Processor Allocation Policies for Message-Passing Parallel Computers

Catherine M. McCann

Department of Computer Science and Engineering

University of Washington

Technical Report No. 95-07-03

September 1994

University of Washington

Abstract

Processor Allocation Policies for Message-Passing Parallel Computers

by Catherine M. McCann

Chairperson of the Supervisory Committee: Professor John Zahorjan
Department of Computer Science
and Engineering

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. This dissertation investigates the issues involved in constructing a processor allocation policy for large scale, message-passing parallel computers supporting a scientific workload.

First, the issues that affect the performance of scheduling policies for message-passing parallel systems are examined. We argue that reasonable policies must provide nearly equal resource allocation to all runnable jobs and allocate, to a single job, processors that are in close proximity to one another.

Second, the concept of *efficiency preservation* is defined as a characteristic of processor allocation policies. Efficiency preservation captures the impact of scheduling overheads such as reallocation cost and system-induced load imbalance on processor efficiencies. We show how efficiency preservation can be used in a first-order evaluation to identify promising scheduling policies.

Third, we address short-term scheduling, that is, how to allocate processors among a set of jobs where resource requirements allow them to be executed concurrently. The details of two families of processor allocation policies, Equipartition and Folding, are specified. Both policy classes employ dynamic allocation, but differ in the way they address the their costs. Folding achieves good application load balance at the cost of higher reallocation overhead, while Equipartition achieves low reallocation cost at the expense of higher system-induced load imbalance. Through performance evaluation of these policies, this dissertation shows that while maintaining low reallocation overhead is imperative to the good performance of dynamic allocation policies in message-passing systems, load balancing is also a dominant factor in the policy performance.

Fourth, this dissertation investigates medium-term scheduling policies, that is, how to choose subsets of the runnable jobs to execute concurrently when the total resource requirements of all the jobs exceeds the system's capacity. We argue that, in general, an efficient optimal solution to this problem is unlikely. As a result, this dissertation concentrates on restricted problem domains, and shows that, for these cases, optimal solutions exist.

Table of Contents

List of Figures

Chapter 1

Introduction

This dissertation is concerned with (kernel) processor allocation policies for large scale, distributed memory, message passing parallel computers. These systems are being developed to meet the ever increasing demand for processing power. Significant decreases in the cost of microprocessors and memory chips, coupled with increasing speed of interconnection networks, has made the development of large-scale distributed memory parallel computers more affordable. Systems with the high computational speed, large memories, and high-speed interconnection networks have the potential for achieving high performance, scalability, and extensibility, and offer an attractive platform for solving many large applications.

Effective operating system support to manage the resources of these systems will continue to be an important goal of research and development. To fully utilize their parallel processing power, it is imperative that every available opportunity be taken to exploit the parallelism of applications and realize the parallel computing potential of the architecture. According to [1], "A true general purpose supercomputer system will need to effectively exploit all parallelism available in an application set and do so in a manner that is convenient to its users."

Many avenues are being explored to efficiently execute parallel applications on these machines. At the most rudimentary level, applications are carefully written and structured to exploit parallelism and control the data distribution to maximize the parallelism of the application. To alleviate the complexity of such efforts, new parallel languages are proposed that efficiently express parallelism and data distribution. Runtime systems serve as intermediaries between applications and the kernel. They aid specific classes of applications in load balancing, dynamic adaptation of parallelism, and expression of resource needs. New operating system constructs reduce the communication latency between processing nodes, increase kernel support for synchronization, and export control over system resources to users, all in an attempt to reduce the overhead of kernel support for parallel activities.

In addition to achieving high performance on single applications, multiprogramming parallel systems can provide higher processor utilization and better aggregate response time to parallel programs than is achievable under uniprogramming. This dissertation investigates this avenue to realizing the full potential of parallel machines, that is, the implementation of processor allocation policies that decide how to divide the available processing resources of a parallel computer among competing parallel applications in order to achieve good application response time and the highest possible utilization of processors.

Processor scheduling in parallel computers is a two-level procedure. While some schedulers combine these two levels into a single scheduler, they are two separate functions. At the lower level, the kernel allocates processors to applications. Kernel-level scheduling is thus also referred to as processor allocation. The kernel-level scheduler decides how many and which processors to allocate to a job, and represents the needs of the system as a whole to provide the best overall system utilization and responsiveness to all applications. The kernel-level schedulable unit is called a kernel thread. A kernel thread is dispatched on each processor assigned to a job, and the job executes within those threads.

At the higher level, the applications decide which of their ready parallel operations (called user-level threads) to execute on each processor. The application scheduler assigns one or more user-level threads to each kernel thread. The application-level (or user-level) scheduling function may be contained within the application itself, may be determined by a parallelizing compiler, or may be provided by a runtime system. Its goal is to provide the best response time for that application, given its current processor allocation.

This dissertation is concerned with the lower level (kernel scheduling). As such, we make a conservative assumption about the interaction between the kernel scheduler and the application scheduler. We assume the kernel has little knowledge of what computation or communication occurs within the threads, and can not use any such information in making its scheduling decisions. In Chapter 2, we describe fully our assumptions about the interaction between these two schedulers.

While this dissertation focuses on distributed memory, message-passing computers, many of the ideas and some of the results can be extended to distributed memory systems with support for shared memory; however, in shared memory systems, which have support for remote memory reference, there is the additional scheduling issue of whether to migrate a job's data upon reallocation of its processor or allow it to access the data remotely. This dissertation does not address this issue.

1.1 Taxonomy of Scheduling Policies

Kernel-level scheduling policies may be categorized according to two types of decisions they make: when to change the partitioning of processors and when to change the allocation to jobs. With respect to partition changes, the simplest policies employ *static* partitioning of the processors, in which the multiprocessor is divided into fixed size partitions. The scheduling policy decides which partition to allocate to jobs, but the size and configuration of the partitions is not controllable. *Adaptive* partitioning policies are more aggressive, adapting the size and configuration of partitions to changes in the workload and system environment. When system load is light, few partitions are defined; when load is heavy, more partitions are defined to accommodate more jobs simultaneously.

With respect to allocation changes, scheduling policies may employ *static* or *dynamic* allocation. Static allocation policies allocate a fixed number of processors to each job for the job's entire execution. The number of processors allocated to a job is determined when it is loaded, based on the job's parallelism, on job size, or on system environmental considerations such as current system load. Dynamic allocation policies may change the number of processors allocated to each job throughout its execution lifetime in response to changes in the job's parallelism or changes in system load.

By varying the combination of partitioning and allocation policies, we define four classes of scheduling policies that differ in the aggressiveness with which they pursue the goals of high system utilization and good application performance. We briefly discuss each of these classes in turn.

1.1.1 Static Partitioning and Static Allocation

The simplest scheduling policies employ static partitioning and static allocation. Processors are divided into a fixed number of partitions, and jobs are allocated to a single partition for their durations. The earliest systems implemented only a single partition, and jobs were executed one at a time until they completed. Later, job scheduling consisted of allocating time-slices during which a single job was executed alone on the whole machine. After its time-slice, the job was preempted and a new job scheduled. Another approach divided the system into some number of partitions. Each job was assigned to one of the partitions, The processors of a partition were time-shared among multiple jobs assigned to the partition. Often the partitions were used to separate different classes of jobs, such as non-parallel interactive jobs from parallel jobs, or long-running from short term jobs.

In contrast to the time sharing policies above, *space sharing* policies allocate resources to jobs by partitioning processors among multiple jobs. Space sharing policies make more effective use of processors than uniprogramming. Although most applications benefit from increasing allocations, their typically sub-linear speed-up implies that at some point additional processors would be better utilized by another application. Also, for applications that can not make use of all processors, other jobs are able to utilize processors that might otherwise be idle.

Within static partitioning, policies either assign multiple jobs to a partition or require jobs to be queued waiting for available partitions. If multiple jobs are assigned to each partition, the processors within a partition are time-shared among the jobs in that partition. This has the potential benefit of allowing a smaller queueing time, which is especially beneficial to short jobs. There is also a benefit to overlapping the computation time of one thread with the time spent blocked on some event by another thread. The scheduler for Thinking Machine's CM5 supports static partitioning and time-shares partitions.

The static division of the machine provides a relatively stable resource allocation on which an executing job can depend. Both data and computation placement may be decided at load time to maximize the performance for the specific processor allocation. However, if the number of jobs in the system is less than the number of partitions, processors will remain unallocated even though the executing jobs could use them, thus adversely impacting performance.

1.1.2 Adaptive Partitioning and Static Allocation

More aggressive partitioning policies allow changes in the size and configuration of the partitions in response to changes in the workload and system load. In this way, the number of partitions can be adjusted to correspond to the number of jobs in execution, thereby reducing the potential for idle processors. Further, the size of a partition allocated to a job may be chosen to better reflect the job's parallelism, thus increasing the efficiency of the allocated processors.

Adaptive partitioning policies have been proposed for cube-connected architectures such as the Hypercube [7, 8]. These policies assume a job requests a subcube of a particular size. The scheduling policy allocates subcubes to newly arriving jobs. The subcube allocated to a job remains the same throughout the job's lifetime. When a job finishes, its subcube is combined with free neighboring subcubes, if any, so that a larger idle subcube becomes available. Since jobs request subcubes of varying sizes, these adaptive partitioning policies try to reduce the number of idle processors and provide better processor utilization than static partitioning policies.

Another form of adaptive partitioning allows the number of processors assigned to a job to be determined when the job is first scheduled. The number of processors allocated to a job can be a function of the number of jobs waiting to be scheduled, the number of processors currently available, and the parallelism of the jobs to be scheduled.

1.1.3 Static Partitioning and Dynamic Allocation

Dynamic allocation coupled with static partitioning may be thought of as a means of load balancing among static partitions. Here, the processors are divided into fixed-size partitions, and parallel jobs are assigned to a partition. The scheduler may migrate parallel jobs to other partitions during their execution in order to balance the load across the partitions. Most load balancing policies reallocate across identical sized partitions; thus, dynamic allocation is not involved. It is doubtful that the cost of dynamic allocation would be justified for load balancing across static partitions. Thus, the combination of static partitioning and dynamic allocation is of benefit, and few systems employ this combination. For this reason, the term *static partitioning* generally implies static allocation and the term *dynamic allocation* generally implies adaptive partitioning.

1.1.4 Adaptive Partitioning and Dynamic Allocation

Adaptive partitioning with dynamic allocation offers the most aggressive form of scheduling. Both the configuration of the partitions and the number of processors allocated to a job can changes during the job's execution. To achieve high utilization, processors are reassigned to other jobs when a job departs. In fact, dynamic allocation could be done at any time: the same mechanisms required to reallocate at job departure can be used to accommodate an arriving job, with each running job relinquishing some processors to the new job. Alternatively, the number of processors allocated to a job may change in response to changing parallel needs of the job, again making more effective use of the processors.

The overhead of reallocation determines the frequency with which it is appropriate. In shared memory systems, it has been shown that the benefits of dynamic allocation outweigh the cost of more frequent reallocations. However, there has been only limited experience with dynamic allocation of processors to jobs in distributed memory systems. While an aggressive manipulation of the system offers a greater potential for increased performance, the costs of effecting these changes in a distributed memory environment may outweigh the benefits. Existing scheduling policies for distributed memory systems generally employ a static form of allocation 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 [45, 41] show how to reduce the latency currently imposed by large message startup costs to a negligible level. These trends indicate that dynamic allocation should be investigated as a viable scheduling strategy for future parallel machines.

This dissertation explores this category of scheduling policies for message-passing distributed memory systems that both dynamically change the allocation of processors to jobs and change the size and configuration of partitions. In doing so, it lays the foundation for the design of processor allocation policies to be employed in the next generation of multiprocessor kernels, which will enjoy high network bandwidths and low message startup costs, thus making the reallocation cost feasible.

1.2 Previous Work on Processor Allocation Policies

An early and fundamental result on processor scheduling for parallel machines is due to Ousterhout [32]. He notes 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 "coscheduled", that is, run at once. If the kernel threads assigned to a job are not coscheduled, there can be significant inefficiencies due to synchronization losses: application-level threads executing on one processor may be forced to wait for a thread that is blocked on another processor. Ousterhout [32] shows that coordinated scheduling is imperative to good performance in time-shared scheduling of parallel systems. This effect also forces processor allocators to operate on a per-job granularity, rather than a per-thread one. This highlights the need to view scheduling as a two-level procedure, separating the kernel-level processor allocation decisions from the application-level decisions of which user-level threads to execute on each processor. Work by Ousterhout [32], Zahorjan et al. [47, 48] and Gupta et al. [20] shows that *single-level* schedulers, where a kernel schedules all threads of all applications (usually from a single queue of ready threads), provide poor performance to individual jobs. These schedulers also provide unfair service when applications contain different numbers of threads.

Base on these early results, we restrict our attention to *two-level* schedulers and focus on the kernel-level scheduling policy. This section reports on previous work on kernel-level scheduling. First, we survey results from the study of scheduling policies specifically designed for a uniform memory access (UMA) shared memory environment. Many of the key results from this domain are also relevant to the distributed memory environment. Next, we report on work specific to a distributed memory environment, where the scheduling policy must reflect the specific environmental considerations of distributed memory in making its allocation decisions.

1.2.1 UMA Shared-Memory Systems

Much research on shared memory systems compares the performance of time-sharing scheduling policies, where each job is allocated a time-slice during which it executes on all the processors, to space sharing policies, where the processors are divided among multiple running jobs. Tucker and Gupta [42] 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 coscheduling, 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 [49] compare time sharing to space sharing, and conclude that space sharing is preferable. Space sharing policies make more efficient use of processors that might otherwise be idle when running a single application, and in addition jobs with sublinear speedup execute more efficiently when run on fewer processors. They also describe a more aggressive form of coscheduling 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. [28] further refine this policy, and report on results from a prototype implementation of it. Their results showed that the benefits of dynamic reallocation of processors among jobs in a UMA environment outweigh the costs of more frequent reallocations.

Many studies examine the relationship between job characteristics and the performance of various scheduling disciplines. In examining the tradeoff between application speedup and processor efficiency, Eager et al. [16] show how allocating a job's "average parallelism" provides a good approximation of the number of processors that best balance processor efficiency with job execution time. Majumdar et al. [27], Sevcik [38, 39], Leutenegger and Vernon [24], and Chiang et al. [10] examine the relationship between job characteristics, such as average or maximum parallelism, variance in parallelism, or 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.

Zahorjan et al. [47, 48] use modelling to examine the effect of scheduling discipline on spinning overhead and show that preemption of processors must be done in a coordinated manner. If a scheduling policy preempts a processor from a running job, and that processor was executing a thread that held a spin lock, then other executing threads on other processors waiting for that lock can not make progress until the blocked thread is rescheduled. Further experimental studies [28, 12] confirm this. Anderson et al. [2] propose kernel-level mechanism to support coordinating processor preemption.

Gupta et al. [20] and Vaswani and Zahorjan [44] 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 conclusions, based on results from an analytic model [40].)

1.2.2 Distributed Memory Systems

The first parallel systems used very simple scheduling policies that allocated the entire parallel machine to each waiting job in turn for some time quantum or until the job completed. Early modelling studies [14, 25] expose the potential performance gain of multiprogramming over uniprogramming.

Early work on multiprogramming distributed memory systems extends policies developed for the shared-memory environment to the distributed memory environment. Crovella et al. [12] report on an implementation of a time-slicing policy, a coscheduling policy and a dynamic allocation policy on a NUMA system (BBN Butterfly). Their experiments show that the dynamic allocation policy outperforms the coscheduling and time-slicing policy, in part due to less cache corruption and fewer remote references than required under coscheduling or time-slicing. These results are consistent with results from the shared memory environment demonstrating that space sharing is preferable to time-sharing policies and that uncoordinated preemption is detrimental to system performance.

Setia et al. [37] compare static partitioning policies where one job executes in each partition, to multiprogrammed policies, where partitions may be time-sliced among multiple jobs. They show that at moderate to high system loads, multiprogramming policies can outperform static partitioning policies.

Feitelson and Rudolph [17, 18] examine variants of gang scheduling. They propose a hierarchical scheme for assigning applications to processors, and examine the fragmentation associated with gang scheduling 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.

Many studies of distributed memory systems focus on partitioning policies for parallel system. Zhou and Brecht [50] describe a pool-based scheduling policy. 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.

Dussa et al. [15] examine the benefits of dynamically repartitioning processors on job arrival and departure, with 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 among jobs based on their total durations and because the sub-linear speedup of typical applications makes them more efficient when run on fewer processors.

Rosti et al. [35], examine adaptive partitioning policies for multiprogrammed multiprocessors where the number of partitions changes upon arrival and departure, and where the number of new partitions created from freed processors depends on system load. They analyze robustness, where robustness is a measure of a policy's performance when given little or no knowledge of job characteristics.

Chen and Shin [8] examine processor allocation policies for cube-connected message passing machines. Like Feitelson and Rudolph, they assume that arriving jobs declare the number of processors required for execution. The scheduling policy allocates subcubes to a newly arriving job. The size of the subcube allocated to a job remains the same throughout the job's lifetime. When a job finishes, its subcube is combined with free neighboring subcubes, if any, so that a larger idle subcube becomes available. The goal of their work is to describe efficient schemes for finding sub-cubes to satisfy arriving jobs and minimizing the fragmentation of available subcubes. Later work [9] describes a global parallel task migration strategy to reduce the available subcube fragmentation.

Other studies consider batch scheduling algorithms where the number of jobs, the parallelism of the jobs, and jobs' execution time needs are known in advance. Instead of minimizing overall response time, they consider *makespan*, which is the total time necessary to complete all jobs. Chen and Lai [7] study the problem of allocating subcubes in a hypercube environment. Li and Cheng [26] consider a mesh-connected environment where jobs require square submeshes. They propose a two-dimensional buddy system partitioning scheme. Both studies give worst case bounds over an optimal schedule.

Turek et al. [43] consider scheduling policies in which jobs may execute on any number of processors, and their execution times, a function of the number of processors allocated, are known for all jobs. They show that the general scheduling problem of finding an allocation of processors to jobs that produces the minimum makespan schedule is NP-hard. They propose a shelf scheduling strategy, which defines intervals (called shelves) to which jobs are allocated processors. At the end of an interval, all processors are reallocated to jobs in the next shelf. They prove their self-scheduling algorithm finds the minimal length schedule within the class of shelf algorithms.

There has been limited experience with dynamic allocation in distributed memory systems. Setia et al. [29] compare a dynamic allocation policy to static partitioning and adaptive partitioning for varying workloads. Under their dynamic allocation policy, partition sizes are changed upon arrival and departure to accommodate newly arriving jobs or to utilize processors freed by departing jobs. They assume the application redistributes its workload at the next safe point in its execution after is is notified of the reallocation. The cost of reallocation is determined by experimentation using a conservative redistribution scheme that collapses all threads of a job onto some minimum number of processors in the new partition and then expands the threads onto the full partition. Their results show that dynamic allocation can provide the best overall performance. However, they observe that dynamic allocation performs worse for large jobs than adaptive partitioning at high system loads, since dynamic allocation allocates fewer processors to those jobs at high loads than do the other policies. Their policies do not consider adjacency of multiple processors allocated to a job, and thus ignore the potential performance impact of interconnection network contention among multiple applications competing for the same interconnection network links.

Several studies [30, 29, 38] explicitly demonstrate the necessity of reducing the allocation of processors to jobs as the system load increases. Peris et al. [33] examine the effects of increased paging overhead accompanying a reduced allocation on job performance. Their results illustrate the necessity to consider the effects of memory on scheduling large memory-intensive applications.

1.2.3 Building on the Results of Previous Work

In examining the results of prior work in both the shared memory and distributed memory environments, we conclude that high performance 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.

- Equal allocation — In the absence of reliable information on the resource requirements of each job, each running job should be allocated a reasonably equal portion of the resources, for reasons of fairness and performance.

This dissertation concentrates on dynamic allocation policies as the most promising approach to scheduling these systems.

1.3 Thesis Outline

This dissertation is organized as follows. Chapter 2 examines the issues in multiprogramming message-passing parallel systems, with a special emphasis on issues particular to dynamic allocation policies necessary for a comprehensive evaluation of the performance of dynamic allocation policies. Chapter 3 defines a new metric, called efficiency preservation, for evaluating the performance of processor allocation policies for parallel systems. We show how this metric can be used in a first-order evaluation to identify promising scheduling policies. We also use this metric to show the potential benefits afforded by dynamic allocation over static partitioning policies.

In Chapter 4, two specific classes of scheduling policies are proposed. Both policy classes employ dynamic allocation and adaptive partitioning: the size and configuration of the processors allocated to jobs changes throughout their execution. These policies differ in the way they address the costs of dynamic reallocation. One policy, called Equipartition, always partitions the processors equally among the competing jobs. Reallocation occurs only when jobs complete or new jobs arrive in the system, relatively rare events. However, since the number of processors allocated to a job may not evenly divide the number of threads in a job, a system induced load imbalance can degrade job performance. The second policy, called Folding, avoids system induced load imbalance by always halving or doubling the number of processors allocated to a job. The Folding policy does not allocate equal-size partitions. To ensure equal resource allocation, the jobs' allocations alternates between larger and smaller partitions. Here, the frequency and cost of these reallocations affects the job's performance.

Chapter 5 evaluates the performance of the policies presented in Chapter 4 to ascertain the relative impact on performance of reallocation overhead and system induced load imbalance. In comparing the performance of the policies, both the response time equal resource allocation among competing jobs (a measure of fairness) are considered. The performance evaluation is performed using quantitative analysis and simulation.

In Chapter 6, we examine policies that take into account the memory constraints of large applications. The existence of a lower bound on possible allocations clearly complicates scheduling policies. Here, it is not always possible to schedule all jobs concurrently, so some time-shared scheduling policy must be employed. We look at scheduling policies that incorporate another level of scheduling to accommodate the situation where large, memory-intensive applications have a minimum number of processors on which they can run.

Chapter 7 summarizes the conclusions of this dissertation and identifies areas of potential future research.

1.4 Thesis Contributions

This dissertation makes the following contributions:

- The important issues relevant to the performance of scheduling policies for message-passing systems are identified, with special attention given to aspects affecting dynamic allocation of processors.

- A new metric, Efficiency Preservation, is defined for evaluating the performance of processor allocation policies for message-passing parallel systems.

- Two dynamic allocation policies are developed that consider the major aspects of processor allocation necessary for a comprehensive understanding of their performance in real systems.

- A performance evaluation of these policies identifies the relative importance of reallocation overhead verses system induced load imbalance. Results show that avoiding system induced load imbalance is important to the performance of scheduling policies.

- Scheduling policies relevant to the scheduling of parallel applications with large memory requirements are studied, and feasible policies appropriate to this domain are defined and analyzed.

- Through the study of scheduling algorithms for large memory-bound applications, a new variant to the class of two-dimensional orthogonal bin-packing problems is introduced. While the complexity of this problem remains open, we show that it is a subset of a similar problem that is NP-complete, and thus is unlikely to be solved efficiently.

Chapter 2

Issues in Designing and Evaluating Scheduling Policies for Parallel Computers

This chapter examines the issues that affect the performance of scheduling policies for message-passing parallel computers, and as such, must be considered in the design and evaluation of such policies. We discuss both the issues relevant to multiprogramming scheduling policies in general, and considerations specific to adaptive partitioning and dynamic allocation policies. We describe the decisions made in selecting among the design alternatives and present the rationale behind those decisions.

2.1 The Hardware and Software Environment

A scheduling discipline manages the resources within a specific environment. Its behavior is intimately tied to the environment for which it is defined. Therefore, to assess the performance of scheduling policies, we first need to identify the environment in which they are appropriate.

This dissertation focuses on scheduling policies for non-uniform memory access (NUMA) architectures. Specifically, it examines policies appropriate to systems in which each node consists of a processor and memory, and nodes communicate via message-passing. Examples of such systems are the Intel Paragon, the Intel Touchtone Delta, and the Thinking Machine CM5.

We consider the particular case of a mesh-connected parallel machine consisting of 2^M x 2^N nodes, as shown in Figure 2.1. 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 the work applies to machines with a tree interconnection structure (such as the CM5), and to machines where each processing node contains a small cluster of processors. Additionally, it should not be hard to translate many of our ideas to other interconnection structures.

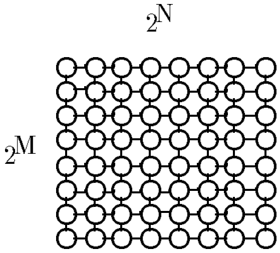


Figure 2.1: *Hardware Environment*

To undertake a substantive analysis, it is also necessary to make assumptions about the software environment. We consider workloads consisting of large parallel scientific applications. Jobs consist of multiple concurrently executing threads that need to communicate and exchange data relatively frequently throughout their execution.

This dissertation assumes scheduling policies have no knowledge of job characteristics that can be used to ascertain expected completion time or parallelism needs. Information on job characteristics (such as the expected computation time and speedup of applications, or their minimum, maximum or average parallelism) is useful in making better allocation decisions. However, such detailed information often is not known. Furthermore, false job characterizations could be used by jobs to circumvent scheduling policy decisions: for example, a job could artificially inflate parallelism requirements in order to be allocated more processors than it would otherwise be entitled to; or long jobs could represent themselves as short jobs in order to acquire service more quickly. Additional measures must be taken to counteract these attempts.

