to metacomputing environments in which users (or system agents) select among various resources according to the predicted time until they are available (among other things).

Thus, although we consider non-FCFS queueing strategies an interesting area for further exploration, we have not included them in this study.

1.2 Related work

In most existing systems, users choose cluster sizes for their jobs by hand, and the system does not have the option of allocating more or fewer than the requested number of processors. Thus, there have been no reports yet on the effectiveness of adaptive allocation strategies in real systems.

Many simulation and analytic studies have examined the performance of system-centric allocation strategies; *i.e.*, strategies designed to maximize an aggregate performance metric, without regard for individual jobs. In most cases, this metric is average turnaround time [12] [21] [19] [20] [11] [13] [4] [18], although some studies also consider throughput [16]. Rosti, Smirni *et al.* [17] [22] use *power*, which is the ratio of throughput to mean response time. In [6] 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.

Recently, there has been interest in application-centric scheduling, in which individual jobs select resources in order to minimize their own run times, without regard for the performance of the system as a whole. The AppLeS project at the University of California at San Diego [2][3] and the MARS project at the Paderborn Center for Parallel Computing [10] are applying this approach in a wide-area, heterogeneous environment.

Atallah *et al.* [1] have proposed system agents that choose cluster sizes (and specific sets of hosts) to minimize the turnaround time of individual applications. Since their target system is a timesharing network of workstations, they consider the problem of contention with other jobs in the system, but they do not have the problem of predicting the time until a cluster becomes available.

1.3 Outline

Section 2 describes an abstract workload model we have derived from observations at the San Diego Supercomputer Center. Section 3 describes the statistical techniques we use to predict queue times. Section 4 describes the simulator we use to evaluate various allocation strategies. Section 5 presents our evaluation of these strategies from the application's point of view, and Section 6 discusses the effect these strategies have on the system as a whole.

2 Workload 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. These measured workloads depend on the characteristics of the job mix, the properties of the system 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 effect of the job mix from the effect of the system.

In [5] we propose a model of malleable 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 a any number of processors. Section 2.1 summarizes this *application model.*

Once we have a model of individual applications, we can construct a *workload model* that describes the system load, the arrival process, and the distribution of application parameters. In [6] we presented observations of scientific workloads on the Intel Paragon at SDSC and the IBM SP2 at CTC. We summarize those observations in Section 2.2, and use them to derive our abstract workload model.

2.1 A model of malleable jobs

Our model of parallel speedup is based on a family of curves that are parameterized by a job's average parallelism, A, and its variance in parallelism, V. To do this, we construct a hypothetical *parallelism profile*[1] with the desired values of A and V, and then use this profile to derive speedups. We use two families of profiles, one for programs with low V, the other for programs with high V. In [5] we show that this family of speedup profiles captures, at least approximately, the behavior of a variety of parallel scientific applications on a variety of architectures.

2.1.1 Low variance model

Figure 1a shows a hypothetical parallelism profile for an application with low variance in parallelism. The potential 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 of parallelism is $V = \sigma(A-1)^2$.

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

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

2.1.2 High variance model

Figure 1b shows a hypothetical parallelism profile for an application with high variance in parallelism. This profile has a sequential component of duration σ and a parallel component of duration 1 and potential parallelism $A + A\sigma - \sigma$. By design, the average parallelism is A; by good fortune, the variance of parallelism is $\sigma(A-1)^2$, the same as that of the low-variance model. Thus for both models σ is approximately the square of the coefficient of variation, 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.

