Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling

Thu D. Nguyen, Raj Vaswani, and John Zahorjan

Department of Computer Science and Engineering, Box 352350
University of Washington
Seattle, WA 98195-2350 USA

Technical Report UW-CSE-95-10-01

Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling

Thu D. Nguyen, Raj Vaswani, and John Zahorjan

Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

October 23, 1995

Abstract

We address the design of practical scheduling policies for scalable, shared memory multiprocessors. In particular, we propose and evaluate experimentally processor allocation policies that use information about parallel job execution characteristics in making their decisions. In contrast to existing processor allocation work, which for the most part has relied on perfect information supplied before job execution, we require no *a priori* information, instead relying on *runtime measurement* of executing jobs.

The experimental results we present validate the following observations:

- The use of runtime measurements of workload characteristics can significantly improve performance relative to disciplines that are oblivious to these characteristics, even given the inherent inaccuracies in the measurements and the overhead of the dynamic reallocations that approaches based on them require.
- Runtime measurements are sufficient for the scheduler to achieve performance surprisingly close to that possible when perfect, *a priori* information is available.
- The primary performance loss, relative to the use of *a priori* information, is due to the transient poor decisions of the scheduler as it acquires information on the running applications, rather than to the overhead of measurement itself.

Additionally, we consider both interactive environments, in which a response time directed scheduler is appropriate, and batch environments, in which maximizing useful instruction throughput is the primary goal.

Our experiments are performed on a prototype implementation running on a 50-node KSR-2 shared memory multiprocessor. A final novel aspect of our work is the use of both hand and compiler parallelized programs in our test workloads.

1 Introduction

It seems intuitively clear that knowledge of job characteristics is an advantage to a processor scheduler in any computer system. For example, in both sequential and parallel machines, estimates of basic job length allows some bias towards shortest-job-first, leading to reduced average response times.

This paper considers the use of information on job speedup and efficiency in the scheduling of parallel jobs on parallel machines. There has been considerable work on this topic in the past. A great deal of this work has been involved with the interesting question of how best to schedule jobs

when given perfect information about speedup, a fundamental attribute of parallel workloads (e.g., [21, 8, 3, 2], among many others). While it would be useful to understand how to best schedule a set of jobs given *a priori* information on their speedups, such information is difficult, if not impossible, to specify accurately in practice[1] because of the sensitivity of job performance to the input data set and to the relative locations of allocated processors on the machine's interconnection network.

As an example, consider the MP3D application from the SPLASH [23] benchmark suite when run on the KSR-2 multiprocessor. The KSR-2 has an interconnection network that is a hierarchy of rings. The basic communication time between two rings is roughly six times that for communication within any one. Because of this, MP3D, which has poor locality, achieves optimal speedup at a number of processors that depends strongly on the location of those processors. In particular, if all processors are located on a single ring, MP3D peaks at 12 processors. If allocated processors are split across two rings, performance peaks at 24 processors. In both the cases that the user requests 12 processors, but the ones allocated are on two rings, and the user requests 24 processors, but they are all on one ring, the achieved speedup is roughly half that of the actual optimum. Thus, even historical information on job performance is not sufficient for the user to provide accurate information on the optimum number of processors to allocate in its next execution.

The purpose of this paper is to determine whether runtime measurements can be obtained sufficiently cheaply, and with sufficient accuracy, to be used in making scheduling decisions in a way that approaches the performance attainable when omniscient *a priori* information is available, but without requiring that information. Taken in this context, this paper describes work intended to address the way in which *realizable* schedulers might make use of information on job characteristics, focusing particularly on job speedup and efficiency.

While at first glance it would appear that runtime measurements of job behavior are clearly useful, the actual situation is considerably more complicated, especially in parallel systems. In particular, the value of runtime measurements to parallel processor allocation policies depends on the answers to the following questions.

- *How can speedup and efficiency be measured at runtime with acceptably high accuracy and low overhead?*
- *Do parallel applications have sufficiently stable characteristics that their recent past is a good indicator of the near future?* (For example, given the well known phase behavior of these workloads, one might reasonably guess that this is not the case.)
- *How can the measures taken when an application is run on p processors be used to estimate its performance when run on q?*
- *Do the costs of the potentially many reallocations required in the searches (which are inherent in this approach) to find appropriate final allocations outweigh the benefits?*

Our goal is to help answer these questions.

We begin by presenting a scheme that allows the runtime estimation of speedup and efficiency, at low overhead. The key to our approach is the measurement of the primary sources of *inefficiency*: communication, idleness due to competition for critical sections and load imbalance, and system overheads.

We then consider how to use the information so obtained in making scheduling decisions. We examine two distinct scenarios: interactive systems, where minimizing response time is the goal,

[1] Of course, supercomputer users running the same application repeatedly on similar data sets are accustomed to providing this information, based on the performance of prior runs. However, at the very least this is an inconvenience. At the worst, apparently insignificant changes in the data set may in fact have a substantial effect on the optimum allocation, although this could go undetected by the user.

and batch systems, where maximizing the rate at which useful work is completed is the goal. Both kinds of computing already have significant roles on existing large scale parallel platforms [7]. For the interactive environment, we use measured efficiencies to guide adjustment of the number of processors given to each running job, attempting to maximize its speedup. For batch environments, our scheduler allocates processors in an attempt to maximize current system efficiency.

Evaluation of the extent to which runtime measurements can be used to help the scheduler is done through the execution of benchmark programs on prototype implementations running on a KSR-2. The benchmarks we use are taken from the SPLASH [23] and Perfect [1] benchmark suites, the best benchmarks available to us for this work.

Our central result is that the use of runtime measurements of job behavior can improve scheduler performance substantially, despite the inevitable noise in the gathered data and the overheads involved in its use, and that such schedulers can approach the performance attainable when perfect information on job speedups is available *a priori*.

1.1 Outline

In the next section we discuss our techniques for measuring job speedup and efficiency characteristics at runtime. Section 3 presents a response time oriented scheduler that makes use of these measurements, and experimental results for it. In Section 4 we turn to the problem of scheduling batch work to maximize its completion rate, again proposing a policy that employs runtime measurements and evaluating its performance experimentally. Section 5 concludes our work.