2.2 Equal Resource Allocation

One goal of scheduling policies is to provide equal allocation of resources among competing jobs. 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. It has been shown that in the absence of information on the parallel characteristics of the workload, policies providing equal resource allocation not only provide a measure of fairness to jobs, but also serve as an approximation to the 'shortest job first' policy, which is desirable from a performance perspective.

There are two basic measures of equality that may be applied. Both measure equality in terms of processor-seconds/second over a window of time $(t-T,t)$. For $T \rightarrow 0$, policies provide equal resource allocation via space sharing, where the machine is divided into an equal number of processors for each job. (Of course, when the number of jobs does not evenly divide the number of processors, some inequality must be tolerated.) For $T > 0$, policies combine some time and space sharing. These policies may allow allocations to differ among jobs at any one time, but provide the same average number of processors to all jobs within an interval T.

The quest for equal resource allocation can sometimes come with a cost to performance if additional shuffling of processors among jobs is needed to achieve equality. The tradeoff of equality and performance is one focus of our analysis of scheduling policies.

2.3 Processor Adjacency

On uniform memory access (UMA) shared-memory systems, processors share a common global memory, and the cost of accessing that memory is the same from all processors. Therefore, scheduling policies may decide how many processors to allocate to a job without concern for which processors the job is allocated. (However, studies have been done to investigate the effects of cache affinity on the performance of scheduling policies in these systems [44, 40].)

In a distributed memory environment, the scheduling policy must decide *which* in addition to *how many* processors to allocate to each competing job. Parallel application performance is affected by the proximity of the job's data to the threads that access the data, as well as by the proximity of the threads of a job to each other. To attain good performance, an application is structured so that most data references are to data in local memory.

Still, parallel threads of an application often communicate data and synchronization information. The cost of this communication affects the performance of the applications. If communicating threads reside on different processors, the time to complete the communication depends on the the network interconnection bandwidth, the message latency, and the contention for network resources. Hardware advances continue to improve the bandwidth of interconnection networks. Also, recent hardware and software advances [45, 41] show how to reduce the latency imposed by large message startup costs to a negligible level. However, despite high bandwidth and low latency, contention for network resources could severely affect job performance.

The potential for network contention affects the design of kernel scheduling policies. It is important that scheduling policies allocate to a job processors in close proximity to one another and forming regular shapes. If processors are not in close proximity, messages between processors belonging to one job would be routed over links connecting processors allocated to other jobs. Not only would tuning the performance of applications be difficult, but contention for links could severely degrade job performance. Also, a mismatch between the application's model of thread configuration and the actual configuration of the processors allocated to the job could result in excessive communication overhead and poor load balancing of the application's threads among the processors of a partition.

To reduce the interference between concurrently executing jobs, we restrict our attention to partitions that have *processor adjacency*. By processor adjacency, we mean that the communication between any two processors of a partition does not involve crossing links connecting processors of another partition. Allocating adjacent processors to a job also reduces the cost of reallocation. For the two-dimensional mesh-connected environment we are considering, this requires rectangular allocations of processors to jobs.

Achieving processor adjacency is complicated by the additional constraints of providing equal resource allocation and dynamically changing partitions. When a new job arrives, a new partition must be created to accommodate the job. To maintain equal allocation, processors must be preempted from one or more other partitions and assigned to a new partition. However, simply creating a new partition by selecting processors in other partitions will not result in the new partition having processor adjacency. Instead, some global rearrangement of the processors into new partitions is needed. Similarly, upon job departure the processors in the vacated partition must be distributed among the remaining partitions in a manner that preserves partition adjacency and maintains equality of allocation. Furthermore, this method of rearranging processors among partitions must be determinable for all multiprogramming levels.

2.4 Job Distribution (Mapping) Policy

Schedulers employing dynamic allocation can change the number of processors allocated to a job as the job executes. When the number of processors allocated to a job changes, the job's computation and data must be redistributed on the new set of processors assigned to the job. The manner in which the this redistribution is accomplished is called the *mapping* problem. There are three approaches to handling the mapping problem:

Application-level Mapping. The job itself could be responsible for the distribution. In this case, the kernel-level scheduler notifies the job of a change in allocation and the identification of the new set of processors assigned to the job. The job remaps its threads on the new partition. When the mapping is complete, the job signals the kernel, and the kernel continues scheduling.

This approach has the advantages that the application can redefine the granularity of threads to match its allocation, and can use knowledge of thread load and communication structure to find the best mapping for that particular allocation. Also, a runtime system could wait until the application-level threads reaches a safe state before redistributing its computation. However, waiting for the job to remap itself may cause excessive delays that slow the execution of other jobs assigned to those processors. This also requires the kernel to trust the job to relinquish the processors in a timely manner.

Kernel-level Mapping. With kernel-level remapping, the kernel is responsible for mapping the jobs threads onto the allocated processors. It should do so in the best way possible in the absence of application-specific knowledge of thread load and communication structure.

To do kernel-level mapping, a job's computational structure is defined in terms of a virtual machine. The virtual machine structure reflects the processor topology. (Thus, in our case, the virtual machine is a 2^Mx2^N mesh of processors.) This mapping 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 [22, 23, 34, 21]. Each thread of the virtual machine becomes a kernel thread, and the kernel makes load balancing and data placement decisions for the application in the context of a fixed virtual machine.

When a job's allocation changes, the the kernel-level scheduler takes immediate action to reallocate the processors among the jobs. The kernel decides which processors to allocate to which jobs and remaps the data and kernel threads of the jobs onto the new set of processors allocated to those jobs. The kernel then notifies the job of the allocation change. A runtime system may still relocate the job's threads after the kernel reallocation.

This immediate preemption is advantageous in that reallocation is completed quickly and kernel threads for a new application may be immediately started on the vacated processors. This also relieves that application of the burden of adapting to allocation changes. However, this approach requires the kernel to make decisions about the mapping of the job's threads and data based only on the initial virtual machine mapping provided by the application and without specific knowledge of the application's structure.

Application Mapping with Timeout Under this approach, the kernel-level scheduler notifies the job of a change in the allocation to a job and the identification of the new processor partition assigned to the job. The job performs an application-level remapping. If the job has not completed the remapping after some appropriate time interval, the kernel-level scheduler preempts the processors from the job and performs a kernel-level remapping.

This has the advantages of allowing the job to redistribute its own load, while also ensuring that uncooperative jobs can not circumvent kernel-level scheduling decisions. However, determining an appropriate time-out interval is not easy, since reallocation overhead depends on many factors (e.g, the size of the job to be remapped and network contention).

Also, even if a time-out interval were used to allow the job to redistribute its load, a kernel-level redistribution policy must be defined and implemented to handle expiration of the interval.

Our approach: Kernel-level Mapping This dissertation assumes kernel-level remapping is used when reallocating processors among jobs. Some applications have support for mapping onto partitions at load time. However, few applications support dynamic remapping during execution. Considerable effort is required to implement application-level dynamic remapping, and to determine when to remap can be complicated [31]. This effort may be justified only for very irregular and dynamic computations [3]. We assume a kernel scheduling policy can not presuppose this support nor wait for a runtime system to remap an application. A general-purpose kernel-level scheduling policy must be able to take immediate action to support global system changes affecting all jobs, and can not rely on the cooperation of applications to voluntarily relinquish processors in a timely manner.

The kernel-level scheduler must incorporate some general purpose remapping policy applicable to a broad range of applications. In defining a remapping policy, it is important for the kernel-level scheduler to remap an application's threads on its new allocation in a reasonable manner. The kernel threads should be evenly distributed among the processors allocated to the job. Also, remapping should preserve the adjacency of kernel threads assigned by the application to the virtual machine. In other words, the remapping policy maintains thread adjacency of the virtual machine when running on a restricted set of processors as when running on the entire machine. This results in a mapping that corresponds to the application's computational structure, which minimizes communication costs for an allocation. It also provides a similar physical communication structure across varying allocations, which is necessary for application tuning.

As previously stated, the kernel defines a computational structure in terms of a virtual machine. For the environment considered here, the virtual machine is a two-dimensional structure identical to the parallel machine (see Figure 2.2). Our kernel remapping policy uses a straightforward modulo division of the kernel-threads along both dimensions of a partition. For example, to distribute the threads of an 8 x 8 virtual machine onto a 3 x 5 partition, the threads are divided into 3 rows and 5 columns, as shown in Figure 2.3. (The mapping policy is fully defined in Chapter 4.) Notice that there is a possible load imbalance in that some processors have more threads assigned to them than other processors. This potential load imbalance can severely affect the performance of applications.

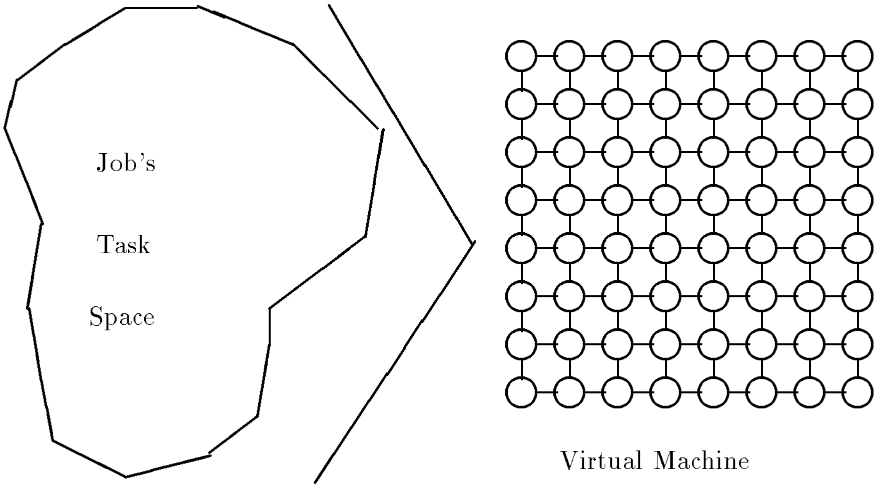


Figure 2.2: *Kernel-level remapping*

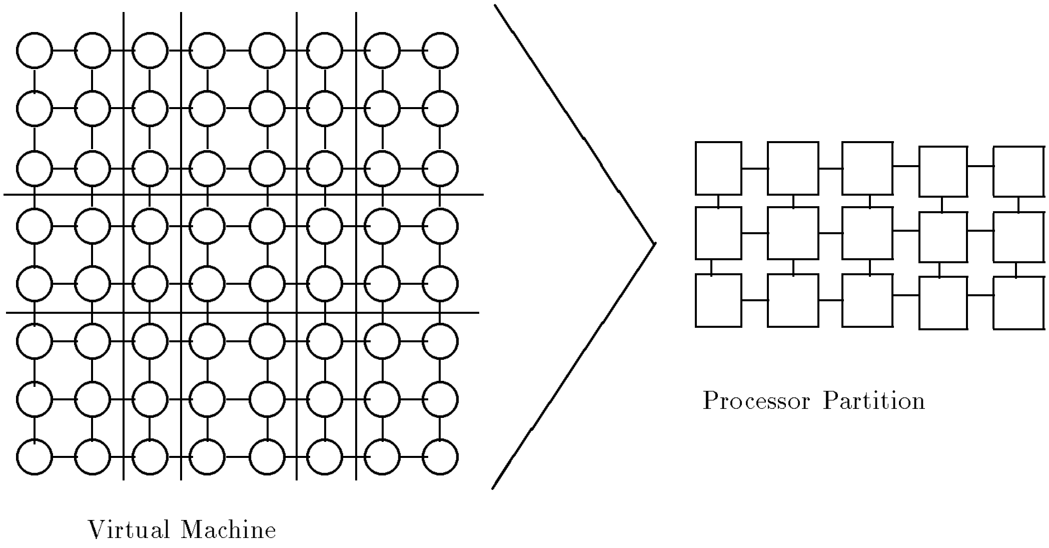


Figure 2.3: *Example kernel mapping onto 3x5 partition*

2.5 Scheduling Overhead

Two primary factors affect the overhead of scheduling policies for distributed memory systems: the overhead of reallocation, and the subsequent effect on job performance. Each of these factors must be considered in the design of scheduling policies for distributed memory parallel systems. We briefly discuss each of these factors. In Chapter 4, we describe more fully how these factors affected the design of the policies proposed in this dissertation.

2.5.1 Overhead of Dynamic Reallocation

The design of scheduling policies must consider and try to minimize the potentially high cost of job reallocation in a distributed memory environment. The overhead of reallocating processors among jobs can be significant in a message-passing environment. Reallocation involves migrating not only the job's state, but also the job's locally resident data. When processors are reallocated from one job to another, the tasks currently running on the processors and the data for the job local to those processors must be redistributed to remaining processors assigned to the job. Similarly, when additional processors become available to a job, the tasks and program data for that job must be redistributed among a new set of processors. The cost of doing this depends on the number of tasks and the amount of data being migrated, the latency and bandwidth of the interconnection network, and the potential cost of contention for network resources if more than one job is reallocated.

In addition, overhead due to network contention with other executing jobs also affects the cost of reallocation. Contention is a function of the relative position of the old and new processor partitions allocated to jobs within the processor topology and the extent to which these migration paths between reallocating jobs intersect.

The scheduling policies presented in Chapter 4 seek to minimize the amount of data migrated during a reallocation and the contention between multiple reallocating jobs.

2.5.2 Effect on Job Performance

In addition to the overhead of reallocating processors, there is a potential overhead due to the subsequent effect on job performance.

It is inevitable that a program's parallel execution involves overheads not present in its sequential execution. These overheads result from the need to communicate data and synchronization information among the threads of the parallel job, and synchronization losses due to load imbalance. 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. Of course, changing the number of processors allocated to a job affects its rate of progress. Furthermore, the size and configuration of processors allocated to a job affect the job's performance. For instance, if an application that has partitioned its load into eight threads is allocated exclusive use of five processors, it is likely that three of the five processors will have twice the number of threads of the other two. If the threads synchronize or communicate often, a great deal of processing capacity will likely be lost. 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 designs of the scheduling policies presented in Chapter 4 consider this effect of the partition configuration on the performance of applications assigned to those partitions.

2.6 Summary

Based on the discussion of scheduling issues in this chapter, we summarize here the design criteria for scheduling policies for message-passing parallel systems:

- We assume that scheduling policies have no knowledge of job characteristics that can be used in making scheduling decisions.

- Scheduling policies should provide some measure of equal resource allocation to competing jobs.

- Because of potential network contention among multiple jobs, it is important that scheduling policies allocate processors to jobs that are in close proximity to one another. This adjacency must be preserved at all multiprogramming levels.

- A kernel-level mapping policy is defined to specify the distribution of a job's computation and data on varying sized partitions.

- The design and performance evaluation of scheduling policies should consider both the overhead of reallocation and the subsequent effect on application performance.

Chapter 3

Efficiency Preservation

Evaluating the performance of dynamic allocation scheduling policies is difficult in the face of the many factors affecting performance, as discussed in Chapter 2. This chapter introduces a simple performance metric, efficiency preservation, that measures the extent to which a policy induces efficiency losses. Efficiency preservation can be used as a first-order evaluation to identify promising scheduling policies. This chapter defines efficiency preservation and discusses its use as an early predictor of scheduling policy performance. As an example, we demonstrate how efficiency preservation can be used to show the potential benefits of dynamic allocation over static partitioning.

3.1 Efficiency Preservation

We define *efficiency preservation* as a characteristic of processor allocation policies that measures the extent to which an allocation policy induces processor efficiency losses. Processor efficiency is a measure of the time processors spend in productive computation. Processor efficiency is affected by both application behavior and scheduling policy characteristics. Efficiency losses attributable to application behavior result from insufficient parallelism to utilize the processors allocated to the job or excessive synchronization among the application's threads. Efficiency losses due to scheduling policy behavior result from unallocated processors, poor matches between an application's parallelism needs and its allocation, and high scheduling overhead. Efficiency preservation measures these impacts of scheduling discipline on processor efficiency, factoring out the inefficiency due to inherent workload characteristics.

To define efficiency preservation, we first define *application efficiency*, $AE_{policy,app}(p)$, as the ratio of the total computation time of an application, *app*, when run under scheduling policy, *policy*, on p processors (excluding waiting time due to load imbalance synchronization losses) to the product of the application's elapsed time and p:

$$AE_{policy,app}(p) \equiv \frac{comp(p)}{elapsed(p) * p} \tag{3.1}$$

Here, $comp(p)$ is the total computation time of an application and $elapsed(p)$ is the elapsed time of the application when run on p processors.

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. The efficiency preservation measure is

$$EP_{policy,app}(J,P) \equiv \frac{\sum_{j=1}^{J} \mathcal{A}_j \frac{AE_{policy,app}(\mathcal{A}_j)}{AE_{Uniprogramming,app}(P)}}{P} \tag{3.2}$$

where $\mathcal{A}_j$ is the number of processors the policy allocates to copy j of the application. By using the ratio of application efficiency on A_j processors to that on P processors, we factor out the inefficiency inherent in the application regardless of the number of processors allocated to the application.

Because allocation policies can impose overheads, EP can in theory be as small as zero. Although it may seem counterintuitive, efficiency preservation can also be greater than one, since at least some applications exhibit significantly sub-linear speedups, and thus run more efficiently on fewer processors.

The efficiency preservation measure provides a natural explanation for a number of conclusions reached in prior work. For example, efficiency preservation can be used to illustrate the importance of space sharing over time-sharing scheduling policies. 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 as a growth in application efficiency with diminishing allocation, and thus the growth in efficiency preservation.

While efficiency preservation provides useful information about a processor allocation policy, it is not a complete characterization. It is a static measure, and thus 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. A policy that dedicates the entire machine to individual jobs in FCFS order will have an 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 useful in comparing alternative processor allocation policies.

Efficiency preservation must be measured relative to a particular workload. The dependence of the efficiency preservation measure on the application considered is necessary. This dependence arises because the interplay between the kernel's processor allocation policy and the application's load management policy can have a profound effect on performance. For 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. For the example presented in the next section, we use the simplistic characterization of application speedup to measure efficiency preservation. A more detailed model of application characteristics is presented in Chapter 5 for the evaluation of the dynamic allocation policies defined in Chapter 4.

3.2 Efficiency Preservation Example

In this section, we illustrate the use of efficiency preservation in a first-order performance evaluation of three candidate scheduling policies. All policies space share the system among multiple jobs. Two policies statically divide the system into four and eight partitions, respectively. Each job is assigned to a partition and executes there until completion. The third policy is a hypothetical dynamic allocation policy that divides the number of processors as evenly as possible among the jobs in the system.

For this example, we use an application's speedup function to estimate its efficiency preservation under the three scheduling policies. Assume for simplicity that the total computation time of an application is the same for all allocations (i.e., *comp*(p) in Equation (3.1) is the same for all p). Then the ratio $\frac{AE_{policy,app}(a_j)}{AE_{Uniprogramming,app}(P)}$ in Equation 3.2 can be rewritten as:

$$\frac{AE_{policy,app}(a_j)}{AE_{Uniprogramming,app}(P)} \equiv \frac{elapsed(P) * P}{elapsed(a_j) * a_j} \equiv \frac{S_{app}(a_j) * P}{S_{app}(P) * a_j} \tag{3.3}$$

Consider an application with a speedup curve S_{app} as shown in Figure 3.1. The x-axis denotes the fraction of the total machine allocated to an application, rather than the actual number of processors. This speedup function is typical of many applications that exhibit increasing speedup for increasing numbers of processors until some point at which speedup levels off.

Figure 3.2 plots the efficiency preservation under the three scheduling policies for the application whose speedup curve is shown in Figure 3.1. This figure shows the drastic decrease in efficiency preservation for the static partitioning policies when the number of jobs in the system is less than the number of partitions. The dynamic allocation policy is able to utilize all the processors, whereas the static partitioning policies have significant loss of efficiency due to idle processors. The dynamic allocation policy also has a higher efficiency preservation at high loads. When the number of jobs in the system increases, the dynamic allocation policy allocates fewer processors to each job. Since the application exhibits sub-linear speedup (as most applications do), it runs more efficiently on fewer processors. This is also shown in Figure 3.2 by the higher efficiency preservation for the eight-partitioned system over the four-partitioned system when the number of jobs in the system exceeds four.

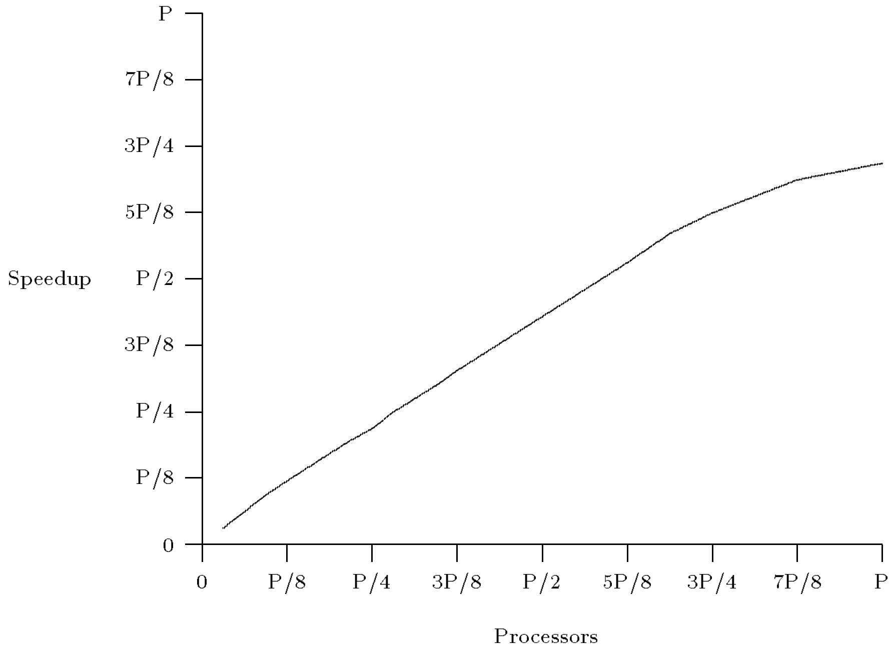


Figure 3.1: *Speedup curve for hypothetical application*

For this simple analysis, Equation 3.3 ignores the cost of scheduling. For the static partitioning policies, this is the cost of time-sharing a partition among multiple jobs when the number of jobs exceeds the number of partitions. Efficiency preservation would be diminished by the overhead required to implement a time-sharing scheme, so on the whole, efficiency preservation argues against its use.

Our analysis of the hypothetical dynamic allocation policy ignores the cost of dynamic

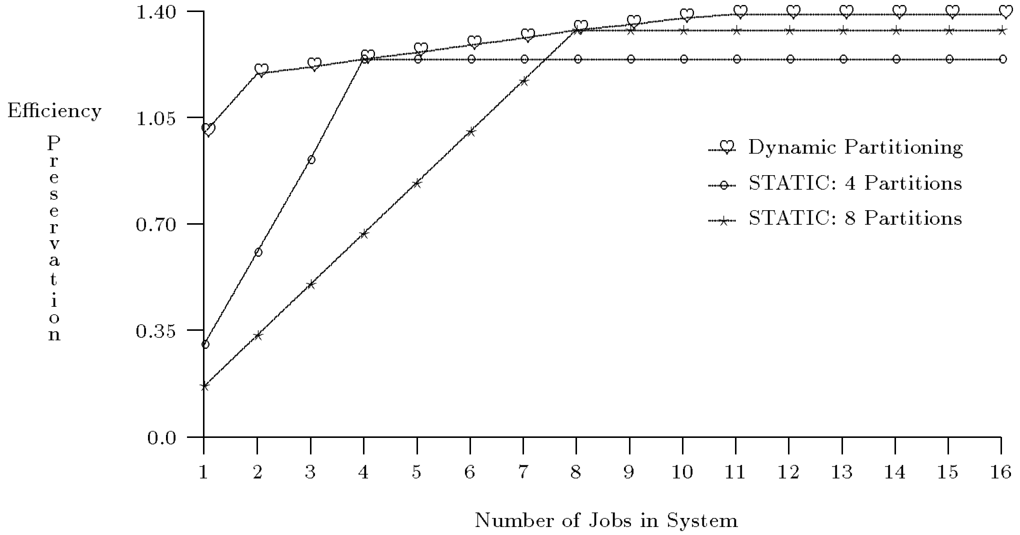


Figure 3.2: *Efficiency preservation for three policies*

reallocation and the effect of varying allocations on the performance of applications. As discussed in Chapter 2, these are important factors affecting the performance of dynamic allocation policies. In Chapter 5, we reexamine efficiency preservation using a more detailed analysis of application efficiency that reflects both the cost of dynamic reallocation and the performance effect of load balancing for the specific policies in Chapter 4.

3.3 Summary

This chapter defined efficiency preservation as a measure of processor allocation disciplines, and showed how it can be used to gain 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. We compared the efficiency preservation of static allocation policies to a hypothetical dynamic allocation policy and showed how efficiency preservation can highlight the potential benefits of dynamic allocation.

Chapter 4

Dynamic Allocation Scheduling Policies

4.1 Introduction

In this chapter, we define processor allocation policies for message-passing parallel systems. Our specific policy proposals fall into two families, called Equipartition and Folding. Both classes employ dynamic allocation and adaptive partitioning: the size and configuration of the processors allocated to jobs can change throughout their execution. Both the Folding and Equipartition families of policies assume no *a priori* information is available on job characteristics, and so strive for equal allocation of resources to the jobs. While we do not consider it here, it is possible to implement variants of these policies that allow unequal allocation to reflect priorities.

