Processor Allocation Policies for Message-Passing Parallel Computers

Cathy McCann and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 93-11-01
Revised 2-28-94

To Appear in Proceedings of the ACM Sigmetrics Conference, May 1994.

Processor Allocation Policies for Message-Passing Parallel Computers

Cathy McCann and John Zahorjan
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
mccann,zahorjan@cs.washington.edu

Abstract

When multiple jobs compete for processing resources on a parallel computer, the operating system kernel's processor allocation policy determines how many and which processors to allocate to each. In this paper we investigate the issues involved in constructing a processor allocation policy for large scale, message-passing parallel computers supporting a scientific workload.

We make four specific contributions:

- We define the concept of *efficiency preservation* as a characteristic of processor allocation policies . Efficiency preservation is the degree to which the decisions of the processor allocator degrade the processor efficiencies experienced by individual applications relative to their efficiencies when run alone.
- We identify the interplay between the kernel processor allocation policy and the application load distribution policy as a determinant of efficiency preservation.
- We specify the details of two families of processor allocation policies, called Equipartition and Folding. Within each family, different member policies cover a range of efficiency preservation values, from very high to very low.
- By comparing policies within each family as well as between families, we show that high efficiency preservation is essential to good performance, and that efficiency preservation is a more dominant factor in obtaining good performance than is equality of resource allocation.

1 Introduction

Processor scheduling in parallel computers is a two-level procedure: at the lower level, the kernel allocates processors to applications, while at the higher level the applications decide which of their ready parallel operations to execute on each processor. In this paper we are concerned with (kernel) processor allocation policies for large scale, message passing machines.

Current kernels for message passing multiprocessors employ only very simple processor allocation policies. The typical scheme is to define a basically static set of processor partitions that may be allocated to jobs. Once a job is assigned to a partition, it runs there to completion. This static division of the machine provides a relatively stable resource allocation on which the executing job can depend, but the fragmentation problems associated with it impact performance adversely.

Existing systems employ this static form of partitioning because dynamic reassignment of processors among jobs has been thought to be too expensive, due to the communication costs associated with relocating code and data. However, hardware advances continue to increase the bandwidth of interconnection networks, and recent software advances [20, 17] show how to reduce the latency currently imposed by large message startup costs to a negligible level. Our goal in this paper is to lay the foundation for processor allocation policies to be employed in the next generation of multiprocessor kernels, which will enjoy high network bandwidths and low message startup costs.

When a parallel application is run in a multiprogramming environment, the processor allocation policy employed in the kernel may induce efficiency losses for that application beyond those inherent to its parallel execution. These losses result from a mismatch between the resources actually allocated to the application and the model of the available resources used in making application decisions, such as load assignment.

The extent to which an allocation policy induces efficiency losses is one of its important characteristics. To quantify this, we define the notion of *efficiency preservation* as a characteristic of processor allocation policies. We then use this measure to help evaluate two specific proposed policies.

Our specific policy proposals fall into two families,

[0] 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.

called Folding and Equipartition. Each trades efficiency preservation against equality of resource allocation among competing jobs. Under Folding, a newly loaded job is allocated a partition of processors obtained by dividing the largest currently allocated partition in half, with the threads[1] running on those processors "folded" onto the remaining processors allocated to their job. In this way, the policy ensures both that no processors are needlessly idle and that jobs that exhibit good load balance when run alone will be load balanced when run in a multiprogrammed environment. To achieve equal long-term allocation of resources, the Folding policies periodically reallocate processors among the running jobs. By varying the rate of reallocations, the Folding policies vary from one emphasising high efficiency preservation to one emphasising equal resource allocation.

The Equipartition family of policies reallocates processors as equally as possible whenever a job arrives or departs, but makes no other reallocations. Equipartition has very low overhead and good equality of allocation, but potentially poor efficiency preservation. By comparing members of the Folding policy family to each other and to members of the Equipartition family, we are able to compare the importance of efficiency preservation and equal resource allocation to the performance of processor allocation policies.

1.1 Previous Work on Processor Allocation Policies

Distributed Memory Systems. An early and fundamental result on processor scheduling for parallel machines is due to Ousterhout [12]. He noted that because the threads of a parallel application synchronize frequently, the rate of progress of the application will be determined by the scheduling quantum unless all threads of the processor are "co-scheduled", that is, run at once. This effect forces processor allocators to operate on a per-job granularity, rather than a per-thread one.

Feitelson and Rudolph [5, 6] build upon these observations, describing variants of gang scheduling. They propose a hierarchical scheme for assigning applications to processors, and examine the fragmentation it experiences when used in conjunction with gang scheduling for jobs whose sizes differ according to a number of distributions. Because many jobs may be allocated to each processor, the processors must be time-shared among them.

Zhou and Brecht [22] describe a pool-based scheduling mechanism. Pools are a logical construct, used by the kernel to balance the allocation of jobs across the processors. Unlike the work of Feitelson and Rudolph, there is only a single level to the allocation structure in pool-based scheduling. Additionally, in pool-based scheduling the kernel can effectively restrict a job's choice of parallelism by restricting its threads to only one or a few pools. Pool-based scheduling may time-share partitions.

Setia et al. [13] examine the benefits of time-sharing and dynamic partitioning. They conclude that time-sharing can be effective in reducting mean response time.

Chen and Shin [2] examine processor allocation for cube-connected message passing machines. Like Feitelson and Rudolph, they assume that arriving jobs declare the number of processors required for execution. The goal of their work is to describe efficient schemes for finding sub-cubes to satisfy arriving jobs.

Dussa et al. [4] examine the benefits of dynamically repartitioning processors on job arrival and departure, using experiments on a ring connected Transputer-based system, as well as a simple analytic model. They conclude that for this hardware and the two-job workload considered that dynamic repartitioning can be beneficial, primarily because of its ability to discriminate between jobs based on their total duration and because the sub-linear speedup of typical applications makes them more efficient when run on fewer processors.

Shared-Memory Systems. Much of the recent work on processor allocation policies has considered shared memory machines. Tucker and Gupta [18] describe "process-control", which is fundamentally a space-sharing approach. Under process control, an arriving job creates a thread per processor of the machine, but based on feedback from the kernel, idles threads in excess of its current processor allocation. This allows a natural form of co-scheduling, which would not be possible if the number of active threads exceeded the processor allocation, even with space sharing. Processors are reallocated among jobs only on job arrival and departure.

Zahorjan and McCann [21] compare time sharing to space sharing, and conclude that space sharing is preferable. They also describe a more aggressive form of co-scheduling than is used in process control, as well as a more aggressive approach to reallocating processors in response to transient changes in job parallelism. McCann et al. [10] further refine this policy, and report on results from a prototype implementation of it.

Gupta et al. [7] and Vaswani and Zahorjan [19] examine the importance of cache affinity to the decisions made by the kernel processor allocator. Their results agree in showing that cache affinity is not exploitable by the kernel, except in very special circumstances. (Squillante and Lazowska come to somewhat contradictory

[1] Throughout this paper we will use "thread" to mean "kernel thread". While user-level threads can be useful in the message-passing environment we consider, exploiting them fully requires application structures more complicated than are currently routinely employed.

conclusions, based on results from an analytic model [16].)

Majumdar et al [9], Sevcik [14, 15], and Leutenegger and Vernon [8] examine the relationship between job characteristics, such as maximum parallelism and total service time, and the performance of various scheduling disciplines. They find that, as in uniprocessor systems, average response time can be improved by allocating resources preferentially to smaller jobs, and that in the absence of *a priori* job characterizations, allocating an equal fraction of total processing power to each job is an effective heuristic.

