Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers [*]

Cathy McCann and John Zahorjan

Technical Report #94–10–05

October 27, 1994

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington, U.S.A. 98195

[*]This material is based upon work supported by the ARPA Fellowship for High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland (MDA972-92-J-1017), the National Science Foundation (Grants CCR-9123308 and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation Systems Research Center and External Research Program. Authors' addresses: Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195; *mccann@cs.washington.edu, zahorjan@cs.washington.edu.*

Abstract

While the parallel use of many processors is a major attraction of scalable multiprocessors for large applications, another important feature of such machines is the large amount of physical memory they make available. Despite these resources, truly large applications may be limited not only by the amount of computation they require, but also by the amount of data they must operate on to solve problems of interest.

In this paper we consider the problem of multiprocessor scheduling of jobs whose memory requirements place lower bounds on the fraction of the machine required in order to execute. We address three primary questions in this work:

1. How can a parallel machine be multiprogrammed with minimal overhead when jobs have minimum memory requirements?
2. To what extent does the inability of an application to repartition its workload during runtime affect the choice of processor allocation policy?
3. How rigid should the system be in attempting to provide equal resource allocation to each runnable job in order to minimize average response time?

1 Introduction

Much of the work on scheduling policies for multiprogrammed multiprocessors has focused on how many processors to allocate to each runnable job without considering the memory requirements of those jobs [7, 14, 15, 6, 17, 18, 13, 8, 2, 9, 16]. In this paper we consider jobs whose memory requirements imply a lower bound on the amount of machine resource they can be allocated for execution.

The interaction of processor scheduling and job memory usage has been considered in [12]. However, they examined a paging environment, with the intent of exposing the influence that paging traffic has on performance. Unlike that work, we make the more conventional assumption that because parallel applications requiring the use of tightly coupled machines synchronize often, the asynchronous blocking of individual threads induced by a paging system would lead to unacceptable performance. Thus, to run efficiently, jobs must be loaded in physical memory in their entirety.

While most previous work in this area has considered *static* disciplines, disciplines that do not alter the number of processors allocated to a job after an initial decision has been made, we consider *dynamic* disciplines, meaning that the number of processors allocated to a job may be changed during its execution. Static disciplines can suffer from significant inefficiences following job terminations, since the processors released by the terminating job cannot be assigned to any running job. Even if there are jobs waiting to execute, the released processors will remain idle if all waiting jobs require more machine resource than has been freed[1]. Dynamic disciplines avoid this problem, since they can reallocate idle processors to any running job.

There are two problems associated with dynamic disciplines that have discouraged their use. One is that changing allocations involves overhead. Because our disciplines change allocations only on job arrival or departure, relatively rare events, this problem does not seem crucial. We do not consider the impact of this overhead in detail here, but instead undertake the first step of defining reasonable dynamic policies and examining their properties. A next step, left for future work, is to examine in detail the impact of this overhead, and ways of reducing it.

The other problem associated with dynamic policies is that applications may not be able to respond to changes in their allocations. In particular, if an application is capable of partitioning of its data and computation into pieces only at load time (or even earlier), allocation of an incompatible number of processors can significantly degrade application efficiency [15, 9]. We do consider this problem in detail in examining our policies.

[1] In fact, static disciplines also face possibly severe problems at job arrivals. For instance, if a job arrives to an idle system and is allocated the full machine, this will have a negative impact on jobs arriving soon after it. If some portion of the machine is reserved for such arrivals, that portion goes unused if no arrivals occur.

In Section 2 we describe the hardware and software we consider, and give a precise statement of the problem we address. Sections 3 and 4 define members of two classes of dynamic policies, and show some of them to be optimal within their classes for a measure of the overhead they impose. Section 5 gives simulation results comparing the policies in terms of policy induced overhead and job average response time, for jobs that can and jobs that cannot repartition dynamically in response to changes in allocation. Section 6 summarizes our conclusions.

2 Problem Environment and Definition

2.1 The Hardware and Software Environments

Because we are interested in scalable multiprocessors, we adopt a distributed memory hardware model typical of current and envisioned parallel machines in that class. Figure 1 depicts such a machine. The important characteristic of these machines to our work is that associated with each processor is a local memory module that can be accessed much more efficiently than any other memory module in the machine. While not all machines have a one-to-one mapping of memory modules to processors (e.g., [5]), the notion of local and remote memories is fundamental to nearly all.

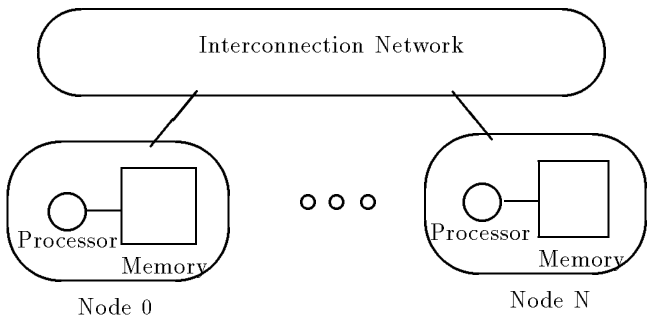


Figure 1: *Hardware Environment*

Because access to remote memories is inefficient on shared-memory machines and impossible on message-passing machines, the association between memory modules and processors implies that a job's memory requirement can be viewed as a lower bound on the number of processors it may be allocated. Thus, we phrase our policy decisions in terms of the number of *nodes* allocated, where a node is a processor-memory pair.

The workload we consider is representative of scientific applications. These applications have high computational requirements, and so run for extended periods (e.g., minutes or hours). For these workloads, response time is the most important measure of performance. Minimizing response time is the goal of the policies we consider. Another important class of applications, those representative of transaction processing systems, is not considered here. It is not unlikely that different disciplines would be appropriate in that environment.

As with most previous work in this area, we do not consider the influence of application IO on scheduling decisions. Because the proper integration of IO into parallel machine hardware is by no means well understood, it seems premature to consider this aspect of the problem at this time. Also, because we consider only how many nodes to allocate, and not which to allocate[2], our policies should be applicable as long as IO bandwidth is spread reasonably uniformly throughout the machine.

[2]Our policies are compatible with the lower level policies in [9], however. Those policies address the question of which nodes to allocate.

One assumption we will make about IO hardware, however, is that it is highly parallel. This seems like a reasonably conservative assumption about successful future multiprocessors.

2.2 Scheduling Principles

Our goal is to design resource allocation policies for distributed memory parallel machines that minimize average response time[3]. To better identify the promising approaches to scheduling distributed memory multiprocessors, it is instructive to examine established results on processor scheduling for both sequential and parallel machines. From the sequential world we take the lessons of Shortest-Job-First scheduling, and the notion that because reliable information on job duration cannot be obtained *a priori*, some runtime approximation must be applied. The overwhelming choice in sequential systems is multi-level feedback scheduling, which consists of two primary mechanisms. First, all jobs considered likely to have similar times to completion are allocated equal machine resources. Because there is only a single processor, this is achieved through time-sharing (i.e., round-robin scheduling). Second, because it has been observed that the remaining time to completion of a job is very strongly correlated with the processing time it has already consumed [4], jobs move down in priority level as they consume resources, and round-robin scheduling is applied only to the set of jobs in the highest non-empty priority level.

Similar results have been obtained for parallel machines, primarily in the context of shared-memory systems [15, 6]. For parallel machines, however, *space-sharing*, in which resources are divided equally by partitioning the processors among the jobs, has proven to be more effective than time-sharing the machine by rotating all resources among the jobs in a round-robin fashion [15, 17]. Among the benefits of space-sharing is that it interacts well with *co-scheduling*, which requires that either all processes of a single parallel application be scheduled at once or that none be scheduled. Co-scheduling has long been known to be critical to parallel machine scheduling [11].

Based on these results, we consider only policies that provide co-scheduling and that use space sharing as the primary mechanism to approach equal resource allocation. We will also assume that the jobs in the set to be scheduled have similar remaining execution time characteristics, as far as the kernel can tell. (For example, they might be the set of jobs currently residing at the highest non-empty priority level in a feedback scheme.) While there has been interest in how to use job-supplied information on execution characteristics in making scheduling decisions [7, 14, 2], this approach runs contrary to what has proven successful in sequential systems, and relies on the user to be both reliable at estimating job characteristics and honest in relating them to the system.