Given J jobs in the running set, the processor allocation policy determines which processors to assign to each. Both policies are restricted to assignments of rectangular blocks of processors: if other shapes were allowed, messages between processors belonging to one job would be routed over links connecting processors belonging to other jobs. As discussed in Chapter 2, for reasons of predictability and contention, we wish to avoid this sort of interference.

Our allocation policies apply to sets of jobs that fit simultaneously in memory and the processors of the system. 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 briefly in Section 5.4 and discuss it more extensively in Chapter 6. We also assume each job may be scheduled on as few processors as possible, even one. Thus, these policies ignore the possibility of applications that may require some minimum number of processors to run. Chapter 6 discusses policies appropriate to the scheduling of these applications.

The Equipartition and Folding policies differ in the way they address the costs of dynamic reallocation. 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 scheduling overhead and good equality of allocation, but potentially poor balance of an application's load across its allocation. Since the number of processors allocated to a job may not evenly divide the number of threads in a job, a system induced load imbalance can degrade job performance.

The other policy, called Folding, avoids system induced load imbalance by always halving or doubling the number of processors allocated to a job. A newly loaded job is allocated a partition of processors obtained by dividing the largest currently allocated partition in half, with the threads 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 exhibiting good load balance when run alone will be load balanced when run in a multiprogrammed environment.

The Folding policy does not allocate equal-size partitions. To ensure equal resource allocation, the jobs' allocations alternate between larger and smaller partitions. Here, the frequency and cost of these reallocations affect the job's performance. By varying the rate of reallocations, the Folding policies vary from emphasizing high efficiency preservation to emphasising equal resource allocation.

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 reallocation cost, application load balance, and equal resource allocation on the performance of processor allocation policies. This chapter defines each of these policy classes in detail.

4.2 Mapping

As discussed in Section 2.4, 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 contraction from a 2^M x 2^N grid onto an R x C 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

$$map(i,j) \ \equiv \ \left(\left\lfloor \frac{i * R}{2^M} \right\rfloor , \left\lfloor \frac{j * C}{2^N} \right\rfloor \right) \tag{4.1}$$

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 for the maximum 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 (4.1)) 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.

4.3 The Equipartition Policy

The Equipartition policy takes a straightforward approach to allocating processors among competing jobs, that is to partition 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: when a new job arrives, each currently executing job relinquishes some of its processors, so that the arriving job is assigned it's equitable allocation. Similarly, when a job departs, the allocation of each remaining job is increased.

There are several factors that complicate an equipartition policy for the distributed memory mesh connected environment we are considering. First, it is not always possible to allocate an equal number of processors to all jobs, since the number of jobs may not evenly divide the number of processors.

Second, to maintain adjacency among the processors allocated to a job, the partitions must be rectangular, which further complicates finding equitable allocations. Additionally, maintaining rectangular partitions in the face of dynamic allocation is difficult. It is not sufficient to reassign processors freed by a departing job among the remaining jobs. Rather, a shift in the allocation of processors is necessary so that the new partitions remain rectangular. The cost of reallocation depends in part on the number of processors that have to be reassigned to different partition sets. Furthermore, this shift will invariably involve all running jobs. In reassigning processors, all jobs will have to remap their applications on new partition sets. The cost of this remapping depends in part on the contention for network resources among all jobs reallocating at once. A viable equipartition policy must both reduce the number of processors shifted among different partition sets and orchestrate the partition changes so as to minimize interference among multiple reallocating jobs. Finally, this reallocation must be defined for all multiprogramming levels.

A third difficulty with defining an equipartition policy relates to the mapping policy used to distribute an application load among the processors allocated to the job. As pointed out earlier, the thread mapping function (Equation (4.1)) 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. If the number of threads does not evenly divide the number of physical processors in the dimensions, the mapping can result in a considerable load imbalance. Because the progress of the job is limited by its slowest thread, reducing this maximal loading is an important goal. In Section 4.3.2, we describe a variant of the Equipartition policy that specifically addresses this concern.

4.3.1 Basic Equipartition: $EQUI$

Our approach to reducing the maximal processor loading is to ensure that at least one dimension of each allocated partition is a power of two, since the thread mapping function would evenly divide the threads among the processors in that dimension. 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. It also chooses partition configurations so that threads of multiple jobs migrating to different processors do not contend for the same network resources.

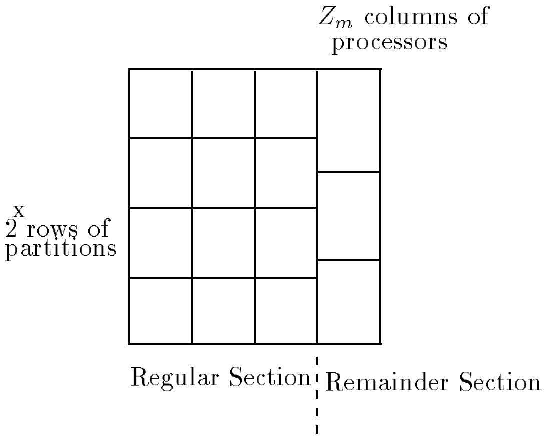


Figure 4.1: *Partitioning under Equipartition*

Figure 4.1 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 $W \equiv J - 2^X Y$ partitions. (For some values of J 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 $X = \left\lceil \frac{\lfloor logJ \rfloor}{2} \right\rceil$. Hence, $2^X Y$ is the largest number less than J with a power of two factor.

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.

For the partitions in the remainder section, the number of processors in each row of the partitions is defined by dividing the number of partitions into the number of processors in a row. One of the dimensions must have a power of two number of processors. If W, the number of partitions in the remainder section, is a power of two, each partition will have a power of two number of processor rows. Otherwise, we must ensure that the number of columns of processors in each partition of the remainder section is a power of two.

The columns of processors will be divided among the partitions in the regular and remainder sections so that each partition will have approximately its fair share of processors. Each partition in the regular section is guaranteed a minimum of $Z_g \equiv \left\lfloor \frac{\frac{2^{M+N}}{J}}{2^X} \right\rfloor$ columns of processors. For the remainder section, let Z_m be the number of columns of processors in each partition. If W is a power of two, each partition is assigned a minimum of $Z_m = \left\lfloor \frac{\frac{2^{M+N}}{J}}{\frac{2^M}{W}} \right\rfloor = \left\lfloor \frac{2^N * W}{J} \right\rfloor$ columns of processors. Otherwise, we ensure that Z_m is a power of two by setting it to:

$$Z_m = 2^{\left\lfloor log(1 - \lfloor \frac{J}{2^X} \rfloor \frac{2^X}{J}) + N \right\rfloor} \tag{4.2}$$

This results in a largest partition possible for the remainder section which has a power of two number of processor columns, and whose size is smaller than the fair share number of processors.

The remaining $2^N - Z_g * Y - Z_m$ columns of processors are allocated among the partitions of the regular and remainder sections in a greedy manner, while ensuring that Z_m remains a power of two if W is not a power of two.

Finally, the orientation of the regular and remainder sections (i.e, positioned horizontally or vertically) is alternated every 2^k, $k > 1$ partitions with the sections being aligned along the larger dimension of the mesh first. This allows the partitions to be added vertically, then horizontally, so that the dimensions of the partitions remain close in size.

Figure 4.2 illustrates how this partitioning scheme allocates a 16x16 processor grid

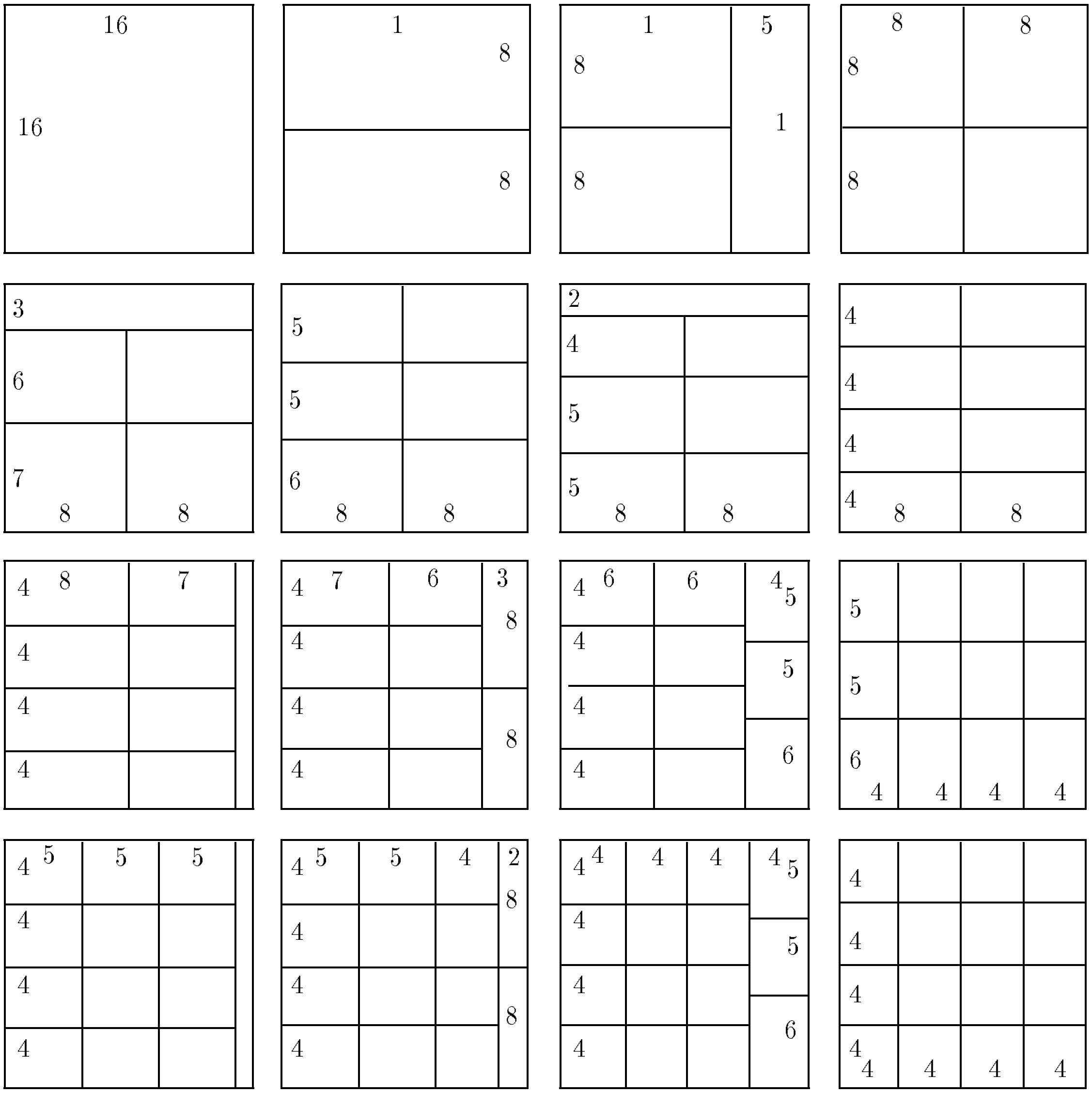


Figure 4.2: *Equipartition of* 16*x*16 *mesh for* 1 *to* 16 *partitions*

among 1 to 16 jobs. As an example, consider partitioning the system into 11 partitions. The regular section would contain 8 partitions (the largest number less than 15 with a power of two factor), with 4 rows of partitions and 2 columns of partitions. The remainder section would contain 3 partitions. The rows of processors are divided evenly among the partitions in the regular section, with 4 processor rows per partition. In the remainder section, the number of partitions does not evenly divide the number of processor rows; two partitions will be allocated 5 processor rows and 1 partition will be allocated 6 processor rows. The fair share number of processors is $\lfloor 16 * 16/11 \rfloor = 23.27$, so each partition in the regular section is guaranteed $\lfloor 23.27/4 \rfloor = 5$ processor columns. Solving Equation 4.2, each partition in the remainder section is guaranteed 4 processors in each column. Thus, there are 2 columns of processors that are unallocated. One column each is assigned to each partition column of the regular section. Thus, each partition in the regular section has size 4x6, and two partitions in the remainder section have size 5x6, and one partition in the remainder section has size 4x6.

4.3.2 A Higher Efficiency Preservation Equipartition: $EQUI_+$

Consider a single partition allocated by $EQUI$, and denote its size as 2^i x C, 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. For this discussion, we will assume that a partition allocated to a job has a power of two number of processor rows, although as discussed in Section 4.3.1, the dimension with the power of two number of processors may be the row or the column depending on the orientation of the regular and remainder sections.

$EQUI_+$ 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 (4.1)) in order to reduce the maximal loading to $\left\lceil 2^{M+N-i}/C \right\rceil$, the theoretical minimum for this partitioning. The $EQUI_+$ mapping policy distributes the threads of an application evenly along the processor rows. In addition, the columns of threads allocated to a processor row are distributed evenly among the processor columns within a processor row. There are a total of 2^{M+N-i} threads assigned to each processor 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 addition, the new mapping policy maintains thread adjacency defined by the virtual machine.

Figure 4.3 illustrates how the $EQUI_+$ mapping policy works. Suppose a system consisting of a 16x16 processor mesh were divided into 13 partitions, as illustrated at the top of Figure 4.3. Consider the shaded partition consisting of a 4x5 partition of processors. The bottom of Figure 4.3 illustrates how $EQUI_+$ maps an application consisting of 16x16 threads onto a 4x5 partition of processors. The threads along each thread column are divided evenly into 4 processor rows. Each processor row then divides its threads among the processor columns as evenly as possible.

Consider the first row of processors in the partition. To maintain adjacency within a processor row, a 'snake-like' ordering of the $\frac{2^{M+N}}{R}$ threads within the processor row is defined. This ordering is illustrated by a dashed line in Figure 4.3. This ordering among threads within a processor row is used to map the threads evenly onto the processor columns within a processor row. For the example in Figure 4.3, the $2^4 * 2^4/4 = 64$ threads of each row are mapped onto 5 columns of processors, so that each processor is assigned either 12 or 13 threads.

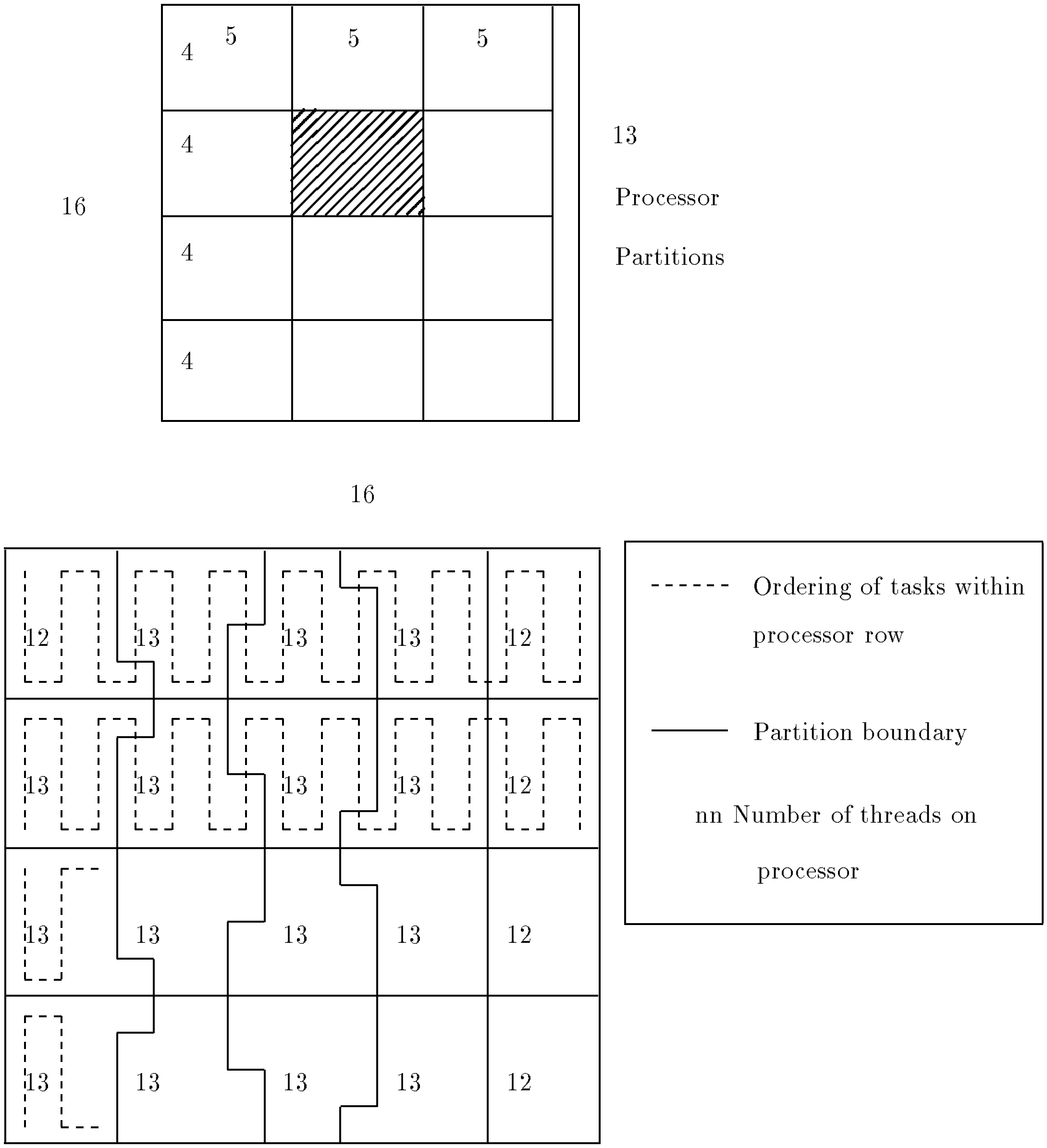
Next, in order to preserve adjacency among threads between rows, the snake-like ordering among threads of a processor row is row-wise transposed, as illustrated in Figure 4.3. (For clarity, the third and fourth rows show only part of the ordering).

The function that realizes this mapping is:

$$map(i,j) \equiv \left(\left\lfloor \frac{i * R}{2^M} \right\rfloor , \left\lfloor \frac{index(i,j) * C}{2^{M+N}/R} \right\rfloor \right) \tag{4.3}$$

where $index(i,j)$ defines the relative position of thread coordinate (i,j) within the $2^{M+N}/R$ threads of a processor row. There are two functions for $index$, corresponding to the two orderings of the threads within the processor row. Consider any even processor row containing $2^{M+N}/R$ threads to be divided evenly into C columns. The relative position of coordinate (i,j) within the $2^M/R$x2^N threads is illustrated in Figure 4.4(top) and defined by:

$$index(i,j) = \left(\left\lfloor \frac{j}{2} \right\rfloor * \frac{2^{M+1}}{R} \right) + \left((j \bmod 2) * \frac{2^M}{R} \right) + \left(\frac{2^M}{R} - \left(i \bmod \frac{2^M}{R} \right) \right) - 1$$



Equi+ mapping onto 4x5 partition

Figure 4.3: *Equi+ mapping of* 16*x*16 *application onto* 4*x*5 *partition*

$$\text{if } \left(i \bmod \frac{2^M}{R}\right) = 0 \quad (4.4)$$

The first term represents the number of threads in the $\lfloor\frac{j}{2}\rfloor$ pairs of thread columns preceding column j. The second term represents a single column of $\frac{2^M}{R}$ threads preceding column j if j is odd. And the third term represents the offset of the coordinate within the last thread column.

For odd processor rows, the third term changes, since the ordering of the threads within the processor row is transposed (Figure 4.4(bottom)). For odd processor rows, $index(i,j)$ is defined as:

$$index(i,j) = \left(\left\lfloor\frac{j}{2}\right\rfloor * \frac{2^{M+1}}{R}\right) + \left((j \bmod 2) * \frac{2^M}{R}\right) + \left(\left(i \bmod \frac{2^M}{R}\right) + 1\right) - 1$$

$$\text{if } \left(i \bmod \frac{2^M}{R}\right) = 1 \quad (4.5)$$

In general, using this indexing within a processor row instead of within a thread row 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. We make the optimistic assumption that the added messaging complexity results in a negligible increase in communication costs.

4.4 The Folding Policy

The goal of the Folding policy is to eliminate the system induced load imbalance resulting from mapping an application onto rectangular partitions whose dimensions do not factor the application's virtual thread space. The Folding policy allocates partitions in such a way 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.

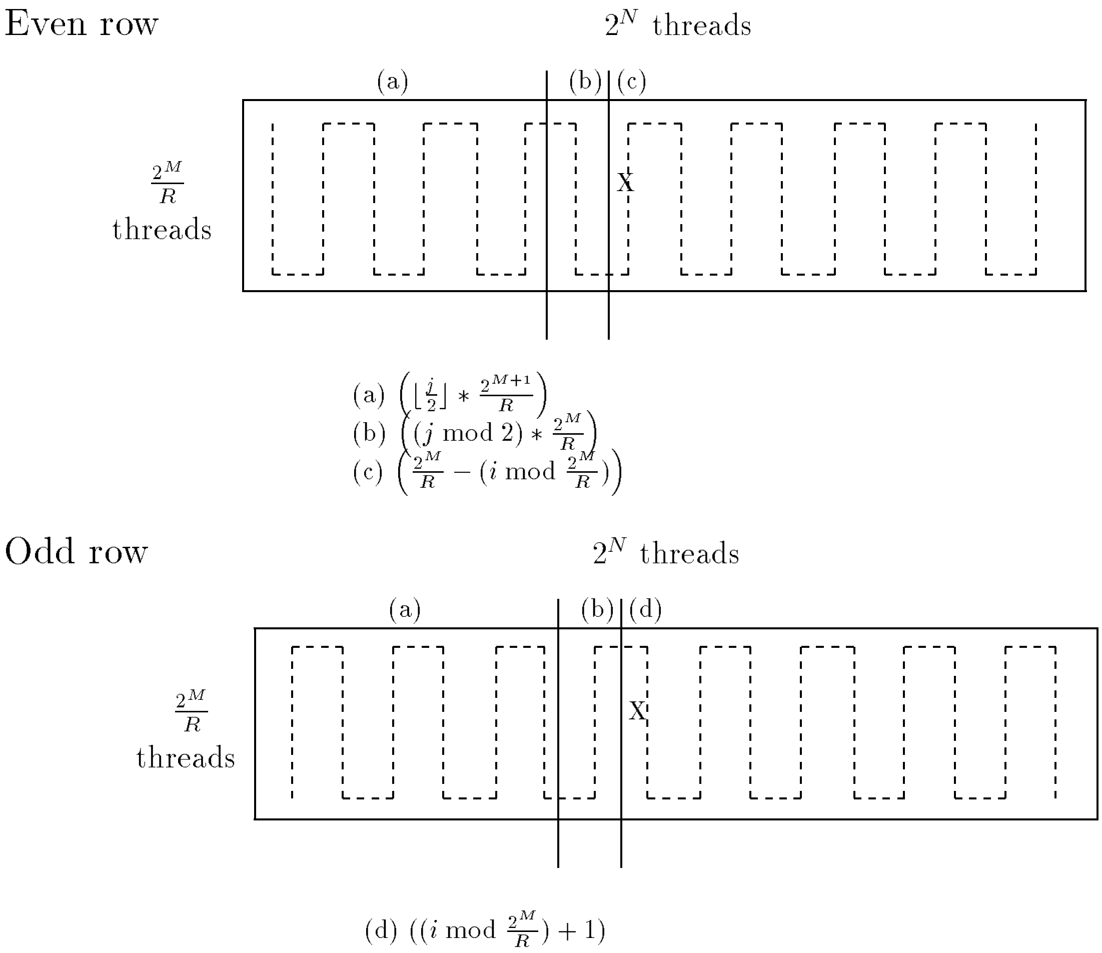

Even row
2^N threads
(a)
(b)
(c)
$\frac{2^M}{R}$ threads
X
(a) $\left(\lfloor \frac{j}{2} \rfloor * \frac{2^{M+1}}{R}\right)$
(b) $\left((j \bmod 2) * \frac{2^M}{R}\right)$
(c) $\left(\frac{2^M}{R} - (i \bmod \frac{2^M}{R})\right)$
Odd row
2^N threads
(a)
(b)
(d)
$\frac{2^M}{R}$ threads
X
(d) $((i \bmod \frac{2^M}{R}) + 1)$

Figure 4.4: *$index(i, j)$ for even processor row (top) and odd processor row (bottom)*

4.4.1 Basic Operation

Folding chooses new partition sizes whenever a job arrives or departs. On job arrival, if the machine is idle, all processors are allocated 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 in Section 4.2. 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 4.4.3.

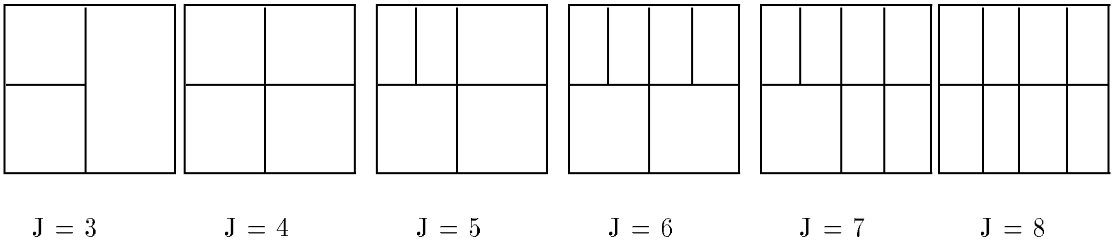


Figure 4.5: *Partitioning of processor mesh for 3 to 8 jobs*

Figure 4.5 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.