Building on the Results of Previous Work. In examining the results obtained by prior work, we conclude that feasible processor allocation policies should have the following characteristics:

- Co-scheduling — The dispatching of a job's threads must be coordinated, so that no thread waits to synchronize with a thread that is not running.
- Space-sharing — A set of processors should be shared among multiple jobs by giving each exclusive use of a subset, rather than by alternating assignment of the full set among them.
- High processor efficiency — When a job departs, its processors must be reassigned to another job.
- Equal allocation — Each running job should be allocated a reasonably equal portion of the resources, for reasons of fairness as well as performance.

1.2 Paper Outline

In the next section we describe the hardware and software environments considered. Section 3 discusses the efficiency preservation measure of processor allocation policies. In Section 4 we define two specific processor allocation policies, called Folding and Equipartition. In Section 5 we use simple algebraic models to perform a static analysis, obtaining the fundamental efficiency preservation behaviors of the two disciplines in the absence of job arrivals or departures. In Section 6 we use a Markovian birth-death model to obtain mean response times under homogeneous job arrivals and departures. In Section 7 we use simulation to deal with variance, examining the fairness of the policies and their characteristics under heterogeneous workloads. Section 8 summarizes our results.

2 The Hardware and Software Environments

We consider the particular case of mesh-connected parallel machine of $2^M x 2^N$ nodes, as shown in Figure 1a. Each node contains a single processor and sufficient memory to multiprogram a number of threads of a single application. Nodes communicate only by messages; there is no shared memory. This model captures the important attributes of the Intel Paragon, and we will make parameterizations of it based on the specifications of that machine. However, much of our work applies to machines with a tree interconnection structure (such as the CM-5), and to machines where each processing node contains a small cluster of processors. Additionally, it should not be hard to translate our ideas to other interconnection structures.

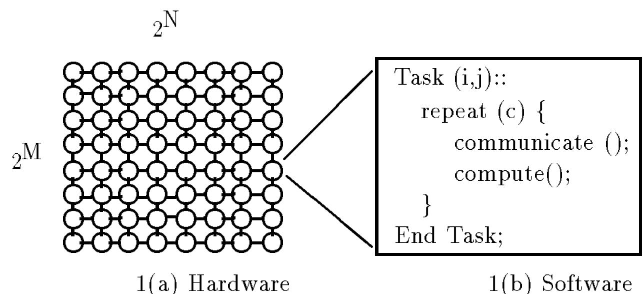

Figure 1: *Hardware and software configurations*

The performance afforded by a particular kernel processor allocation policy is intimately tied to decisions made by the applications it supports. We consider in this paper a workload composed of scientific applications written in a single-program-multiple-data (SPMD) style. This is a common style for both hand-written codes and those produced by parallelizing compilers.

Figure 1b shows the particular software structure we consider, the simple but quite common "communicate-compute" model, in which all nodes pass through coordinated phases of communication followed by local computation. We assume the particular case of nearest-neighbor communications, although most of our quantitative results are not strongly affected by this, and the policies and analysis techniques are applicable for more general patterns.

As mentioned earlier, the interaction of the kernel-level and application-level policies has an important effect on performance. In this paper, we make a very weak assumption about the requirements placed on the application regarding the policies it employs. This conservative assumption is appropriate for general purpose system software, which must accommodate as wide a variety of application software as possible. We assume that load balancing and data placement decisions for the application have been made in the context of a fixed "virtual machine", which in our case is identical to the full physical machine (i.e., a grid of $2^M x 2^N$ processors). This targeting may be done either explicitly, by the programmer, or implicitly, by the compiler. The use of a virtual machine model for this purpose is a common concept in many languages intended for implementation of parallel programs.

We assume that when a job begins execution, it spawns a number of threads equal to the number of processors in the virtual machine, and partitions its workload as equally as possible among them. After that, it is unable to alter the load distribution. We also assume that the application does not alter the number of threads in response to changes in processor allocation. Considerable effort is required to implement dynamic remapping, and determining when to remap can be complicated [11]. This effort may be justified only for very irregular and dynamic computations [1].

Our hardware and software models reflect an important class of system and applications, and capture the most central aspects of the processor allocation problem for other classes. However, we have not attempted to include all aspects of all parallel systems. We intend our results to be appropriate for systems supporting the large number of applications of the type we model, and as well to serve as the basis for policies intended to support other forms of parallel applications.

3 Efficiency Preservation

When a parallel program is run in a multiprogramming environment, the number of processors it is allocated at any point in time is determined by the kernel, through its processor allocation policy. If not carefully designed, this policy can induce efficiency losses beyond those inherent in the application. For instance, if an application that has partitioned its load into eight pieces is allocated exclusive use of five processors, a great deal of processing capacity will likely be lost.

To quantify this notion, we define *efficiency preservation* as a characteristic of processor allocation policies. To do so, first define *application efficiency* for some application *app* running under policy *policy* on p processors, $AE_{policy,app}(p)$, as the ratio of the total computation time of the application when run on p processors (excluding waiting time due to load imbalance synchronization losses) to the product of the application's elapsed time and p.

We define the efficiency preservation of a processor allocation policy *policy* for some application *app* by considering what happens when J copies of *app* are run under *policy* on a machine with P processors. Then efficiency preservation measure is

$$EP_{policy,app}(J,P) \equiv \frac{\sum_{j=1}^{J} \mathcal{A}_j \frac{AE_{policy,app}(\mathcal{A}_j)}{AE_{Uniprogramming,app}(P)}}{P} \quad (1)$$

where $\mathcal{A}_j$ is the number of processors the policy allocates to copy j of the application. Because allocation policies can impose overheads, EP can in theory be as small as zero. Because at least some applications exhibit significantly sub-linear speedups, EP can in theory be much larger than one.

The efficiency preservation measure provides a natural explanation for a number of conclusions reached in prior work. For instance, it is easy to see that a policy that statically divides the processors into K equal sized partitions and runs a single job in only one partition will have difficulty achieving an efficiency preservation much greater than J/K, and so should be expected to perform poorly. Similarly, because applications typically exhibit sub-linear speedup, it is beneficial to schedule many of them at once, assigning each a few processors, rather than rotating possession of many processors among them. This is expressed in the efficiency preservation measure by the growth in application efficiency with diminishing allocation, and thus the growth in efficiency preservation. Finally, because efficiency preservation is diminished by the overhead required to implement a time-sharing scheme, efficiency preservation argues against their use.

While efficiency preservation provides useful information about a processor allocation policy, it is not a complete characterization. For one thing, it ignores the cost of reallocating processors when jobs arrive and depart. For most policies these costs are small, since job arrivals and departures are relatively infrequent. However, some policies can experience longer term ill effects from arrivals or departures. For instance, a policy that dynamically partitions a machine but never changes the assignment of any job once made will not respond well to job departures. Another shortcoming of efficiency preservation as a measure is that high efficiency preservation alone does not guarantee good performance, as measured, say, by mean response time. For instance, a policy that dedicates the entire machine to individual jobs in FCFS order will have efficiency preservation of 1.0. However, this policy is as unattractive for parallel machines as it is for sequential ones. Despite these shortcomings, efficiency preservation does provide important information that is useful in comparing alternative processor allocation policies.

Finally, note that the dependence of the efficiency preservation measure on the application considered is critical, as the interplay between the kernel's processor allocation policy and the application's load management policy can have a profound effect. Revisiting our earlier example, if an application that has divided its work into eight pieces is allocated five processors, its application efficiency will be very poor if it is incapable of reallocating its work in response to this processor allocation. On the other hand, if the application is able to dynamically repartition its work into an arbitrary number of equally balanced pieces, application efficiency may not suffer. Thus, in general it is not reasonable to compare the performance of alternative processor allocation policies without specifying at least some of the characteristics of the applications they are intended to support.