2 Measuring Job Speedup and Efficiency at Runtime

2.1 What We Measure

One approach to using measured job characteristics in formulating scheduling policies is to exploit general, long-term characteristizations of the workloads (e.g., [12]). An example of this from sequential systems is the use of feedback scheduling: the success of this approach is implied by the empirical validity of the assumption that a job that has run for a long time is likely to continue running for a proportionally long time. Exploiting workload characteristics in this way does not require runtime measurements of individual job characteristics.

A second approach is to use historical information from previous executions of individual jobs to provide job-specific information when the job is submitted. Finding and determining how to exploit such long-term characterizations for production parallel workloads is an important topic, and has been an aspect of much work in this area (e.g., [8, 20, 22, 3]).

In this paper we concentrate on a third approach, using the specific, recently measured characteristics of the currently running jobs to optimize performance for the specific workload in execution. (Sobalvarro and Weihl [24] take a similar approach in an attempt to relax the constraints of coscheduling.) As we will show, this approach can offer substantial benefits. It also has the advantage of being applicable independently of long-term production workload characteristizations, which is especially important in light of the nascence of production parallel programming: many of these long-term properties are likely to change in response to new hardware architectures, to improvements in parallelizing compiler technologies and runtime support systems, and to a continuing expansion in the diversity of applications run on parallel platforms.

The particular job characteristics we use, and so must measure, are speedup and efficiency. These are obviously key aspects of parallel job execution, and so seem like good candidates for use in making

scheduling decisions. In the next subsection, we explain how we obtain them through runtime measurement.

2.2 Obtaining Runtime Measurements of Speedup and Efficiency

The basic parallel job characteristics we wish to exploit are speedup and efficiency. While these measures are normally applied to the complete execution of an application (for example, speedup on P processors is the time to complete divided by the time to complete on a single processor), to be useful in our work we must take a more short-term view. In particular, we wish to measure speedup or efficiency over the fairly short-term past, with the intention of relying on it as a predictor of the near-term future. We therefore use the terms speedup and efficiency in this (more instantaneous) sense.

While speedup and efficiency are of course intimately related, in practice one (efficiency) is rather easily measured, whereas the other (speedup) is not. Imagine trying to estimate speedup directly by measuring the time required to complete some identifiable unit of application work. If we take that measurement when the application is running on P processors, we have no way to roll the application state back and remeasure when running on a single processor. The best we can do is to measure the next similar unit of work; in the common case of applications with an iterative structure, we might measure the next iteration, for instance. Unfortunately, the amount of work inherent in the application can vary from iteration to iteration, and so we have no way to determine if comparing times from separate iterations is valid.

Because of this difficulty, we instead measure efficiency, or more precisely, we measure the inefficiencies due to overheads and subtract these from 1.0. When speedup information is required, we then infer it from our efficiency estimate.

It is well known that loss of efficiency in shared memory systems arises from a combination of parallelization overhead (e.g., per-processor initialization, work partitioning, and synchronization), system overhead (e.g., events such as page faults and clock interrupts), idleness (e.g, due to load imbalance, synchronization constraints, and sequential portions of execution) and communication. Our experience with a wide variety of benchmark programs shows that we can accurately predict application efficiency by measuring only system overhead, idleness, and communication; parallelization overhead is typically small. Thus, we require only estimates of the other three components to accurately assess efficiency.

On the KSR-2, we rely on a combination of hardware and software support to measure system overhead, idleness, and communication costs. The KSR-2 has per-node event monitors that maintain three critical hardware counters: elapsed wallclock time, elapsed user mode execution time, and accumulated processor stall[2]. Thus, measuring system overhead and communication is simply a matter of reading these three registers periodically. Measuring idleness is only slightly more involved: we instrument the Cthreads synchronization code [4] to keep elapsed idle time using the wallclock hardware counter. Our idleness measurement scheme is relatively overhead free because idleness accounting is performed when the processor would otherwise not be doing any useful work.

[2] On shared memory systems, such as the KSR-2, communication is required whenever data does not currently reside in the local cache, or is not in an appropriate state. Processors in many shared memory systems stall in this situation, that is, they execute no instructions until the remote data becomes available. Thus, processor stall corresponds to communication cost. On message passing machines, measuring performance loss due to communication would be even more straightforward, requiring only software support

2.3 The Interval of Measurement

There are two possible approaches to choosing measurement intervals: doing so in a way that is transparent to the application, and relying on application assistance. Obviously, the application independent approach is preferable, if it can be made to work.

The simplest application independent measurement technique is to take samples at fixed intervals. However, choosing the interval over which a single measurement is taken is a difficult problem. On the one hand, for the individual samples to be reliable, the measurement interval must be long enough that the job completes a representative sample of its computation. On the other hand, long intervals increase the latency of the scheduler in responding to changes.

Our experience with the benchmark programs indicates that this application independent approach is not sufficiently accurate to be useful: there is no fixed interval that is suitable across these applications. While it might be possible to design a dynamic scheme for choosing an appropriate interval independently for each application, we have not yet pursued this approach. Instead, we rely on what we argue is an acceptable level of application involvement.

The measurement interval we use is an application "iteration." Iterative program structures are very common in parallel applications. For example, the overwhelming majority of the applications in our benchmark suites have an inherently iterative structure; in particular, there is an outermost sequential loop that drives the execution. We don't require this, however. At a minimum, what we do require is that there be some identifiable point in the application's execution where it can indicate that a unit of work has been completed. For example, in a fairly coarse grained application employing user level threads as the basis of parallel execution, iterations might be defined to be the work between the kernel thread's dequeueing and subsequent enqueueing (or termination of) a user level thread.

We rely on the application to call appropriate runtime routines at the beginning and end of each iteration. For this study, we have inserted these calls into each application in our application suite by hand. In many instances, however, we believe that it would be possible for compilers to automatically detect iterative behavior and insert the calls to the runtime system appropriately. For example, Jeremiassen [9] has shown that it is possible to automatically detect phase behavior for many hand coded applications.