4.4.2 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 twice as many processors as the smaller partitions.

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 4.5.) Similarly, we rotate twenty jobs as four systems of five jobs, each running on one quarter of the machine.

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 4.6 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 inductively that because of the way in which we form partitions, such a walk must exist. Section 4.4.5 discusses this further.) In Figure 4.6 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.

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 in each such cycle. For example, in Figure 4.6 the cycle length is five. The second measure, $f(J)$, is the number of rotations participated in by each job

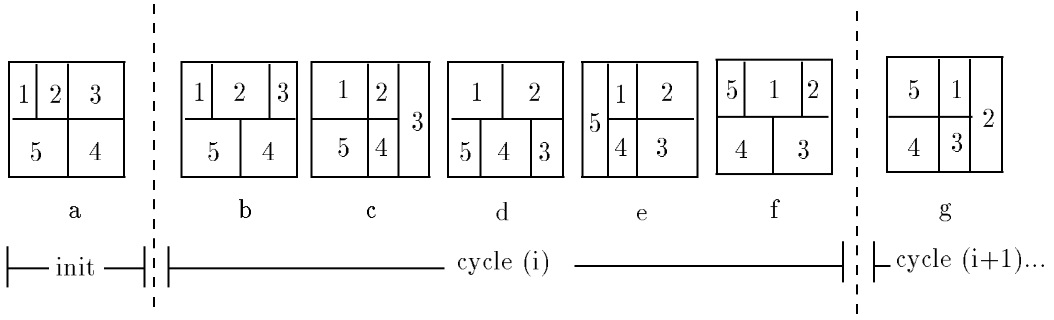


Figure 4.6: *Folding rotation for $J = 5$ jobs*

within the rotation cycle of $N(J)$ rotations. In Figure 4.6, $f(J)$ 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 as described above. For instance, the string corresponding to Figure 4.6a is 'ssbbb', and for Figure 4.6b 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 rotation steps during a cycle, and so equal allocation is assured.

This analysis leads to the following expressions for the cycle length, $N(J)$, and the number of reallocations per job per cycle, $f(J)$:

$$N(J) = \begin{cases} N(K) & \text{if } J = K * 2 \\ 0 & \text{if } J = 1 \\ J & \text{otherwise} \end{cases} \tag{4.6}$$

$$f(J) = \begin{cases} f(K) & \text{if } J = K * 2 \\ 0 & \text{if } J = 1 \\ 2 * min(S, B) & \text{otherwise} \end{cases} \tag{4.7}$$

4.4.3 Job Departures with Rotations

An examination of Figure 4.6 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 4.6, 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. It does offer a much simpler implementation, in the simulation model of Section 5.4, and presumably in real systems. We believe this is an indication that it would be an attractive approach.

4.4.4 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.

4.4.5 Defining Partition Cycles

The Folding rotation policy relies on the property that there is a cyclic ordering of the partitions. Each partition knows the partitions immediately preceding and immediately following it in a cyclic partition ordering. This ordering is used to determine what action (fold, unfold or no change) is taken for each partition at each rotation instance. This section describes the algorithm for maintaining the cyclic ordering of the partitions in the face of changes to the partition configuration.

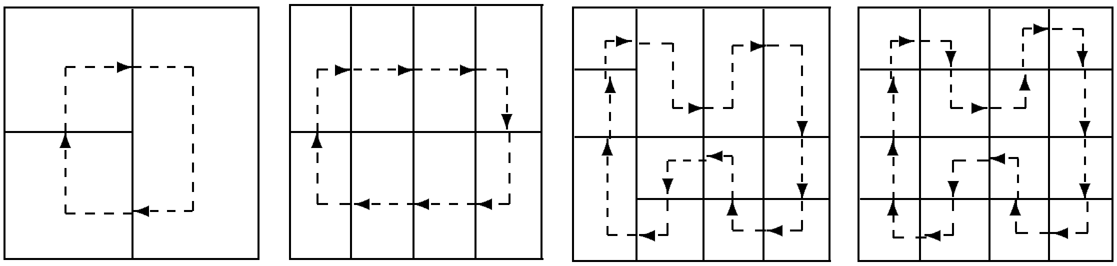

(a) 3 partitions (b) 8 partitions (c) 12 partitions (d) 16 partitions

Figure 4.7: *Cycles for 16 or fewer partitions*

It is highly desirable that as jobs enter and leave the system and the number of partitions changes, changes to the partition cycle involve only those partitions affected by the allocation change. In this way, only local updates to the cyclic ordering are necessary.

The partitioning policy described in Section 4.4.1 has the property that only local

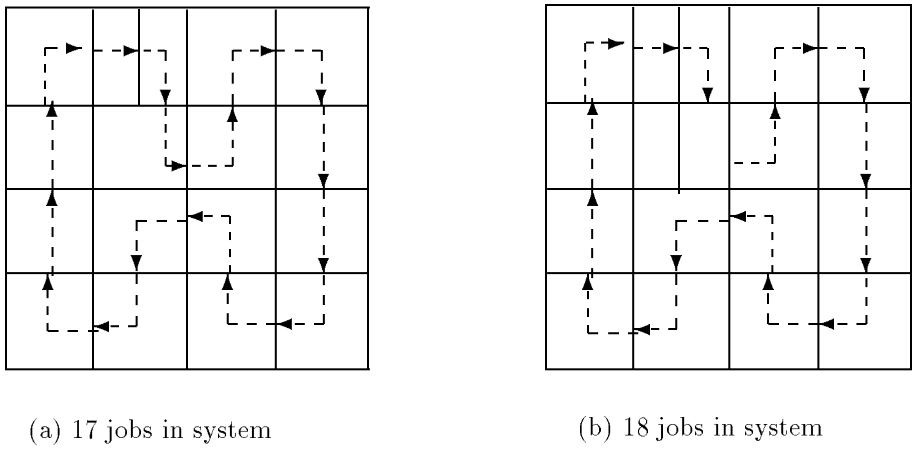


Figure 4.8: *Cycles for 17 and 18 partitions*

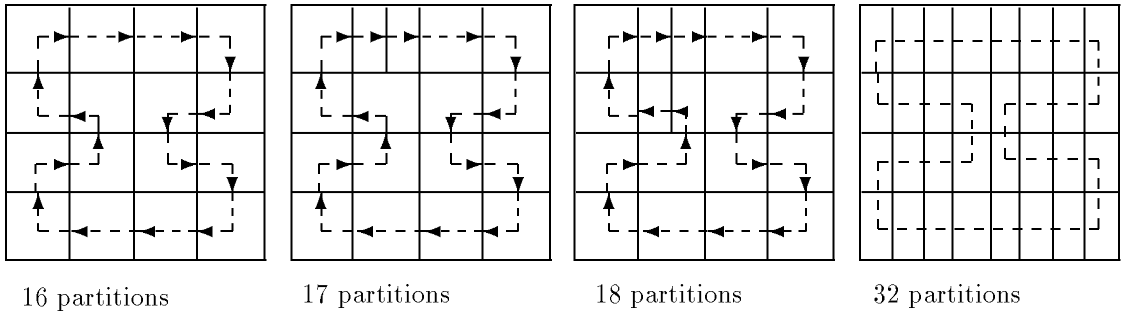


Figure 4.9: *Cycles for 17 and 18 partitions*

changes are necessary to the cyclic ordering for up to 16 processors in the system. To see this, Figure 4.7 illustrates the cyclic ordering for several configurations with up to 16 processors in the system. When a large partition is split into two, the currently defined cycle is changed so that the new partitions replace the old partition within the cycle, i.e., the two new partitions are adjacent to each other in the new cycle, and the neighboring partitions of the old partition become the preceding neighbor of one new small partition, and the succeeding neighbor of the other new small partition. Similarly, when two adjacent partitions that are combined to become a large partition upon job departure, the two partitions are replaced by the single large partition in the cyclic ordering, while updating only the adjacency of the new partition and its immediate neighbors in the cyclic ordering.

However, when the number of partitions exceeds 16, it is not always possible under the partitioning policy defined in Section 4.4.1 to make local changes to the partition cycle. Consider the configuration of Figure 4.7(d) with 16 processors in the system. Figure 4.8(a) illustrates the partition cycle after a 17^{th} partition is added if only adjacency for the two new smaller partitions and the neighboring larger partitions is changed to define the new cycle. Now suppose an 18^{th} partition is added as illustrated in figure Figure 4.8(b). Given the current cycle, it is not possible to create a new cycle simply by changing adjacency information local to the new partitions and their neighbors.

To solve this problem, a new global cyclic ordering is defined when the number of partitions reaches 16. Figure Figure 4.9(a) illustrates the global change to the partition cycle at 16 processors. Once the cycle is changed, subsequent increases in partition changes (from 16 to 32 processors) result again in only local changes. Figure 4.9(b)-(d) illustrate the partition cycles for 17, 18, and 32 partitions. (In Figure 4.9(d), we omit the arrowheads for clarity.) In general, every 2^k partitions, $k >= 4$, a redefinition of the cyclic ordering is necessary so that subsequent changes to the partitions result in local changes to the cyclic ordering. (This corresponds to the partitioning policy defined in Section 4.4.1 which alternates the direction along which a large partition is split every 2^k partitions.) Finally, note that there is more than one partition cycle possible for 16 partitions that would allow local cycle changes for up to 32 partitions. Figure 4.9 illustrates only one such cycle.

Globally changing the cycle definition requires synchronization among all processors at processor reallocation time, which is expensive. While global change occurs only for the single allocation point within 2^k partitions, if the system oscillates within this number of partitions, system performance can degrade due to significant reallocation costs. Still, no such global change is needed for a moderate numbers of partitions (up to 16).

Another possible solution exists for systems with the same number of processors in each dimension ($M = N$) when the number of partitions ranges between 16 and 32. For these systems, when the number of partitions reaches 16, each partition has the same number of processors in each dimension. To create 17 to 32 partitions, a partition may be split along either direction. By repeating (rather than alternating) the direction along which the machine was split for 8 to 16 partitions, a global cyclic ordering of the partitions is possible to maintain by making only local changes to the cyclic ordering. This results in the ordering illustrated in Figure 4.9, without requiring a global reordering. After 32 partitions, however, a global change to the cyclic ordering is required. For these systems, however, this simple approach doubles the number of partitions that may be created before a global change to the cyclic ordering.

4.5 Summary

In summary, we have defined two specific dynamic allocation policies. These policies are designed for a 2-dimensional mesh-connected processor topology, where each dimension has a power of two number of processors. In defining both policies, we address not only how many processors are allocated to arriving jobs, but also which processors. Both policies maintain partitions that are composed of adjacent processors, so that there is no contention for network resources among multiple executing jobs. Each policy also takes great pains to reduce the cost of reallocation, since in a distributed memory environment, this cost can be significant. Consideration of all of these factors is necessary for a realistic evaluation of the performance of scheduling policies for distributed memory parallel systems.

The Equipartition policy attempts to divide the system into rectangular partitions of approximately equal size, so it attempts to provide low reallocation overhead but incurs a system induced load imbalance. In order to reduce the potential for system-induced load imbalance, each rectangle defined has one dimension that has a power of two number of processors. Two variants of Equipartition are proposed: the first, $EQUI$, uses a simple modulo division to divide the threads of an application as evenly as possible along both dimensions of the partition allocated to the job. The second policy, $EQUI_+$, uses a more complicated mapping policy that provides the minimum load imbalance for a partition. By always halving or doubling a partition, the Folding policy provides good load balance but incurs additional reallocation overhead in guaranteeing equal resource allocation. We defined a rotation policy that tries to reduce the number of reallocations and the cost per reallocation that must occur in order to guarantee equal resource allocation.

In Chapter 5, we compare the Folding and Equipartition policies in order to ascertain the importance of cost of reallocation and load balancing – two central overheads of dynamic reallocation – on the performance of dynamic scheduling policies.

Chapter 5

Performance Evaluation of Dynamic Allocation Policies

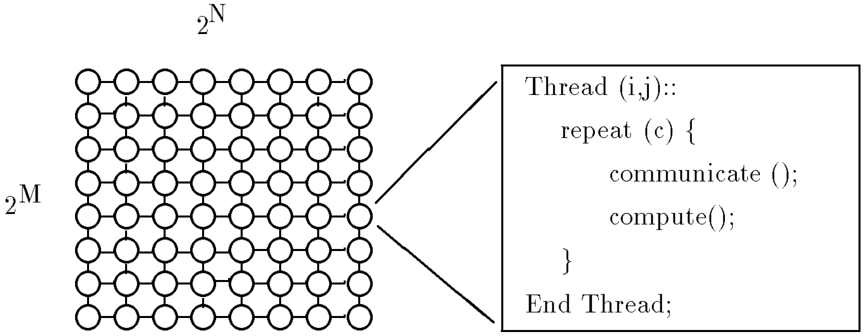
This chapter compares the Folding and Equipartition families of policies in order to ascertain the relative importance of reallocation cost, load balancing, and the impact of equal resource allocation on the performance of dynamic allocation scheduling policies for distributed memory, message-passing systems.

Section 5.1 details the assumptions made about the hardware and software environment. Section 5.2 derives the efficiency preservation of the two classes of policies in the absence of job arrivals or departures. Unlike the efficiency preservation metric derived for the hypothetical dynamic allocation policy defined in Chapter 3, we incorporate reallocation overhead and load imbalance inefficiencies into the efficiency preservation derivation. This gives us a simple, first order metric determining the effects of this overhead on scheduling performance. We also examine the degree of fairness afforded by each of the policies for a static number of jobs in the system.

In Section 5.3, we consider the effects of arrivals and departures on the performance of the policies. We use a Markovian birth-death model to obtain mean response times under homogeneous job arrivals and departures. We also examine the effects of imposing a limit on the multiprogramming level, as might be appropriate to accommodate limited memory capacity and large memory-bound applications. In Section 5.4 we use discrete event simulation to ascertain examine the performance of the policies under heterogeneous workloads, again including the effects imposed by a multiprogramming limit. We also re-examine the issue of fairness in this dynamic environment. Section 5.5 summarizes our results.

5.1 The Hardware and Software Environments

As stated in Chapter 2, we consider the particular case of mesh-connected parallel machine of 2^Mx2^N nodes, as shown in Figure 5.1a. 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.



```
Thread (i,j)::
   repeat (c) {
        communicate ();
        compute();
   }
End Thread;
```

(a) Hardware

(b) Software

Figure 5.1: *Hardware and software model*

The performance afforded by a particular kernel processor allocation policy is intimately tied to decisions made by the applications it supports. We consider 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 5.1b 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 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. While some systems allow the granularity of the job to be specified at load time in response to the job's initial allocation, in the environment we are studying where the job's allocation changes, this choice could be suboptimal. By choosing a parallelism at least as great as the number of processors in the system, if given the opportunity, the job can utilize all processors. We also assume that the application does not alter the number of threads in response to changes in processor allocation.

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.

We use the following notation throughout this chapter. The values in parentheses indicate the baseline setting for the experiments discussed throughout the chapter. These values are based, in part, on the workload characteristics described in [13], and on the characteristics of the Intel Paragon hardware.

The parameters relevant to this hardware environment are:

- 2^Mx2^N, the size of the mesh of processors (2^4x2^4).

- c, the context switch time required to schedule a new thread (0.5 msec.).

- X, the interconnection network link bandwidth (200MB/sec.).

- 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, [45, 41] has shown that these costs can be almost entirely avoided. Our parameterization anticipates the next generation of system software that will incorporate these advances.

For the SPMD applications, for a quantitative analysis, we assume the particular case that during the communication phase, threads exchange data only with their four nearest-neighbors. Most of our results are not strongly affected by this, and the analysis techniques are applicable for more general patterns.

We use the following parameters to model the software:

- 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 size of a thread's code segment (512KB).
- D, the size of the thread's data and stack segments (4096KB).

Finally, for the Folding policy, we define I as the inter-rotation time, which varies throughout the experiments.

5.2 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.

5.2.1 Efficiency Preservation: Derivation

As explained in Chapter 3, efficiency preservation is given by

$$EP_{policy,app}(J,P) \equiv \frac{\sum_{j=1}^{J} \mathcal{A}_j \frac{AE_{policy,app}(\mathcal{A}_j)}{AE_{Uniprogramming,app}(P)}}{P} \tag{5.1}$$

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

$$AE_{policy}(\mathcal{A}_j) = \frac{\sum_{i=1}^{2^{M+N}} t_i}{\max_{p \in \mathcal{P}(j)}(CMP/step_p + CS/step_p + COM/step_p)} * (1 - \%OV_{policy}) \tag{5.2}$$

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, $CMP/step_p$ and $COM/step_p$ are the total compute and communication times respectively that processor p spends per application step, $CS/step_p$ is the 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.

Under most applications where computation time exceeds the cost of synchronizing between threads, the ratio in the expression is dominated by $CMP/step_p$ term, which is determined by the maximally loaded processor. In the specific case of perfectly load balanced computations ($t_i = t$), we can simplify $CMP/step_p$ using the fact that the maximum number of threads mapped to a single processor for a partition of size R_j x C_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 $\left\lceil \frac{2^M}{R_j} \frac{2^N}{C_j} \right\rceil$. The numerator of the ratio can also be simplified under an assumption of perfect load balancing. When thread compute times can vary, we use Monte-Carlo simulation to obtain numerical results for this term.

In computing the cost of the communication of a job, $CS/step_p$, we make the following assumptions:

1. the cost of synchronization is dominated by the cost to synchronize between threads on remote processors; the cost of communicating between threads on the same processor can be overlapped with the remote communication.

2. two pairs of threads of a job communicating between the same pair of processors must send distinct messages.

Recall that s is the time for two threads located on neighboring nodes to communicate the amount of data necessary for an application step. For a processor with X_j x Y_j threads, the synchronization cost for a single application phase is $(X_j + Y_j) * s$. Again, under the assumption of a perfectly load balanced system, this term, determined by the maximally loaded processor of a R_j x C_j partition, is $\left(\left\lceil \frac{2^M}{R_j} \right\rceil + \left\lceil \frac{2^N}{C_j} \right\rceil\right) * s$.

The last term, $COM/step_p$ is $(X_j * Y_j - 1) * c$ for a processor with X_j x Y_j threads. For the maximally loaded processor of a perfectly load balanced application running in a R_j x C_j partition, $COM/step_p = \left(\left\lceil \frac{2^M}{R_j} \right\rceil * \left\lceil \frac{2^N}{C_j} \right\rceil - 1\right) * c$.

$\%OV_{Uniprogramming,p}$ and $\%OV_{Equipartition,p}$ are zero (for both variants of Equipartition). To compute $\%OV_{Folding,p}$ 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 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 threads 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

$$\%OV_{Folding} = \frac{(\#rotations\ in\ N(J)I) * (\#procs/rotation) * (\#seconds/(proc/rotation))}{2^{M+N}\ N(J)I} \tag{5.3}$$

The denominator represents the total number of processor-seconds available in an interval of length $N(J)I$, the first term of the numerator the total number of rotations in that interval, the next term the total number of processors involved in each rotation, and the final 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 $2^{M+N}/2^{\lfloor logJ \rfloor}$.

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 from a small to a large partition, 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 $2^{\min(M,N)}/2^{\left\lfloor \frac{\lfloor \log_2 J \rfloor}{2} \right\rfloor}$.

Combining these terms and simplifying, we get

$$\%OV_{Folding,j} = \frac{1}{I} * f(J) * 2^{\max(M,N) - \left\lfloor \frac{\lfloor \log_2 J \rfloor}{2} \right\rfloor - 1} * \frac{(2D+C)}{X} \tag{5.4}$$

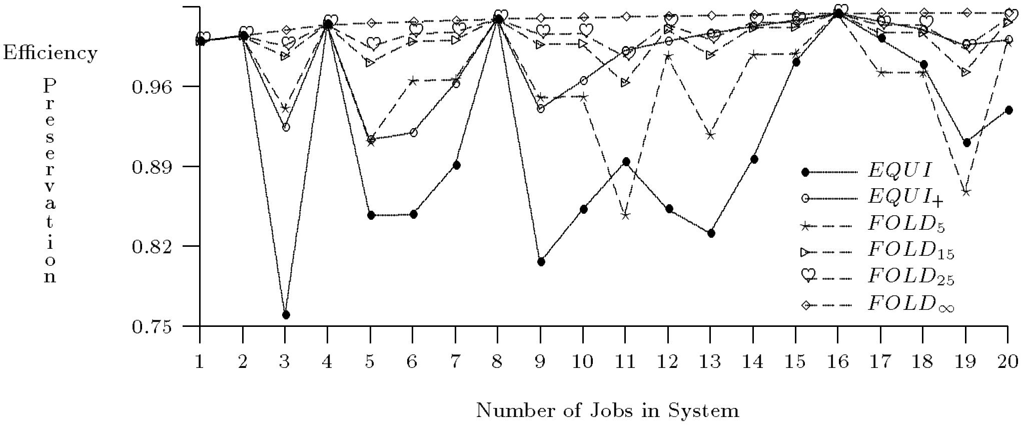


Figure 5.2: *Efficiency preservation for perfectly load balanced jobs ($\delta = 0$)*

5.2.2 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 5.2 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 application is folded onto smaller and smaller partitions.

Figure 5.3 shows how variation in application thread compute times affect efficiency preservation for the $FOLD_\infty$ and $EQUI_+$ policies. We present results for thread 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

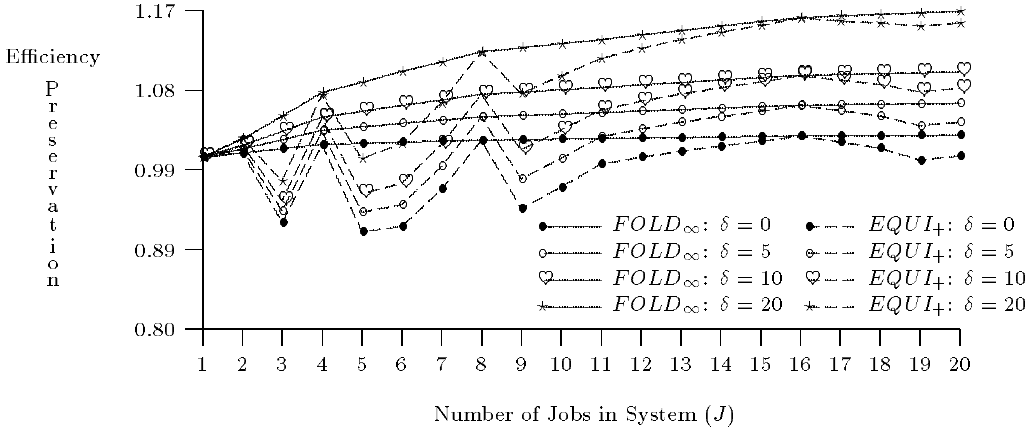


Figure 5.3: *Efficiency preservation for thread compute times from* $U(100 - \delta, 100 + \delta)$ *msec.*

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.

5.2.3 Fairness: Results

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

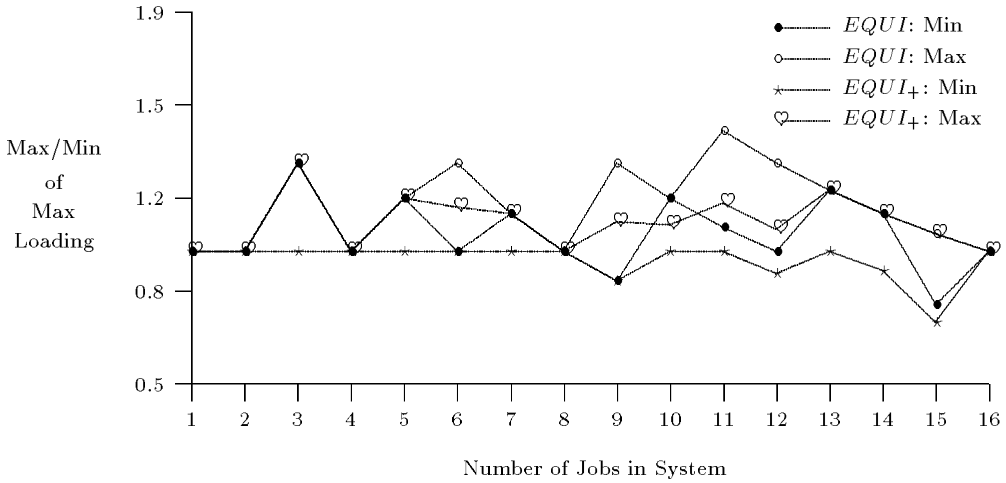


Figure 5.4: *Fairness of Equipartition: relatively most heavily and most lightly loaded jobs*

Figure 5.4 shows the fairness ratios for $EQUI$ and $EQUI_+$, and Figure 5.5 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 5.5 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

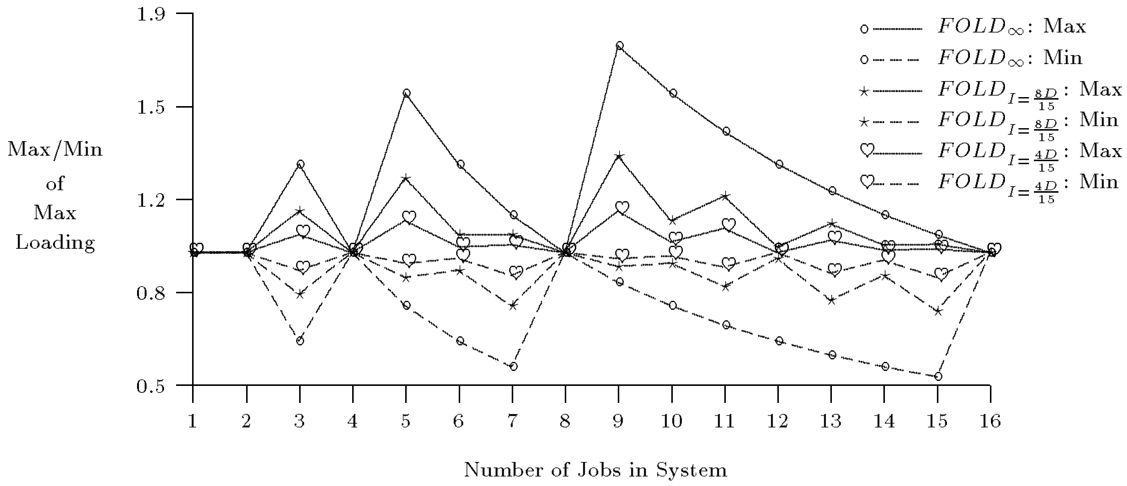


Figure 5.5: *Fairness of Folding: relatively most heavily and most lightly loaded jobs*

$I \geq E$ ($q \geq 1$), 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 5.5 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 alone.