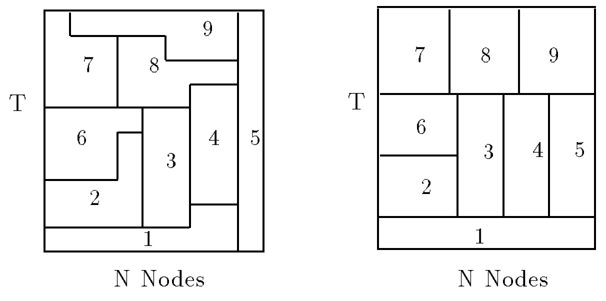
Because we allow jobs to have minimum node requirements, it is not possible for our policies to avoid time-sharing altogether, since it may not be possible to schedule all jobs at once. While it is certainly possible to design run-to-completion policies that load only a subset of the jobs and run each until it completes [2], this approach violates the principle of equal resource allocation required to approximate shortest-job-first, and there is strong evidence that this results in performance inferior to policies that instead employ some amount of time-sharing [2, 9].

2.3 Problem Definition

The system we consider is defined by three parameters: N, the number of nodes; J, the number of jobs; and $\vec{M} \equiv (M_1, M_2, ..., M_J)$, the vector of job minimum node requirements. We introduce an additional parameter, T, the length of a scheduling interval. We require acceptable policies to schedule every job during each interval of length T. In practice, the duration T would be set so that the overhead of time-sharing the machine would be acceptably low. Because in our model T is the only time related parameter, we simply let it be the unit of time, i.e., we let $T = 1$.

Figure 2(a) illustrates the general scheduling problem, which can be pictured as partitioning a $Nx1$ rectangle into pieces and assigning each piece to a job, under the constraints that no job j is assigned any

[3]A secondary issue in designing scheduling disciplines is fairness. While we do not address this issue directly here, the policies we propose naturally allow fair service among the jobs that are eligible for execution.



(a) General Scheduling Problem (b) Restricted Scheduling Problem

Figure 2: *The Scheduling Problem*

piece of width less than M_j and that each job is allocated pieces whose total area is N/J. (The area of each piece is measured in node-seconds, and represents a resource allocation. Thus, insisting that equal total areas be allocated to each job meets the goal of equal resouce allocation.)

Because reallocating nodes involves writing the current contents of their local memories to disk and loading the previously stored contents of the next job to be scheduled there, such reallocations can be fairly expensive. (Remember, however, that the impact of these reallocations can be made arbitrarily small by lengthening the scheduling quantum, T, although at the risk of increasing average response time through an overly FCFS-like policy.) We thus look for policies that minimize this reallocation overhead.

While Figure 2(a) illustrates the general problem, we restrict our attention to policies of the sort shown in Figure 2(b), which invoke each job only once per scheduling quantum and do not change the allocation to any job during a quantum. The desirability of these restrictions seems compelling, given the cost of reallocating nodes and the difficulty that individual jobs may have in dealing with allocations of varying sizes. Thus, the policies we consider change the allocation to an individual job at most at instants when some job arrives to or departs from the system.

Let A_j be the width of the subrectangle assigned to job j, i.e., the number of nodes on which it runs when in execution. We define the overhead of a schedule to be $\sum_{j=1}^{J} A_j$, and look for policies that minimize this overhead. We note that minimizing overhead is equivalent to minimizing the average number of processors allocated to each job, which should have a beneficial impact on system utilization (and thus job response time) because individual jobs nearly always run more efficiently on smaller numbers of processors than on larger numbers.

3 The BUDDY Scheduling Policy

This section presents a scheduling policy, called BUDDY, that is optimal for the restricted class of problems in which the number of nodes and the number of jobs are powers of two, and among the restricted class of policies that make only allocations that are powers of two. Figure 3(a) gives an example of the kind of schedule produced by BUDDY. We describe below how such schedules are computed.

BUDDY fills the $Nx1$ scheduling rectangle from bottom to top. A *partial schedule* (see Figure 3(b)) is the set of assignments made at some intermediate point during the scheduling process. The *frontier* of a partial schedule is the set of (horizontal) line segments below which the $Nx1$ scheduling rectangle is filled and above which it is empty.

Crucial to the operation of BUDDY is the observation that the height of each subrectangle is a (non-positive) power of two. Remembering the restricted problem domain to which BUDDY applies, let $N = 2^n$, $J = 2^j$, and the number of processors allocated to job z be a_z. Since each job is allocated an equal percentage of the processing resources, it must be given a subrectangle of area $N/J = 2^{n-j}$. Thus, job z will be allocated

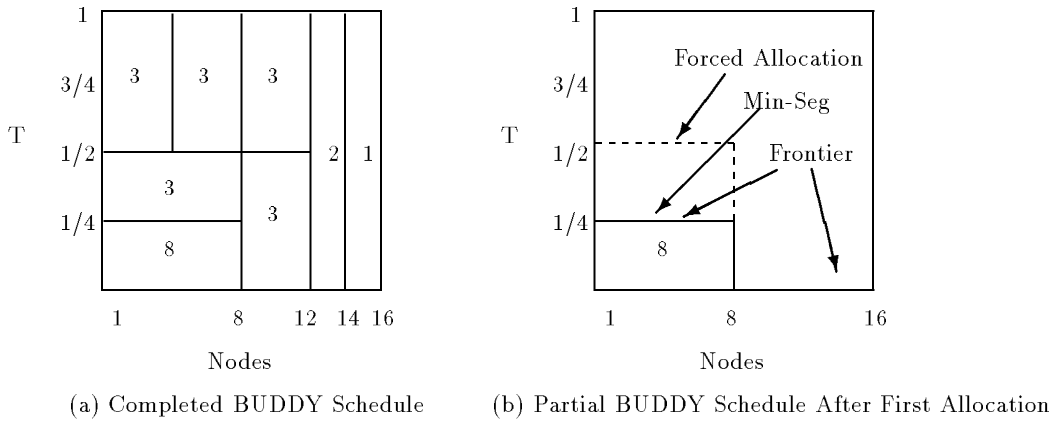


Figure 3: *BUDDY Schedule for* $N = 16$,$J = 8$, $\vec{M} = [1,2,3,3,3,3,3,8]$

2^{a_z} nodes for duration 2^{n-j-a_z}.

The fact that each individual subrectangle allocated has height equal to a negative power of two implies that the height of a segment of the frontier (which is the sum of the heights of the subrectangles below it) can be represented as an irreducible fraction $C/2^x$, for C odd (or zero) and $x \geq 0$. (By definition, the frontier before any jobs have been allocated subrectangles has height $0/2^0$.) Lemma 1 uses this fact in defining a property of all valid schedule completions that forms the basis for the BUDDY policy.

Lemma 1 *For each frontier line segment of height $C/2^x$, for some odd C and integer $x \geq 0$, at least one allocation above the line segment must have an allocation of height less than or equal to $1/2^x$.*

In effect, this property states that for any height of a partial schedule, there is an upper bound on the shortest subsequent allocation height to be added above that height. Thus, there is a corresponding lower bound on the largest node allocation to be placed above that point on the graph.

BUDDY assigns subrectangles to jobs in non-increasing order of their minimum node requirements, using the property expressed in Lemma 1 to decide how many processors to allocate to each . Let the *min-seg* be the (provably unique) segment of the current frontier whose height, $C/2^x$, has the largest exponent in the denominator. BUDDY always allocates the next job a subrectangle that is placed above the min-seg, aligned with its left edge. The width of the allocated piece is the minimum of the smallest power of two no smaller than the job's minimum node requirement and 2^{n-j-x}, the minimum allocation "forced" by the min-seg's height.

Figure 3(b) shows the partial schedule obtained for its example system after a single job has been allocated. At this point the min-seg is at height $1/2^2$, and has width 8. The min-seg forces the next allocation to have height no greater than $1/2^2$, corresponding to a width of 8. In this particular case, the next job which has minimum requirement 3, is allocated 8 processors, even though it's self-imposed lower bound is 4. Figure 3(a) shows the final schedule.

3.1 Analysis of BUDDY

BUDDY runs in time $O(J \log J)$: each of J jobs must be scheduled, and for each the up to J segments of the current frontier must be searched to find the min-seg. Thus, BUDDY executes efficiently.

The quality of BUDDY's schedules is capture by Theorem 1, which, due to space limitations, we state here without formal proof. (The reader is referred to [10] for the complete proof.)