2.4 Use of Job Length Predictions in Scheduling

Application characteristics other than speedup and efficiency certainly are also important in scheduling policy design. One such characteristic is remaining execution time, which might be predictable by measuring acquired execution time, or number of processors used, or amount of memory consumed, if it were known that these measurable quantities had some reliable correlation to (the unmeasurable) future execution time. Many previous works have hypothesized the existence of such correlations, and have considered them when evaluating their policies (e.g., [11] and [3], among other). Feitelson and Nitzberg [7] present evidence that there is such a correlation for execution time consumed and number of nodes used, at least for one production installation. (Because they measured a message passing machine, in which processors and memory are allocated together, it is difficult to determine from their data whether a similar correlation holds for memory use alone.)

In the work presented here, we do not attempt to make use of these predictions. Rather, we assume that some other mechanism, such as the feedback scheduling employed in sequential systems, is used for this purpose. (Parsons and Sevcik [18] present the design and evaluation of two such schemes, for example.) We consider the workload mixes we schedule to be the subset of a larger job mix chosen for current execution by such a mechanism.

3 Interactive Environments: Improving Response Time

In this section, we describe and evaluate a scheduling policy designed to improve mean response time in interactive environments through the use of runtime gathered job characterizations. This new policy is called ST-EQUI. Before presenting it, we first describe the baseline dynamic equipartition policy from which ST-EQUI is derived.

3.1 The Equipartition Policy: EQUI

The basic scheduling policy on which we build is dynamic equipartition [25], which we call EQUI. Under EQUI, each currently executing job is allocated an equal number of processors. Processor reallocations take place at job arrival and departure times. EQUI is representative of the space sharing approach to processor allocation that has been found to perform well for multiprogrammed shared-memory multiprocessors [25, 14], and exemplifies schemes actively used in such environments [10].

We have used the EQUI policy as our baseline despite two common objections to it (and other dynamic policies). The first is that the cost of periodic processor reallocations is too high. We do not find this to be the case in practice. The basic kernel path cost of reallocating a processor on our poorly-tuned system is about 5 milliseconds. In addition to this, processor reallocations cause some loss of cache state, and so entail a cache refill overhead. However, this cost is not large, especially relative to the rate at which reallocations take place, even for the KSR's 32 megabyte local caches.

The second common objection to dynamic policies is that parallel applications often rely on having a fixed number of processors available to them during their entire execution. This dependence stems from the implementation of application level work partitioning in a static way, often based on the number of processors available to the job when it first begins execution. For instance, a compiler parallelized code might make a single decision at program initialization time about how many threads to use in executing parallel loops; the program will then run very poorly if the actual number of processors available changes.

While it is true that existing parallel applications often rely on the number of available processors being fixed, we have found that they do so needlessly, that is, this reliance is an artifact of environments in which the programs were developed (which made this fixed allocation guarantee) rather than on some fundamental requirement of the software. For example, we were able to successfully transform all of our benchmark applications to accommodate changes in processor allocation with little difficulty. In the case of the compiler parallelized codes, this was done by a simple change to the threads library that is their target. In the case of the hand coded programs, simple changes in the selection of the number of threads to use was sufficient.

One limitation of our quick conversion of these programs, though, is that we have implemented application level dynamic scheduling only at the level of iteration boundaries: the applications examine and adjust to the number of available processors each time they begin an iteration, but do not do so while executing any one iteration. It is clearly possible to do much more dynamic scheduling (see, for example, [19, 6, 13]); we did not do so because of the very large incremental implementation cost relative to our more restrictive change, and because we anticipated that the additional benefits of this added flexibility at the application level would be quite modest.

3.2 Using Runtime Measurements: ST-EQUI

The specific policy we propose to take advantage of runtime measured speedup characteristics is called *ST-EQUI*. At the highest level, each job is allocated an equal number of processors, just

as with EQUI. However, each time a reallocation takes place, each affected job engages in a *self-tuning* procedure by which it estimates how many of its allocated processors it should actually use to maximize its current speedup. The self-tuning is performed by repeatedly adjusting the number of processors the application uses and measuring its resulting efficiency. From those measurements the application infers its speedup under the current allocation decision, and potentially attempts to adjust the allocation in a way that will improve it.

The advantage of ST-EQUI over EQUI arises when at least some jobs determine that they are better off using fewer than their fair share of processors. In this case, they release their processors back to the system, which reallocates them as equally as possible among those jobs that can profitably make use of more than their fair share. It is reasonable to expect the jobs to release these processors because they have no incentive to keep them (they have determined that they run more slowly using them than without them). Additionally, in any system charging for resource use, there is a positive incentive to release excess resources.

The disadvantages of ST-EQUI relative to EQUI are that reallocations, and their associated costs, are more frequent, and that there is potentially considerable overhead associated with the search procedure followed during self-tuning. To understand better the source of that overhead, we next present the self-tuning procedure in somewhat more detail.

3.2.1 Self-Tuning

We present here an overview of the self-tuning procedure we employ. Comprehensive details can be found in [16], which examines the use of this technique in a static (essentially uniprogramming) environment.

Self-tuning is based on a simple optimization technique, the method of golden sections [15], which searches for the maximum of a unimodal function over a finite interval by iteratively computing function values and narrowing the interval in which the maximum must occur. In our case, the function to be maximized is job speedup. Evaluating the function value at p involves running the job for a single iteration using p processors, measuring the resulting efficiency, and inferring from it the job's speedup at p. The initial interval of interest in the search is 1 to P, the number of processors currently available for use by this job.

There are two basic problems we must address in using golden sections for our purpose. The first is that speedup functions are not, in general, unimodal. We address this in a simple, greedy way. When the results of the function evaluations in the current interval of interest demonstrate that the function is not unimodal, we reduce the interval of interest to the largest contained subinterval that includes the maximum speedup value observed so far and for which the known speedup values are conformal with a unimodal function. The experiments in [16] show that this procedure works remarkably well, nearly always converging to a near optimal value.

The second problem we face is one of efficiency. While the golden sections method converges in a number of probes proportional to the log of the interval length, the cost of an individual probe can be quite large (if the job has poor speedup at the probed number of processors). We address this by exploiting the assumption that speedups cannot be superlinear[3]. In particular, we begin self-tuning by executing one application iteration using all P available processors. This allows us to estimate $S(P)$, the job's speedup with P processors. Since speedups can never be superlinear, we know that the globally best number of processors must fall in $[S(P), P]$. Our search therefore starts in this interval. We use the same observation at later points in the search: the lower bound of the current interval of interest must always be no smaller than the largest speedup so far observed.