4 Description of the Policies

In this section we describe the Folding and Equipartition processor allocation policies. These policies assume that no *a priori* information is available on job characteristics, such as duration. Experience with uniprocessor systems, as well as prior work on parallel machine scheduling [8], indicates that providing roughly equal allocation of resources to the jobs is appropriate in these circumstances.

Our allocation policies apply to sets of jobs that fit simultaneously in memory and the 2^{M+N} processors of the mesh. If more jobs are ready to run than can be supported by the hardware resources, another level of scheduling is required. We confront this issue in Section 7.

Given J jobs in the running set, the processor allocation policy determines which processors to assign to each. We limit our attention to assignments of rectangular blocks of processors: if other shapes are allowed, messages between processors belonging to one job will be routed over links connecting processors belonging to other jobs. For reasons of predictability, we wish to avoid this sort of interference.

In addition to choosing the processor partition for a job, the allocation policy must also choose a location for each of its threads. To make it possible for users to tune their applications, thread adjacency when running on a restricted rectangle of processors must be the same as when running on the full grid.

The simplest thread mapping scheme, and the basis of all the schemes used in our proposed policies, is simply a linear contraction from a $2^M x 2^N$ grid onto an RxC grid. A thread that would be located on processor (i, j) of the full machine is located on node $map(i, j)$ of the RxC subgrid, where

$$map(i,j) \equiv \left(\left\lfloor \frac{i*R}{2^M} \right\rfloor , \left\lfloor \frac{j*C}{2^N} \right\rfloor \right) \qquad (2)$$

The maximum number of threads assigned to any of a job's processors is an important measure, since the synchronization constraints of the job limits its performance to that of its most slowly progressing thread. The theoretical minimum possible maximal number of threads assigned to a single processor under any thread mapping function is $\left\lceil \frac{2^M}{R} \frac{2^N}{C} \right\rceil$. In general, the adjacency preserving mapping (Equation (2)) gives maximal loading $\left\lceil \frac{2^M}{R} \right\rceil \left\lceil \frac{2^N}{C} \right\rceil$, which can be considerably larger. Addressing this shortcoming is one of the problems confronting processor allocation policies.

4.1 The Folding Policy

The motivation for the Folding policy is high efficiency preservation. The Folding policy allocates partitions such that the adjacency preserving mapping of threads to processors results in a perfectly equal allocation of threads. Under the assumption that the threads of the application are load balanced when run on the full machine, they will also be load balanced when run under Folding in competition with other jobs.

Folding: Basic Operation. Folding chooses new partition sizes whenever a job arrives or departs. On job arrival, if the machine is idle, it is allocated as a whole to the new arrival. If not, the largest currently allocated partition is divided in half, with the new job taking one of the resulting partitions and the existing job the other. Both jobs have their threads mapped to their allocated processors in the manner described above. Thus, each job arrival disturbs at most one currently running job.

When a job departs, two partitions must be recombined. Unfortunately, this is not always a simple operation. We defer discussion of job departures to Section 4.1.

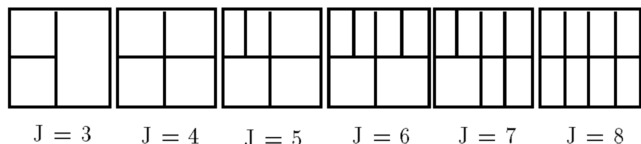

Figure 2: *Partitioning of processor mesh for 3 to 8 jobs*

Figure 2 shows the partitions produced by Folding for $J = 3$ to 8 jobs, for $M = N$. In the general case, when a job arrives and a single other job holds the entire machine, the machine is split along its largest dimension. After that, splits alternate directions, so that a partition of size $2^{M+N}/2^i$ is split along the machine's largest dimension if i is even and along the other dimension if i is odd.

Folding: Realizing Equal Resource Allocations. Unless J, the number of jobs, is a power of two, Folding allocates partitions of two different sizes, with the larger partitions containing a factor of two more processors than the smaller partitions. There are two potential drawbacks to unequal processor allocations. First, the jobs will not experience fair service, which might be one of the goals of the kernel resource allocator. Additionally, mean job response time might suffer if there are jobs of significantly different sizes presented to the system.

To provide equal resource allocation for all jobs, the Folding policy must rotate ownership of the large and small partitions. To achieve this, we define a *rotation policy*. The rotation policy is invoked at fixed intervals, and determines how many and which processors to move from one job to another. The goal of the rotation policy is to ensure that over a sufficiently long interval, each job will receive an equal percentage of the total processing power of the machine.

Many different rotation policies are possible. In designing a rotation policy, it is desirable to limit processor reassignments to jobs running on adjacent rectangular

subgrids, so that rotation traffic does not interfere with the execution of other, uninvolved jobs. It is also desirable to use a scheme that achieves equal allocation after only a small number of rotations, that is, with small overhead.

One technique for reducing the required number of rotations is to perform them hierarchically: a system running an even number of jobs is treated as two systems, each with half the jobs and half the processors. For example, when six jobs are present, we rotate two independent systems of three jobs, each on half the machine. (See Figure 2.)

The rotation policy we propose here has the property that each job can determine whether or not it is involved in an exchange of processors at each rotation instant using information about only two neighboring jobs, rather than the state of the entire machine. Figure 3 shows the sequence of partition allocations made under our rotation policy for a system running five jobs. To understand the procedure used, consider performing a cyclic walk that touches each partition and moves only between adjacent partitions. (It is easy to show that because of the way in which we form partitions, such a walk must exist.) In Figure 3 the jobs are numbered according to their position in this cyclic walk. At rotation instants, the rotation policy we use reassigns half the processors from a job holding a large partition to a job holding a small partition if the former immediately follows the latter in this cyclic walk. This reallocation results in the exchange of the large and small partitions between the two jobs, and so preserves the cyclic walk. Other partitions remain unchanged.

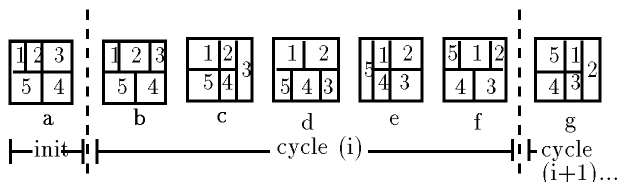

Figure 3: *Folding rotation for $J = 5$ Jobs*

We evaluate rotation policies according to two measures, rotation cycle length and rotations per job. Imagine running the system at a constant multiprogramming level for a long period. After some initial transient, the sequence of allocations produced by a well behaved rotation policy will be partitionable into cycles, each of which delivers the same total processing power to each job. The rotation cycle length, $N(J)$, is the number of reallocation intervals or rotations in each such cycle. For example, in Figure 3 the cycle length is five. The second measure, $f(J)$, is the number of rotations participated in by each job within the rotation cycle of $N(J)$ rotations. In Figure 3, $f(J)$ is four.

To understand the behavior of the rotation policy we have proposed in terms of $N(J)$ and $f(J)$, we represent it as an operation on strings. Let the symbol 's' represent a small partition, and 'b' a large one, and form a string R by performing the cyclic walk over the jobs described above. For instance, the string corresponding to Figure 3a is 'ssbbb', and for Figure 3b is 'sbsbb'. Our rotation policy is simply to replace each occurrence of 'sb' in R with 'bs', considering the first symbol of R to follow the last.