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

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

[1] The parallelism profile is the distribution of potential parallelism during the execution of a program[21].

a) Low-variance model

b) High-variance model

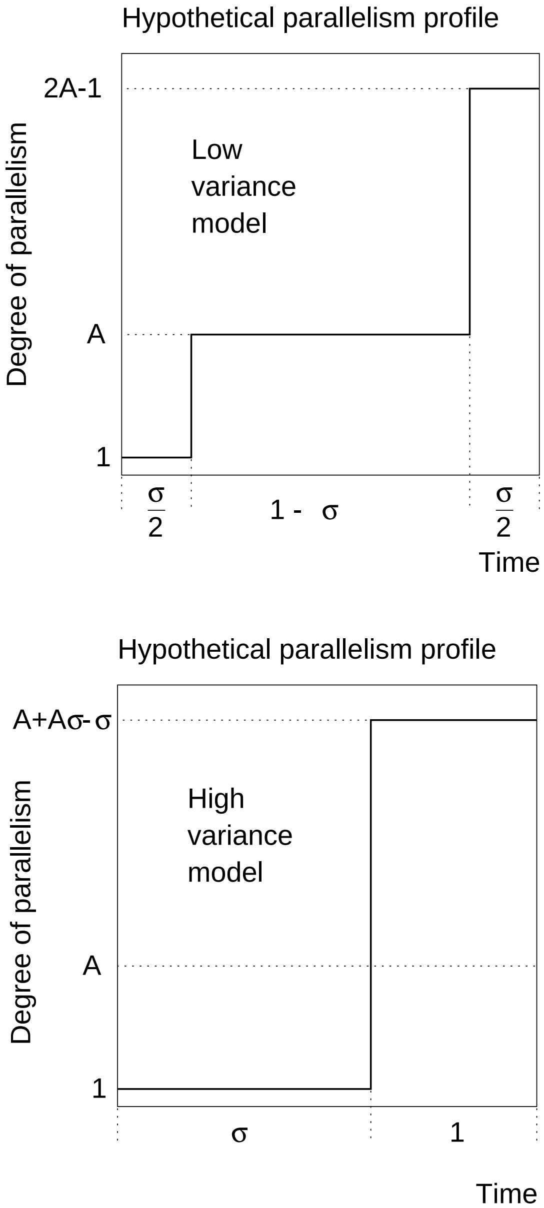


Figure 1: The hypothetical parallelism profiles we use to derive our speedup model.

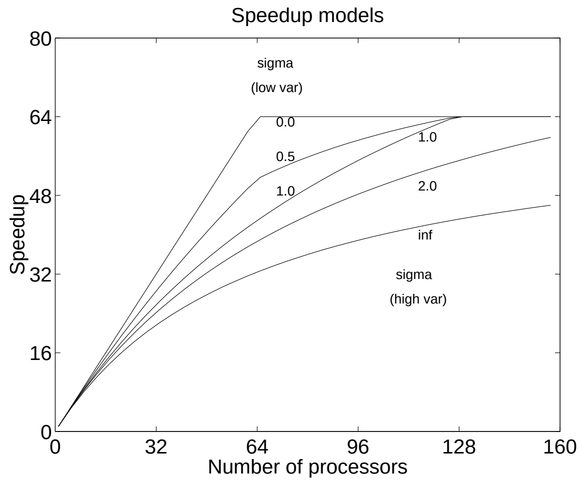


Figure 2: Speedup curves for a range of values of σ.

Figure 2 shows a set of speedup curves for a range of values of σ (and $A = 64$). 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.* [8]:

$$S_{min}(n) = An/(A + n - 1) \tag{3}$$

When $\sigma = 1$ the two models (high and low variance) are identical.

2.2 Distribution of parameters

With the speedup model in the previous section, we can use L, A, and σ to calculate the run time of a job on any number of processors: The speedup on n processors is $S(n, A, \sigma)$; the run time is L/S. 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. Thus we qualify our application model with the understanding that for each application there may be a limited range of viable cluster sizes.

2.2.1 Lifetimes

Ideally, we would like to know the distribution of L, the *sequential lifetime*, for a real workload. But the accounting data we have from real systems does not contain sequential lifetimes; since most jobs run on more than one processor, we do not know what their sequential lifetimes would be. Furthermore, sequential lifetime is not always defined, since many jobs cannot run on a single processor because of memory requirements. Thus, L is really an abstract measurement of the total work a job does, rather than a literal measurement of its run time on one processor.