Theorem 1 *The BUDDY algorithm produces an optimal schedule under the restrictions that N, J, and allocations A_j for all jobs j are powers of two, and that all jobs are allocated equal resources.*

While the formal proof that BUDDY is optimal is quite long, its intuitive justification is relatively simple. BUDDY makes each allocation in a way that minimizes the widths of the pieces forced by the frontier segments in the current partial schedule. It does so by allocating each new pieces above the min-seg, and by choosing its width to be the smallest possible width that respects both the job's minimum requirement and what is forced by the min-seg. The key to the formal proof is to show that when BUDDY is forced to allocate a piece wider than the job's minimum, so would be any hypothetical optimal schedule. While in general one would not expect a greedy algorithm like BUDDY to be optimal, the restrictions of the problem domain to which it applies make the currently most conservative decision the best global strategy.

3.2 Removing the Restrictions of BUDDY

BUDDY requires that N and J be powers of two, and is provably optimal among schedules whose allocation widths A_j are powers of two. It is natural to ask if these restrictions are necessary.

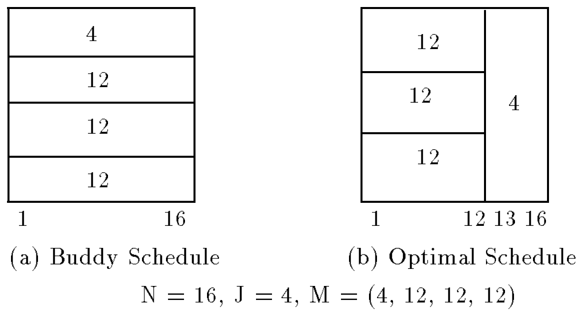


Figure 4: *Example Showing BUDDY Is Not Optimal When* $A_j \neq 2^{a_j}$

Figure 4 shows that BUDDY is not optimal if the restriction on allocation widths is removed. Finding an optimal policy for this situation seems difficult, and we have not yet pursued that question.

Similarly, the restriction that N be a power of two is deeply embedded in the BUDDY policy, and it appears difficult to make any significant extension along that line.

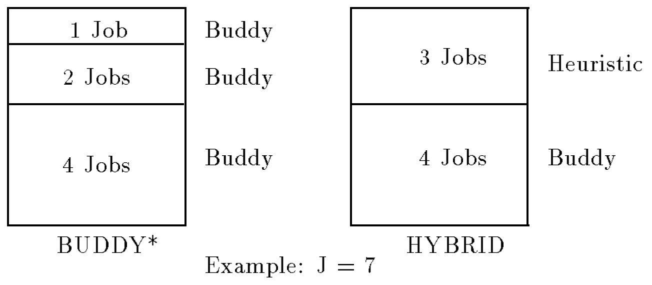


Figure 5: *Heuristic Extensions to BUDDY For* $J \neq 2^j$

While finding an optimal policy for general J is also difficult, BUDDY lends itself naturally to heuristic extensions in this dimension. Figure 5 shows two approaches we have considered. The first, BUDDY*, is simply to partition the J jobs into subsets such that each subset contains a power of two number of jobs, to obtain the BUDDY schedule for each subset, and then to concatenate these schedules (after appropriately rescaling their time dimensions) to form an overall schedule. We place jobs in sorted order by increasing minimum node requirement into subsets of decreasing numbers of jobs, in an attempt to minimize overhead.

BUDDY* has the advantage that each subset is scheduled optimally, but it has the significant disadvantage that the number of subsets is determined by J rather than by the memory requirements of the jobs. Since each subset requires at least N reallocations, this property implies that values of J with many one's in their binary representations may have unnecessarily high reallocation costs.

To reduce this effect, we define the HYBRID policy. A HYBRID schedule is obtained by concatenating the BUDDY schedule obtained for the largest power of two number of jobs no larger than J with a schedule obtained using a suboptimal scheme that is applicable to an arbitrary number of jobs. (We define a number of such policies in the next section.) We place the jobs with the smallest minimum node requirements into the set scheduled by BUDDY, since BUDDY is relatively more advantageous for them than it is likely to be for the larger jobs.

Both BUDDY* and HYBRID are equivalent to BUDDY when J is a power of two. We consider all three disciplines in the performance comparisons of Section 5.

4 Epoch Scheduling

In this section, a restricted class of scheduling policies, called epoch policies, is considered. Figure 6 shows an example epoch schedule. Under epoch polices, all nodes are reallocated at each reallocation moment. The time between successive reallocation instants is called an epoch.

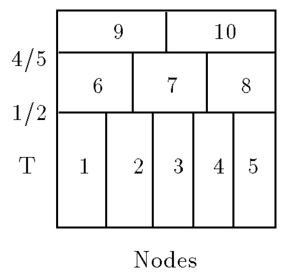


Figure 6: *An Example Epoch Schedule for 10 Jobs*

It is apparent that an epoch schedule always exists for any values of N, J, and $\vec{M}$, since the time-sharing policy of rotating the entire machine from one job to the next is an epoch policy. Our goal is to find good (and, if possible, optimal) epoch policies. As before, schedule optimality is defined as a minimization of the total number of node reallocations that take place in each scheduling quantum. For the class of epoch scheduling policies, this is equivalent to minimizing the number of epochs.

We consider a number of different epoch policies. We begin with a policy that efficiently determines an optimal epoch schedule under the constraints that N is a power of two and that all jobs are allocated exactly equal resources each scheduling quantum. We then generalize this policy to remove the restriction on N and at the same time to allow bounded inequality in allocations. Computing schedules under this (still optimal) policy requires exponential time in the worst case. Our final policy is a heuristic derived by applying the ideas of the optimal policy in a greedy way.

4.1 The EQUI-EPOCH Policy

EQUI-EPOCH is an epoch policy that allocates equal resources to each job. In the epoch class of policies, this implies that an equal number of nodes must be allocated to all jobs in any single epoch, although the numbers of nodes allocated can vary from epoch to epoch.

EQUI-EPOCH is applicable to all problem parameters N and J. It uses a greedy approach to fill epochs. Jobs are sorted from to largest minimum node requirement, and then considered in order. As many jobs as possible are placed in the current epoch, under the restrictions that each job is allocated exactly the same number of nodes. When the epoch is filled in this way, a new epoch is begun. When all jobs have been assigned to epochs, each epoch is given a duration that is proportional to the number of jobs it contains.

4.1.1 Analysis of EQUI-EPOCH

EQUI-EPOCH runs in time $O(J\sqrt{N})$. For each of up to J epochs we must find the largest number of jobs that fit in the epoch with equal allocations. Since that number must divide N, and since we are considering jobs in strict order of non-decreasing minimum node requirement, there are up to $\sqrt{N}$ possibilities to be tested.

We state here the major result regarding the quality of the schedules produced by EQUI-EPOCH. The reader is referred to [10] for the details of the proof.

Theorem 2 *When $N = 2^n$, for some integer $n \geq 0$, EQUI-EPOCH is optimal among epoch scheduling policies that guarantee equal resource allocations.*

Informally, EQUI-EPOCH is optimal in this case because it is never advantageous to schedule fewer jobs in the epoch under consideration than will fit there. This property holds because the set of jobs remaining to be scheduled under EQUI-EPOCH *dominates* the sets remaining under any other possible decision, in the sense that the set corresponding to EQUI-EPOCH can be scheduled in no more epochs than those of any other possible set.

To help clarify this key property, we present two similar scenarios in which a greedy algorithm is non-optimal.

4.1.2 EQUI-EPOCH Is Not Optimal When $N \neq 2^n$

If N is not restricted to a power of two, the EQUI-EPOCH scheduling policy is not optimal. To illustrate this, consider scheduling $N = 140$ nodes. Assume there are 10 jobs, each can run with no fewer than 20 nodes each. Figure 7 shows the EQUI-EPOCH and the optimal schedules in this case. The major fault with the EQUI-EPOCH scheme in this case is that the three jobs remaining after the first epoch is filled necessarily require two epochs in order to achieve equal allocation (independently of the minimum node requirements of the remaining jobs). By leaving a number of jobs unscheduled that evenly divides N, the optimal schedule is able to place the remaining jobs in a single epoch.