5.3 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 that it captures the most important aspects of the problem, but is simple and has 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 5.4) 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.

5.3.1 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 5.6 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 5.2. 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.

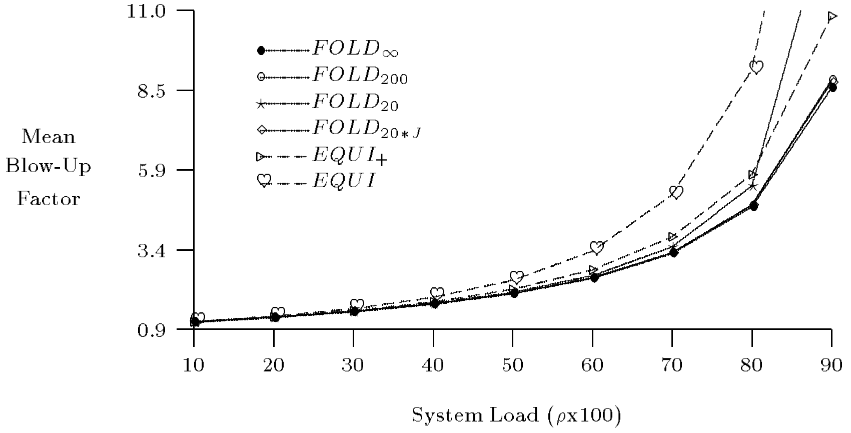


Figure 5.6: *Mean blow-up factor versus system load ($\delta = 0$)*

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 5.6 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 chapter, we will use inter-rotation interval lengths that vary with J.

5.3.2 Limited Memory Resources

In Figure 5.7, 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.

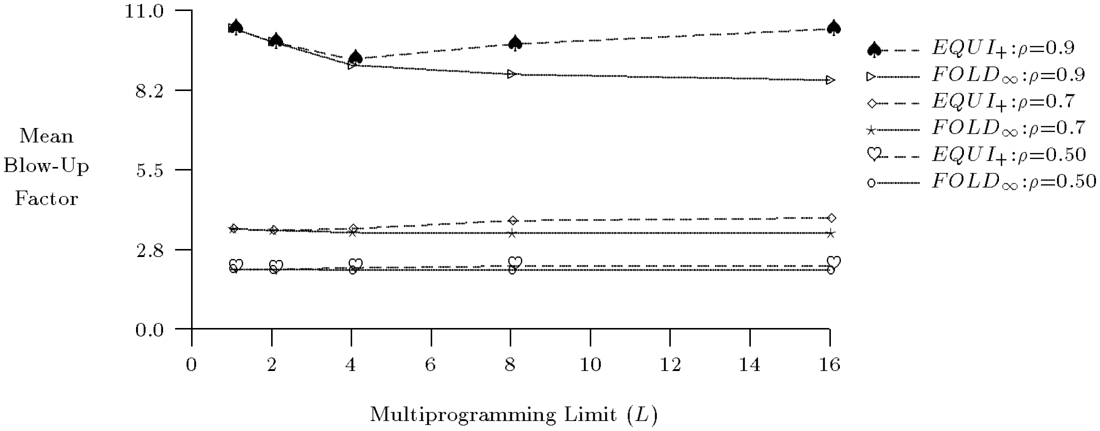


Figure 5.7: *Mean blow-up factor versus multiprogramming limit ($\delta = 0$)*

The results show that performance under the the Folding policy improves with an 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.

5.4 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. (This level of confidence required 30 minutes or more of Decstation time per point.)

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.

5.4.1 Unlimited Memory Resources

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, and p is the fraction of arrivals that are short jobs.

Figure 5.8 shows the mean response times of the long and short job classes against p. 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 ($EQUI_+$). The performance of $FOLD_{15*J}$ is comparable to $FOLD_{50}$ for this workload intensity.

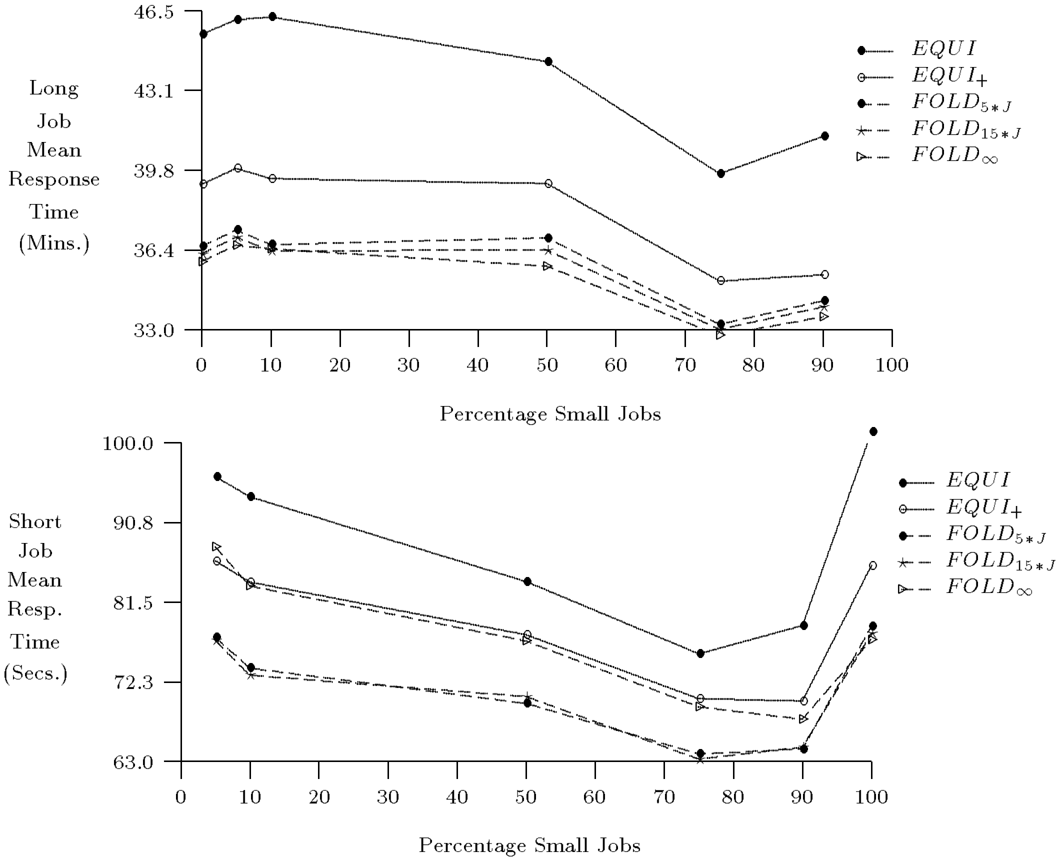


Figure 5.8: *Mean response times* ($\rho = 0.5$, $\delta = 5$)

Figure 5.9 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.

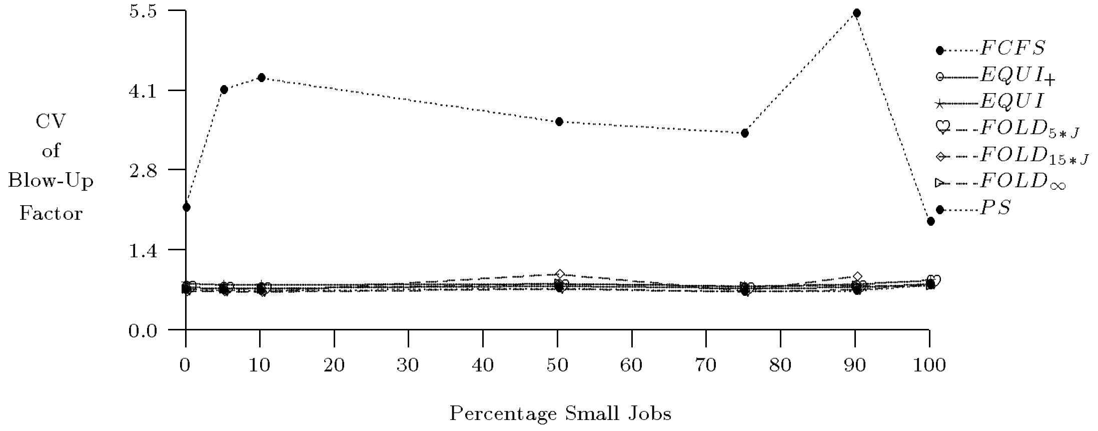


Figure 5.9: *Coefficient of variation of blow-up factor ($\rho = 0.5$, $\delta = 5$)*

5.4.2 Limited Memory Resources

Figure 5.10 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. 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%.

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

5.5 Conclusions

This chapter compares the performance of the Folding and Equipartition policies defined in Chapter 4. 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.

Based on these results, we conclude while maintaining low reallocation overhead is imperative to the good performance of dynamic allocation policies, load balancing for highly parallel applications is also a dominant factor in the performance of policies

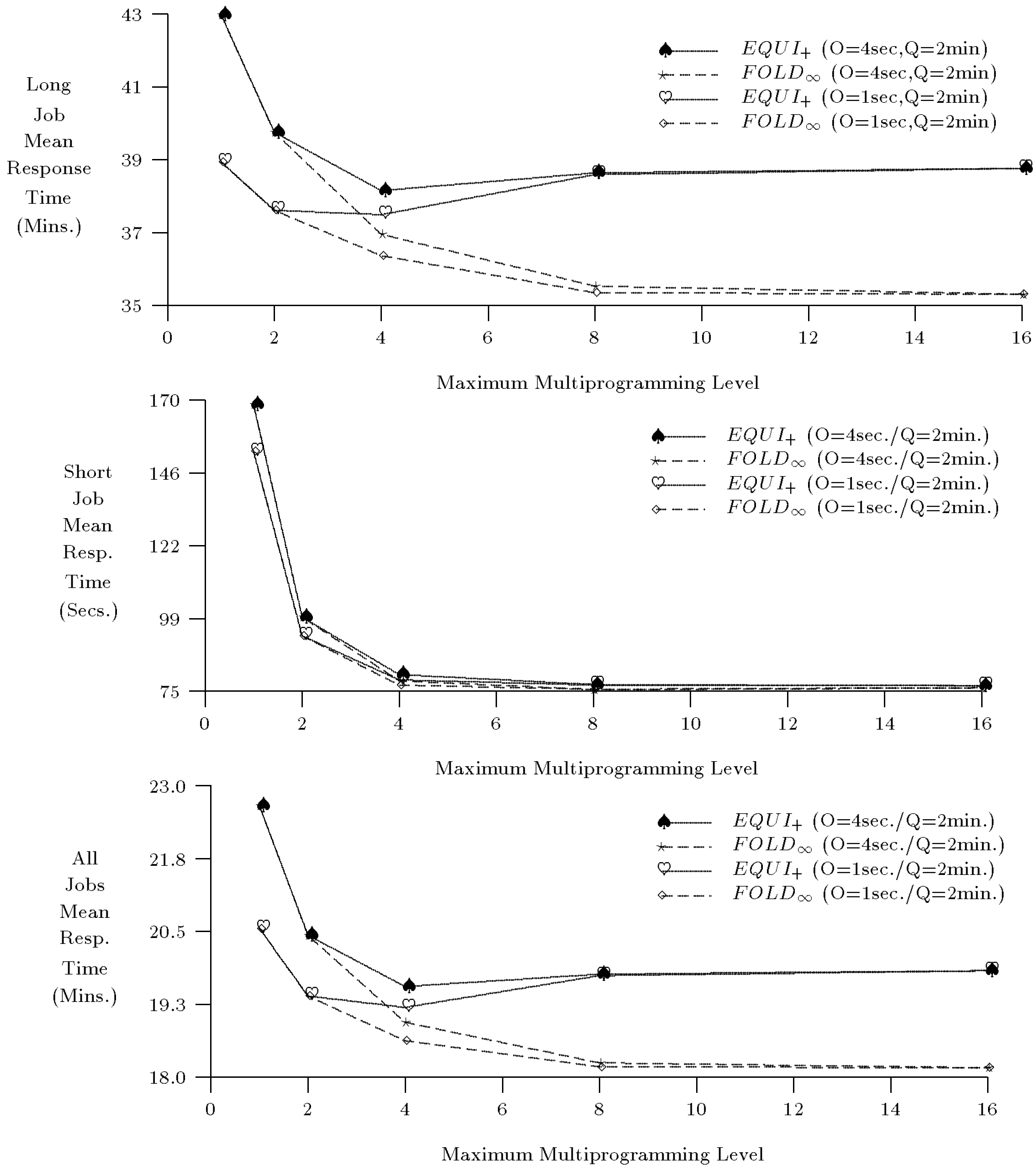


Figure 5.10: *Response time versus multiprogramming limit ($\rho = 0.5$, $\delta = 5$)*

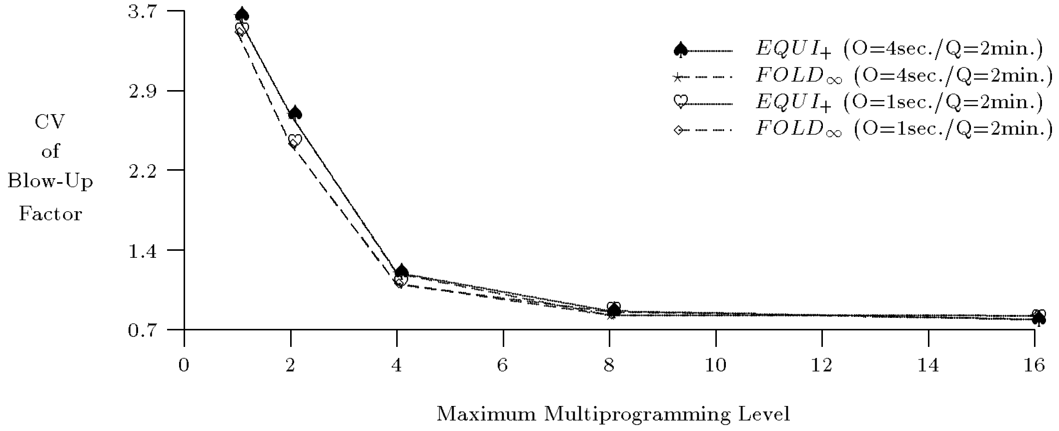


Figure 5.11: *CV of blow-up factor versus multiprogramming limit ($\rho = 0.5$, $\delta = 5$)*

for distributed memory systems. The results also show that with careful attention to processor adjacency and load balancing considerations, dynamic allocation offers the potential for high performance in distributed memory parallel systems.

Chapter 6

Scheduling Policies for Memory-constrained Applications

6.1 Introduction

Processors are not the only scarce resource in message-passing parallel systems; there is also a finite amount of memory. These systems do not generally support virtual memory addressing. However, parallel applications often operate on data sets that require large amounts of memory to run. These applications may not fit in an arbitrarily small portion of the machine's memory. For the distributed memory environment we are considering, where memory is local to processors, this results in a minimum processor requirement for these jobs. The existence of a lower bound on possible allocations clearly complicates scheduling policies: it is not always possible to schedule all jobs concurrently, so some other level of scheduling must be employed to determine which of the available jobs to schedule at any one time. This chapter addresses this additional constraint imposed on scheduling policies for distributed memory systems.

To address large, memory-constrained applications, the scheduling problem for distributed memory parallel systems is defined in two parts, corresponding to two levels of kernel scheduling. First, a medium-term scheduling policy decides which of the available jobs to run at a given time and for how long. In the absence of reliable *a priori* information on job characterization, medium-term scheduling involves time-sharing. Second, for each set of jobs to be scheduled, a short-term scheduling policy decides how to schedule the set of jobs on the parallel system. These policies employ space-sharing. The scheduling policies studied in Chapters 4 and 5 focused on the space-sharing scheduling policy – deciding how to distributed the processors among the running jobs. This chapter is concerned with the time-sharing scheduling policy, deciding *when* to schedule jobs and *for how long* when not all jobs may be scheduled at once.

In general, the goal of the time-sharing scheduling policy is to schedule J jobs on P processors, where each job has a minimum processor requirement to run. We assume each job can potentially use all the processors, and job execution time is not known to the scheduler. In this environment, it makes sense to rotate the processors among the jobs, so that each job gets an equal percentage of the processing resources within a reasonable time interval.

Define a scheduling quantum T within which all jobs are scheduled. The schedule is repeated every T time units. When the number of jobs in the system changes, the schedule is changed to reflect the new multiprogramming level before the start of the next interval.

Within the time interval, T, each job receives an equal percentage of the total processing resources with the constraint that no job receives an allocation of processors less than its minimum requirement. Resource allocation is measured by the product of the number of allocated processors and the duration of the allocation.

To illustrate, consider a graph with the number of processors P on the x axis and the time interval T on the y axis, as shown in Figure 6.1. A feasible schedule will partition the PxT rectangle into areas representing the allocation of processors to jobs over time. (We do not allow processors to be unallocated.) A schedule is an assignment of the J

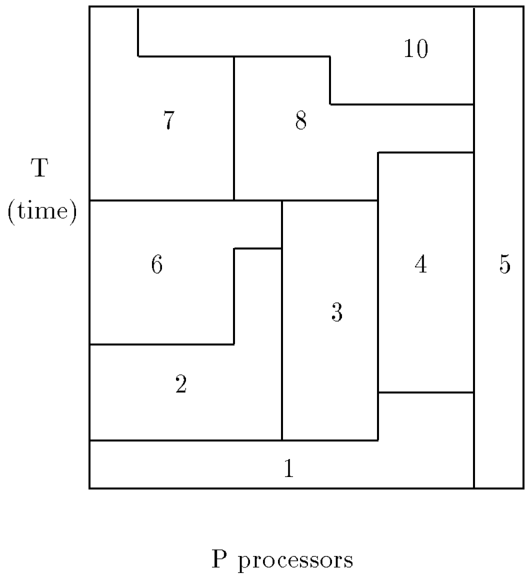


Figure 6.1: *Multiprocessor Scheduling (J,P)*

jobs to the areas in such a way that the total area for each job is $P * T/J$, and the minimum width of the area assigned to each job is greater than or equal to that job's minimum processor requirement.

The parameter T represents a time interval over which equal allocation is guaranteed. Choosing an appropriate value for T involves a tradeoff between fairness and performance. Fairness cannot be guaranteed over an infinitely small time interval, since the overhead of swapping within a small time interval would be prohibitive. On the other hand, performance of short jobs would suffer if an interval is too long, causing excessive queueing delays. Thus, T is a system definable parameter, set to bound the percentage of reallocation overhead and provide fair resource allocation within T.

It is always possible to find a feasible schedule for any inputs. Specifically, each job can be given P processors for time T/J (i.e., uniprogramming). What we want is the "best" policy.

Intuitively, an optimal policy minimizes the reallocation overhead – the total cost within time T of swapping out one or more jobs running on several processors and swapping in another set of parallel jobs. This cost depends on many factors, including the I/O bandwidth available, the connectivity of the I/O channels (IOC) to the processors, and the size and number of jobs being reallocated at any one time. In the extreme case of one IOC per processor, each processor can be reallocated in parallel; therefore swapping cost is independent of the number of processors swapping concurrently. At the other extreme, if there is a single IOC for all processors, then swapping must occur sequentially among all the processors. In general, the cost of each reallocation depends on both the number of processors competing simultaneously for each IOC and the total amount of data that must be swapped out.

Systems vary in I/O availability and configuration. As a result, it is not possible to define a single model of IOC connectivity to encompass all system configurations. For our analysis, we assume reallocation cost is independent of which processors are allocated to a job. In other words, we assume that all I/O parallelism is available to each processor, but that current reallocations among multiple processors serialize. Some of the schemes we propose tend to minimize the number of processors simultaneously involved in swapping; so this assumption is not critical to them.

As discussed in Chapter 2, reallocation cost alone is not the sole determinant of scheduling policy performance, since the number of processors allocated to a job affects its performance. We assume that specific information (such as speedup) about the performance of a job under varying allocations is unknown to the scheduler. In the absence of such job information, we make the general assumption that jobs exhibit sublinear speed-ups, implying that each job runs more efficiently on fewer processors (down to the job's minimum processor requirement).

Returning to the time sharing scheduling problem above, we define an optimal scheduling policy as one that produces a feasible schedule and minimizes the total number of processor reallocations within T. This objective function reflects a desire both to minimize I/O and to promote efficiency through allocation of fewer processors to each job. It also favors solutions for which jobs with small processor requirements are able to run uninterrupted if their minimum processor requirement is no greater than P/J.

We restrict our attention to schedules allocating processors to each job at most once within T. This simplifies the scheduling problem, and eliminates the need for jobs to adjust to varying allocation sizes, a potentially expensive operation. Also, it is a reasonable restriction, given the objective function of minimizing the number of reallocations.

This scheduling problem can be defined as follows: Given a set of J jobs, each with a minimum processor requirement, min_j, schedule the jobs over an interval [0,T], in such a way that, at each moment, each job, j, has either no fewer than min_j processors, or else has 0 processors, and each job is allocated a total of $P * T/J$ processor-seconds within the interval. Let A_j be number of processors allocated to job j, and let job j within T. An optimal policy minimizes:

$$\sum_{j=1}^{J} O_j \tag{6.1}$$

where $O_j = A_j$, if $A_j > P * T/J$, and $O_j = 0$, otherwise. This objective function minimizes the number of reallocations within T. (Jobs for which $A_j = P{*}T/J$ execute for the entire interval T, and thus are never reallocated). For the illustration of Figure 6.1, this translates to an objective function of minimizing the sum of the lengths of the horizontal lines, excluding the top line of the scheduling grid, and excluding the jobs that are never reallocated.

We characterize our scheduling problem by defining the corresponding decision problem, ***EQUAL_MIN_SCHED***, as follows (making the assumption that $T = 1$, without loss of generality):

EQUAL_MIN_SCHED(P,J,min[],L):

Instance: An integer P, an integer J, an array of J integers $min[]$, and an integer L.

Define a *valid schedule* for ***EQUAL_MIN_SCHED*** as a partitioning of a Px1 rectangle into J subrectangles, where each of the J rectangles is denoted by its lower left coordinate (x_i, y_i) and its dimensions $(width_i, height_i)$: x_i and $width_i$ are integers, y_i and $height_i$ are real numbers, $width_i \geq min_i$, $width_i * height_i = P/J$.

Question: Is there a *valid schedule* for which $\sum_{i=1}^{J} O_i <= L$, where $O_i = width_i$, if $height_i < 1$, $O_i = 0$, otherwise?

At first glance, ***EQUAL_MIN_SCHED*** appears similar to the physical memory allocation problem studied extensively a decade ago [4, 11]. At that time, only those jobs that could fit into real memory could be scheduled for execution. Memory scheduling policies determined which jobs to load into memory next so as to maximize the number of jobs in memory and minimize the fragmentation. However, the objective function of ***EQUAL_MIN_SCHED*** is to minimize the number of times processors are reallocated instead of minimizing unallocated processors (analogous to memory fragmentation). Furthermore, ***EQUAL_MIN_SCHED*** is a two-dimensional packing problem, whereas memory allocation is a one-dimensional problem. Our problem is distinguished from most other two-dimensional packing problems in that our subrectangles are *malleable*: the height and width of the rectangles can be varied, subject to the constraints of minimum width to each piece and equal area to all pieces.

Recent studies [43, 46] also consider the problem of scheduling malleable rectangles within the context of multiprocessor and data base query scheduling. In these studies, each rectangle represents the execution of a parallel task or database query: the width represents the number of processors executing the task and the height represents the duration of execution. Malleable rectangles represent different execution times of the tasks under varying allocations. The objective function is to minimize the total height of the schedule (i.e., makespan). Schedules that leave processors unallocated for some periods of time are allowed.