The accounting data we do have is 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. Nevertheless, we assume that the distributions

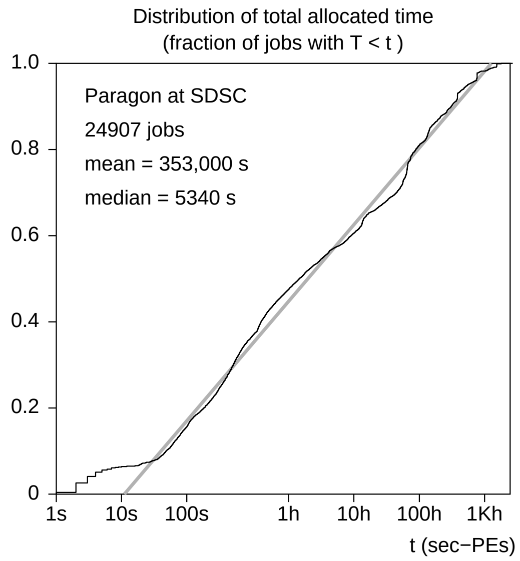


Figure 3: Distribution of total allocated time (wall clock time multiplied by number of processors) for 24907 batch jobs from the Intel Paragon at the San Diego Supercomputer Center (SDSC). The gray line shows the model used to summarize the distribution.

of L and T have the same shape, but different parameters. In our simulations, this is true: the distribution of L (which is an input to the simulator) and the distribution of T (which depends on the application characteristics and scheduling policy) have the same shape, although the values of T are higher due to non-linear speedups.

This result allows us to use observed distributions of T to construct the distribution of L for our workload. We have examined accounting logs from the Intel Paragon at the San Diego Supercomputer Center (SDSC); Figure 3 shows the distribution of total allocated time for 24907 jobs that ran on this machine during a nine-month perion in 1995. The distribution is approximately linear in log space, which implies that the cumulative distribution function (cdf) has the form:

$$cdf_T(t) = Pr\{T \leq t\} = \beta_0 + \beta_1 \ln t \qquad t_{min} \leq t \leq t_{max} \tag{4}$$

where 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 $t_{max} = e^{(1.0-\beta_0)/\beta_1}$.

Since this distribution is uniform in log space, we call it a *uniform-log distribution*. 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 IBM SP2 at the Cornell Theory Center and the Cray C90 at SDSC, and other authors have reported similar distributions on other systems [9][23].

Since T overestimates the sequential lifetimes of jobs, our workload model uses a distribution of L with a somewhat lower maximum than the distributions we observed. In our simulations, L is chosen from a uniform-log distribution with minimum e^2 and maximum e^{12} seconds. The median of this distribution is 18 minutes; the mean is 271 minutes.

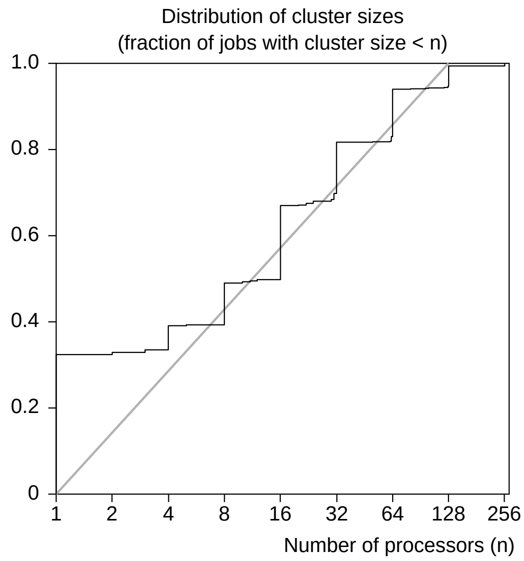


Figure 4: Distribution of cluster sizes for the workload from SDSC. The gray line is the distribution we used for our workload model.