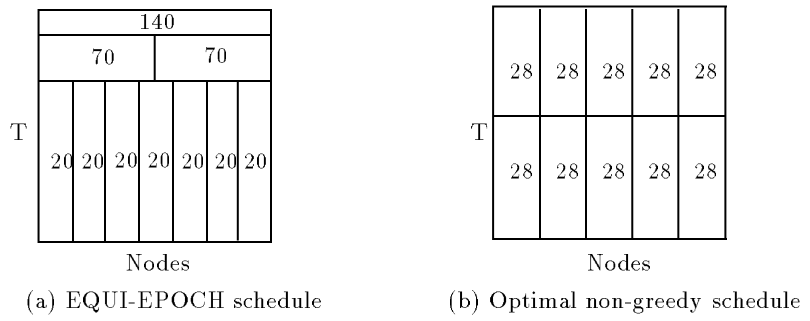
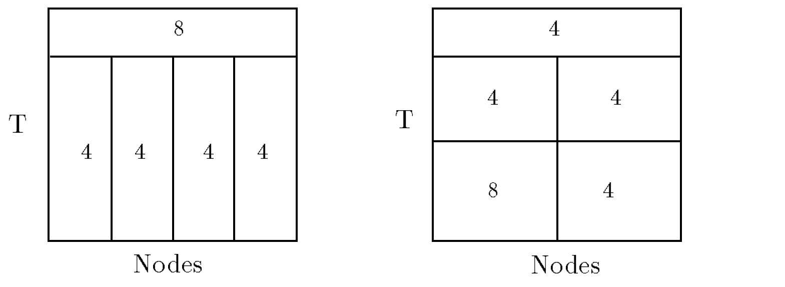


Figure 7: *EQUI-EPOCH versus Optimal Epoch Policy When $N \neq 2^n$* ($N = 140$, $J = 20$, $M_j = 10$ for all j)

4.1.3 Non-optimal Greedy Algorithm

We note that there is a greedy algorithm similar to EQUI-EPOCH, but starting with the jobs with the largest minimum node requirements instead of the smallest. However, as shown in Figure 8 such a policy is not optimal. Once again, the problem is that the set of jobs remaining after an epoch is filled in a greedy manner may not dominate other possible remaining sets.



(a) EQUI-EPOCH schedule (b) Non-optimal greedy schedule

Figure 8: *EQUI-EPOCH versus Non-optimal Greedy Epoch Policy* ($N = 16$, $J = 5$, $\vec{M} = [4, 4, 4, 4, 8]$)

4.2 The OPT-EPOCH(k) Policy

OPT-EPOCH(k) is an optimal epoch policy for general N that allows a limited amount of inequity in the resource assignments to each job. The inequity is measured as the maximum permissible difference in the number of nodes allocated to any two jobs in a single epoch, and is given by the parameter k of the policy. For instance, OPT-EPOCH(0) results in schedules with perfectly equal resource allocations, while OPT-EPOCH(2) would allow epochs with allocation widths of size c, $c + 1$, and $c + 2$ (for some $c > 0$).

By varying parameter k we are able to evaluate the extent to which allowing increasing amounts of inequity improves or degrades performance. Choosing $k > 0$ has the advantage that the number of jobs inserted into an epoch need not evenly divide the number of nodes, N, a shortcoming of the exactly equal allocation policies. For instance, when $J = 7$ and N is a power of two, any schedule produced by OPT-EPOCH(0) must contain at least three epochs. In contrast, it is possible that an OPT-EPOCH(1) schedule could contain only a single epoch. On the other hand, large values of k may lead to very unequal resource allocations, which could be detrimental to performance.

OPT-EPOCH(k) schedules are constructed by performing a recursive evaluation of all possible schedules. This procedure is obviously very expensive. In practice, it can be made more efficient by exploiting a few properties of the problem at hand.

Consider enumerating all possible vectors of h positive integers whose sum is N, for $h = 1, ..., N$. To enumerate all possible schedules, we must recursively try filling each such partitioning with each possible subset of the as yet unscheduled jobs. However, it is clear that when filling a particular partitioning, it is advantageous to place the largest possible jobs into that partitioning, since this leaves the most easily scheduled set of remaining jobs. (All possible remaining sets have the same number of jobs in them, and any set of jobs that is pairwise no smaller than another set of equal size can be scheduled in no more epochs than that other set.) Thus, the search for the optimal schedule can be narrowed by eliminating choices at each step that leave remaining sets that are dominated by other possible remaining sets, in the sense of the pairwise comparisons above.

When $k \geq 1$, the same argument can be expanded to avoid examining potential epoch assignments that contain a subset of an already examined assignment, since the remaining set of jobs to be scheduled would be a superset of the other assignment's remaining set, and so could not be scheduled in fewer epochs. Combining these observations leads to a search that considers jobs in largest to smallest order, and possible assignments for the current epoch from largest to smallest number of jobs.

Our implementation exploits only a subset of the possible early search terminations. In particular, given a remaining set of jobs to be scheduled, we try possible assignments from those containing the most jobs to the fewest, and allocate the largest jobs possible in each. The search at this level is terminated whenever the jobs loaded are an uninterrupted, consecutive sequence starting with the largest remaining job.

While this procedure appears to have combinatorial worst-case complexity, it is quite efficient in practice, at least for $k = 1$. In 111 experiments on OPT-EPOCH(1) of 10,000 trials each, the average number of epoch assignments tested per epoch in the final assignment was 1.08, and the worst case average for any one of the experiments was 2.98. Thus, it appears that OPT-EPOCH(1) schedules can be reasonably computed. On the other hand, we did not extend our implementation to $k > 1$, because of the complexity of that implementation. Additionally, for increasing values of k it is likely that the early search termination criteria becomes less valuable, while the same time the number of possible epoch assignments increases dramatically. (Table 1 shows the number of valid partitions of $N = 128$ for various values of k.) Thus, to compute schedules that allow greater inequity than is possible under OPT-EPOCH(1), we used the heuristic technique described in the next subsection.

Allowed Inequality (k)	0	1	2	3	4	5	6
Number of Partitions of $N = 128$	8	128	2,144	21,527	144,055	692,693	2,560,378

Table 1: *Number of Valid Partitions of* $N = 128$ *Nodes versus Allowed Inequality*

4.3 HEURISTIC-EPOCH(k)

HEURISTIC-EPOCH(k) is a greedy implementation of the OPT-EPOCH(k) policy. Rather than look at every possible assignment at each level of the search, we simply choose the first one encountered in a specified search order. In particular, we add jobs to epochs from largest minimum requirement to smallest, while respecting the inequity limit k. If at any point a job is encountered that requires more nodes than are left unallocated in the epoch, and the number of unallocated nodes is within k of the largest allocation, that job is skipped and the largest job that will fit is placed in the epoch if one exists. If no such job can be found, the unallocated nodes are distributed as evenly as possible among the jobs in the epoch, with any excess allocated to jobs with the smallest total allocation.

HEURISTIC-EPOCH(k) has running time $O(J^2)$: for each of up to J epochs we must scan the list of jobs to find the maximum number that will fit.

4.4 Policy Summary

Table 2 summarizes the characteristics of the six policies we have defined. (The HYBRID(k) policy is formed by combining BUDDY and HEURISTIC(k), as explained in Section 3.2.) The columns marked "Restricted to" reflects the ability of the policy to determine any schedule for a given problem. The column marked "Optimal?" refers to optimality within the given policy's class.

5 Performance Comparisons

We compared the overhead and response time characteristics of our disciplines using a set of simulation experiments. For the purpose of these experiments, we define the *load factor* to be the number of jobs in the system times the average minimum node requirement of the jobs. We group result showing the effect of varying the basic problem parameters into sets that keep the load factor constant.

Policy	Restricted to $N = 2^n$?	$J = 2^j$?	Equal Allocations?	Optimal?	Efficient?
BUDDY	Yes	Yes	Yes	Yes	Yes
BUDDY*	Yes	No	Yes	No	Yes
HYBRID(k)	Yes	No	Partially	No	Yes
EQUI-EPOCH	No	No	Yes	If $N = 2^n$	Yes
OPT-EPOCH(k)	No	No	No	Yes	No
HEURISTIC(k)	No	No	No	No	Yes

Table 2: *Summary of Proposed Scheduling Policies*

Quantity	**Symbol**	**Default Value**
Number of Nodes	N	128
Number of Jobs	J	8
Load Factor	L	100%
Min. Node Requirement	M_j	Uniform($1 \ldots 2\frac{L*N}{J} - 1$)
Allowed Inequity	k	1

Table 3: *Default Model Parameter Values*

Table 3 gives the default parameters used in our experiments. In the results that follow, we vary a single parameter at a time. To keep the load factor constant when varying the number of jobs or the number of nodes, the average minimum node requirement was also adjusted as a side-effect.