[3] In fact, this property holds by definition in our system, since we estimate efficiency by measuring inefficiencies and subtracting them from 1.0.

3.2.2 Implementation of Self-Tuning

We have implemented the code required to perform self-tuning in the Cthreads library. Thus, the self-tuning procedure is independent of the specific application to be run. Additionally, there is no code development overhead involved in using it: the application programmer simply links his program with the modified version of Cthreads.

3.3 Performance

To evaluate whether runtime measurements are sufficiently accurate for use in making scheduling decisions, and to illustrate the benefits of self-tuning as just described, we compare the multiprogramming performance of three policies:

- *EQUI.* The dynamic equipartition policy described in Section 3.1. EQUI serves as a baseline for the performance of an achievable policy.
- *ST-EQUI.* The EQUI policy augmented with self-tuning (Section 3.2).
- *AP-EQUI.* AP-EQUI operates in the same way as ST-EQUI, except that it uses perfect *a priori* information on job speedup rather than imperfect runtime estimates. Given this information, AP-EQUI needs to reallocate processors only at job arrival and departure times. When it reallocates, it gives each job no more processors than the number that maximizes its throughput. Distinctions in performance between AP-EQUI and ST-EQUI serve to illustrate the impact of errors in our runtime measurements, as well as the overhead of dynamic self-tuning.

We next describe the workload used to exercise these policies.

3.3.1 Workload

As explained in Section 1, we are interested in a diverse workload composed of both hand-coded (SPLASH) and compiler-parallelized (Perfect) applications.

Our previous detailed study of these applications [17] made clear that these programs could be divided into three broad classes:

- *Good speedup.* Most of the hand-coded applications fall into this class, which is characterized by fairly good speedup that rises monotonically as the job receives processors. Some of these applications exhibit slowdown beyond a certain number of allocated processors.
- *Poor speedup.* Almost all of the compiler-parallelized applications fall into this class, which is characterized by nearly negligible speedup at most processor values. Most of these applications exhibit slowdown beyond a certain number of allocated processors.
- *Erratic speedup.* This class consists of applications whose speedup is irregular, e.g., it varies over time, or its curve exhibits multiple local maxima. Such behavior can be observed in both hand-coded and compiler-parallelized applications [17].

Because it is clearly infeasible to run experiments with all possible combinations from our benchmark suites, we instead use our taxonomy to reduce the number of jobs that must be considered. In particular, we chose to select a single application from each of the three classes as the representative of that class, specifically, the application from each class exhibiting the best speedup. Thus, we choose Barnes from the SPLASH suite to exemplify good speedup, FLO52 from the Perfect codes

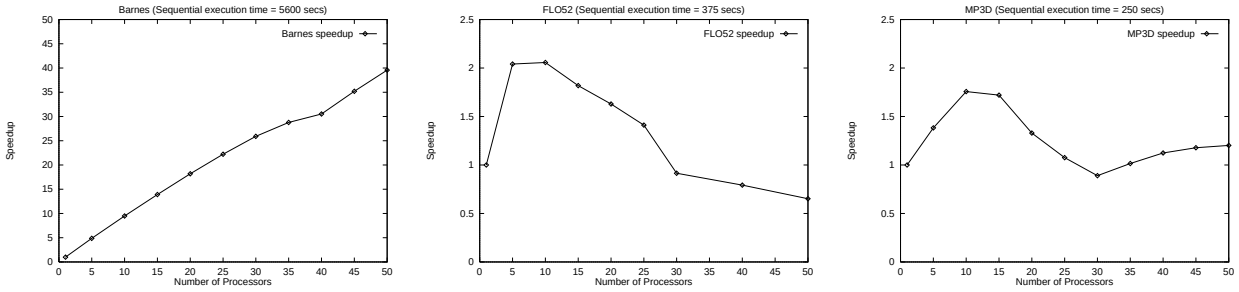

Figure 1: *Speedup Characteristics of the Representative Jobs*

to exemplify poor speedup, and MP3D from the SPLASH suite to exemplify erratic speedup. The measured speedup curves for these applications are given in Figure 1.

Next, we chose to set a maximum multiprogramming level of four, reasoning that (a) given our 50-processor machine, higher multiprogramming levels would increase processor demand to an extent that would render allocation decisions trivial, and (b) such a limit is prevalent in practice, since memory constraints dictate that only a relatively small number of jobs can be allowed to run concurrently. This decision is supported by the measurements in [7], which indicate that a multiprogramming levels of 2, 3, and 4 are the three most common during daytime hours in their production environment.

Thus, given the three representative applications and a maximum multiprogramming level of four, we constructed and evaluated all 31 of the possible workload mixes containing more than a single job type. Figures 2–4 show job response time for 10 of these 31 workload mixes. We now discuss the performance of these workload mixes when running under the three processor allocation policies previously described.

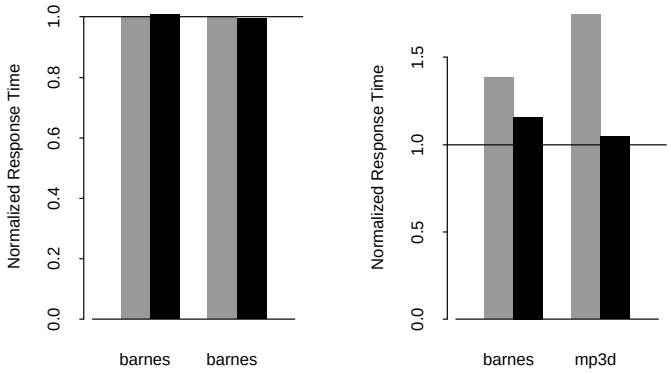

Figure 2: *Response Time Results at Multiprogramming Level = 2. (Grey bars are results for EQUI; black bars are results for ST-EQUI; results are normalized with respect to AP-EQUI)*

3.3.2 Performance Results

Figures 2, 3, and 4 depict the performance of a representative sample of the workload mixes under the EQUI and ST-EQUI policies for multiprogramming levels 2, 3, and 4 respectively. Response times under these two policies are shown normalized to those under AP-EQUI (the horizontal line on each graph). These results lead us to the following observations.