2.2.2 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, 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 the distribution of cluster sizes for the workload from SDSC. Almost all jobs have cluster sizes that are powers of two. The Paragon does not require power-of-two cluster sizes, but the interface to the queueing system suggests powers of two and few users have any incentive to resist the combination of suggestion and habit. We hypothesize 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, the distribution of A is uniform-log with parameters $min = 1$ and $max = 128$. The gray line in Figure 4 shows this distribution. At present observed distributions contains more sequential jobs than our model, but this surplus is disappearing as the workload matures. In the first two months of 1996, the fraction of sequential jobs at SDSC was 21%, down from over 40% a year prior. Thus more recent workloads match our model well.

2.2.3 Variance in parallelism

In general there is no way to measure the variance in potential parallelism of existing codes explicitly. In [5] we propose a way to infer this value from observed speedup curves. To test this technique, we collected speedup curves reported 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.

3 Predicting queue times

In [7] we present statistical techniques for predicting the remaining queue time for a job at the head of the queue. We summarize these 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 $n' = n - n_{free}$ 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 $Q(n')$.

3.1 Predictor A: median

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 equal 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 that turns out to be sufficiently accurate for our intended purposes.

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

$$\begin{aligned} 1 - cdf_{L|a} &= Pr\{L > t | L > a\} \\ &= \frac{1 - cdf_L(t)}{1 - cdf_L(a)} \\ &= \frac{1 - \beta_0 - \beta_1 \ln t}{1 - \beta_0 - \beta_1 \ln a} \qquad t_{min} \leq a \leq t \leq t_{max} \end{aligned} \tag{5}$$

and so the probability of a given event is

$$Pr\{b\} = \prod_{i|b_i=0} cdf_{L|a_i}(t) \cdot \prod_{i|b_i=1} \left(1 - cdf_{L|a_i}(t)\right) \tag{6}$$

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

$$F(b) = \sum_i b_i \cdot n_i \tag{7}$$

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:

$$Pr\{F \geq n'\} = \sum_{b|F(b)\geq n'} Pr\{b\} \tag{8}$$

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:

$$Pr\{F < n'\} \approx \prod_{i|n_i \geq n'} 1 - cdf_{L|a_i}(t) \tag{9}$$

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.

3.2 Predictor B: mean

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 (Equation 5) to calculate the probability that it will have completed before time t.

We will 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:

$$F = \sum_i n_i \cdot cdf_{L|a_i}(t) \tag{10}$$

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

3.3 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 Predictor A to do well when there are many jobs in the system that can single-handedly satisfy the request (benefactors). When there are few benfactors, we expect Predictor B to be better (especially since, if there are none, we cannot calculate Predictor A at all). Thus, in our simulations, we use Predictor A when the number of benefactors is 2 or more, and Predictor B otherwise. The particular value of this threshold does not affect the accuracy of the combined predictor drastically.

4 Simulations

To evaluate the benefits of using predicted queue times for processor allocation, we use the models in the previous section to generate workloads, and then use a simulator to construct schedules for each workload according to the proposed allocation strategies. We compare these schedules using several summary statistics as 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 [9] [23]):

- In the early morning there are few arrivals, system utilization is at its lowest, and queue lengths are short.
- During the day, the arrival rate exceeds the departure rate and jobs accumulate in queue. System 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 "day-time" phase, jobs arrive according to a Poisson process and either begin execution or join the queue, depending on the state of the system and the current scheduling strategy. During the "night-time" phase, 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). We observe that the number of jobs per day is between 160 (when $\rho = 0.5$) and 320 (when $\rho = 1.0$).

5 Results: application-centric

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 individual jobs. We examine how users might subvert such a system, and measure the potential benefit of doing so.

5.1 AVG: incentive for subversion

Our baseline strategy is AVG, which assigns free processors to queued jobs in first-come-first-served 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 [21] [11] [15] [22] [4] [6].

An important feature of this strategy is that it is *work-conserving*: if even one processor is free, the job at the head of the queue will be forced to begin execution immediately. From the system's point of view, work-conservation is expected to yield high utilization, but from the application's point of view, it often makes decisions that are contrary to the interests of the users. At the least, users will object; in many cases, they will subvert the system.