Turek et al. [43] show this scheduling problem to be NP-complete, and present efficient approximate solutions for schedules restricted to those consisting of shelves. At shelf boundaries, all processors are reallocated to jobs scheduled on the next shelf. Between shelf boundaries, no processors are reallocated. The height of a shelf is determined by the longest running job on that shelf. The total length of the schedule is determined by the sum of the heights of all shelves. $\boldsymbol{EQUAL\_MIN\_SCHED}$ differs from this scheduling problem in that job completion times are not known: the allowable dimensions of our rectangles are determined by the constraint of equal resource allocation (i.e., equal area) rather than the speedup properties of the jobs.

In Section 6.2, we discuss the complexity of $\boldsymbol{EQUAL\_MIN\_SCHED}$. We introduce a similar problem known to be NP-complete and argue that, while the complexity of $\boldsymbol{EQUAL\_MIN\_SCHED}$ remains an open problem, an efficient solution is unlikely. As a result, we take two approaches to finding good solutions for many inputs. In Section 6.3, we restrict our attention to problems in which the number of processors and the number of jobs are powers of two, and to policies that make power of two allocations. We present a scheduling policy in this class for these problems, and in Section 6.4 prove that policy to be optimal.

In Section 6.5, we impose a different restriction on the class of schedulers considered, requiring all processors to be reallocated at the same time. We call this class the *epoch scheduling* policies. Epoch scheduling policies differ from shelf-scheduling policies discussed above in that, within a shelf, the jobs run to completion and the duration of a shelf is determined by the execution time of the longest running job on that shelf. With epoch scheduling, jobs execute for some time interval, after which they are all swapped out and a new epoch (with a new set of jobs) begins. The duration of each epoch is chosen to result in equal allocation over the full period [0,T]. Unlike shelf scheduling, under epoch scheduling policies all processors are fully utilized each epoch.

This chapter concludes with a discussion of the interaction between the medium-term, time-sharing policies it define and the short-term, space-sharing policies studied in Chapters 4 and 5.

6.2 *MIN_SCHED* is NP-Complete

This section formulates a decision problem, called ***MIN_SCHED***, that considers a superset to the class of problems addressed by ***EQUAL_MIN_SCHED*** and shows that this decision problem is NP-Complete.

Define ***MIN_SCHED*** as follows:

MIN_SCHED(P_m, U_m, L):

Instance: Given an integer P_m and a set U_m of J pairs $(min\_width_i, max\_height_i)$, where min_width_i is an integer, $0 < max\_height_i \leq 1$, $1 \leq i \leq J$, and an integer L.

Define a *valid schedule* for ***MIN_SCHED*** as a set of J non-overlapping rectangular areas positioned within a rectangular area P_m x 1. Each of the J rectangles is denoted by its lower left coordinate (x_i, y_i) and its dimensions $(width_i, height_i)$, where x_i and $width_i$ are integers, and y_i and $height_i$ are real numbers. A *valid schedule* requires that $width_i \geq min\_width_i$ and $width_i * height_i = min\_width_i * max\_height_i$ for all $1 \leq i \leq J$.

Question: Is there a *valid schedule* for which $\sum_{i=1}^{J} width_i \leq L$?

To prove ***MIN_SCHED*** is NP-complete, consider also the following problem,

RECTANGLE_FIT:

RECTANGLE_FIT(U_r):

Instance: A set U_r of J pairs $(width_i, height_i)$, where $width_i$ is an integer and $height_i$ is a real number. $(width_i, height_i)$ corresponds to a box of width $width_i$ and height $height_i$.

Question: Is there a way to fit the J non-overlapping boxes into a rectangle of width $\sum_{i=1}^{J}(width_i * height_i)$ and height 1.0?

MIN_SCHED can be shown to be NP-complete by transforming the well-known NP-Complete problem PARTITION [19] to ***RECTANGLE_FIT***, and then transforming ***RECTANGLE_FIT*** to ***MIN_SCHED***.

PARTITION(A):

Instance: Finite set A of integers a_i.

Question: Is there a subset $A' \subseteq A$ such that $\sum_{a_i \in A'} a_i = \sum_{a_i \in (A-A')} a_i$.

Theorem 6.1 ***RECTANGLE_FIT*** *is NP-complete.*

Proof: Reduce ***PARTITION***(A) to ***RECTANGLE_FIT***(U_r). Define a set U_r of pairs $(a_i, \frac{1}{2})$ for each $a_i \in A$. Let $P = \frac{1}{2}\sum_{i=1}^{J} a_i$. If ***RECTANGLE_FIT***$(U_r)$ is true, then it must be the case that the boxes of U_r are arranged in two rows (of height $\frac{1}{2}$ each). Since the width of the boxes in each row must sum to P, the rows define a solution to ***PARTITION***. Conversely, if A can be partitioned into two subsets, then the corresponding boxes of U_r can fit into a P x 1 rectangle using 2 rows, each row corresponding to a subset of A.

Finally, to see that ***RECTANGLE_FIT*** is in NP, encode a solution by ordering the rectangles in bottom-to-top order of lower left coordinate, and within the same base height, left-to-right order. An oracle can verify in polynomial time that the rectangle positions represented by the ordering fit within the P x 1 area.

QED

Theorem 6.2 $\boldsymbol{MIN\_SCHED}$ *is NP-complete.*

Proof: Reduce $\boldsymbol{RECTANGLE\_FIT}(U_r)$ to $\boldsymbol{MIN\_SCHED}(P_m, U_m, L)$. Let $min\_width_i = width_i$, for each $width \in U_r$ and $max\_height_i = height_i$ for each $height_i \in U_r$, $1 \leq i \leq J$. Let $P_m = \sum_{i=1}^{J}(min\_width_i * max\_height_i)$. Let $L = \sum_{i=1}^{J} width_i$.

If $\boldsymbol{MIN\_SCHED}$ produces a schedule S_m with $\sum_{i=1}^{J} width_i \leq L$, then it must be the case that each $width_i \in S_m = min\_width_i$ and each $height_i \in S_m = max\_height_i$. Thus S_m is also a valid fitting of boxes of U_r. Conversely, if $\boldsymbol{RECTANGLE\_FIT}(U_r)$ is true, then the fit produced by $\boldsymbol{RECTANGLE\_FIT}$ corresponds to a valid schedule in $\boldsymbol{MIN\_SCHED}$, where the $\sum_{i=1}^{J} width_i \leq L$.

Finally, to show $\boldsymbol{MIN\_SCHED}$ is in NP, encode the solution space to include both an ordering of the rectangles in bottom-to-top, left-to-right order, and a list of widths corresponding to each rectangle in the ordering. An oracle may generate in polynomial time a schedule from these lists by allocating each rectangle in bottom-to-top, left-to-right order for the corresponding width, and verify that the rectangles fit within the P x 1 area and that the sum of the widths of the rectangles is less than or equal to L.
QED

The general scheduling problem, $\boldsymbol{EQUAL\_MIN\_SCHED}$, defined in Section 6.1 is a restriction on $\boldsymbol{MIN\_SCHED}$ in which $min\_width_i * max\_height_i = P_m * T_m/J, \forall 1 \leq i \leq J$. While the complexity of $\boldsymbol{EQUAL\_MIN\_SCHED}$ remains an open problem, we believe that restricting the input rectangles to having the same total area is not likely to reduce this complexity.

6.3 The *BUDDY* Scheduling Policy

Given the unlikelihood of an efficient optimal solution to $\boldsymbol{EQUAL\_MIN\_SCHED}$, it is natural to ask under what conditions an optimal solution can be found. This section presents a scheduling policy, called *BUDDY*, for a restricted class of problems, those in which the number of processors and the number of jobs are powers of two, and a restricted class of policies, those that allocates powers of two numbers of processors. In Section 6.4, we show $BUDDY$ is optimal for these restrictions.

In $BUDDY$ scheduling, processors are allocated to jobs in order of non-increasing minimum processor requirement of the jobs. $BUDDY$ attempts to assign each job the smallest power of two number of processors greater than its minimum processor requirement. However, in some cases, it may be necessary to assign a job a larger allocation.

Consider P, J, and allocations A_i, for all jobs i, to all be powers of two; thus $P = 2^p$, $J = 2^n$, and $A_i = 2^{a_i}$. Since each job is allocated an equal percentage of the processing resources, job i will be allocated 2^{a_i} processors for duration $2^p/(2^n * 2^{a_i})$. Thus, each rectangle allocated in the schedule has width 2^{a_i} for some a_i and height $1/2^{n+a_i-p}$.

Represent the scheduling area as a Px1 rectangular grid with P on the x axis. Define a *partial schedule* as an allocation to a subset of the jobs in the system, where each allocation is given by a coordinate (x_i, y_i) and dimensions (A_i, t_i). This represents a rectangle of width A_i and height t_i whose lower left corner is at (x_i, y_i). As will be seen, allocations are scheduled within the Px1 scheduling area in a bottom-up fashion, whenever possible. Define the $frontier$ of a partial schedule to be the lowest possible set of (horizontal) line segments within the Px 1 scheduling grid above which no allocations have been made.

Lemma 6.1 defines a property of partial schedules created by the $BUDDY$ scheduling policy that forms the basis for its allocation decisions. 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 processor allocation to be placed above that point on the graph. This restriction on the maximum height (minimum width) of the largest allocation placed above a current partial schedule is used by the $BUDDY$ policy to determine the size of the next allocation to be scheduled.

To illustrate this restriction, consider a system with 16 processors where 8 jobs

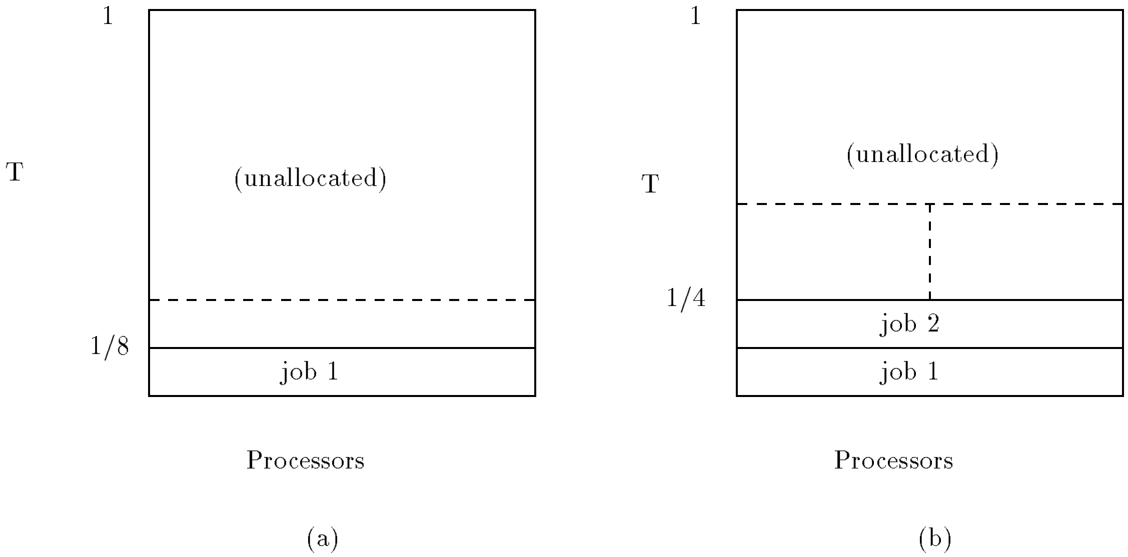


Figure 6.2: *Example Buddy schedule*

are to be scheduled, and the minimum processor requirement for the jobs is $min = [2, 2, 4, 8, 8, 8, 8, 16]$ [1]. Since one job requires all the processors, it will be scheduled on 16 processors. Since it must be allocated an equal percentage of the processing power, it must be allocated for 1/8 of the time. (See Figure 6.2(a).) To fill the remaining 7/8 of the schedule above this allocation with allocations of height $1/2^x$, at least one of the other jobs must get an allocation of height no more than 1/8. This corresponds to an allocation of no fewer than 16 processors. Therefore, even though no other job requires 16 processors, at least one other job must run on 16 processors. This is represented by the dashed line in Figure 6.2(a). Similarly, as showing in Figure 6.2(b), if the height of a current partial schedule summed to 1/4 of T at some position in partial schedule, to complete the rest of the schedule above that position, at least one other job placed above that position must have an allocation of height 1/4 or less, which corresponds to a width of 8 processors or more, regardless of minimum processor requirements of the

[1] The minimum processor requirement is not restricted to a power of two, but since all allocations must be powers of two, for simplicity we assume here the minimum requirement is also a power of two.

remaining jobs to be scheduled.

We begin by defining the following property of the height of any segment of a frontier of a *BUDDY* schedule.

Property 6.1 *If P, J, and A_i for all i are powers of two, then any line segment of the frontier for a partial schedule will have height $C/2^x$ for some odd integer C, integer $x \geq 0$.*

Since each allocation is a power of two, the duration of each allocation (e.g., each allocation height) will be $1/2^i$ for some i. Therefore, the sum of any allocations can be written in the form $C/2^x$ for some odd C and some x. This is the irreducible fractional representation of the segment's height.

We now prove Lemma 6.1.

Lemma 6.1 *For each line segment of a frontier of height $C/2^x$ for some odd integer 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 $1/2^x$.*

Proof: Let B_v be the height of the current set of allocations along a vertical cut of the PxT scheduling area. By definition, the sum of the heights of the allocations placed above a line at height B_v must sum to $1 - B_v = (2^x - C)/2^x$ or $K/2^x$ for some odd K.

In order for subsequently added allocations of height $1/2^i$, $i \geq 0$, to sum to $K/2^x$ for some odd K and some $x \geq 0$, there must be at least one allocation whose height is less than or equal to $1/2^x$. Thus, given any frontier of the PxT scheduling area, there is an upper bound on the smallest allocation height to be added above it. Since the area of each allocation is P/J, there is a corresponding lower bound of $P/(J * 1/2^x)$ on the largest allocation width to be added above B_v.

QED

Figure 6.3 gives the *BUDDY* algorithm. Jobs are represented by their minimum processor requirement within an array, *min*[], sorted in non-decreasing order. *Frontier* represents the frontier of a partial schedule. It is composed of a set of a segments given

```
$\boldsymbol{BUDDY}(P, J, min[])\{$
    int min[0..J-1];                                   /* minimum processor requirement */
    linked_list $Frontier$ = new linked_list(0,0);      /* list of segments representing /*
                                                       /* current top of allocations */
    For (int $j = J - 1$; $j \geq 0$; j–) {
        $min\_width\_exp = \lceil log_2 min_j \rceil$                  /* job's min width = $2^{min\_width\_exp}$ */
        $max\_height\_exp = log_2(J/P) - min\_width\_exp$;     /* $1/2^{max\_height\_exp}$ height */
                                                       /* corresponding to $min\_width\_exp$ */
        $(x, C/2^{min\_seg\_exp}) \leftarrow$ Get_min_frontier_segment();  /* returns $min\_seg$ */
        if ($max\_height\_exp \leq min\_seg\_exp$)             /* allocation forced by $min\_seg$ */
            Allocate($j, P * 2^{min\_seg\_exp}/J, 1/2^{min\_seg\_exp}$);
        else                                           /* allocation not forced */
            Allocate($j, 2^{min\_width\_exp}, 1/2^{max\_height\_exp}$);
    }

    Get_min_frontier_segment() {
        /* Returns the first segment with the largest denominator */
        /* when represented as an irreducible fraction */
    }

    Allocate(int $j$, int $width$,float $height$) {
        /* Assigns job $j$ an allocation starting at coordinate $(x, C/2^{min\_seg\_exp})$, */
        /* for width, $width$, and height, $height$. */
        Update_frontier($x$, $C/2^{min\_seg\_exp}$, $width$);
    }

    Update_frontier(int $x$,float $y$,int $box\_size$) {
        float $prev\_height$ = $Frontier$.get_prev_height($x$,$y$); /* height of segment preceding */
                                                           /* $(x, y)$, or 1 if $(x, y)$ not found */
        int $segment\_width$ = $Frontier$.get_width($x$,$y$);      /* width of segment */
        $Frontier$.remove($x$,$y$);
        float $new\_height = y + P/(J * box\_size)$;
        if ($new\_height \neq prev\_height$))
            $Frontier$.add($x$,$y + new\_height$));
            if ($box\_size < segment\_width$)
                $Frontier$.add($x + box\_size$, $y$);
    }
}
```

Figure 6.3: *BUDDY Scheduling Algorithm*

by coordinates in the scheduling graph. Each coordinate (x,y) is the starting point of a segment. The ending point and length of segment can be computed from the x coordinate value of the next segment.

$BUDDY$ schedules jobs in decreasing order of minimum processor requirement. It first determines the power of two minimum width permitted by the job's minimum processor requirement (represented in Figure 6.3 by the exponent min_width_exp of this power of two number) and the corresponding maximum height (represented in Figure 6.3 by the exponent of the power of two denominator, max_height_exp).

The function $Get\_min\_frontier\_segment()$ returns the frontier segment that has the largest denominator when represented as an irreducible fraction. Call this segment min_seg. For this segment, $BUDDY$ determines the maximum allocation height of the largest possible allocation to be placed above min_seg. (This value is represented in Figure 6.3 by the exponent of the power of two denominator, min_seg_exp). We call this the $forced$ height of the min_seg segment. (Similarly, the corresponding width of that allocation is called the $forced$ width of the min_seg.).

$BUDDY$ gives the next job to be scheduled an allocation whose height is the minimum of

1. its maximum height, and

2. the height forced by the min_seg.

This allocation is placed in the left-most position on top of the min_seg segment. After each allocation, $Frontier$ is updated.

As an example, consider scheduling $P = 16$ processors and $J = 8$ jobs, where the minimum processor requirement of the jobs is $min_n = [2, 2, 4, 4, 4, 8, 8, 8]$. Figure 6.4 illustrates the schedule created by the $BUDDY$ scheduling algorithm. The numbers inside the rectangular allocations represent the minimum processor requirement of the job assigned to that allocation. The first three jobs, all with minimum processor requirements of eight, are scheduled consecutively on eight processors each. The fourth job, with a

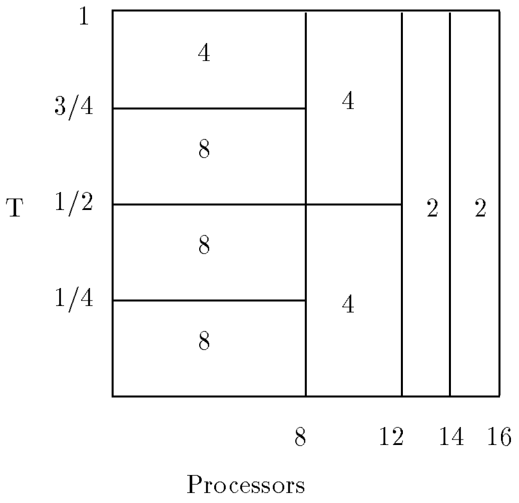


Figure 6.4: *BUDDY Schedule for* $P = 16$,$J = 8$, $min = [2, 2, 4, 4, 4, 8, 8, 8]$

minimum processor requirement of four processors, is forced onto an allocation of eight processors in order to complete the schedule above the first allocations. The rest of the jobs are allocated their minimum processor requirements. The *BUDDY* algorithm is able to schedule two jobs on only two processors, so that these jobs never have to incur the overhead of reallocation.

6.4 Analysis of *BUDDY* Scheduling Policy

In this section, we prove the *BUDDY* schedule always produces a valid and optimal schedule when P, J, and a_i for all i are powers of two.

6.4.1 *BUDDY* Produces a Feasible Schedule

Theorem 6.3 *The BUDDY algorithm always produces a valid schedule.*

Proof We prove this by showing that *BUDDY* can always schedule the next job. Let $P = 2^p$, $J = 2^j$, and all allocations to jobs be of the form 2^a. Consider following invariant of the frontier of a partial schedule created by *BUDDY*:

Inductive Hypothesis: After job $i < J$ is added to the schedule and given 2^{a_i} processors, each segment x of the frontier has width greater than or equal to 2^{a_i} and height $C_x/2^{j+x-p}$ for some integer C_x, $x <= a_i$.

Given this invariant, it follows that the next job, $i+1$, can be scheduled. *BUDDY* schedules job $i+1$ on segment *min_seg*. It gives the job either its minimum processor requirement, if it fits, or a larger allocation forced by min_seg.

Base: $i = 1$. Trivial.

Induction: Assume the hypothesis is true after scheduling job i. *BUDDY* schedules the next job on the *min_seg* segment, the segment with the largest denominator, when represented as an irreducible fraction. Let $C/2^x$, for some odd C, be the height of *min_seg*. *BUDDY* will give the job one of two possible allocations.

If job $i+1$ has a maximum height greater than the height forced by *min_seg*, then the job will be given a partition of height $1/2^x$ placed above *min_seg*. It remains to be shown, for this case, that the resulting frontier satisfies the invariant for $i+1$ jobs. Adding the $i+1^{th}$ allocation to the partial schedule results in either:

1. the *min_seg* segment of the frontier replaced by two segments, one segment $1/2^x$ higher than the height of the previous segment, and the other segment the same height,

2. the *min_seg* segment of the frontier is replaced by a single segment, whose height is $1/2^x$ higher.

The higher segment has height $(C+1)/2^x$, for some odd C, which equals $K/2^{x-k}$, for some odd K, $k \geq 1$. Thus, if the segment height is not 1, then it will have enough height above to support another segment of identical height (and sufficient width). The lower segment, if one exists, will be at least as wide as the higher segment (and, of course have sufficient height). Finally, according to the inductive hypothesis, the remaining segments have width greater than a_i, which is greater than or equal to a_{i+1}. Furthermore, the denominators of these segments all have values 2^y, $y <= x$. Thus, they can support allocations above the segments with heights of $1/2^x$.

If $m = \lceil log_2 min_{i+1} \rceil$, the maximum possible height of job $i+1$ is less than or equal to the height forced by min_seg, then the job will be given a partition of height $1/2^{j+m-p}$ placed above min_seg. It remains to be shown, for this case, that the resulting frontier satisfies the invariant for $i+1$ jobs. Adding the $i+1^{st}$ allocation to the partial schedule results in the min_seg segment of the frontier replaced by two segments, one segment $1/2^m$ higher than the height of the previous segment, and the other segment the same height. The higher segment has height $C/2^m$, for some odd C. Thus, if the segment height is not 1, then it will have enough height above to support another segment of identical height (and sufficient width). The lower segment will be at least as wide as the higher segment and will have space above to support a smaller segment height as indicated by the induction hypothesis. Finally, according to the inductive hypothesis, the remaining segments have width greater than a_i, which is greater than or equal to a_{i+1}. Furthermore, the denominators of these segments all have values 2^y, $y <= x$, Thus, they can support allocations above the segments with heights of $1/2^m$, $m >= x$.

QED

6.4.2 *BUDDY* Scheduling Policy is Optimal

In this section, we show the $BUDDY$ schedule is optimal when P, J, and each allocation to jobs of A_i processors are powers of two. To do so, we first need to establish several properties of schedules created by the $BUDDY$ algorithm. First, Lemma 6.2 states that all segments of the frontiers of a $BUDDY$ decrease in height (e.g, the frontier resembles a staircase). This property is used to show that all segments of a frontier, when represented as an irreducible fraction have unique denominators. This implies there is a single segment, called min_seg, which has the largest denominator (Corollary 6.1 of Lemma 6.3). This segment determines the maximum allocation of the next job to be scheduled. This lemma is used in Lemma 6.5 to show if the $BUDDY$ algorithm forces an allocation to some job i to be of size 2^{b_i}, then there is no possible partial schedule for jobs 1 to $i-1$ for which the all segments of the frontier have heights $K/2^{j+x-p}$ where

$x < b_i$. In other words, there is no way to complete a partial schedule without giving a job an allocation width greater than or equal to 2^{b_i}. This property is exploited in Theorem 6.4 to prove the *BUDDY* algorithm produces an optimal schedule.

Lemma 6.2 *Every segment of a frontier created by the BUDDY schedule has height less than or equal to the height of the segment immediately to the left.*

Proof: by induction on the number of allocations i in a partial schedule.
Base: $i = 1$. Allocation for job 1 is placed at the bottom-left of the PxT scheduling grid. Thus, the new frontier has either a single segment (of width P) or two segments, with the height of the second segment less than the height of the first.
Induction: Assume true for a partial schedule of i jobs. *BUDDY* allocated job $i+1$ above the segment with the largest denominator when represented as an irreducible fraction. Let *min_seg* be this segment (returned from Get_min_frontier_segment()). If *min_seg* is not the first segment, Let the height of *min_seg* be $C/2^x$, for some odd C, and let $K/2^a$, for some odd K, be the height of the segment immediately preceding *min_seg*. $K/2^a = K * 2^{x-a}/2^x > C/2^x$, $a < x$. The allocation to job $i+1$ will have height $1/2^x$ or less, and the new frontier would involve increasing part or all of the segment by this height. Still, the new height must be less than or equal to $K/2^a$, since $K * 2^{a-x}/2^x \geq (C+1)/2^x$. Thus, it is impossible for the new segment height to be greater than the segment to the left.
QED

The next two lemmas together prove that no two segments of a frontier created by the *BUDDY* scheduling policy have the same power of two denominator when represented as a irreducible fraction. This implies there is a single segment, called *min_seg*, which has the largest denominator, when represented as an irreducible fraction.