We ran many more experiments than are included here, varying all the parameters above. (For example, we looked at results obtained when job minimum node requirements are chosen from a binomial distribution, as well as from the uniform distribution.) We present a subset of these results that we believe convey the essence of the comparisons of our policies.

In what follows, we use "BUDDY" to mean the BUDDY discipline when it applies, and the BUDDY* discipline otherwise.

5.1 Overhead

We compared the overhead of the policies, that is, the average number of node reallocations required during a scheduling quantum of length T, using Monte-Carlo simulation[4]. In each individual experiment, a random set of job minimum requirements was selected, and the jobs were then scheduled according to each discipline. We present results in terms of *normalized overhead*, which is the average number of reallocations each node experiences during a scheduling quantum.

Figure 9 shows how normalized overhead varies with the number of jobs in the system, for load factors of 100% and 200%. We see that BUDDY and EQUI-EPOCH react badly to numbers of jobs that must be broken into many subsets in order to divide N evenly. OPT-EPOCH(1) and HEURISTIC-EPOCH(1), which have no *a priori* restrictions on the number of jobs they can assign to a single epoch, are relatively unaffected by the value of J. HYBRID(1), which combines BUDDY and HEURISTIC-EPOCH(1), falls somewhere in between.

Figure 10 shows how overhead varies with number of nodes. Here the BUDDY-based algorithms offer clearly superior performance, as they enjoy a system with power of two numbers of jobs. The most noticeable other feature in the figures is the relatively regular oscillations of EQUI-EPOCH. These are caused by changes

[4]All simulations were run to achieve a 90% confidence interval no greater than 1% of the point estimate.

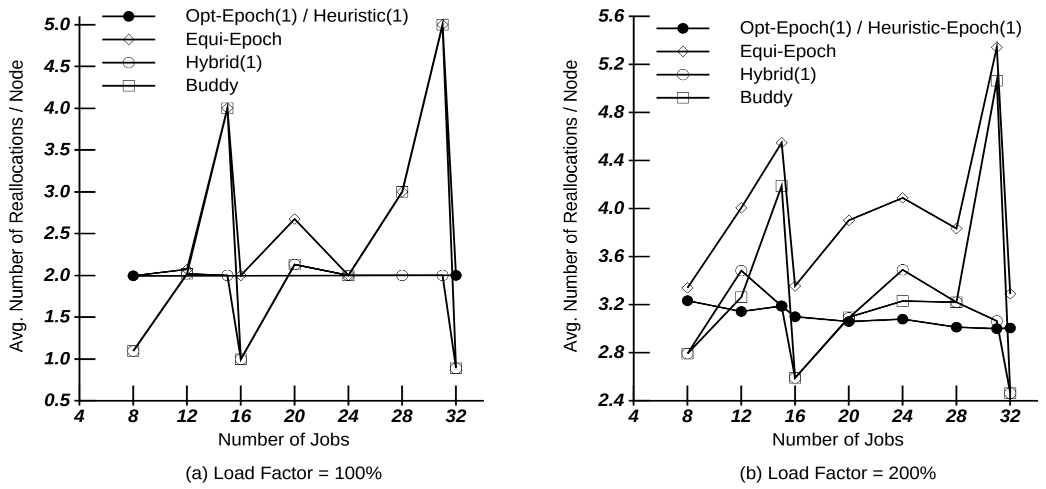


Figure 9: *Normalized Overhead versus Number of Jobs*

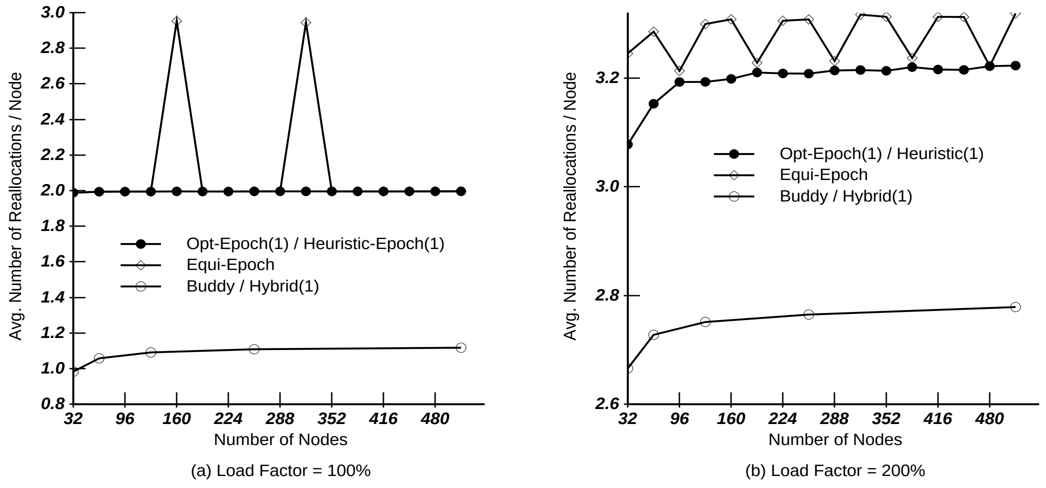


Figure 10: *Normalized Overhead versus Number of Nodes*

in the prime factors of the number of nodes, which determine the possible number of jobs in each epoch under this schedule. For example, the spike at $N = 160$ for load factor 100% results from sometimes placing the five smallest jobs in the first epoch, leaving three to schedule (and so necessarily requiring two more epochs). In these cases it would be better to allocate only four jobs to the first epoch, placing the remaining jobs in a single epoch. This effect illustrates why the optimality of EQUI-EPOCH rests on N being a power of two: a smaller remaining set of jobs does not necessarily dominate a larger set in the general case. As a heuristic for general N and J, it is probably better to allocate in a greedy manner starting from the largest jobs, as does HEURISTIC(k), rather than from the smallest, as does EQUI-EPOCH.