For example, a user might try to avoid long run times by imposing a minimum cluster size for his 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 problem is that inflated jobs (ones that wait in queue for more resources than they strictly require) decrease utilization by leaving processors idle, increase queue times not only for themselves, but also for the other jobs in queue, and may not even improve their own performance. As a result, the performance of the system degrades.

In [6] we evaluate a simple policy in which users require that each job allocate at least 40% of its requested cluster size. The overall result is a 24% *increase* in average turnaround time. Imposing larger minimum cluster sizes degrades the performance of the system even more dramatically. This result suggests that AVG will not perform as well in the presence of self-interested users as it does in simulation.

5.2 OPT: scheduling with perfect prediction

Our application-centric strategy uses queue time predictions to minimize the turnaround time of each job. By evaluating each possibility, we find the value of n that minimizes $Q(n) + R(n)$, where $Q(n)$ is the queue time until n processors are available, and $R(n)$ is the run time of the job in n processors. As in AVG, n is restricted to be no greater than A. In this section we will assume that $Q(n)$ is known deterministically by oracular prediction, and find the maximum benefit individual jobs might gain. In the next section we will see how much of this benefit we can achieve with realistic predictors.

We ran AVG for 120 simulated days with the offered load, ρ, fixed at 0.75. For each of the 30421 jobs, we found the cluster size that would minimize its turnaround time. Many jobs (63%) were allocated the maximum cluster size, A processors. We call these jobs *spoiled*, since they get everything they want.

Of the remaining jobs, we identified two groups: *docile* jobs are the ones that do what the system wants them to do — if they are offered fewer than A processors, they run immediately on the small cluster. *Rebels* are the jobs that would benefit by defying the system and waiting for a larger cluster.

In our simulations, 38% of the non-spoiled jobs (14% of all jobs) should rebel. For each rebel, we calculate the *time savings*, which is the difference between the job's actual turnaround time and its best possible turnaround time. The median savings for a rebel is 15 minutes; the average is 1.6 hours. These are substantial savings, considering that the median duration for all jobs is 3 minutes and the average duration is 1.3 hours. We conclude that using queue time predictions for processor allocation has great potential to reduce turnaround times.

5.3 PRED: using realistic predictions

Of course, scheduling is easy with the benefit of an oracle. We would like to know how much time rebels will save using less accurate predictions about queue times. In this section we will compare the performance of the optimal strategy to an implementable strategy called PRED. PRED is the same as OPT except that instead of knowing $Q(n)$ deterministically, we estimate it using the predictors described in Section 3.

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

Our metric of performance is total time savings divided by the total number of jobs. OPT maximizes this *time savings per job*. We simulated PRED with the same workload, job-for-job, we used to simulate OPT (120 days, 30421 jobs, $\rho = 0.75$). The average time savings per job was 12.8 minutes, compared to the optimal 13.8 minutes. Thus, from the point of view of individual applications, PRED is within 7% of optimal.

Of the rebels, only a few (8%) end up taking longer than they would if they were docile, and for these the time lost is small (3.7 minutes on average). For the majority, the benefit is substantial; the median time savings is 55 minutes; the average is 2.7 hours[2].

5.4 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 Figure 5a is 0.48 (all statistical calculations are performed 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 data. For a perfect predictor, the slope would be 1 and the intercept 0; for our predictors the slope of this line is 0.6 and the intercept 1.7.

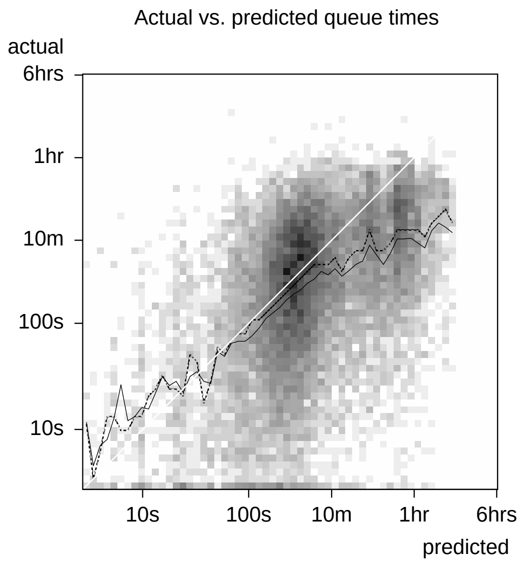
Fortunately, if we know that a predictor is biased, and we estimate the parameters of this bias (based on previous predictions), we can correct the bias by applying a transformation to the calculated values. In this case, we estimate the intercept (β_0) and slope (β_1) of the trend line, and apply the transformation $p_{corr} = p * \beta_1 + \beta_0$, where p is the calculated prediction and p_{corr} is the modified 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 2% 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.