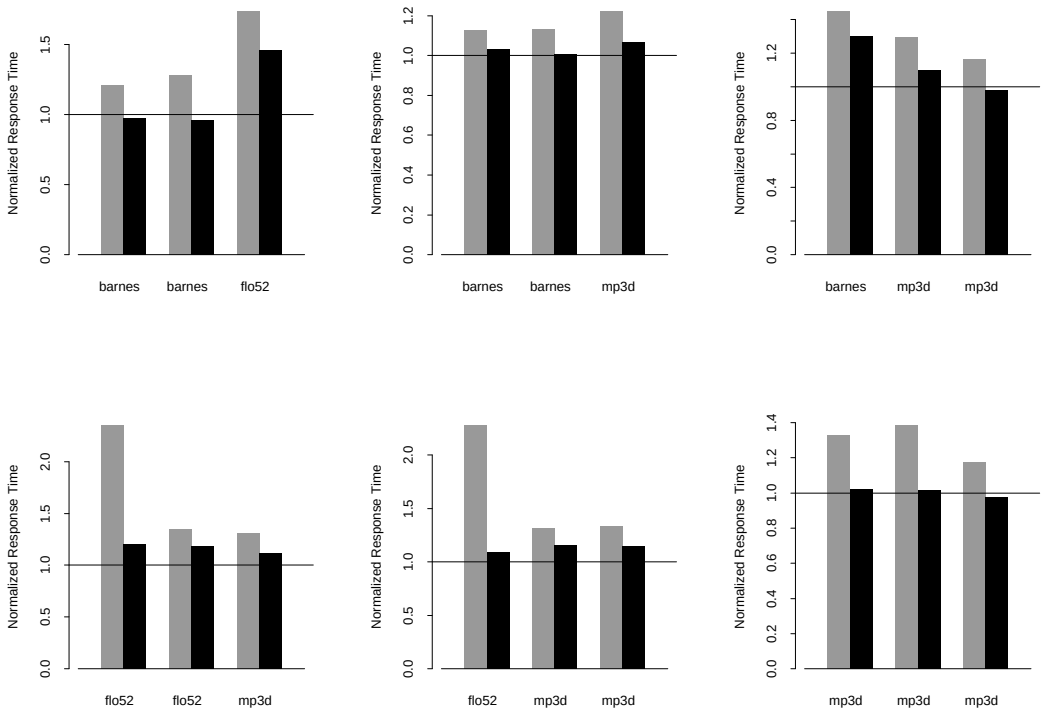

Figure 3: *Response Time Results at Multiprogramming Level = 3. (Grey bars are results for EQUI; black bars are results for ST-EQUI; results are normalized with respect to AP-EQUI)*

- *The policy using runtime measurement (ST-EQUI) outperforms the similar policy that does not (EQUI).*

  This effect arises because all jobs can benefit by participating in cooperative processor allocation. In scenarios with high demand for processors (e.g., all jobs request their equipartition share), ST-EQUI behaves exactly as does EQUI, so its performance is no worse. However, in scenarios with more complex processor demands, ST-EQUI performs much better than does EQUI: jobs exhibiting slowdown points run faster by shedding excess processors that only degrade their performance; jobs exhibiting good speedup can then use these processors to run faster as well.

- *The policy using runtime measurements (ST-EQUI) performs nearly as well as the policy provided perfect a priori information (AP-EQUI) in most cases.*

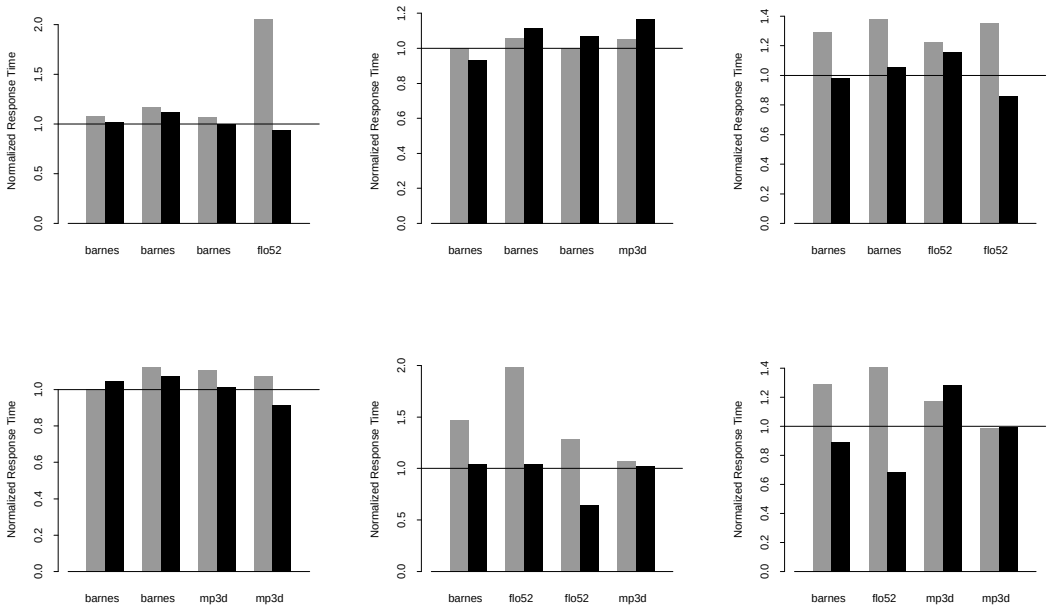

Figure 4: *Response Time Results at Multiprogramming Level = 4. (Grey bars are results for EQUI; black bars are results for ST-EQUI; results are normalized with respect to AP-EQUI)*

The difference between the performance of the two policies stems mainly from the overhead of self-tuning, which ST-EQUI must accept as the price of determining at runtime the jobs' processor needs (i.e., of obviating *a priori* information). This overhead is, as described in Section 3.2.1, the cost of running for some time at processor allocations that yield poor performance. (See also [16].) Despite this, with few exceptions, ST-EQUI performs nearly as well as does AP-EQUI.