Lemma 6.3 *Let S be a partial schedule created by the BUDDY algorithm. Every segment i of the frontier of S has height $C_i/2^{x_i}$, for some odd C_i, and $x_i \neq x_j, \forall$ segments $i, j \in thefrontier$.*

Proof: By induction on the number of allocation in the partial schedule, j.

Base: $j = 1$. Trivial.

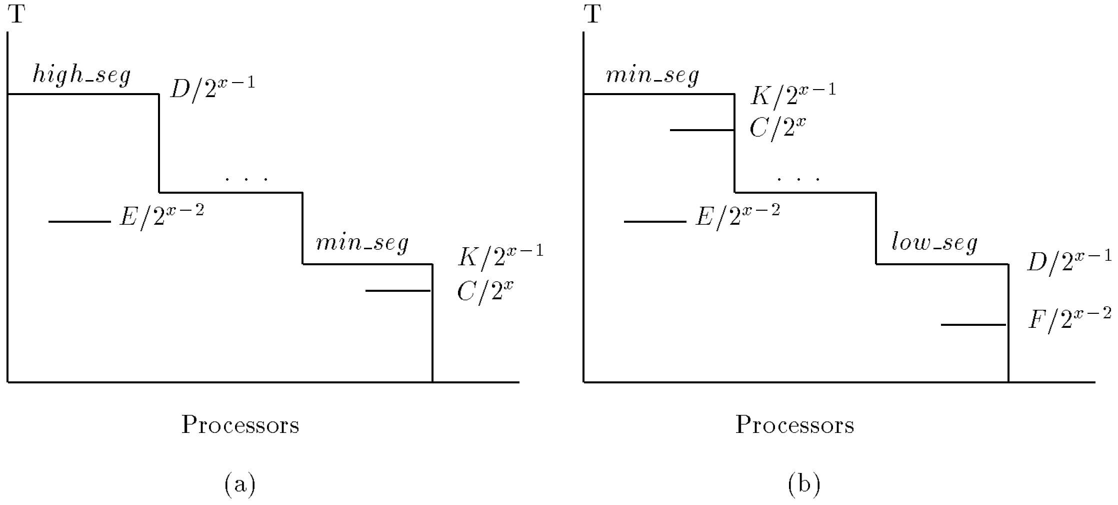
Induction : Assume the lemma (hypothesis) is true for partial schedule with j allocations. Let $C/2^x$, for some odd C, be the height of the *min_seg* segment of the frontier. $BUDDY$ will give the job one of two possible allocations. If job $i+1$ has a maximum height greater than the height forced by *min_seg*, then the job will be given a partition of height $1/2^x$ placed above *min_seg*. Adding the $j+1^{th}$ allocation to the partial schedule results in either:

1. the *min_seg* segment of the frontiers replaced by two segments, one segment $1/2^x$ higher than the height of the previous segment, and the other segment the same height,

2. the *min_seg* segment of the frontier is replaced by a single segment, whose height is $1/2^x$ higher.

The higher segment has height $(C+1)/2^x$, for some odd C, which equals $K/2^{x-k}$, for some odd K, $k \geq 1$. By Lemma 6.2, this segment has the same height or less than its immediately preceding neighbor. If it has the same height, then it is combined into a single segment of height $K/2^{x-k}$. Otherwise, it remains to be shown there can not be another segment of the frontier of height $D/2^{x-k}$, for some odd D.

Suppose, by contradiction, there is another segment of height $D/2^{x-k}$, for some odd D. If $D > K$, then this segment is located to the left of *min_seg*. Denote this left-most, higher segment as *high_seg*. Figure 6.4.2(a) illustrates this scenario. Corollary 6.2 of Lemma 6.4 states that *high_seg* must have previously had a height of $(D-1)/2^{x-k} = E/2^{x-k-j}$, for some odd E, $j \geq 1$. At that time, *min_seg* was at or below $C/2^x$. If *min_seg* was at $C/2^x$, then the $BUDDY$ schedule would not have added allocations above $E/2^{x-k-j}$, and *high_seg* could not have attained its current height of $D/2^{x-k}$. If *min_seg* was below $C/2^x$, then in order for *high_seg* to have advanced beyond the height of $E/2^{x-k-j}$, *min_seg* would have to be at a height $F/2^{x-k-j-l}$, for some odd F, $l \geq 1$. But then after *high_seg* advanced to its height of $D/2^{x-k}$, the $BUDDY$ schedule would



not have advanced min_seg from $F/2^{x-k-j-l}$ before advancing $high\_seg$. So it is not possible for another segment to have a height of $D/2^{x-k}$, for some odd $D > K$.

A similar argument is used when $K < D$ (when the other segment with the same denominator is to the right of min_seg). Denote this segment as low_seg. Figure 6.4.2(b) illustrates this scenario. According to Corollary 6.2, in order for min_seg to have reached the height of $K/2^{x-k}$, it must have previously had a height of $(K-1)/2^{x-1} = E/2^{x-k-j}$ for some odd E, $j \geq 1$. At that time, low_seg was at or below $D/2^{x-k}$. If low_seg was at $D/2^{x-k}$, then the $BUDDY$ schedule would not have added allocations above $E/2^{x-k-j}$ until low_seg reached a height greater than its height $D/2^{x-k}$. If low_seg was below $D/2^{x-k}$, then in order for min_seg to have advanced beyond the height of $E/2^{x-k-j}$, low_seg would have to be at a height $F/2^{x-k-j-l}$, for some odd F, $l \geq 1$. But then $BUDDY$ would have advanced min_seg to a height of $K/2^{x-k}$ before any advancements of low_seg from its height of $F/2^{x-k-j-l}$.

Finally, if job $i+1$ has a maximum height less than or equal to the height forced by min_seg, then the job will be given a partition of height $1/2^{j+m-p}$ placed above min_seg, where $m = \lceil log_2 min_{i+1} \rceil$. This corresponds to an allocation height of $1/2^{j+m-p} = 1/2^{x+k} < 1/2^x$. This will result in a new segment height of $C/2^x + 1/2^{x+k} = (C * 2^k + 1)/2^{x+k}$, where $k \geq 1$, which is equal to $D/2^{x+k}$, for some odd D, which has a denominator unique (and greater than) the denominators of the other segments.

QED

Corollary 6.1 *For any segment of a frontier created by the BUDDY scheduling algorithm, there is a single segment, called min_seg, with the largest denominator, when represented as an irreducible fraction.*

Lemma 6.4 *If a segment of a frontier of a schedule created by the BUDDY algorithm has a height $C/2^x$, for some odd C, $1 \leq x \leq n$, then there exists a set of allocations lying immediately below the segment and having a combined height of $1/2^x$.*

Proof by induction on x.

Base: $x = n$. Trivial.

Induction: Assume the hypothesis holds for $C/2^y$, $y \geq x$. We show the hypothesis is true for $C/2^x$. If the segment has a height $C/2^x$, for some odd C, then the next allocation must have height less than or equal to $1/2^x$. Since all previous allocations must have heights less than or equal to all subsequent allocations, then all previous allocations must have height less than or equal to $1/2^x$. If the highest possible allocation, $1/2^x$, occurred in the previous allocation, then the inductive hypothesis holds for x.

Otherwise, let $1/2^{x+i}$ $(i \geq 1)$ be the height of the previous allocation below this segment. Before this allocation, the previous height of the segment was $C/2^x - 1/2^{x+i} = (2^i * C - 1)/2^{x+i} = K/2^{x+i}$, for some odd K. According to the inductive hypothesis, in order for the height of $K/2^{x+i}$ to be achieved, the height of the previous allocations totaled $1/2^{x+i}$. Thus the total height thus far of allocations below the segment at height $C/2^x$ is $1/2^{x+i} + 1/2^{x+i} = 1/2^{x+i-1}$. Therefore, a previous height of the segment before these allocations was $C/2^x - 1/2^{x+i-1} = (2^{i-1} * C - 1)/2^{x+i-1} = K/2^{x+i-1}$, for some odd K. Again, according to the inductive hypothesis, in order for the height of $K/2^{x+i-1}$ to be achieved, the height of the previous allocations totaled $1/2^{x+i-1}$. Thus the total height thus far of allocations below the segment at height $C/2^x$ is $1/2^{x+i-1} + 1/2^{x+i-1} = 1/2^{x+i-2}$. Therefore, a previous height of the segment before these allocations was

$C/2^x - 1/2^{x+i-2} = (2^{i-2} * C - 1)/2^{x+i-2} = K/2^{x+i-2}$, for some odd K. It is easy to see this pattern repeats until the allocations below the segment of height $C/2^x$ have total height $1/2^{x+i-i} = 1/2^x$.

QED

Corollary 6.2 *If a BUDDY frontier segment has a height $C/2^x > 1/2^{x-1}$, then it had to have passed through height $(C-1)/2^x = D/2^{x-k}$ for odd $D = (C-1)/2^k$, for some $k \geq 1$.*

Lemma 6.5 *Let S be a partial schedule produced by the BUDDY algorithm for jobs 1 to i. If min_seg has height $C/2^s$ for some odd C, then it is not possible to rearrange the allocations of jobs 1 to i so that a resulting frontier has all segments of heights representable $K/2^x$ for K odd and $x < s$.*

Proof: Consider each column of the partial schedule, where a column represents the allocations to a single processor. The height of any allocation to a processor is $1/2^x$ for some integer x which can be represented as a binary fraction, where the $(n+1)^{th}$ binary place represents $1/2^n$. The sum of all allocations to a processor is $C/2^x$, for some odd C, which can also be represented as a binary fraction with no less than $x+1$ binary places. The processor columns comprising the *min_seg* have the greatest number of necessary binary places $(s+1)$. Ignoring the width of allocations, consider rearranging the allocation heights among the processor columns, so that all processor columns have heights whose binary fractions have fewer than $s+1$ binary places. Since all columns not directly under the *min_seg* have a height whose binary fraction has fewer than $s+1$ binary places, the only way to eliminate the $s+1$ binary place from the sum of the heights of the *min_seg* columns is to remove a height of $1/2^x$ from $\frac{1}{2}$ of the *min_seg* columns, and place those allocations above the other half of the processor columns of *min_seg*.

It is not possible to divide the allocations lying below *min_seg* in the manner described above. To see this, consider the set of allocations lying within a height $1/2^s$

immediately below *min_seg*. (According to Lemma 6.4, such a set of allocations exists). It is not possible for these allocations to be halved in the manner described above, for otherwise, in adding allocations above height $(C-1)/2^s$, *BUDDY* would have added allocations in such a way that the right half of the allocations would have been added only after the left half of the allocations, since the segment height of the left half would always have had a greater denominator. But, after the left half of the allocations were added, the frontier below *min_seg* would have consisted of two segments of height $(C-1)/2^s$ and $C/2^s$. *BUDDY* would have placed further allocations above the left half (above $C/2^s$) instead of filling the right half. Therefore, at least some of the allocations within the set of allocations lying within a height $1/2^s$ immediately below *min_seg* are wider than $\frac{1}{2}$ of the *min_seg*. Finally, since these allocations are at least as wide as any allocations lying below these allocations of the *min_seg*, it is not possible for any allocations below the *min_seg* to be rearranged in such a way that a set of allocations of width $\frac{1}{2}$ of the *min_seg* and height $1/2^x$ are placed above $\frac{1}{2}$ of the *min_seg*.
QED

Theorem 6.4 *The BUDDY algorithm produces an optimal schedule when P, J, and allocations A_i are powers of two (for all i).*

Proof by Contradiction. Let $A_i = 2^{a_i}$ for some $a_i \geq 0$. Let $P = 2^p$ and $J = 2^j$ for some $j, p \geq 0$. Assume there is some optimal schedule, O, for which the total number of processors reallocated within T is less than that of a schedule, B, produced by the *BUDDY* algorithm. Now, consider the list of jobs as ordered within *min*[] (sorted by non-decreasing minimum processor requirement) where $i > j$ implies job i appears after job j in this list.

Without loss of generality, assume schedule O has the following property:

Property 6.2 *For all job pairs i and j, if $min_i \geq min_j$, $a_i \geq a_j$.*

Any feasible schedule can be transformed into a schedule meeting this property by switching the allocations for all jobs pairs, i and j not meeting this criteria.

Consider the first job i represented by $min[]$ allocated a different number of processors under the optimal schedule O than the schedule produced by the $BUDDY$ Algorithm. Let 2^{o_i} be the number of processors allocated to job i under schedule O and 2^{b_i} be the number of processors allocated to job i under schedule B.

$o_i < b_i$ implies the $BUDDY$ algorithm gave a job an allocation greater than the minimum power of two greater than its minimum processor requirement. When allocating job i, a segment of the frontier of Schedule B must have had a lower bound on the largest processor allocation to be placed above that segment equal to 2^{b_i} and greater than $2^{\lceil \log_2 min_i \rceil}$. In other words, at least one segment of the frontier of schedule B for $i-1$ allocations had height $\frac{K}{2^{j+b_i-p}}$, for some odd K.

Consider, for the schedule O, the partial schedule representing the allocation of jobs 1 to $i-1$. Consider $Frontier_O$ to be the set of line segments defining the boundary between allocated and unallocated space in this schedule. Define $Frontier_B$ to be the frontier in the $BUDDY$ algorithm before job i is allocated. Assume for now schedule O contains no unallocated space below allocated space.

Because of Property 6.2, for an optimal schedule to give an allocation of size $2^{o_i} < 2^{b_i}$, all subsequent allocations to jobs $i+1$ to J will have allocations fewer than 2^{b_i}. Hence, all $Frontier_O$ line segments must height of the form $K/2^x$ where $x < j + b_i - p$. Lemma 6.5 proves this is not possible; if the $BUDDY$ algorithm forces a segment to be of size 2^{b_i}, then there is no possible partial schedule for jobs 1 to $i-1$ for which the all segments of the frontier have heights $K/2^{j+x-p}$ where $x < b_i$.

Now, consider the case where schedule O contains unallocated space below allocated space. Since any subsequent allocations must have a width less than or equal to 2^{o_i}, any future allocations must be of a height greater than or equal to $1/2^{j+o_i-p}$. Thus, the height of any unallocated space below allocated space of schedule O must have height $C/2^{j+x-p}$, for some odd C, where $x > b_i$. Since any partial schedule for jobs 1 to $i-1$ will have some frontier height $C/2^{j+b_i-p}$, and no unallocated space within or below the previous allocations may be of height less than or equal to $1/2^{j+b_i-p}$, then some line segment of schedule O above which no allocations occur must have a height $K/2^{j+b_i-p}$, for some odd K, and therefore, it is not possible to complete schedule O without an allocation of width 2^{b_i}.

Finally, consider the case in which $o_i > b_i$, or the optimal schedule O gives an allocation greater than the $BUDDY$ schedule for job i. Since the $BUDDY$ schedule was able to allocate 2^{b_i} processors to job i, then the min_seg of the frontier before job i is allocated had height $C/2^{j+x-p}$, for some odd C, where $x \leq b_i$. Since 2^{b_i} is the largest possible height of the previous allocations, it is not possible to rearrange the $i-1$ previous allocations, so that the height of some unallocated space does not sum to $K/2^{j+x-p}$, for some odd K. If schedule O gives 2^{o_i} processors to job i, where $o_i > b_i \geq x$, then the unallocated space above or below the allocation to job i must sum to $D/2^{j+o_i-p}$, for some odd D. As a result, in order for all allocations lying directly above or below the allocation to job i to sum to 1, some future allocation to one of jobs $i+1$ to J must be given an allocation of height $1/2^{j+o_i-p}$, and which lies directly above or below the allocation to job i. But then, schedule O may be modified as follows to result in a new schedule better than schedule O, thus contradicting the assumption schedule O is optimal:

1. Rearrange schedule O, so that the allocations to jobs i and j lie directly above one another. This is possible since both jobs have the same width and encompass the same processors.

2. Replace the allocations to jobs i and j with two allocations lying side by side with one half the width and twice the height of the previous allocations. The smaller allocation to job i is possible since job i was allocated $b_i < o_i$ processors in the $BUDDY$ schedule. The allocation to job j is possible since its minimum processor requirement is less than or equal to that of job i. Thus, the new schedule has fewer reallocations overall than schedule O, a contradiction to the assumption that schedule O is optimal.

QED

Generalization of *BUDDY* Policy

The *BUDDY* policy is defined for a power of two number of jobs in the system. When J is not a power of two, an additional level of scheduling is needed to divide the jobs into groups, where each group has a power of two number of processors. A buddy schedule is created to allocated the processors to the jobs within each group, and the duration of a group is determined by the number of jobs within the group.

There are many possible approaches to dividing the jobs among the groups. One reasonable division would fill the smallest groups first, allocating the jobs with the largest memory requirements first. This strategy would provide the greatest opportunity for jobs with small processor requirements to run uninterrupted if their minimum processor requirement is less than P/J. Also, jobs within a group may form a job class with similar memory requirements. Thus, it would be possible to adjust the frequency or duration for which a group is scheduled to give priority to one group over another.

6.5 Epoch Scheduling

In this section, a restricted class of scheduling policies, called epoch scheduling, is considered. In epoch scheduling policies, all schedules consist of epochs: at each reallocation moment, all processors are reallocated at once to a (possibly) new set of jobs, and within epochs, no reallocations occur. For example, the schedule in Figure 6.5 consists of three epochs. During the first epoch, job 1 is allocated all the processors for $1/4 * T$. During the second epoch, jobs 2 and 3 are allocated 1/2 of the processors each for $1/2 * T$, and during the last epoch, job 4 is allocated all the processors.

As before, schedule optimality is defined as a minimization of the total number of processor reallocations that take place in providing (at least nearly) equal service to all jobs. For the class of epoch scheduling policies, this is equivalent to minimizing the number of epochs.

In the work presented here, we consider only epoch scheduling policies providing

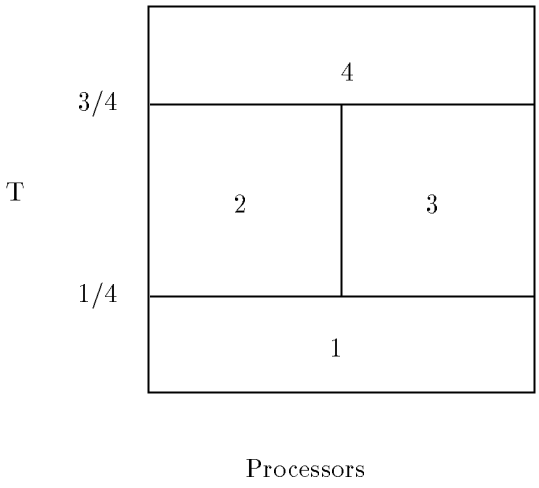


Figure 6.5: *An epoch schedule for 4 jobs*

exactly equal allocations. This implies each job scheduled within an epoch is allocated the same number of processors. The duration of an epoch is determined in such a way that all jobs receive the same number of processor-seconds of execution time. We define a scheduling policy, called $EQUAL\_EPOCH$, within this class and prove it is optimal when the number of processors is a power of two.

6.5.1 The $EQUI\_EPOCH$ Policy

We define an epoch scheduling policy, $EQUI\_EPOCH$, that allocates an equal number of processors to each job within an epoch. $EQUI\_EPOCH$ uses a greedy approach to fill epochs with as many jobs as possible, starting with the jobs with the lowest minimum memory requirement, while ensuring that the number of jobs in an epoch evenly divides the number of processors. In general, $EQUI\_EPOCH$ does the following:

1. Sort jobs by non-decreasing minimum processor requirement, $min[]$.
2. Fill the first epoch with the largest number of jobs that evenly divide P and for which the last minimum requirement of the jobs in the epoch does not exceed its equitable share in that epoch.

3. Fill subsequent epochs similarly from remaining jobs.

4. Determine the duration of each epoch by the percentage of total jobs executing in that epoch, so that each job is allocated an equal number of processor-seconds within the interval T.

Figure 6.5.1 gives the *EQUI_EPOCH* algorithm.

```
$\boldsymbol{EQUI\_EPOCH}(P, J, min[])$ {
    int $min[0..(J-1)]$;           /* non-decreasing order of minimum processor requirement */
    $start\_index = 0$;             /* remember first job in candidate set */
    while ($start\_index < J$) do {
        for ($j = J - start\_index$; $j > 0$; $j - -$)
            if (($P$ mod $j \equiv 0$) && ($min[start\_index + j] < P/j$))
                break;
        assign_epoch($start\_index, start\_index + j$);
        $start\_index + = j$;
    }
}
```

Figure 6.6: *EQUI_EPOCH Scheduling Algorithm*

For example, consider scheduling eight jobs on 12 processors, where the minimum processor requirements for the jobs are $min[] = \{2, 3, 3, 4, 4, 4, 5, 6\}$. Figure 6.7 illustrates the *EQUI_EPOCH* schedule for this job mix. The numbers in the figure refer to the minimum allowable allocation of the job assigned to that pertion of the schedule. The first three jobs with the lowest minimum processor requirements are assigned to the first epoch. The fourth job can not be added to this epoch, since its minimum processor requirement of four is greater than the three processors that would be allocated to each job in the epoch. Thus, the fourth, fifth and sixth jobs are scheduled on four processors each in the second epoch, and the remaining two jobs are given an allocation of eight processors each in the third epoch.

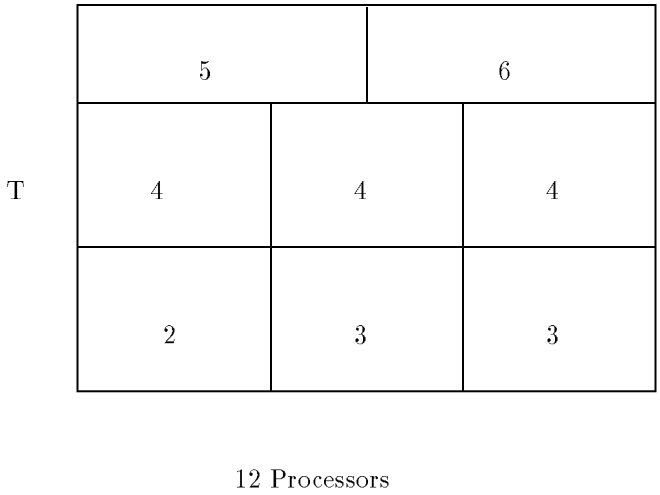


Figure 6.7: *EPOCH Scheduling Policy: Example for* $P = 12$*,* $J = 8$*,* $min[] = \{2,3,3,4,4,4,5,6\}$

6.5.2 Analysis of $EQUI\_EPOCH$

This section presents a proof that $EQUI\_EPOCH$ produces an optimal schedule when P is a power of two.

Theorem 6.5 *$EQUI\_EPOCH$ is an optimal epoch scheduling policy for $P = 2^p$, for some integer $p \geq 0$.*

Proof: Let S be an optimal epoch schedule for a set of J jobs with minimum processor requirement $min[]$. Consider the list of jobs sorted by non-decreasing minimum processor requirement where $i < j$ implies job i appears before jobs j in this list. Define the size of an epoch be the number of jobs scheduled in that epoch. Define $ORD_S(i)$ as the i^{th} allocation in S, where allocations are ordered by increasing job index within decreasing epoch size. Let a_i be the number of processors allocated to job i. Without loss of generality, assume S has the following properties:

Property 6.3 *For all job pairs i and j, if $i < j$, $a_i <= a_j$.*

Property 6.4 *For all job pairs i and j, if $i < j$, $ORD_S(i) < ORD_S(j)$.*

Any schedule can be changed to a schedule meeting these properties and having the same number of epochs by simply exchanging the allocations of any pairs not meeting this criteria.

Both S and $EQUI\_EPOCH$ produce schedules where jobs appear in the same order within epochs of decreasing size. It remains to be shown that S and $EQUI\_EPOCH$ have the same number of epochs; therefore, $EQUI\_EPOCH$ is optimal.

Let S_1,S_2...S_s be the sizes of the epochs of S, where $S_x >= S_y$ if $x < y$. Let E_1,E_2,...E_e be the sizes of the epochs produced by $EQUI\_EPOCH$ where $E_x >= E_y$ if $x < y$. Since P is a power of two, the number of jobs in each epoch of both schedule S and the $EQUI\_EPOCH$ schedule must also be a power of two. Thus, E_1,E_2...E_e, and S_1,S_2,...S_n are powers of two.

Consider the minimum x where $S_x \neq E_x$. $E_x < S_x$ is not possible, since the size of the epochs prior to epoch x and the set of jobs in these epochs are the same under the two schedules, and by definition, the $EQUI\_EPOCH$ algorithm packs as many jobs into an epoch as possible.