[2] PRED's average time savings per *rebel* is actually better than that of OPT, since PRED chooses fewer, but more successful, rebels (see Table 1). This anomaly is the reason we chose time savings per *job* as our metric.

a) Raw predictors



b) Predictors with bias correction

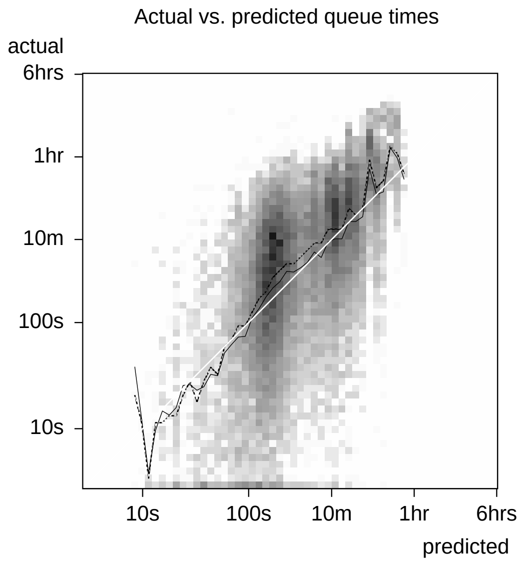


Figure 5: Scatterplot of predicted and actual queue times (log scale). The white line shows the identity function; i.e. a perfect predictor. The solid line shows the average of the actual queue times in each column. The broken line shows the median.

5.5 HEUR: using only application characteristics

We would like to know whether our predictions are really necessary for allocation, or whether we can do as well using only application characteristics. Intuitively, we expect that we can identify good candidates for rebellion by their long lifetimes and high degree of parallelism. Observation of the optimal policy confirms this intuition; the average lifetime of rebels is 8 times the average lifetime of docile jobs.

We propose the following heuristic allocation policy (HEUR): any job with sequential lifetime greater than L_{thresh} and parallelism greater than A_{thresh} will hold out 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.

Like PRED, HEUR requires a form of prompting. In this case, if a job rebels, we calculate its potential time savings, $t_{save} = R(n_{free}) - R(fA)$, where n_{free} is the number of free processors, and fA is the number of processors the job is holding out for. If a period of time, kt_{save}, elapses before the job begins execution, the job is forced to run on the available processors. The parameter k, again, must be tuned according to the workload.

By a semi-systematic exploration of parameter-space, we found that the following values maximized the performance of HEUR: $L_{thresh} = 0$, $A_{thresh} = 1$, $f = 1.0$ and $k = 0.2$. In other words, all jobs, regardless of their characteristics, should hold out for at least A processors, but they should not hold out for long. If they use up 20% of their potential time savings waiting in line, they should concede and start running on the available processors.

Using these parameters, the average time saving per job is 12.8 minutes, which is the same as PRED. Thus we conclude that a well-tuned heuristic allocation strategy can do as well, from the application-centric view, as our predictive strategy. In a real system, it may be difficult to tune the four parameters; we have observed that their optimal values are different for other loads, system sizes and workload models.

5.6 Summary of allocation strategies

Table 1: summary of allocation strategies

	number of rebels (% of total)	average time savings per job	fraction of rebels who lose
OPT	4244 (14%)	13.8 min.	0%
PRED	2388 (8%)	12.8	8%
BIAS	3089 (10%)	13.5	8%
HEUR	11158 (37%)	12.8	68%

Table 1 summarizes the performance of the four allocation strategies. 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.