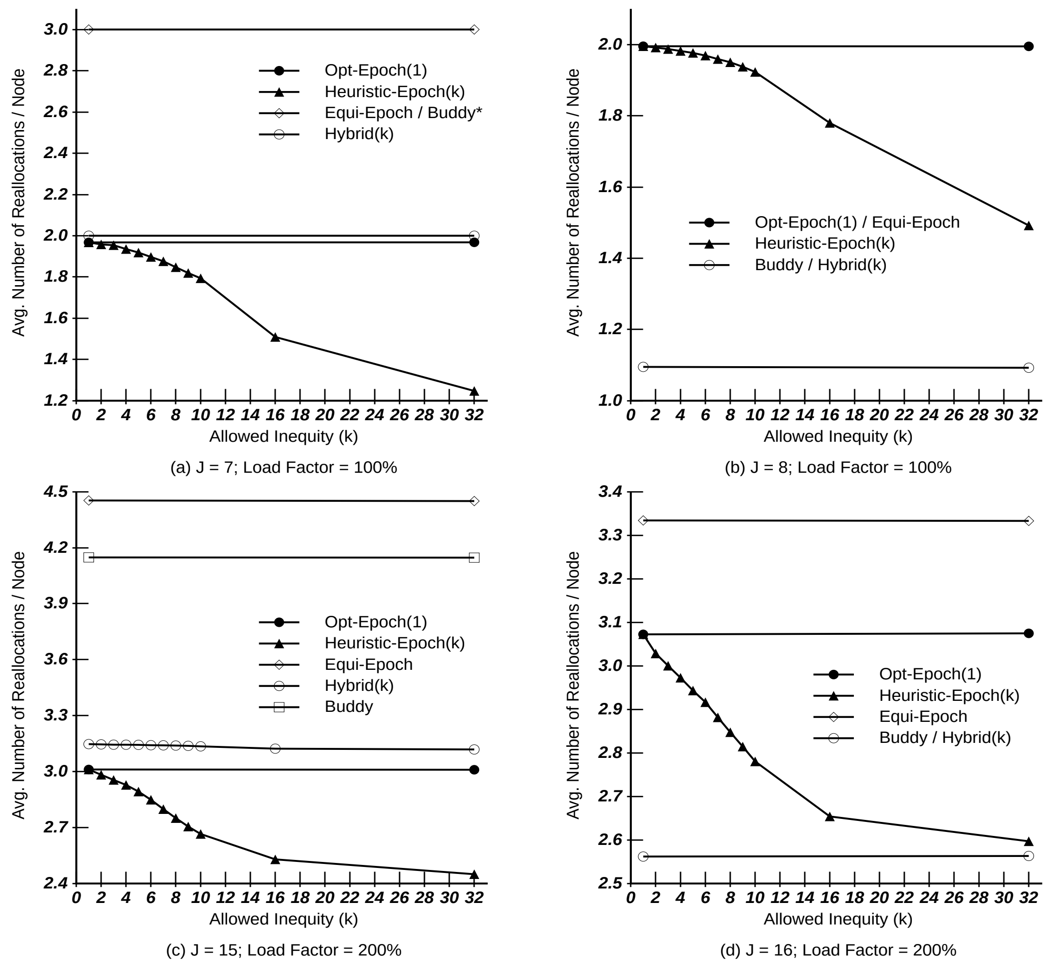


Figure 11: *Normalized Overhead versus Allowed Inequity (k)*

Figure 11 illustrates the benefit of allowing some inequity in allocation. The figure makes clear that increasing inequity, which increases the possibility that many jobs may be packed into a single epoch, is beneficial. It should be noted, however, that a major advantage in allowing inequity comes at $k = 1$, the point at which an arbitrary number of jobs may in theory be loaded into a single epoch. This is illustrated in the figures by comparing the performance of policies that insist on equal allocation to those that allow some inequity when J is not itself a power of two. For example, the large drop in overhead from EQUI-EPOCH to HEURISTIC(1) for $J = 7$ and $J = 15$ results from HEURISTIC's advantage in loading arbitrary numbers of jobs into epochs.

5.2 Response Time

To compare the average response time under each of the policies, we used simulations in which there were a fixed number of jobs: each time a job completed, another was started immediately. We used two job classes, short jobs (requiring 400 time units of CPU service total) and long jobs (requiring 4000 CPU time units). In each case, individual CPU service requirements were chosen from exponential distributions with the appropriate means. A new job had equal probability of being placed in each class. Because the results were not very sensitive to the length of the scheduling quantum, T, as long as T was smaller than the duration of the smaller jobs, we used 10 time units for this quantity in all results shown.

We express response times relative to those achieved under the OPT-EPOCH(1) discipline. OPT-EPOCH(1) was chosen as the basis for normalization because it is applicable to all values of J and N.

We divide our results into two sections, those pertaining to *static* applications and those pertaining to *dynamic* applications. An application is static if it is unable to repartition its data and computation into an arbitrary number of pieces from time to time during its execution. This is the common case in current software. An application is dynamic if it can perform this repartitioning function.

This distinction between static and dynamic applications is captured by the speedup functions used for them. Following [2], for dynamic applications we use a synthetic speedup function (taken from [3]) $S_{dynamic}(n) = (1+\beta)n/(\beta+n)$. The jobs in our workloads choose speedup parameter β uniformly from 30 to 300, the range considered in [2]. For static partitioning jobs, we assume that the jobs have been partitioned into N threads, so that they can make use of the full machine should it become available [9]. When n divides N, these jobs experience speedups equal to $S_{dynamic}(n)$. In general, though, their speedup is degraded by the load imbalance induced by allocating their N threads on an arbitrary number n of nodes, according to

$$S_{static}(n) = S_{dynamic}(n) * \left(\frac{\lfloor \frac{N}{n} \rfloor}{\lceil \frac{N}{n} \rceil}(n - N \bmod n) + (N \bmod n)\right)/n$$

Figure 12 shows $S_{static}(n)$ and $S_{dynamic}(n)$ when $N = 128$.

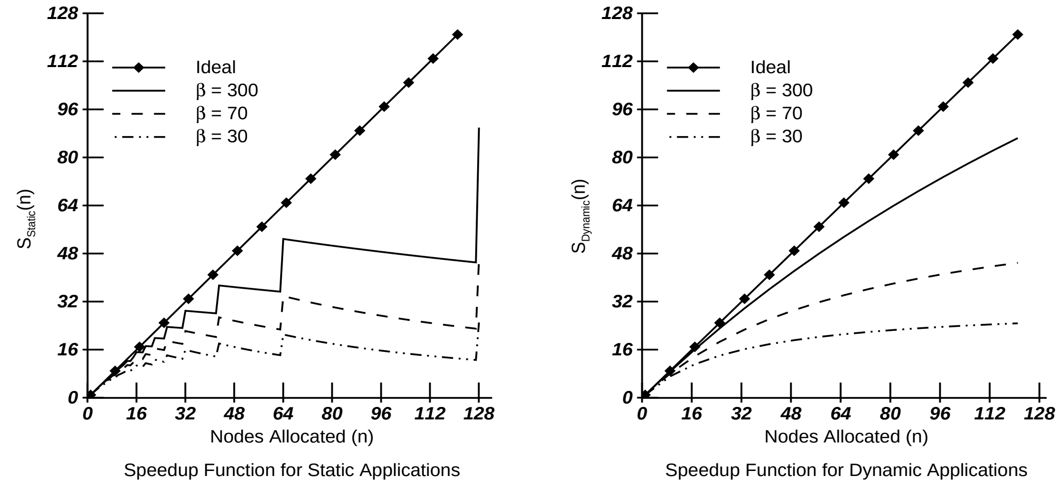


Figure 12: *Workload Speedup Functions (N = 128)*

5.2.1 Static Workloads

Figures 13-15 show the results obtained when jobs are unable to repartition dynamically. The overall lesson is that the disciplines that provide equal allocation perform substantially better than those that allow inequities. Equal allocation implies that each job is given a number of nodes onto which its N threads can be equally distributed, leading to high application efficiency. The penalty of allowing unequal allocations (and so allocations that do not necessarily evenly divide the number of threads of an individual application) grows with increases in the allowed inequity, because of the increased possibility of making these unfortunate decisions.