We also observe that for each of the three jobs types, there are some workload mixes in which the jobs perform better than under AP-EQUI. The reason for this is the one alluded to in the example in the introduction: the number of processors that maximizes a job's speedup depends on which processors are allocated. Because AP-EQUI must be provided a single allocation target, and can not adjust to which processors are allocated as reallocations take place, ST-EQUI is able to outperform it on occasions when jobs are sensitive to processor placement.

- *Dynamic policies incur some cost.*

  Consider the performance of the Barnes jobs in the various workloads. The difference between the performance of ST-EQUI and that of AP-EQUI occurs because ST-EQUI consistently provides Barnes with a lower allocation (averaged over the job's lifetime) than does AP-EQUI. ST-EQUI is unable to provide the full desired allocation because: (a) although the other jobs in the mix might eventually yield processors to Barnes, these processes arrive slowly, at a rate limited by the pace of the other jobs' self-tuning, and (b) vagaries in the search procedure applied by self-tuning can result in the other jobs' holding slightly more processors than actually desirable, denying Barnes these processors. Note that although the idealized AP-EQUI incurs neither of these costs, the first (and more significant) one is a problem for

EQUI as well — delay in processor arrival is a source of overhead for all practical processor allocation policies.

We have shown that ST-EQUI enjoys a clear performance advantage over EQUI, without requiring *a priori* knowledge of application needs (as is the case for AP-EQUI). Furthermore, the policy is fair in the sense that it does not discriminate among job classes: it simply responds to the jobs' own requests to release/acquire processors, giving each job equal weight in the attempt to minimize response time. In the next section, we discuss relaxing this fairness in the interest of maximizing overall system efficiency.

4 Batch Environments: Improving System Efficiency

In batch environments, such as are common for overnight runs of large parallel applications, the critical performance measure is not response time, but rather the rate at which useful work can be completed; the higher this rate, the larger the workload that can processed in a fixed amount of time. In these environments, the goal of the scheduler is to maximize *system efficiency*, the sum of the parallel efficiencies of all processors. In this section, we describe and evaluate a scheduling policy with this goal.

4.1 The EQUAL-EFF Policy

The policy we propose, EQUAL-EFF, employs a heuristic in attempting to maximize system efficiency: it tries to partition the processors among the currently executing jobs in a way that results in all jobs having about equal current efficiencies. The intuitive motivation for this is that reallocating processors from a currently executing job with low efficiency to one with high efficiency is very likely to improve overall system performance; the limiting case of this process is equal efficiency allocation. Additionally, it is possible to find a procedure to compute equal efficiency allocations that is both simple and has low overhead, and that makes decisions that tend to require only small reallocations (if any) at successive scheduling moments. We show below that allocating for equal efficiency also results in near optimal system efficiences in practice.

Allocations under EQUI-EFF are computed in a simple, greedy way. We begin by noting that maximizing system efficiency is equivalent to maximizing the sum of the speedups of the running jobs (since speedup is equal to average efficiency times number of processors). Let $S_j(p)$ be the estimated speedup of job j on p processors. At each scheduling moment, we begin with the base case that all processors are unallocated, and then hand out the available processors to jobs one by one. Let p_j be the number of processors allocated to job j at some point in this process. We hand the next processor to that job for which $S_j(p_j + 1) - S_j(p_j)$ is maximum, so long as this value is positive. We stop allocating when either there are no remaining unallocated processors or no job is estimated to have a positive benefit from additional processors. Note that this procedure guarantees that each job is allocated at least one processor, so that no job can be starved.

This procedure gives the scheduler a set of target allocations for the jobs. If those targets differ from the current allocations, processors are moved among the jobs to achieve the new targets.

There is one difficulty in implementing this process in practice: we do not have available the full speedup functions for the applications. Rather, we have the presumably accurate value measured for the most recent allocation, as well as potentially out of date information obtained for previous allocations. To overcome this problem, we employ a simple analytic speedup function, taken from [5], as a substitute for the jobs' actual speedups. We parameterize the speedup function for job j

so that it intersects the speedup estimate obtained in the most recent measurement interval. Our allocation scheme then walks along these speedup curves.

This procedure uses only the most recent efficiency estimate, ignoring those obtained earlier. The motivation for that is that job efficiency can be affected by which processors are allocated, not just how many are allocated. Because old allocations may have consisted of substantially different processors than the current set (and so than the next set likely to be allocated), we distrust this old information.

4.2 Preliminary Modelling Results for EQUAL-EFF

Before implementing our prototype of EQUAL-EFF, we first evaluated the extent to which equalizing job efficiencies maximizes system efficiency, using an extensive set of simple simulations. The inputs to the simulations were the measured speedup curves of the applications; the outputs were total system efficiency. We ran the allocation procedure described above using the known speedup curves and compared the system efficiencies obtained to those for OPT-EFF, a policy that allocates processors optimally (for this measure). We computed the OPT-EFF allocations and performance measures using a simple dynamic program, taking the same speedup curves as inputs. For N jobs running on P processors, the dynamic program requires time $O(NP^2)$.