On the other hand, HEUR is much more agressive than OPT. More than a third of the jobs rebel, although only for a short time. Of these, the majority (68%) end up wasting time in queue. Since these losses are small, and the occasional win is big, the average over all jobs is the same as PRED, but it is not clear whether users would choose a strategy with such a high chance of being detrimental.

6 Results: system-centric

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 (over all jobs) and utilization. We define utilization as the average, over time and processors, of efficiency. Efficiency is the ratio of speedup to cluster size, $S(n)/n$. 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 shows the results of each allocation strategy, using the same workload as in the previous section. In each case, we compare the results with the baseline allocation strategy, AVG. Since the workloads are identical, we can compare them job-by-job and see which jobs do better under which strategies.

Table 2 : system performance

	Average change in utilization (120 days)	Average turnaround time (30421 jobs)
AVG	—	4795 seconds
OPT	+0.1%	5236 (+9%)
PRED	+2.4%	4652 (-3%)
BIAS	+1.0%	5048 (+5%)
HEUR	-5%	6555 (+37%)

The predictive strategies all yield better utilization than AVG. This is surprising, since these strategies often leave processors idle (which decreases utilization) and allocate larger clusters (which decreases efficiency). Thus 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.

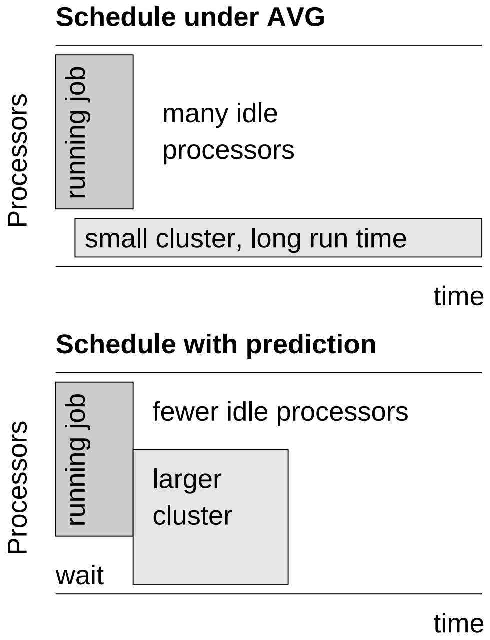


Figure 6: Sample schedules showing how longer queue times and larger cluster sizes can, paradoxically, improve system utilization. Queue time prediction makes it possible to avoid L-shaped schedules and thereby reduce the number of idle processors.

Despite this improvement, both OPT and BIAS lead to longer turnaround times, on average, than AVG. The reason for this degradation is that rebellious jobs impose longer queue times on the other jobs in queue. By using conservative predictions, PRED reduces the number of rebels enough that the average turnaround time is actually better than under AVG. HEUR is much more agressive in its selection of rebels, and the system pays for it. The average turnaround time under HEUR is 37% longer than under AVG.

Since PRED produces the best results for the system, it is tempting to say that it is the best choice for a real system. Unfortunately, it does not satisfy our goal, which is to find an allocation strategy that is robust in the presence of self-interested users. In the case of PRED, users would eventually notice that the predicted queue times were consistently too high; thus, they might apply bias correction on their own behalf. The result would be performance similar to BIAS.

6.1 Improvements

One way to reduce the impact of rebellion on global performance is to *internalize* the cost rebels impose on the jobs behind them. Instead of minimizing the turnaround time of each job, $Q(n) + R(n)$, the system might minimize the sum of run time and the queue time *imposed on all jobs*, $qQ(n) + R(n)$, where q is the number of jobs in queue.

The problem with this approach is that it takes us back where we started; some users will be forced to accept allocations that are contrary to their immediate interests, and these users will have incentive to subvert the system.