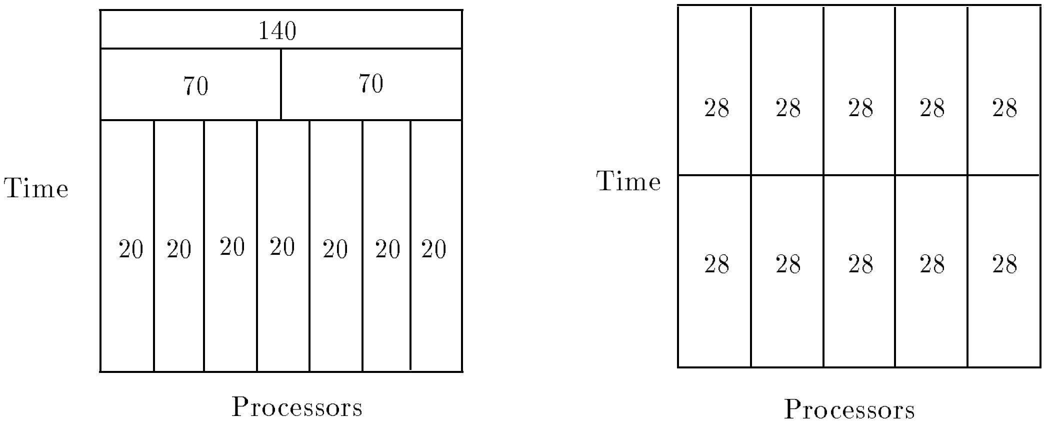
If $E_x > S_x$, it would have to be at least twice the size of S_x since all epochs have a power of two number of jobs. So E_x would have to contain at least the contents of S_{x+1} in addition to S_x. In general, E_x would have to encompass S_x,...S_{x+i} for some i. (Either all of an epoch of S is encompassed by E_x or none of the epoch is included in E_x.) This means that $EQUI\_EPOCH$ would take fewer epochs to schedule the jobs in E_1 up to E_x than the optimal solution S. Similarly, for the remaining jobs not scheduled in epochs E_1 to E_x of $EQUI\_EPOCH$ (or epochs S_1 to S_{x+i} of S), the number of epochs created by $EQUI\_EPOCH$ will be less than or equal to the number of epochs created by an optimal schedule S. Thus $EQUI\_EPOCH$ is an optimal epoch scheduling policy for equal allocation.

QED

To help clarify the key properties on which the optimality of this greedy algorithm rests, we present two similar scenarios in which a greedy algorithm is non-optimal.

$EQUI\_EPOCH$ not optimal when P not a power of two

If P is not restricted to a power of two, the $EQUI\_EPOCH$ scheduling policy is not optimal. To illustrate this, consider scheduling $P = 140$ processors. Assume there are 10 jobs, each can run with no fewer than 20 processors each.



(a) EQUI_EPOCH schedule

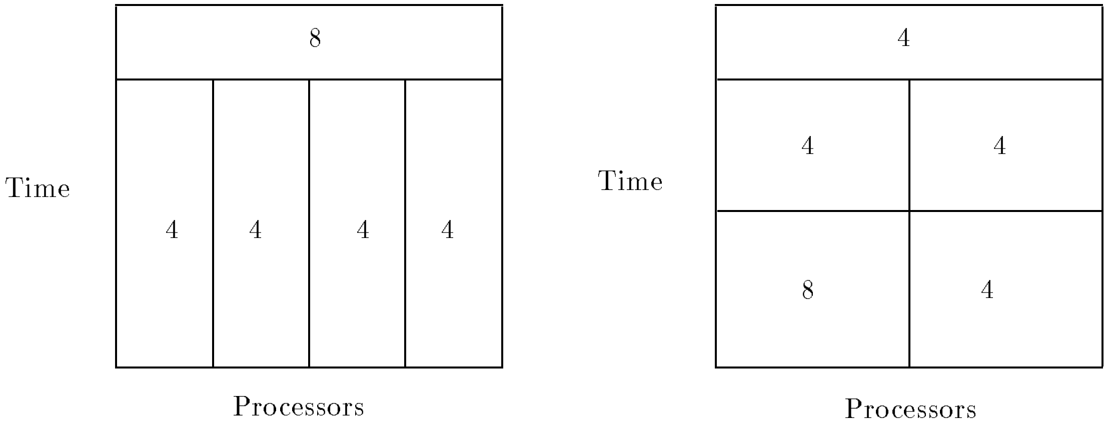
(b) Optimal non-greedy schedule

Figure 6.8: *$EQUI\_EPOCH$ verses optimal epoch policy when P is not a power of two*

As illustrated in Figure 6.8(a), $EQUI\_EPOCH$ schedules 7 jobs with 20 processors each in the first epoch, 2 jobs with 70 processors each in the second epoch, and 1 job with 140 processors in the third epoch. (3 jobs can not be coscheduled on 140 processors equally.) An better solution could schedule the jobs in two epochs, with 5 jobs having 28 processors in each epoch (Figure 6.8(b).

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 processors requirement instead of the smallest minimum processor requirement. However, such a policy is not optimal. To show this, consider scheduling 5 jobs whose minimum processor requirements are $min = [4, 4, 4, 4, 8]$ on $P = 16$ processors. $EQUI\_EPOCH$ would schedule the jobs in two epochs as shown in Figure 6.9(a), while the other greedy schedule would be forced to schedule the jobs in 3 epochs (Figure 6.9(b)).



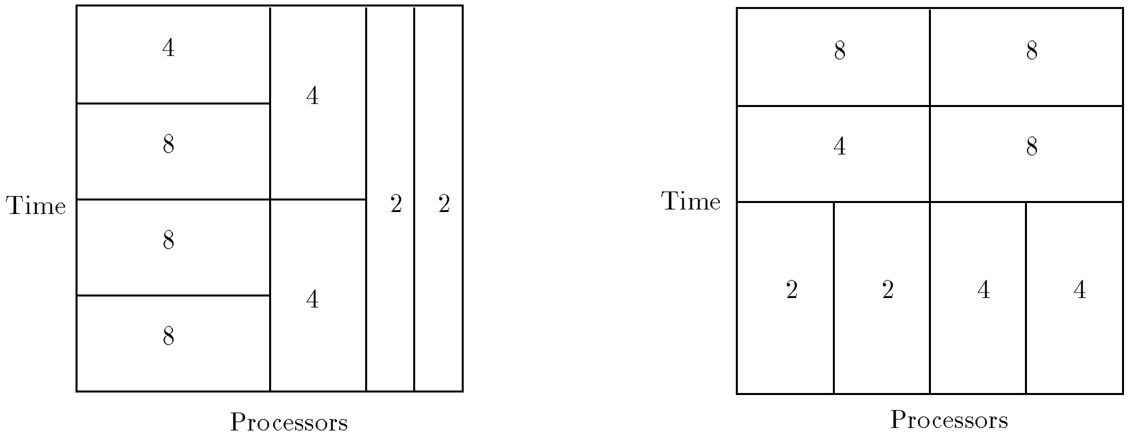
(a) EQUI_EPOCH schedule

(b) Non-optimal greedy schedule

Figure 6.9: *$EQUI\_EPOCH$ verses non-optimal greedy epoch policy:* $P = 16$*,* $J = 5$*,* $min = [4, 4, 4, 4, 8]$

6.6 Comparison of $BUDDY$ to $EQUI\_EPOCH$

Although $EQUI\_EPOCH$ is an optimal epoch scheduling policies within its class of scheduling policies, the non-epoch scheduling policy $BUDDY$ can produce schedules with fewer reallocations, when it is applicable (i.e., when J as well as P is a power of two). Figure 6.10 gives an example for $P = 16$ processors and $J = 8$ jobs, where the minimum processor requirement of the jobs is $min_n = \{2, 2, 4, 4, 4, 8, 8, 8\}$. The $EQUI\_EPOCH$ policy requires three epochs to schedule the jobs. The job's with minimum requirement of 2 are forced to run on four processors. The $BUDDY$ schedule instead is able to schedule those jobs on two processors, so that they never have to incur the overhead of reallocation. Therefore, it's total reallocation overhead is less.



$BUDDY$ Schedule

$EQUI\_EPOCH$ Schedule

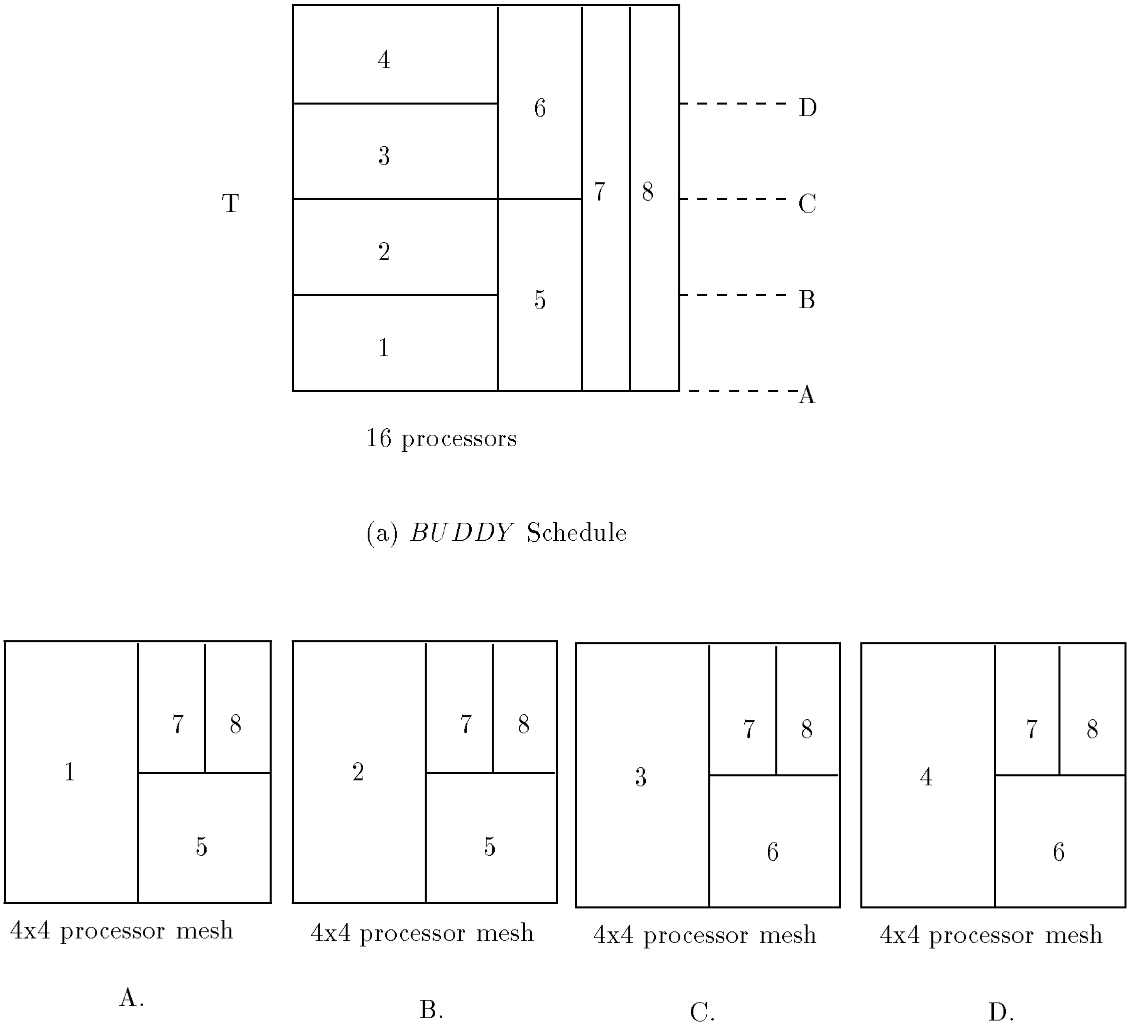
Figure 6.10: $BUDDY$ *vs.* $EQUI\_EPOCH$*:* $P = 16$*,* $J = 8$*,* $min = [2, 2, 4, 4, 4, 8, 8, 8]$

6.7 Integration With Space-sharing Policies

As discussed in Section 6.1, scheduling in distributed memory parallel systems is defined in two parts, corresponding to two levels of kernel scheduling. This chapter focused on the first level, medium-term scheduling that determines the of the available jobs to run at a given time when not all jobs can be scheduled concurrently. Both of the medium-term scheduling policies presented in this chapter must interact with a second-level, short-term policy that determines how to schedule a set of jobs on the processors in order to realize the allocations specified by the time-sharing policy.

For the $EQUI\_EPOCH$ scheduling policy, the Equipartition policy could be used to schedule the jobs assigned to each epoch. This involves a simple partitioning, since the number of jobs scheduled in each epoch evenly divides the number of processors. For the two-dimensional mesh topology, this results in no load imbalance.

When P is not a power of two, the $EQUI\_EPOCH$ still integrates well with equipartitioning policies for a two-dimensional mesh of processors. The dimensions of the mesh must be factors of P. Since the number of jobs in any epoch are always an even divisor of P, partitions can be defined by appropriate factoring along the dimensions. Thus, the equipartitioning results in no system-induced load imbalance for the jobs executing within the epoch.



(a) *BUDDY* Schedule

(b)Realization of *BUDDY* schedule on 4x4 mesh of processors

Figure 6.11: *Implementation of BUDDY schedule on 4x4 processor mesh*

The *BUDDY* policy may be implemented by a simple variant of the Folding policy.

Each allocation consists of a power of two number of processors, which Folding maps onto a two-dimensional mesh of processors with a power of two number of processors in each dimension. Any allocation of a $BUDDY$ schedule rests on only one previous allocation, and there are a power of two number of allocations resting on any allocation. Therefore, each reallocation consists of dividing an allocation into a power of two smaller allocations, a straightforward mapping when the dimensions of the larger allocation are powers of two. Thus, only local changes are needed to reallocate the processors. No Folding rotation policy is needed, since the reallocations necessary to achieve equal resource allocation are determined by the $BUDDY$ schedule. This is consonant with the results of Chapter 5, which show that both good fairness and good performance can be achieved without rotation under Folding.

As an example, Figure 6.11(a) illustrates a $BUDDY$ schedule consisting of 8 jobs, whose minimum processor requirements are $min = [2, 2, 4, 4, 8, 8, 8, 8]$. Figure 6.11(b) illustrates a folding-style partitioning of a 4x4 grid of processors to accommodate the $BUDDY$ schedule. The dashed lines in Figure 6.11(b) denotes the time at which the reallocations occur.

6.8 Summary

This chapter examined medium-term scheduling policies for applications with large memory requirements, resulting in a minimum number of processors required to execute these applications. We defined this scheduling problem as a time-sharing scheduling problem in which all jobs are scheduled at some time within a scheduling interval. A scheduling policy determines which jobs to schedule concurrently within some quantum interval, so that each job receives at least its minimum processor requirement, and each job receives the same total number of processor-seconds at the end of the interval. We define an optimal policy as one that minimizes the total number of reallocations within the scheduling interval.

We showed that this scheduling problem is a subset of a similar problem known to be NP-Complete, and argued that although the complexity of this scheduling problem is open, it is unlikely efficient solutions exist for all inputs. As a result, we present scheduling policies that produce optimal schedules for a restricted class of inputs. One policy, called $BUDDY$, produces an optimal schedule for a power of two number of processors, a power of two number of jobs, and where all allocations are powers of two.

We also consider a restricted class of scheduling policies, called epoch scheduling policies, where all processors are reallocated at the same time, at epoch boundaries, and within each epoch, no reallocations occur. We defined a policy, called $EQUI\_EPOCH$, for this class of epoch schedulers and proved it is optimal when the number of processors is a power of two.

Finally, this chapter discussed the interaction between the medium-term, time sharing policies and the short-term, space sharing policies defined in Chapter 4.

Chapter 7

Conclusions and Future Research Directions

This dissertation investigates kernel processor allocation policies for multiprogramming large scale, message passing parallel computers. Specifically, it examines aggressive approaches to scheduling these systems, using dynamic reallocation of processors among running jobs in order to attain the best overall job performance and system utilization.

This chapter summarizes the contributions of this dissertation and proposes some interesting areas for future research in scheduling on parallel systems.

7.1 Conclusions

This dissertation examines the design factors in multiprogramming message-passing parallel systems, with a special emphasis on issues particular to dynamic allocation. First, we assumed that to achieve good job response time in the absence of knowledge of application run-time behavior, the scheduling policy must provide equal resource allocation.

Second, the design and performance of dynamic allocation policies must consider the overhead of reallocation and the subsequent effect on application performance. Application performance is affected by the configuration of the partition allocated to the job and the distribution of the job's threads across that allocation. A mismatch between the application's structure and the physical partition, or a poor mapping of an application's threads across a partition, could result in an computational imbalance, thus adversely affecting job performance. Policies must also consider the proximity of processors allocated to a job. Furthermore, a policy must consider the manner in which a job's threads are redistributed when its allocation changes. We discuss a number of alternatives to providing this mapping and propose a kernel-level mapping policy that specifies the distribution of a job's threads on varying sized partitions.

This dissertation introduces a new metric, called efficiency preservation, to evaluate the performance of processor allocation policies for parallel systems. This metric measures the effects of a scheduling policy on processor efficiency. We show how efficiency preservation can be used in a first-order evaluation to identify promising scheduling policies.

We presented two families of dynamic allocation policies for mesh-connected machines that differ in the ways they address the costs of dynamic allocation. One policy, called Equipartition, partitions the processors only when the number of jobs in the system changes, either because of a job completion or arrival. Processors are partitioned equally among the competing jobs. Since the number of processors allocated to a job may not divide the number of threads it contains, a system-induced load imbalance can degrade job performance. The other policy, called Folding, avoids system-induced load imbalance by always halving or doubling the number of processors allocated to a job, but it must incur additional reallocation cost to ensure equal resource allocation.

The performance of these policies is compared using both modelling and simulation to ascertain the relative importance of reallocation cost and job load balancing effects of the scheduling policies on job performance. Achieving good load balancing is shown to be a dominant factor in the performance of policies for distributed memory systems. Overall, the results show that with careful attention to processor adjacency and load balancing considerations, dynamic allocation offers the potential for high performance in distributed memory parallel systems.

Lastly, medium-term scheduling policies are examined to accommodate the situation where the combined memory requirements of the submitted jobs exceeds the system's capacity. This scheduling problem is shown to be a variant to the class of two-dimensional orthogonal bin-packing problems, and we argue that it is unlikely to be efficiently solvable. We investigate two scheduling policies that address a subset of the problem space. One approach produces an optimal schedule when the number of processors, the number of jobs, and all allocations are powers of two. The second approach creates schedules that consist of epochs, a scheduling discipline in which all processors are reallocated at once to a (possibly) new set of jobs. We define a policy that is optimal in the class of epoch scheduling policies when the number of processors is a power of two. We also discuss the relationship between this level of scheduling and the short-term, space sharing scheduling policies examined earlier.

7.2 Future Research Directions

7.2.1 Other Workload Characteristics

The dynamic allocation scheduling policies designed and analyzed in Chapters 4 and 5 considered only performance aspects of scheduling policies. However, there are other non-performance aspects that have an impact on the suitability of scheduling policies for large parallel systems. For example, many systems execute interactive jobs or jobs with little parallelism in addition to large parallel applications. It might be best to reserve a portion of the machine for this workload, and to schedule those jobs in a manner similar to traditional time-sharing systems. The Folding family of policies does not lend itself well to this partitioning, since it requires a power of two number of processors in each dimension of the mesh of processors, which would be unlikely if a small number of processors where reserved for sequential work. However, the Equipartition policy could easily accommodate a separate partition while maintaining a power of two number of processors in one dimension of the processor mesh. These non-performance aspects must be considered in a comprehensive evaluation of scheduling policies.

7.2.2 Other Architectures

The results presented in this dissertation are specific to a two-dimensional processor mesh topology. For other processor topologies, such as tree-like structures or hypercube topologies, a partitioning policy needs to be defined that simultaneously maximizes the adjacency of processors within a partition allocated to a job and that reduces the load balancing of an application distributed on the partitions. Achieving this partitioning is not trivial in the face of dynamic allocation. Furthermore, this partitioning should be realizable for all multiprogramming levels.

This dissertation focused on message-passing parallel systems. The basic ideas and some of the results presented here can also be extended to distributed memory systems with support for shared-memory; however, in shared memory systems, which have support for remote memory reference, there is the additional issue of whether to migrate a job's data upon reallocation of its processor, or allow it to access the data remotely.

7.2.3 Approximation Algorithms for Medium-term Scheduling

In Chapter 6, we proposed and analyzed medium-term scheduling policies that were optimal for a restricted class of schedulers. An area of future research investigates approximation algorithms for medium-term scheduling that provide schedules with low (but not lowest) overhead of reallocation. One approach currently being pursued considers a new class of epoch policies in which some inequity is tolerated in the allocations to the jobs. This policy class allows the number of processors allocated to jobs within an epoch to differ by some small amount. Allowing this inequity permits a reduction in the number of epochs needed to schedule the jobs, in many, if not most cases.

7.2.4 Interaction Between Kernel-level and Application-level Scheduling

This dissertation is concerned with kernel-level scheduling, and has assumed that there is minimal interaction between the kernel-level and the application-level schedulers. However, the design and performance of kernel-level scheduling policies can be improved by coordinated interaction with application-level scheduling, especially for aggressive scheduling policies that dynamically change the allocation of processors to jobs. Coordination between the two levels of scheduling may reduce the cost of dynamic reallocation and the effects of dynamic allocation.

We identify two kernel support mechanisms that could facilitate dynamic reallocation. The first mechanism concerns kernel support for faster communication or synchronization between two kernel threads of the same application residing on the same processor. Message-passing between co-resident threads could be replaced by local more efficient message-passing within the same processor in a manner similar to the lightweight remote procedure call (LRPC) mechanism proposed by Bershad et al. for intraprocessor communication on multiprocessors [5]. For this environment, the mechanism used to achieve the communication between application threads must be changed dynamically as the threads become co-located or become reassigned to separate processors.

A more aggressive operating system support mechanism for dynamic reallocation could involve kernel or runtime system support for combining multiple kernel threads executing on the same processor into a single kernel thread. In the absence of kernel support, an application could reduce the number of kernel threads by reassigning application-level threads and data from one kernel thread to another and then deleting the vacated kernel-level thread. Such contraction could reduce memory needs by resolving duplicate copies of shared data and eliminating the overhead of maintaining two address spaces. In addition, communication between application threads that previously executed within separate kernel threads could be implemented via shared memory references. Ashok [3] examines runtime support mechanisms for adjusting an application's parallelism to changes in its allocation. However, it is unclear to what extent the kernel could facilitate the collapsing or expansion of kernel threads. Such mechanisms might be implemented more easily in shared memory systems or systems supporting a single address space [6].

Bibliography

[1] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. *Proceedings of Supercomputing '91*, November 1991.

[2] T. Anderson, B. Bershad, E. Lazowska, and H.M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

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

[4] B. Baker, E. Coffman, and R. Rivest. *Orthogonal packings in two dimensions*, SIAM Journal of Computing, 9(4),846–855, November 1980.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[6] J. Chase, H. Levy, M. Feeley, and E. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12(4), November 1994. To appear.

[7] G.-I. Chen and T.-H. Lai. Scheduling independent jobs on partitionable hypercubes. *Journal of Parallel and Distributed Computing*, 12:74–78, 1991.

[8] 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.

[9] M.-S. Chen and K.G. Shin. Subcube allocation and task migration in hypercube multiprocessors. *IEEE Transactions on Computers*, C-39(9):1146–1153, September 1990.

[10] S.-H. Chiang, R. Mansharamani, and M. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of ACM SIGMETRICS Conference*, pages 33–44, May 1994.

[11] E. Coffman, M. Garey, D. Johnson, and R. Tarjan. *Performance bounds for level-oriented two-dimensional packing algorithms*, SIAM Journal of Computing, 9(4),808–826, November 1980.

[12] M. Crovella, P. Das, C. Dubnicki, T. Leblanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings 3rd IEEE Symposium on Parallel and Distributed Processors*, Dallas, December 1991, pages 590–597.

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

[14] L. Dowdy. On the partitioning of multiprocessor systems. Technical Report, Vanderbilt University, Nashville, TN, July 1988.

[15] 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.

[16] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, C-38(3):408–423, March 1989.

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

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

[19] M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H.Freeman and Company, New York, 1979.

[20] 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.

[21] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):377–383, July 1991.

[22] L. Kale'. The chare kernel parallel programming language and system. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 17-42.

[23] C. Koelbel and P. Mehrotra. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177-186, March 1990.

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

[25] M. Leuze, L. Dowdy, and K. Park. Multiprogramming a distributed memory multiprocessor. *Concurrency: Practice and Experience*, September 1989.

[26] K. Li and K.-H. Cheng. Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):413–422, October 1991.

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

[28] 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.

[29] V. Naik, S. Setia, and M. Squillante. Scheduling of large scientific applications on distributed memory multiprocessor systems. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computation*, pages 913–922, March 1993.

[30] V. Naik, S. Setia, and M. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of Supercomputing '93*, pages 824–833, November, 1993.

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

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

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

[34] M. Rosing, R. Schnabel, R. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[35] E. Rosti, E. Smirni, L. Dowdy, G. Serazzi, B. Carlson. Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation*, To appear.

[36] S. Setia. *Scheduling on Multiprogrammed, Distributed Mmoery Parallel Computers.* PhD thesis, The University of Maryland, UMIACS-TR-93-115, October 1993.

[37] 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.

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

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

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

[41] 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.

[42] 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.

[43] J. Turek, J. Wolf, K. Pattipai, and P. Yu. Scheduling parallelizable tasks: putting it all on a shelf. In *Performance Evaluation Review*, 20(1),158–170, June 1992.

[44] 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.

[45] 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.

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

[47] J. Zahorjan, E. Lazowska, and D. Eager. The effects of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, (2)2:180–189, April 1991.

[48] J. Zahorjan, E. Lazowska, and D. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Sympossium on Performance of Distributed and Parallel Systems*, (Kyoto, Japan, Dec. 1988).

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

[50] 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.