Let S be the number of small partitions and B the number of large partitions. It is straightforward to show for any such string R that after no more than $min(S, B)$ rotations, R contains no consecutive 's's when $S \leq B$, and no consecutive 'b's when $B \leq S$. Once this equilibrium condition is reached, each string R that occurs at all occurs every $J + 1$ rotations, that is, repeats after J rotations. (Because the length of the string must be odd due to our hierarchical rotation scheme, the string may not reoccur any sooner.) To see this, note that if $B \leq S$, each 'b' will be preceded by an 's' in R. Thus, at each rotation, each 'b' will move one symbol to the left, and so after J rotations the string will be in its original configuration. (A similar argument applies when $S \leq B$.) This reasoning also shows that all jobs will be allocated large partitions for the same number of reallocation intervals during a cycle, and so equal allocation is assured.

This analysis leads to the following expressions for the cycle length and the number of reallocations per job per cycle:

$$N(J) = \begin{cases} N(K) & \text{if } J = K * 2 \\ 0 & \text{if } J = 1 \\ J & \text{otherwise} \end{cases} \quad (3)$$

$$f(J) = \begin{cases} f(K) & \text{if } J = K * 2 \\ 0 & \text{if } J = 1 \\ 2 * min(S, B) & \text{otherwise} \end{cases} \quad (4)$$

Folding: Job Departures with Rotations. An examination of Figure 3 shows that reallocating the processors freed by a departing job may not be possible with only local operations. For example, if job 5 departs in any of the configurations shown in Figure 3, we cannot make only local adjustments without violating the constraint on partition sizes imposed by Folding.

One way to handle job departures in such cases is to compute a new set of partitions and perform a global assignment of jobs to those partitions. A simpler way to achieve the same effect is to mark the freed partition as available, but to leave it in the rotation string R that controls rotation reallocations, and then step through the normal rotation sequence as fast as possible until the idle processors have been allocated. (Note that it is possible for the freed processors to constitute a small partition. We do not steal processors from a succeeding large partition in this case, and as well modify the rules

so that a preceding small partition is combined with the free small partition.)

It is not evident whether this scheme is more or less expensive than simply remapping all the jobs to new partitions on any departure. We have adopted it because it greatly simplified implementation of the simulation model of Section 7. We believe that this is an indication that it would be an attractive approach in a real system, for the same reason.

Folding: The Family of Policies $FOLD_I$. The family of Folding policies is generated by varying the rate at which rotations take place. We denote by $FOLD_I$ the Folding policy with inter-rotation time I. At one extreme, $FOLD_\infty$ never rotates. This gives a policy with very high efficiency preservation, but unequal allocation to running jobs. By decreasing the time between rotations, equality of allocation is enhanced, but at the cost of diminished efficiency preservation.

4.2 The Equipartition Policy

Equipartition emphasizes equal allocation over efficiency preservation by partitioning the processors as evenly as possible among the running jobs. The number and size of the partitions change only when jobs enter or leave the system; there is no need to perform periodic rotations. We describe two variants of the Equipartition policy.

Basic Equipartition: $EQUI$. As pointed out earlier, the thread mapping function (Equation (2)) allocates $\left\lceil \frac{2^M}{R} \right\rceil \left\lceil \frac{2^N}{C} \right\rceil$ to at least one processor in an RxC partition. Because the progress of the job is limited by its most slowly executing thread, reducing this maximal loading is an important goal.

The key to reducing the maximal processor loading is to ensure that at least one dimension of each allocated partition is a power of two. We propose a policy, $EQUI$, that has this property, and additionally tries to reduce the reallocation overhead necessary to move from one configuration to another by minimizing the number of processors that must be reassigned to different jobs.

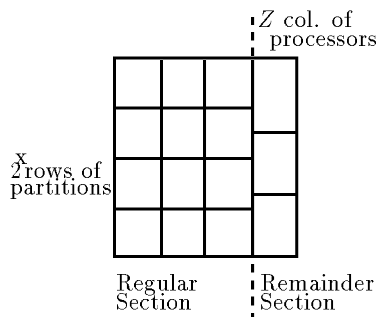

Figure 4: *Partitioning under Equipartition*

Figure 4 shows the general case. We divide the mesh into two sections, the *regular section* and the *remainder section*. We divide the regular section into 2^X rows, each of which contains $Y \equiv \lfloor J/2^X \rfloor$ partitions. The remainder section contains $J - 2^X Y$ partitions. (When J is a power of two, this will be zero, and so there will be no remainder section.) To make the partitions in the regular section as square as possible, we choose $X = \left\lceil \frac{\lfloor log J \rfloor}{2} \right\rceil$.

To ensure that each partition in the regular section has at least one dimension that is a power of two, we simply assign 2^{M-X} rows of processors from the 2^{M+N} mesh to each row of partitions in the regular section.

To ensure that the partitions in the remainder section (if any) have a power of two dimension, we allocate Z columns of processors to them, where

$$Z \equiv 2^{\left\lfloor log(1 - \lfloor \frac{J}{2^X} \rfloor \frac{2^X}{J}) + N \right\rfloor} \tag{5}$$

This is a power of two approximation to the fair share of the 2^{M+N} total processors.[2] The remaining $2^N - Z$ columns of processors are allocated among the partitions of the regular section as evenly as possible.

A Higher Efficiency Preservation Equipartition: $EQUI_+$. Consider a single partition allocated by $EQUI$, and denote its size as 2^ixC, for some i. The maximally loaded processor in that partition will contain $2^{N-i} \lceil 2^M/C \rceil$ threads. The purpose of $EQUI_+$ is to reduce that maximal loading.

$EQUI_+$ performs the same partitioning as $EQUI$, that is, it assigns partitions of exactly the same size. However, it violates the simple mapping function of threads onto processors (Equation (2)) in order to reduce the maximal loading to $\lceil 2^{M+N-i}/C \rceil$, the theoretical minimum for this partitioning. This is done by balancing the thread assignment among the processors of each row of the partition. There are a total of 2^{M+N-i} threads assigned to each row. Thus, it is possible to achieve the minimal load balance by averaging within rows only; there is no need to resort to averaging between rows.

In general, performing this averaging requires moving only a few threads from where the simple mapping scheme would place them. However, the fact that even a few are not where that mapping would place them means that some more complicated procedure must be followed to determine to which processor a message should be sent. A fairly simple scheme to do this is possible (but because this is not central to the purpose of this paper, we omit its details to help shorten the discussion). We make the optimistic assumption that the added messaging complexity results in a negligible increase in communication costs.

[2] A closer approximation to equitable allocation is possible if the number of processors in the remainder section is a power of two. In this case, each partition will have a power of two number of processor rows, so the number of columns allocated to the remainder section need not be a power of two.

5 Static Analysis: Efficiency Preservation and Fairness

In this section we compare efficiency preservation and fairness under Folding and Equipartition when there are a fixed number of identical running programs, that is, in the homogeneous, static case.

We use the following notation. The values in parentheses indicate the baseline setting for the experiments discussed throughout the remainder of the paper. These values are based, in part, on the workload characteristics described in [3], and on the characteristics of the Intel Paragon hardware.

- 2^Mx2^N, the size of the mesh of processors (2^4x2^4).
- J, the (static) number of jobs in the system (variable).
- t, the average per-thread compute time of each application step (100 msec.).
- δ, thread compute time spread: individual thread per-step compute times are chosen from $U(t-\delta, t+\delta)$ (0 msec.)
- s, the per-thread cost of the communication phase of each application step when run on 2^Mx2^N processors. (1 msec.).
- c, the context switch time required to schedule a new thread (0.5 msec.).
- C, the size of a thread's code segment (512KB).
- D, the size of the thread's data and stack segments (4096KB).
- X, the interconnection network link bandwidth (200MB/sec.).
- I, the inter-rotation time for the Folding policy (variable).
- L, the limit on the number of threads per node, or equivalently, the number of jobs in the system (2^{M+N})

While current production systems impose high message start-up costs that can have a significant performance impact, recent work, [20, 17] has shown that these costs can be almost entirely avoided. Our parameterization anticipates the next generation of system software that will incorporate these advances.