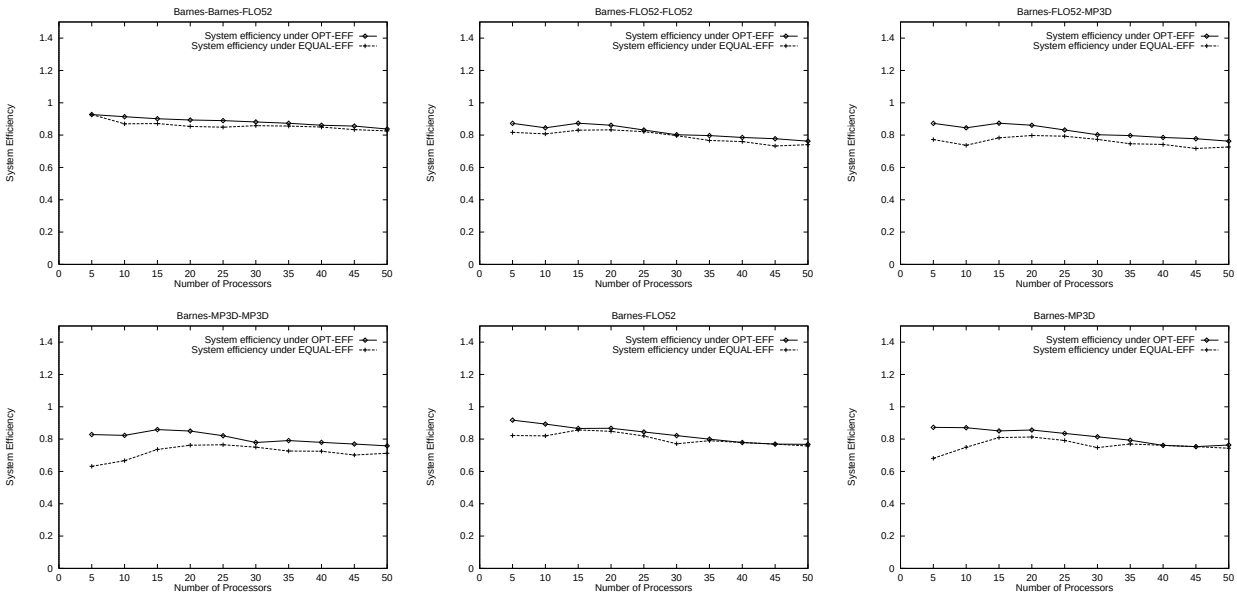

Figure 5: *Modelled Results for EQUAL-EFF and OPT-EFF*

Figure 5 shows the system efficiencies obtained by these simulations for a representative set of workload combinations, as well as the optimal system efficiencies computed for OPT-EFF by the dynamic program. In all cases, the greedy scheme employed by EQUAL-EFF comes very close to the optimum, with the largest differences occurring only for very small numbers of processors.

Based on the results of this simple model, we were motivated to continue to the prototype implementation of EQUAL-EFF. The results obtained from experiments with that prototype are presented next.

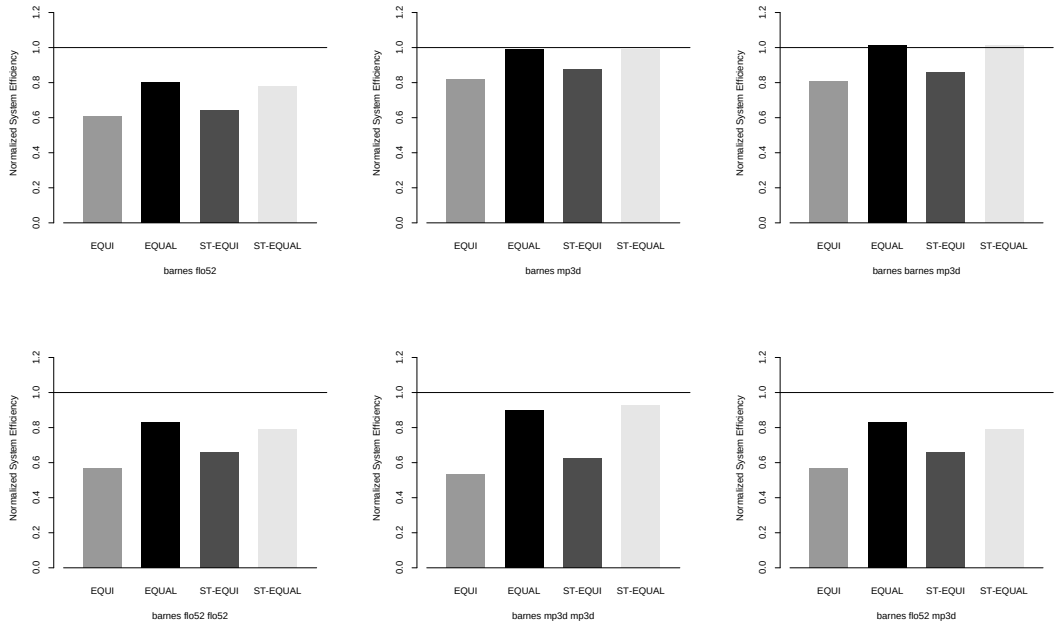

Figure 6: *System Efficiency Results (Multiprogramming Level = 3).*

4.3 Experimental Performance Results

We evaluate the use of runtime measured job characteristics in improving scheduling in a batch environment by considering four related policies:

- EQUI. The basic dynamic equipartition policy (Section 3.1).
- EQUAL-EFF. The equal efficiency heuristic policy (Section 4.1).
- ST-EQUI. The self-tuned equipartition algorithm, which was designed to reduce response times (Section 3.2).
- ST-EQUAL-EFF. The EQUAL-EFF policy with the addition that each job also engages in self-tuning, releasing processors when it determines that it has been assigned more than it can profitably use.

The performance results under these policies are compared against those predicted by OPT-EFF. Because OPT-EFF is computed offline, it does not capture any of the overheads that are inevitable in scheduling in practice, exaggerating its optimism.

We assessed performance with workloads composed of the same representative jobs as were used in the Section 3. However, we used a single multiprogramming level of 3 in all experiments (a reduction from the maximum of 4 considered for the interactive environment) to reflect the likely larger size of jobs submitted for batch execution. (This change is supported by the measurements in [7].) Additionally, we present here results only for those workloads that include a Barnes job, the representative from the class of jobs having good speedup. Workloads without Barnes are relatively uninteresting, as there are 50/3 processors available to each job under simple equipartition, and this

Load	Job	EQUI	EQUAL-EFF	ST-EQUI	ST-EQUAL-EFF
Barnes-FLO52	Barnes	0.29	0.43	0.20	0.41
	FLO52	0.20	0.30	0.35	0.29
Barnes-MP3D	Barnes	0.32	0.38	0.34	0.38
	MP3D	0.34	0.37	0.36	0.38
Barnes-Barnes-FLO52	Barnes	0.32	0.44	0.43	0.44
	FLO52	0.05	0.22	0.31	0.23
Barnes-Barnes-MP3D	Barnes	0.35	0.44	0.37	0.44
	MP3D	0.30	0.22	0.32	0.21
Barnes-FLO52-FLO52	Barnes	0.21	0.31	0.24	0.29
	FLO52	0.30	0.41	0.53	0.46
Barnes-MP3D-MP3D	Barnes	0.18	0.28	0.23	0.36
	MP3D	0.55	0.44	0.63	0.36
Barnes-FLO52-MP3D	Barnes	0.23	0.31	0.23	0.29
	FLO52	0.14	0.20	0.27	0.23
	MP3D	0.31	0.23	0.33	0.26

Table 1: Job Throughput Rates (jobs/ minute).

number exceeds the number that can be used profitably by the other two representative jobs in our mixes. For this reason, as well as space limitations, we therefore omit these results in what follows.