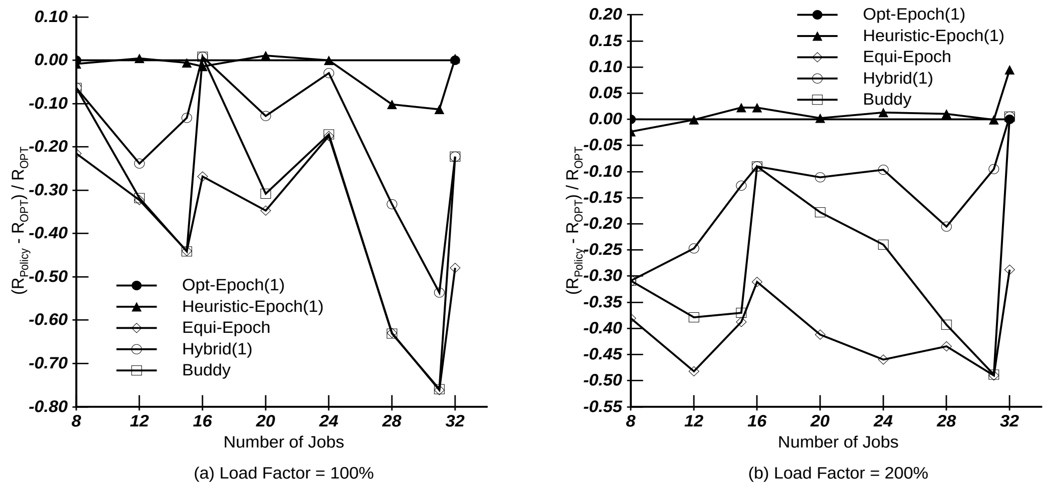


Figure 13: *Static Workloads: Relative Response Time versus Number of Jobs*

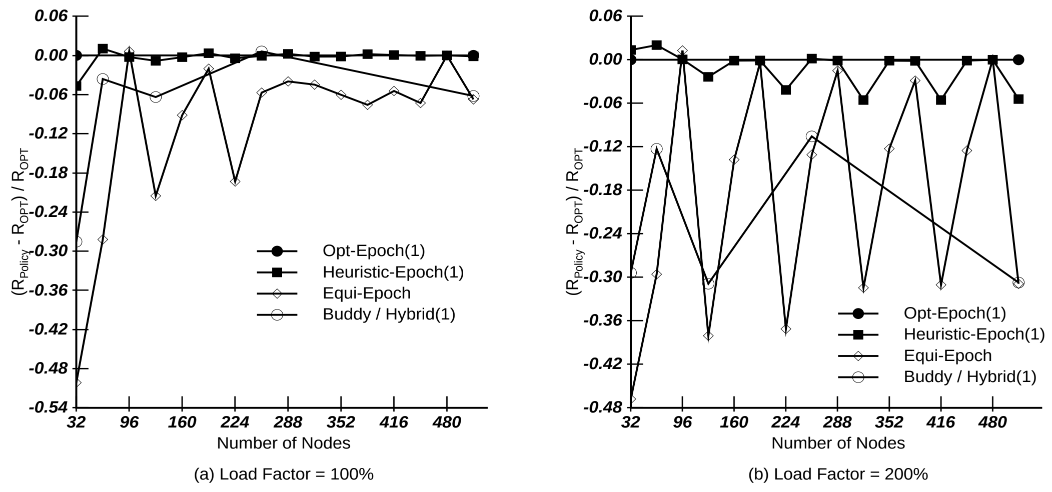
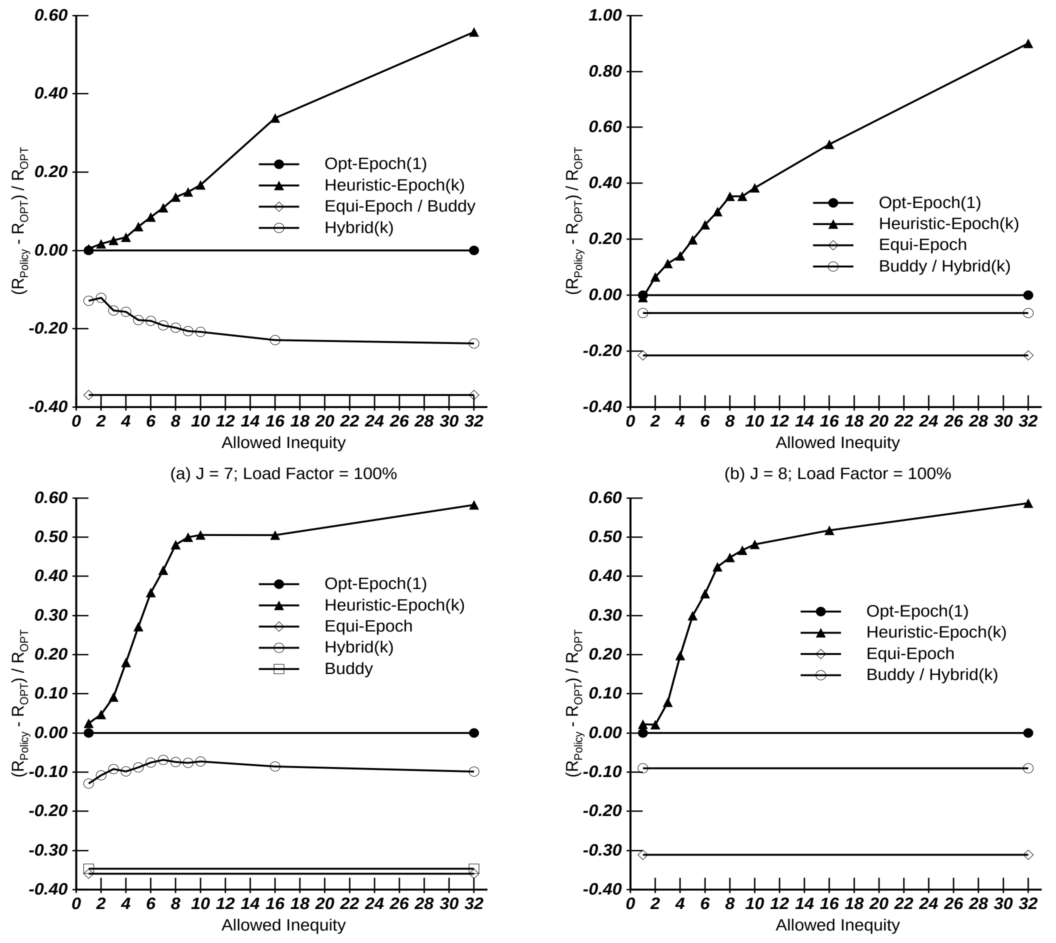


Figure 14: *Static Workloads: Relative Response Time versus Number of Nodes*

5.2.2 Dynamic Workloads

Figures 16-18 shows the results obtained when jobs can repartition dynamically. For these jobs, smaller numbers of allocated nodes lead to improved efficiencies in a monotonic way. In this case, policies that allow unequal assignment are in general preferable, since they can more nearly allocate each job it's minimum requirement. We note, however, that the range of differences among the policies for dynamic workloads is much smaller than that observed for static workloads: almost all distinctions for the former are under 10%, while for the latter the distinctions run as high as 90%.



(a) J = 7; Load Factor = 100%

(b) J = 8; Load Factor = 100%

(c) J = 15; Load Factor = 200%

(d) J = 16; Load Factor = 200%

Figure 15: *Static Workloads: Relative Response Time versus Allowed Inequity (k)*

The other lesson learned from these results is that absolute equality of service allocation is not required to minimize response times. (See Figure 18.) While it has been shown that extreme inequality (running only a subset of the jobs to completion) is detrimental to performance [2, 9], the potential disadvantage of unequal service from our disciplines is outweighed by the smaller allocations (and thus higher efficiencies) possible with them. This result is compatible with those of [1], which showed that the penalty of FCFS scheduling is substantially lower on multiprocessors than on sequential machines.