5.1 Efficiency Preservation: Derivation

As explained in Section 3, efficiency preservation is given by

$$EP_{policy,app}(J,P) \equiv \frac{\sum_{j=1}^{J} \mathcal{A}_j \frac{AE_{policy,app}(\mathcal{A}_j)}{AE_{Uniprogramming,app}(P)}}{P} \quad (6)$$

where $\mathcal{A}_j$ is the size of the partition allocated under *policy* to job j. (Because which application we are considering is clear, we hereafter drop the subscript *app* on all quantities.) Therefore, to compute the efficiency preservation of a policy, we must first give an expression for application efficiency.

Let t_i denote the length of each of thread i's compute phases. Then we have for job j

$$AE_{policy}(\mathcal{A}_j) = \frac{\left(\sum_{i=1}^{2^{M+N}} t_i\right) * (1 - \%OV_{policy})}{\max_{p \in \mathcal{P}(j)}(C_p + S_p + X_p) * A_j} \quad (7)$$

where $\mathcal{A}_j$ is the size of the partition allocated under *policy* to j, $\mathcal{P}(j)$ is the set of processors in the partition assigned to j, C_p and X_p are the total compute and communication times respectively that processor p spends per application step, S_p is a context switch time if more than one thread is mapped to p and zero otherwise, and $\%OV_{policy}$ is the fraction of time each processor spends on policy overhead functions. In the specific case of perfectly load balanced computations ($t_i = t$), we can simplify the ratio in this expression, using the fact that the maximum number of threads mapped to a single processor for a partition of size R_jxC_j is $\left\lceil \frac{2^M}{R_j} \right\rceil \left\lceil \frac{2^N}{C_j} \right\rceil$ under all policies we consider except $EQUI_+$, for which it is $\left\lceil \frac{2^M}{R_j} \frac{2^N}{C_j} \right\rceil$. When thread compute times can vary, we use Monte-Carlo simulation to obtain numerical results for this term.

$\%OV_{Uniprogramming}$ and $\%OV_{Equipartition}$ are zero (for both variants of Equipartition). To compute $\%OV_{Folding}$ we make use of the following assumptions:

- Each processor can send or receive only a single message at a time.
- Because of the large amount of data transferred in moving a thread, communication time is dominated by bandwidth considerations, not latency.
- When folding a job currently allocated a large partition onto half of its processors, only thread data must be sent, since a copy of the code segment already exists on the destination processors. The code segment must be sent, however, to unfold a job onto an expanded partition.
- All processors of a partition stop application processing while folding or unfolding is taking place in it.
- Thread transfer times are sufficiently well synchronized that folding a job along a single dimension from $2K$ processors onto K processors requires K steps. This is accomplished by first transferring the tasks at node $2k$ to node $2k-1$ in parallel for $k = 1..K$ (one step), and then successively transferring the resulting paired sets of threads to their final destinations ($(2K-2)/2$ steps).

With these assumptions one can derive that

$$\%OV_{Folding} = \quad (8)$$
$$(\#rotations\ in\ N(J)I) * (\#procs/rotation) *$$
$$(\#seconds/(proc/rotation))/(2^{M+N} * N(J)I)$$

The last term represents the total number of processor-seconds available in an interval of length $N(J)I$, the first term the total number of rotations in that interval, the next term the total number of processors involved in each rotation, and the next term the time required to complete a rotation.

It is easy to see that the number of rotations that occur in $N(J)$ rotation steps is $N(J)*f(J)/2$, since each rotation is composed of two jobs. Since each rotation involves a fold followed by an unfold, and each of these involves a number of processors equivalent to the size of a large partition, the number of processors per rotation is $2^{M+N}/2^{\lfloor log J \rfloor}$.

We compute an upper bound for $\#seconds/(proc/rotation)$ by assuming that each fold takes place along the larger dimension of a large partition. Both the job folding from a large partition onto a small one, and the job unfolding in the opposite direction, must relocate half (i.e., 2^{M+N-1}) of their threads. It takes time D/X to fold a thread, and time $(D+C)/X$ to unfold one. Finally, parallelism equal to the narrower dimension of a large partition is possible in moving the threads. This is equal to $2^{\min(M,N)}/2^{\lceil \frac{\lfloor \log_2 J \rfloor}{2} \rceil}$.

Combining these terms and simplifying, we get

$$\%OV_{Folding} = \frac{f(J) * 2^{\max(M,N) - \lfloor \frac{\lfloor \log_2 J \rfloor}{2} \rfloor - 1} * (2D+C)}{I * X} \quad (9)$$

5.2 Efficiency Preservation: Results

We present quantitative results on efficiency preservation for each of our policies, using the baseline parameterization given at the beginning of this section.

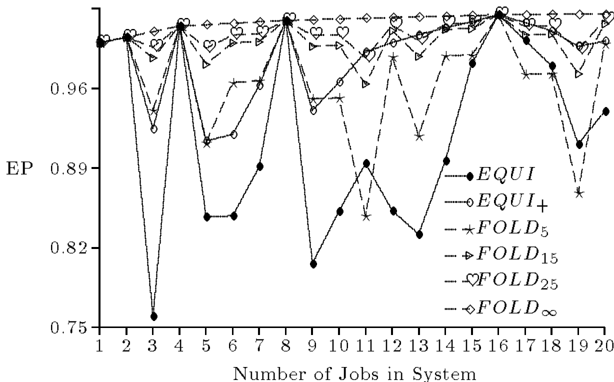

Figure 5: *Efficiency preservation for perfectly load balanced jobs ($\delta = 0$)*

Figure 5 shows efficiency preservation when thread compute times are constant ($\delta = 0$). As expected, $EQUI$ shows significant efficiency losses when J is not a power of two. While $EQUI_+$ is a noticeable improvement, it still experiences large efficiency losses, although the magnitudes of these losses decrease with increasing J.

The efficiency preservation of the Folding policies shows some sensitivity to the choice of inter-rotation interval. In general, though, they have much better behavior than the Equipartition policies for all but unrealistically small inter-rotation times. $FOLD_\infty$ attains values slightly greater than 1.0 as J increases. This reflects the decreased off-processor communication required as the applications is folded onto smaller and smaller partitions.

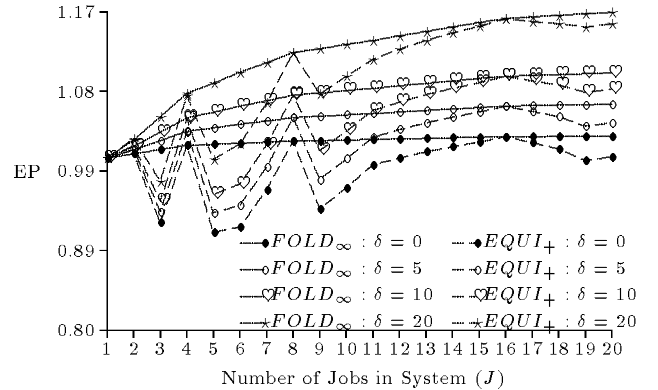

Figure 6: *Efficiency preservation for thread compute times from $U(100-\delta, 100+\delta)$ msec.*

Figure 6 shows how variation in application thread compute times affect efficiency preservation for the $FOLD_\infty$ and $EQUI_+$ policies. We present results for task times taken uniformly from $(100-\delta, 100+\delta)$ msec. for $\delta = 0$, 5, 10, and 20 msec. As can be seen, the efficiency preservation measures of our disciplines increase with increasing variance in thread compute times, indicating that they are even more effective for applications with load imbalance than those that are perfectly balanced. The intuition behind this is quite simple: there is less variation among the total per-processor compute times when many threads are allocated to each processor than when there is only one thread per processor. This is a quite general phenomenon, and is likely to be violated only if the high load threads are located in a clustered way.