Figure 6 presents the experimental results we obtained. From them, we draw conclusions similar to those in Section 3.3.2.

- *Policies using runtime measurements can greatly outperform those without access to such information.*

  This is supported by comparing the results for EQUAL-EFF and ST-EQUAL-EFF scheduling to those for EQUI.

- *Policies using runtime measurements can approach the performance of policies with access to perfect information a priori.*

  For all workloads, the performance of the equal efficiency policies is within 20% of those for the overly optimistic OPT-EFF.

- *The primary policy-induced loss of efficiency is the cost associated with the allocation search procedure.*

  In the cases where EQUAL-EFF and ST-EQUAL-EFF fail to approach closely the (theoretical) optimal performance of OPT-EFF, further examination revealed that it was because of overhead associated with the search procedure, rather than because the search was settling on poor final allocation choices.

4.4 Starvation

Because the EQUAL-EFF policy attempts to maximize throughput without regard to fairness, it is natural to wonder if jobs with poor speedup characterstics are starved under this discipline. Table 1 shows the job throughput rates (in jobs/minute) for each job class under our test workload mixes. If starvation were a problem in practice, we would expect to see sharp drops in throughputs for FLO52 and MP3D when comparing an equal efficiency policy to an equipartition policy. The fact that this does not happen is a reflection of the equal efficiency policies' guarantee that every job be given at least one processor.

5 Conclusions

Our goal in this paper was to determine if parallel processor allocation policies could be built to exploit runtime measurements of application performance. If so, such policies could replace a reliance on *a priori* specification of job characteristics, a troublesome and error prone task.

For a number of reasons, it was not obvious whether runtime measurements would be useful to parallel schedulers: it was not clear how to obtain those measurements; it was uncertain if recent measurements would be good indicators of future behavior; measures taken for an application with an allocation of p processors are not easily interpreted when considering changing the allocation to q processors; and the use of runtime measures requires dynamic allocation schemes, and their concomitant reallocation overheads.

We have formulated policies for both interactive and batch oriented parallel environments that make use of information obtained by runtime measurement of application characteristics. Given the convenience of these policies for the user of the system, their resilience to changes in program behavior due to phase changes within a single run or to changes in datasets between runs, their good performance, and the evidence of our prototype that practical implementations are possible, we believe that the availability of runtime measurements is an important factor to be considered in parallel processor allocation policy design.

References

[1] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[2] T. B. Brecht and K. Guha. Using Parallel Program Characteristics in Dynamic Processor Allocation Policies. Technical report, Department of Computer Science, York University, in preparation.

[3] S.-H. Chiang, R. Mansharamani, and M. Vernon. Use of Application Characteristics and Limited Preemption for Run-to-Completion Parallel Processor Scheduling Policies. In *Proceedings of ACM SIGMETRICS Conference*, pages 33–44, May 1994.

[4] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.

[5] L. Dowdy. On the Partitioning of Multiprocessor Systems. Technical report, Vanderbilt University, June 1988.

[6] D. L. Eager and J. Zahorjan. Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing. *ACM Transactions on Computer Systems*, 11(1):1–32, Feb. 1993.

[7] D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In *Job Scheduling Strategies for Parallel Processing, IPPS '95 Workshop*, pages 337–360. Springer, Apr. 1995.

[8] D. Ghosal, G. Serazzi, and S. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessors. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[9] T. E. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors Through Compile-Time Analysis. In *Proceedings of the 7th SIGPLAN Symposium on Principles and Practice of Parallel Processing*, Sept. 1995.

[10] Kendall Square Research Inc., 170 Tracer Lane, Waltham, MA 02154. *KSR/Series Principles of Operation*, 1994.

[11] S. Majumdar, D. Eager, and R. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of ACM SIGMETRICS Conference*, pages 104–113, May 1988.

[12] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the ACM SIGMETRICS Conference*, pages 104–113, May 1988.

[13] E. Markatos and T. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.

[14] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Strategy for Multiprogrammed, Shared Memory Multiprocessors. *ACM Trans. on Computer Systems*, 11(2):146–178, May 1993.

[15] G. P. McCormick. *Nonlinear Programming*. John Wiley & Sons, Inc., 1983.

[16] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup Through Self-Tuning of Processor Allocation. Technical Report UW-CSE-95-09-02, Department of Computer Science and Engineering, University of Washington, Sept. 1995.

[17] T. D. Nguyen, R. Vaswani, and J. Zahorjan. On Scheduling Implications of Application Characteristics. Technical report, Department of Computer Science and Engineering, University of Washington, in preparation.

[18] E. W. Parsons and K. C. Sevcik. Multiprocessor Scheduling for High-Variability Service Time Distributions. In *Job Scheduling Strategies for Parallel Processing, IPPS '95 Workshop*, pages 127–145. Springer, Apr. 1995.

[19] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, Dec. 1987.

[20] S. Setia, M. Squillante, , and S. Tripathi. Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Systems. In *Proceedings of ACM SIGMETRICS Conference*, pages 158–170, May 1993.

[21] K. Sevcik. Characterizations of Parallelism in Applications and Their Use in Scheduling. In *Proceedings of ACM SIGMETRICS Conference*, pages 171–180, May 1989.

[22] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation*, 19(2-3), 1994.

[23] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[24] P. G. Sobalvarro and W. E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Job Scheduling Strategies for Parallel Processing, IPPS '95 Workshop*, pages 106–126. Springer, Apr. 1995.

[25] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166, December 1989.