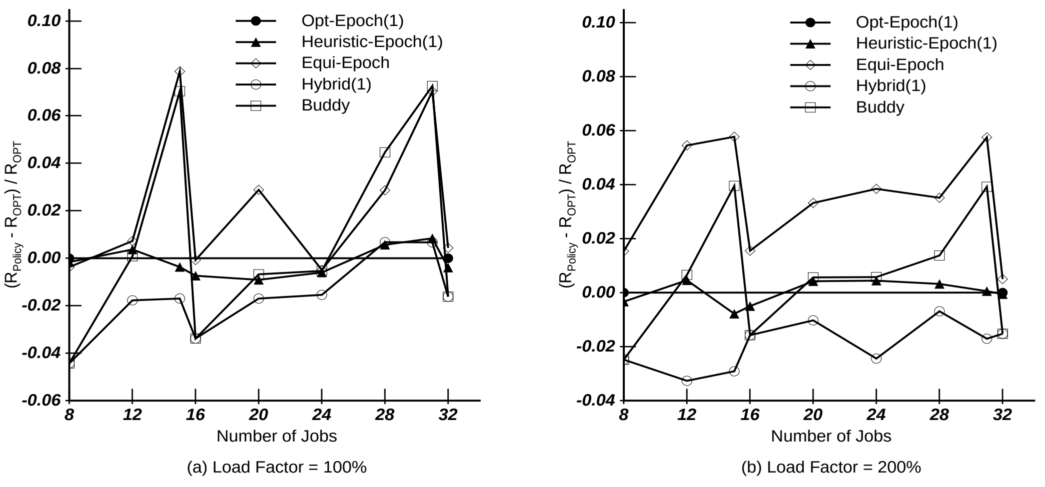


Figure 16: *Dynamic Workloads: Relative Response Time versus Number of Jobs*

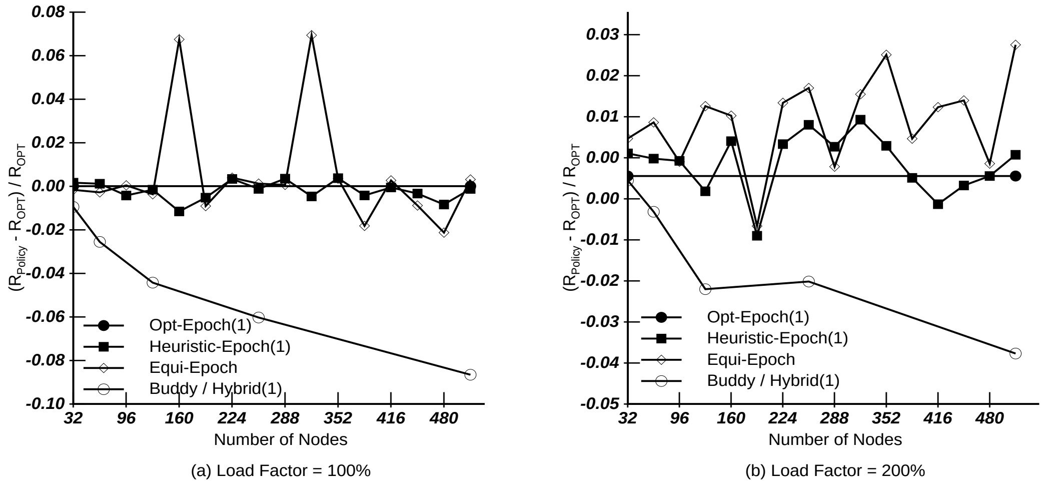
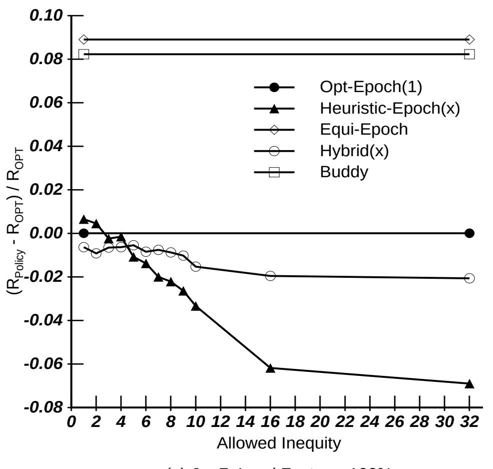


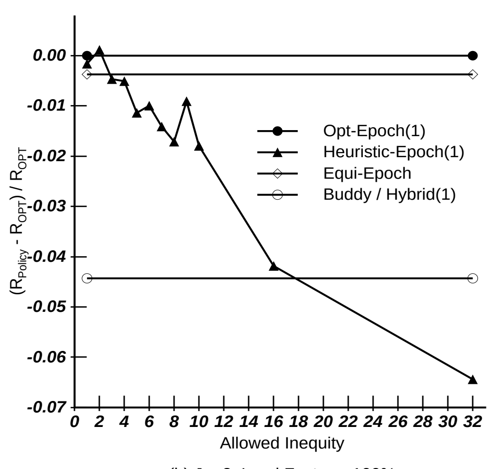
Figure 17: *Dynamic Workloads: Relative Response Time versus Number of Nodes*

6 Summary

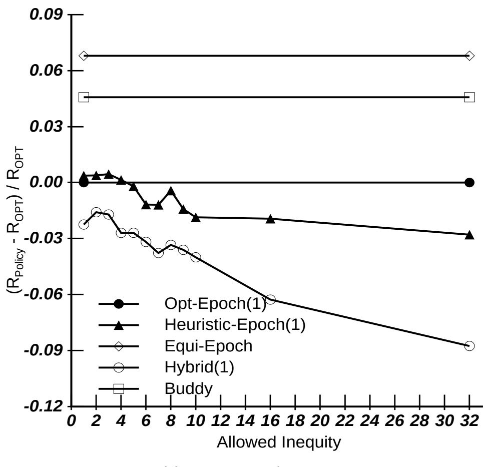
We have considered the problem of scheduling distributed memory parallel machines when jobs have memory requirements that impose lower bounds on the number of nodes they may be allocated. Because there is necessarily an element of time-sharing in this situation, we have looked for policies that minimize the number of node reallocations required. We have proposed a number of specific policies that are optimal



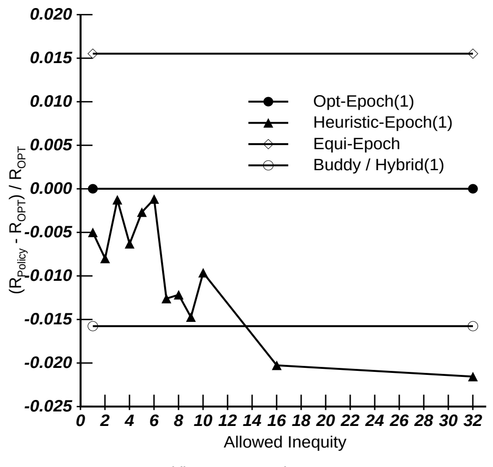
(a) J = 7; Load Factor = 100%



(b) J = 8; Load Factor = 100%



(c) J = 15; Load Factor = 200%



(d) J = 16; Load Factor = 200%

Figure 18: *Dynamic Workloads: Relative Response Time versus Allowed Inequity (k)*

under restricted conditions, and a number of heuristics that are more flexible and are guaranteed to find schedules efficiently.

In comparing the performance of our disciplines, we consider both jobs that can repartition their work dynamically in response to changes in their node allocations and those that cannot. We observed that the choice of discipline is strongly affected by this application characteristic.

We have also examined the extent to which equality of service is necessary to achieve good performance. We found that the additional flexibility afforded the scheduler by the possibility of unequal allocations outweighed the negative impact of the more FCFS-like nature of such allocations. (Although that same flexibility could work against the scheduler if it is unaware of the restrictions required in scheduling static applications, this is not a comment on the fundamental impact of unequal service, but rather a reflection of this other factor involved in scheduling such workloads.)

While we have rejected the use of application supplied information, such as job duration or parallelism, in making scheduling decisions, motivated in part by the fact that successful schedulers for sequential systems have made a similar decision because of the difficulty of obtaining reliable information of this sort, it seems reasonable that a policy implemented in practice should require jobs to classify themselves as static or dynamic. This characteristic is certainly known to the application, and there would be little inducement to misrepresent it. The scheduler could then apply different decisions to the two classes, by combining the basic approaches we have outlined here. Additionally, the advantages that would likely be enjoyed by dynamic applications in such an environment would be an inducement to future applications to provide this function (perhaps through the code emitted by the compiler and its runtime support), making dynamic applications more common in the future. While it requires further work to define precisely how such a hybrid scheduling policy would operate, we believe that the advantages of dynamic node allocation over run to completion policies make this a worthwhile endeavor.

Acknowledgements

The authors thank Raj Vaswani for his careful reading of and comments on an earlier version of this paper. We also thank Martin Tompa for his assistance in refining the proofs referenced in [10].

References

[1] R.M. Brown, J.C. Browne, and K.M. Chandy. Memory management and response time. *Communications of the ACM*, 20(3):153–165, March 1977.

[2] S.-H. Chiang, R.K. Mansharamani, and M.K. 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.

[3] L. Dowdy. On the partitioning of multiprocessor systems. Technical report, Vanderbilt University, June 1988.

[4] W. Leland and T. Ott. Load-balancing heuristics and process behavior. In *Proceedings of ACM SIGMETRICS Conference*, pages 54–69, May 1986.

[5] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[6] S. Leutenegger and M. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of ACM SIGMETRICS Conference*, pages 226–236, May 1990.

[7] S. Majumdar, D.L. Eager, and R. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of ACM SIGMETRICS Conference*, pages 104–113, May 1988.

[8] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[9] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of ACM SIGMETRICS Conference*, pages 19–32, May 1994.

[10] Catherine M. McCann. *Processor Allocation Policies for Message-Passing Parallel Computers.* PhD thesis, University of Washington, 1994.

[11] J. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[12] V.G.J. Peris, M.S. Squillante, and V.K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of ACM SIGMETRICS Conference*, pages 5–18, May 1994.

[13] S. Setia, M.S. Squillante, , and S. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel systems. In *Proceedings of ACM SIGMETRICS Conference*, pages 158–170, May 1993.

[14] K.C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of ACM SIGMETRICS Conference*, pages 171–180, May 1989.

[15] 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.

[16] J.L. Wolf, J. Turek, M.-S. Chen, and P.S. Yu. Scheduling multiple queries on a parallel machine. In *Proceedings of ACM SIGMETRICS Conference*, pages 45–55, May 1994.

[17] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of ACM SIGMETRICS Conference*, pages 214–225, May 1990.

[18] S. Zhou and T. Brecht. Processor-pool-based scheduling for large-scale numa multiprocessors. In *Proceedings of ACM SIGMETRICS Conference*, pages 133–142, May 1991.