Note that, as desired, efficiency preservation reflects on the performance of the processor allocation policy for the workload, not the performance of the application itself. It is clear that application performance decreases with increasing thread compute time variation, since the application is less well load balanced.

5.3 Equal Allocation: Results

The computation rate of a parallel application is limited by its most slowly progressing thread. For this rea-

son, we use the maximal number of threads assigned to any of a job's processors as our measure of fairness, rather than the total number of processors the job is allocated.

Under an ideal policy, the maximal number of threads assigned to any processor would be J for all J jobs. To evaluate fairness, we compute the ratio of two values to this ideal average: the largest (over all partitions) maximal number of threads assigned to any processor in a single partition, and the smallest maximal thread assignment. To compute the thread assignment values for the members of the Folding family that employ rotation, let E be the elapsed time of the application when run alone on the machine. When run in competition with other jobs, the application passes through some number of complete intervals of length $N(J)I$, followed by a partial interval. During the complete intervals, it finishes fraction $(N(J) * I)/(E * J)$ of its work, since it is allowed use of an equal share of the machine in each such interval. We obtain upper and lower bounds on fairness by assuming that during the partial interval some job is allocated large partitions only, and another small partitions only.

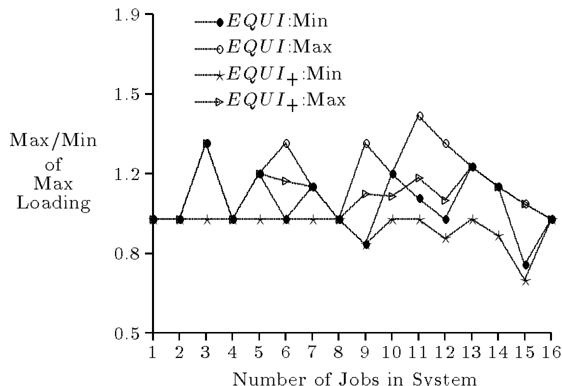

Figure 7: *Fairness of Equipartition: relatively most heavily and most lightly loaded jobs*

Figure 7 shows the fairness ratios for $EQUI$ and $EQUI_+$, and Figure 8 the same results for members of the Folding family. These results demonstrate the benefits of reducing $N(J)$ for J even by dividing the system of rotations into two.

In Figure 8 we express the inter-rotation time of the Folding policies, I, as $q * E$, for various values of $q > 0$. This allows the results to be independent of the value of E. For $I \geq E$ ($q \geq 1$), the bounds are equivalent to the case $I = \infty$. For $I < E$, intuitively the worst case is when I is just greater than $E/2$. Figure 8 shows the bounds on fairness in this case ($q = 8/15$). We also show results for I just greater than $E/4$ ($q = 4/15$). From these samples, it is clear that fairness is very near the ideal once the inter-rotation time is at least a small factor smaller than the duration of the job when run

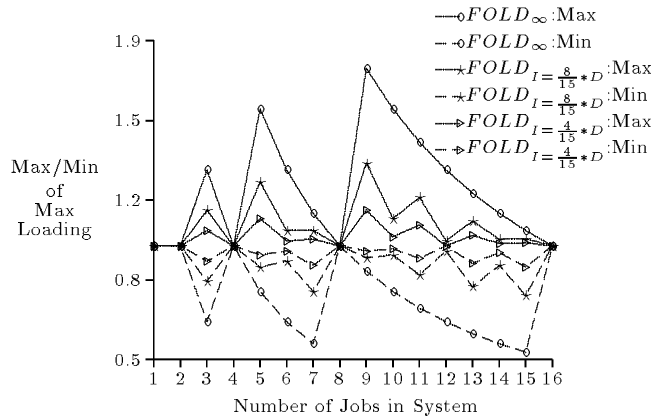

Figure 8: *Fairness of Folding: relatively most heavily and most lightly loaded jobs*

alone.

6 Birth-Death Analysis: Mean Response Times

In the previous section we examined two static properties of the Folding and Equipartition disciplines: efficiency preservation and fairness. In this section we examine a dynamic property, the mean response time afforded under homogeneous arrivals.

To do this, we employ a simple, load dependent Markovian birth-death model, the states of which represent the number of jobs that are ready to run. We define parameter L of the model as a limit on the number of simultaneously runnable jobs. This limit might result, for instance, from the limited memory capacity of the system.

For each processor allocation policy we consider, we set the completion rate, $\mu(S)$, in each state $S \leq L$ to $1/(E * EP_{policy}(S, 2^{M+N}))$, where E is the elapsed time the job would experience if run alone on the machine. This represents the equilibrium rate of job completions under *policy* given a constant workload of J jobs. For each state $S > L$, we set $\mu(S) = \mu(L)$.

We set the arrival rate of the model, λ, to produce a desired target system load, ρ. In particular, if E is the elapsed time required by the application when run alone on the full machine, we set $\lambda = \rho/E$.

This model captures the policy costs of induced load imbalance under Equipartition, of rotations under Folding, and of context switching under both policies. However, it does make a number of approximations; for instance, it ignores the costs of repartitioning due to job arrivals and departures. Its major benefits are its simplicity and low computational cost, which makes it easy to parameterize and allows us to obtain many performance estimates quickly. For example, on a workstation on which our simulator (described in Section 7) requires about 30 minutes to produce a single response time estimate, our birth-death model computes about one hundred such estimates in under a second. Comparisons

of the results of the birth-death model to those of the detailed simulation show that the birth-death model is very accurate, despite the approximations it makes.

6.1 Unlimited Memory Resources

In this subsection we examine response times under the assumption that each node has sufficient memory to multiprogram an arbitrary number of threads. In the next subsection we consider the case of limited memory.

Figure 9 show the estimates of the mean blow-up factor under two Equipartition and three Folding policies against system load for jobs with deterministic thread compute times, using the baseline parameterization of Section 5. The mean blow-up factor represents the factor by which the job's elapsed time exceeds its minimum, and is defined as the mean response time divided by the mean time to complete if run in isolation, E.

We note that, in general, response time is smaller under the Folding than under the Equipartition policies. This reflects their better efficiency preservation properties, and the fact that equal allocation of resources is unimportant to performance when there is a single class of jobs, as considered in this section. However, the results for $FOLD_{20}$ also make clear that there is a danger in choosing too small an inter-rotation interval for the Folding policies. As the system load increases, $FOLD_I$ becomes less efficient, for $I < \infty$. Thus, a value of I sufficiently large to achieve good performance at moderate loads may become unstable at high loads.

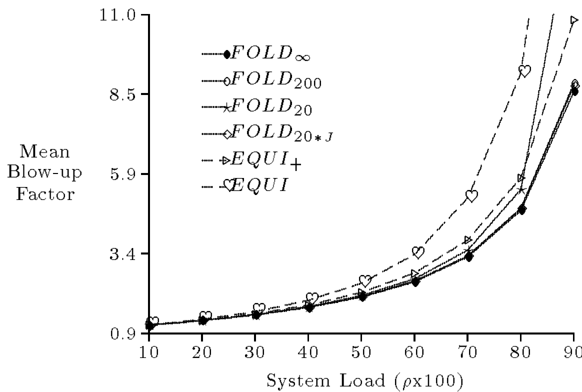

Figure 9: *Mean blow-up factor versus system load ($\delta = 0$)*

The explanation for this phenomenon is that as load increases, the average number of jobs in the system also increases. Since each job makes progress at a rate that is inversely proportional to the number of competitors, when I is a constant, the rotation overhead per unit of application progress grows with increasing load. Eventually, the relative rotation load exceeds capacity.