An alternative, which can be enforced by the system, is *lock-in*. If a job chooses to rebel, it must declare the number of processors it is holding out for; that is, the value of n that minimizes the rebel's expected turnaround time. Then, even if more processors become available, the rebel can allocate only n. We expect this mechanism to reduce the queue time rebels impose on other jobs, and thereby improve average turnaround time. In our simulations, lock-in reduces the average turnaround time under BIAS from 5048 seconds to 5011 seconds (still 4.5% longer than AVG).

Another way to mitigate the impact of rebellion is to discourage jobs from holding out for short gains. We simulate a strategy in which a job only rebels if its predicted time savings are greater than 20% of its run time. Although this strategy does not strictly minimize expected turnaround time, it would be preferable to users who are risk-averse. Adding *risk aversion* to BIAS with lock-in decreases the average turnaround time to 4783 seconds (marginally lower than under AVG).

Thus, with some modification, BIAS satisfies our design criteria: it allows users to maximize the performance of their jobs using all the information at their disposal, yet it prevents this local optimization from degrading the overall performance of the system.

7 Conclusions

- We have proposed an allocation policy that never creates the incentive for users to subvert the system; thus, we expect it to perform well in the presence of self-interested users. We show that this *application-centric* policy yields global performance as good as the best *system-centric* policy.
- Using predicted queue times to choose cluster sizes significantly reduces the turnaround time of individual jobs, even if the prediction is not very accurate. In our simulations, the average time savings per job is 13.5 minutes (the average job duration is 78 minutes). From the point of view of individual applications, our predictive allocation strategy is within 2% of optimal.
- We show that it is possible to improve the performance of some jobs using only application characteristics and ignoring the state of the system, but we observe that this strategy has a disastrous effect on the overall performance of the system.

7.1 Future work

In this paper we have considered a single system size (128 processors), distribution of application characteristics (see Section 2), 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 predictions are not very accurate. In the real systems we have examined, the information provided by users significantly improves the quality of the predictions [7]. We would like to investigate the effect this improvement on our results.

As part of the DOCT project [14] 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.

References

[1] M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing*, 16(4):319–327, Dec 1992.

[2] Francine Berman and Rich Wolski, Principal Investigators. AppLeS Home Page http://www-cse.ucsd.edu/groups/hpcl/apples/apples.html. University of California at San Diego, 1996.

[3] Francine Berman and Rich Wolski. Scheduling from the perspective of the application. In *Proceedings of the High Perfomance Distributed Computing Conference*, August 1996.

[4] 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 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1994.

[5] Allen B. Downey. A model for speedup of parallel programs. In preparation, 1996.

[6] Allen B. Downey. A parallel workload model and its implications for processor allocation. Submitted for publication in SIGMETRICS'97, 1996.

[7] Allen B. Downey. Predicting queue times on space-sharing parallel computers. In *11th International Parallel Processing Symposium*, April 1997. To appear. Also available as University of California technical report number CSD-96-906.

[8] 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.

[9] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 215–227, 1995.

[10] J. Gehring and A Reinefeld. MARS — a framework for minimizing the job execution time in a metacomputing environment. In *Future Generation Computer Systems (FGCS)*, 1996. To appear.

[11] 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.

[12] 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.

[13] Rajesh K. Mansharamani and Mark K. Vernon. Comparison of processor allocation policies for parallel systems. Technical report, University of Wisconsin, December 1993.

[14] Reagan Moore and Richard Klobuchar, Principal Investigators. DOCT Home Page http://www.sdsc.edu/DOCT. San Diego Supercomputer Center, 1996.

[15] 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.

[16] 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.

[17] 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.

[18] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 101–111, 1995.

[19] Sanjeev K. Setia and Satish K. Tripathi. An analysis of several processor partitioning policies for parallel computers. Technical Report CS-TR-2684, University of Maryland, May 1991.

[20] 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.

[21] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, 17(1):171–180, May 1989.

[22] Evgenia Smirni, Emilia Rosti, Lawrence W. Dowdy, and Giuseppe Serazzi. Evaluation of multiprocessor allocation policies. Technical report, Vanderbilt University, 1993.

[23] 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.