There is a remedy to this drawback, which is to use an inter-rotation interval of length $J * I$, where J is the current number of running jobs. Figure 9 shows that the performance of $FOLD_{20*J}$ is stable at high loads and very similar to $FOLD_{200}$. Even in the absence of this modification, it is not difficult to find values of I that provide reasonable fairness and are stable up to very high loads. In the rest of this paper, we will use inter-rotation interval lengths that vary with J.

6.2 Limited Memory Resources

In Figure 10, we graph mean blow-up factor against L, the multiprogramming limit, for the $EQUI_+$ and $FOLD_\infty$ policies, and a number of system load factors (ρ). (We show relatively high load factors because for $\rho < 0.5$ the number of simultaneously present jobs is almost always very small.) The results show how performance would be affected if the memory capacities of the nodes limited the number of threads that could be multiprogrammed at each, or if a policy were to impose such a limit voluntarily in an attempt to improve performance.

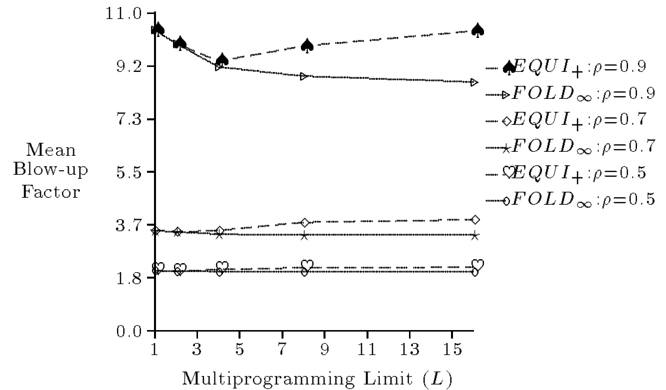

Figure 10: *Mean blow-up factor versus multiprogramming limit ($\delta = 0$)*

The results show that performance under the the Folding policy improves with increasing multiprogramming limit. This occurs because Folding has high efficiency preservation, which grows slightly higher with increased numbers of jobs in the system due to the better locality of communication of those jobs.

In contrast, Equipartition shows some tendency towards a local minimum, especially for high loads. This reflects the efficiency losses that Equipartition induces for most values of J larger than 4. The effect is not extremely pronounced, however, and in the next section we revisit the question of multiprogramming limit in the context of heterogeneous workloads, where there is an additional benefit to high limits.

7 Simulation Analysis: Heterogeneous Workloads

In this section, we use simulation to investigate the behavior of the policies under heterogeneous loads, as

well as their dynamic fairness properties. We obtained simulation point estimates using the batch means method. All results have a 90% confidence interval of width 5% of the point estimate.

The workload we study is composed of two job classes. The basic behavior of both classes is identical to that of the job class we have used to this point, that is, they are SPMD jobs performing repeated communication-compute cycles. The two classes differ from each other only in the mean number of cycles required to complete. For the shorter class of jobs, we set the mean such that the job would complete in 30 seconds if run alone on the full machine. For the longer class, a job would complete in 15 minutes. The number of cycles for an individual job of either class is chosen according to a geometric distribution.

As in the previous section, we divide our discussion into the memory constrained and unconstrained cases.

7.1 Unlimited Memory Resources

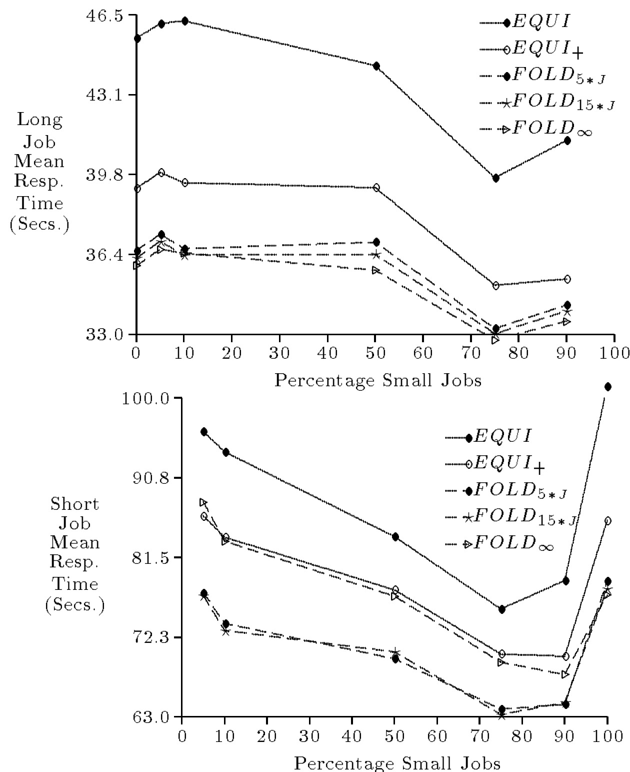

Figure 11: *Mean response times ($\rho = 0.5$, $\delta = 5$)*

We compute mean performance measures for the short and long job class under a variety of workload mixes. In each case, we set the overall arrival rate, λ, so that $p\lambda E_{short} + (1-p)\lambda E_{long} = 0.5$, where E_r is the mean elapsed time experienced by class r jobs if allocated the full machine.

Figure 11 shows the mean response times of the long and short job classes against p, the fraction of arrivals

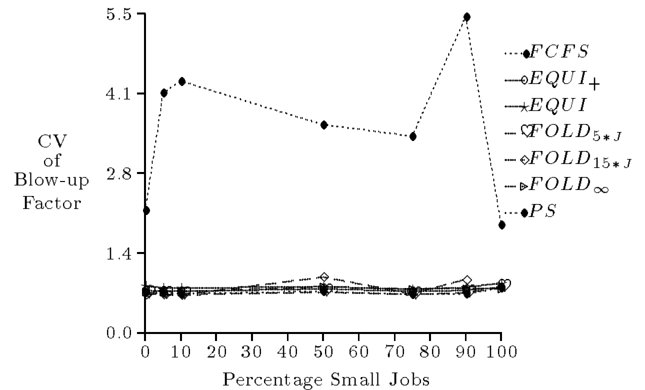

Figure 12: *Coefficient of variation of blow-up factor ($\rho = 0.5$, $\delta = 5$)*

that are short jobs. We see from these results that the Folding policies dominate the Equipartition ones, and that the best Folding policy ($FOLD_\infty$) yields response times about 10% smaller than the best Equipartition policy ($EQUI_+$).

Figure 12 compares the fairness of the policies. Here we graph the coefficient of variation of the blow-up factors of the jobs. In addition to the Equipartition and Folding policies, we also show results for Uniprogramming under both FCFS and processor sharing (PS). These serve to give some scale to the results, since it is well known FCFS has poor behavior for heterogeneous workloads, while PS has very good behavior.

From the figure, it is evident that all our policies behave very similarly with respect to fairness, and that even $FOLD_\infty$ performs about as well on this measure as could be hoped for any policy. We attribute this to the fact that all the policies employ space sharing, and thus provide at least some service to each ready job. In addition, the constant stream of job arrivals and departures provides an opportunity to shift resources without resorting to time-sharing.

7.2 Limited Memory Resources

Figure 13 shows how response time is affected under $EQUI_+$ and $FOLD_\infty$ when memory resources are sufficient to support at most L jobs at a time. The memory admission policy can have a significant effect on performance for heterogeneous workloads. Because there can be a significant performance penalty of small memory limits on short job performance, we modeled preemptive admission policies. Every Q time units, enough jobs are preempted so that any queued jobs can be assigned to processors. Q is a system definable parameter whose value depends on the cost of preempting processors. Define O as the overhead of preempting processors. This overhead is highly dependent on the availability of I/O bandwidth for swapping jobs in and out of the system. The I/O bandwidth varies widely among systems. Given a cost of preempting processors, O, the interval between preemptions, Q, can be set so that a

desired percentage overhead due to preemptions, O/Q, is incurred. Our simulations used two values for O/Q ranging from less than 1% up to 3%.

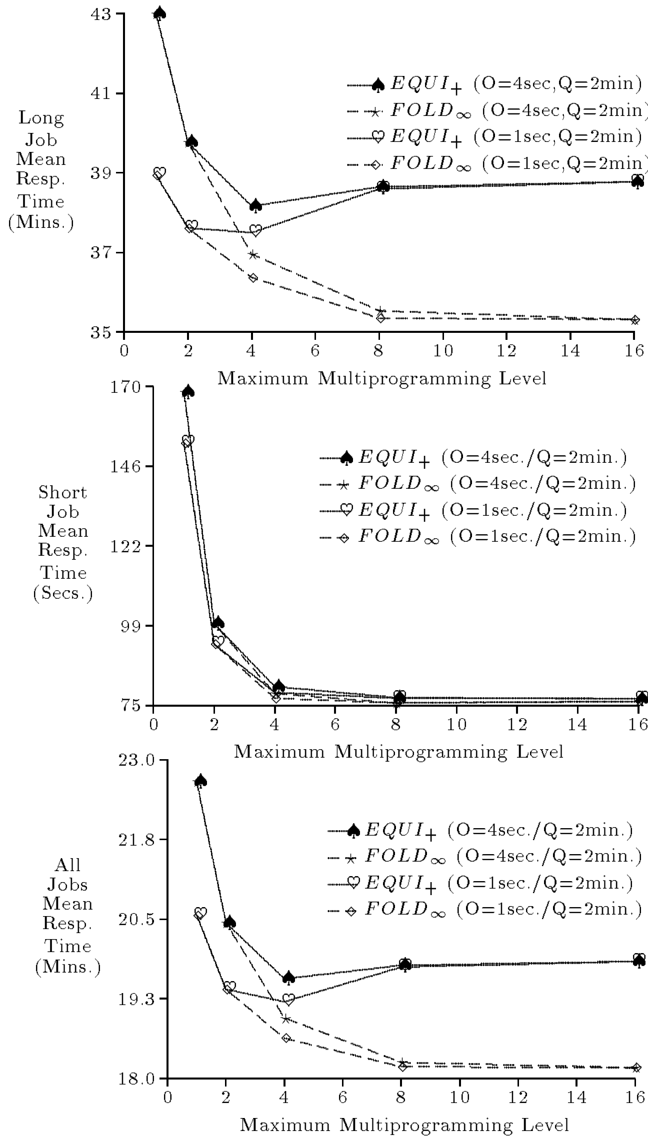

Figure 13: *Response time versus multiprogramming limit ($\rho = 0.5$, $\delta = 5$)*

We note that in terms of overall mean response time, small memory limitations are detrimental to Folding, but may be beneficial to Equipartition. The drawback of small multiprogramming limits under both policies is the reduced opportunity to put an arriving short job into service quickly The drawback to large multiprogramming limits for Equipartition is that it is inefficient for odd numbers of jobs in the system. The fact that the average response time grows with increasing L indicates that the penalty can outweigh the benefit.

Figure 14 shows how the coefficient of variation of blow-up factor for all jobs varies with multiprogramming limit. We see that both Folding and Equipartition

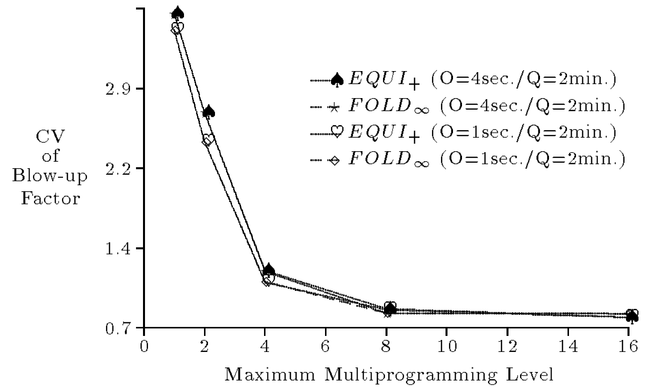

Figure 14: *CV of blow-up factor versus multiprogramming limit ($\rho = 0.5$, $\delta = 5$*

behave about the same, and that the blow-up factor is significantly larger for small values of L than for large ones.

From this data, we conclude that there is little or no penalty to the factor of two difference in resource allocation possible under $FOLD_\infty$, and so this policy appears to dominate the others.

8 Conclusions

We have examined issues in the the design of processor allocation policies for message-passing parallel computers running scientific workloads, making a number of contributions.

First, we have defined a measure of processor allocation disciplines, called efficiency preservation, and showed that it gave useful information for comparing the expected performance under alternative proposed policies. We also argued that this measure must necessarily be taken relative to a particular workload, as the suitability of an allocation policy is intimately tied to the workload to be supported.

In addition, we have proposed two specific families of processor allocation disciplines for mesh-connected machines. One, called Folding, emphasizes efficiency preservation over equality of resource allocation, while the other, Equipartition, is just the opposite. Using a static analysis, we examined the efficiency preservation of the two disciplines, finding that Folding is superior by this measure. Using a simple Markovian birth-death model, we evaluated mean response time under a homogeneous load, and found a member of the Folding family to perform best. Finally, we used simulation to examine both mean response time and fairness for a mixed workload of long and short jobs. We found here that the Folding policy continued to afford better response time performance, at little or no penalty in fairness.

Acknowledgements

The authors thank Martin Tompa for valuable discussions regarding the analysis of the Folding rotation scheme.

References

[1] I. Ashok. *Adhara: A Run-Time Support System for Space-Based Applications.* PhD thesis, University of Washington, In Preparation.

[2] M.-S. Chen and K.G. Shin. Processor allocation in an N-cube multiprocessor using gray codes. *IEEE Transactions on Computers*, C-36(12):1396–1407, December 1987.

[3] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientifc applications with explicit communication. In *Proceedings 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[4] K. Dussa, B. Carlson, L. Dowdy, and K-H. Park. Dynamic partitioning in a transputer environment. In *Proceedings of ACM SIGMETRICS Conference*, pages 203–213, May 1990.

[5] D.G. Feitelson. *In Support of Gang Scheduling.* PhD thesis, Department of Computer Science, The Hebrew University, December 1991.

[6] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, May 1990.

[7] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of ACM SIGMETRICS Conference*, pages 120–132, May 1991.

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

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

[10] 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.

[11] D.M. Nicol and J.C. Townsend. Accurate modeling of parallel scientific computations. In *Proceedings of ACM SIGMETRICS Conference*, pages 165–170, May 1989.

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

[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. Characterization of parallelism in applications and their use in scheduling. In *Proceedings of ACM SIGMETRICS Conference*, pages 171–180, May 1989.

[15] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, To appear.

[16] M.S. Squillante and E.D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[17] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[18] 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.

[19] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings 13th ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[20] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

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

[22] 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.