USING KNOWLEDGE OF JOB CHARACTERISTICS IN MULTIPROGRAMMED MULTIPROCESSOR SCHEDULING

by

Eric W. Parsons

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

USING KNOWLEDGE OF JOB CHARACTERISTICS
IN MULTIPROGRAMMED MULTIPROCESSOR SCHEDULING

Eric W. Parsons
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1997

Abstract

Multiprocessors are being used increasingly to support workloads in which some or all of the jobs are parallel. For these systems, new scheduling algorithms are required to allocate resources in such a way as to offer good response times while being capable of sustaining high system loads. Until recently, research in this area has focussed on processors as the critical resource, but for many parallel workloads, memory will likely also become a concern. In this thesis, we investigate the design of parallel-job scheduling disciplines, considering simultaneously processors and memory as critical resources.

First, we demonstrate that preemption is a necessary feature of parallel-job schedulers in order to obtain good response times given the types of workloads found in practice. Next, we develop analytic bounds on the achievable system throughput with respect to both processing and memory for the case where no knowledge exists about the speedup characteristics of individual jobs. Through the derivation of these bounds, we show that an equi-allocation scheduling discipline, one which allocates processors evenly among jobs selected to run, is the best approach. The key factor to obtaining good performance for such disciplines is to make effective use of memory.

If the scheduler possesses speedup knowledge of jobs, however, then the equi-allocation strategy is no longer recommended for workloads in which there exists a correlation between the memory requirements of jobs and their speedup characteristics. In this case, it is theoretically possible to achieve an arbitrary increase in the sustainable throughput over equi-allocation by using this speedup information in allocating processors.

Finally, we present the implementation of a family of scheduling disciplines, based on Platform Computing's Load Sharing Facility. Each of these disciplines makes different assumptions about the characteristics of the system, such as the type of preemption that is available or the flexibility that the system possesses in allocating processors. Through this work, we demonstrate, in a practical setting, that sophisticated parallel-jobs scheduling disciplines can be successfully implemented and can deliver improved performance relative to disciplines typically being used today.

Acknowledgements

I would first like to thank Ken Sevcik for his excellent guidance throughout the course of my studies. I enjoyed the positive relationship that we developed over the past three years, from which I gained much insight about performance analysis of systems. I also felt that the door was always open to discuss a new idea or ponder fresh results, for which I am very grateful.

I would like to thank Michael Stumm for his valuable advice and unique perspectives on systems research. Although my research did not ultimately involve the Tornado operating system, I appreciated always being included in the group and valued the numerous discussions we had on this topic. I would also like to thank all my other committee members, Tarek Abdelrahman, Marsha Chechik, and Songnian Zhou, and my external examiner, Mary Vernon, for their constructive criticism that helped improve the thesis.

Pursuing a PhD is in many ways an experience shared with fellow students. As such, I have greatly enjoyed working alongside everyone in the systems lab, particularly Karen, Paul, Ben, Orran, and Daniel (roughly in order of distance from my desk). I wish them all the very best in their future careers, and hope that our paths will cross again.

My studies would never have reached this point had it not been for the timeless support from my family. My parents have instilled in me a great sense of passion and excellence in all my activities. But the person who was closest to me during this time was my wife, Jo-Ann, who contributed to this degree in countless ways.

My appreciation also goes to Bell-Northern Research for having sponsored my degree. I would like to particularly thank Elaine Bushnik and Peter Cashin for their efforts in this regard. I now look forward to returning to Nortel to apply some of the skills I have developed.

Finally, I would like to dedicate this thesis to Valerie, who was never given the opportunity to fulfill her dreams, but whose spirit in life was an inspiration to us all.

Contents

Chapter 1

Introduction

As large-scale multiprocessor systems become available to a large and growing user population, mechanisms and policies to share such systems become increasingly necessary. Users of these systems run applications that vary from computationally-intensive scientific modeling to I/O-intensive databases, for the purpose of obtaining computational results, measuring application performance, or simply debugging new parallel codes. While in the past, such systems may have been acquired exclusively for a small number of individuals, they are now being installed so as to be available to a large number of users, each submitting jobs having very different characteristics.

Scheduling is the activity of controlling the execution of the jobs submitted to run on a system. The goal of the scheduler is to make effective use of the physical resources of the system, such as processors, memory, and I/O devices, while satisfying the performance expectations of users. In practice, many factors complicate the decisions that must be made by a scheduler. For instance, scheduling operations, such as migration or checkpointing, have certain costs that must be taken into account; some users may have special requirements, such as guaranteed resource allocations, in order to conduct performance experiments; and different classes of jobs may have different scheduling requirements that may be inconsistent with each other.

The simplest type of scheduling used for large-scale systems is where users "purchase" or "reserve" dedicated portions of the system for certain periods of time. This approach offers users exclusive access to the system, which can be attractive for controlled experiments, but it is inconvenient and unworkable for large user communities and would likely lead to poor system utilization. To serve as a general-purpose resource, any modern system needs to provide much greater flexibility, allowing users to run jobs with as little impediment as possible.

The first multiprogrammed (i.e., multi-user) multiprocessor schedulers were based on their uniprocessor counterparts. Essentially, a parallel program is broken down into a set of cooperating threads, which are executed independently by the different processors of the system. A processor may dispatch threads either from a global pool (in small-scale systems) or a local pool (in large-scale ones). This *thread-oriented* type of dispatching can lead to very poor performance for parallel-job workloads, particularly when the threads of a job communicate (or synchronize) frequently amongst themselves. The reason is that the threads of a job are unlikely to ever all be running concurrently, causing them to block for extended periods of time while waiting for other threads to be scheduled. On large-scale systems, the context switches caused by the blocking can overwhelm the system.

In practice, many jobs tend to be computationally-intensive in nature. Moreover, these types of jobs are the ones most likely to require frequent synchronization and thus suffer from thread-oriented dispatching. A simple and effective alternative is to run each job on a dedicated set of processors (i.e., without any interference from threads of other jobs). In this type of *job-oriented* dispatching, each thread of a job is normally associated with its own processor (termed *coordinated* or *gang* scheduling [Ous82, FR92]). It is sometimes possible, however, to multiplex threads of a job on a reduced number of processors and to still achieve good performance, as long as only threads from a single job are assigned to any given host [MZ94].

A simple job-oriented scheduling strategy is to divide the system into fixed-sized partitions, each one associated with a queue to which jobs are submitted [RSD$^+$94]. Jobs in each queue are executed, in first-come first-served order, using all the processors associated with the queue. To accommodate different partition sizes than the ones configured, the set of active queues can be varied at different times of the day or week (e.g., a queue corresponding to all processors in the system can be made active from midnight until morning). Although such an approach is better than reservation-based systems or thread-oriented schedulers, it still possesses numerous obvious problems: (1) resources can be wasted if jobs do not require all processors within the queue's partition, (2) there can be unevenness in response times between jobs that are directed to lightly-loaded queues versus those that are sent to heavily-loaded queues, (3) the tendency to use first-come first-served strategies within a queue can lead to high response times given the workloads typically found in practice, (4) the user is often required to choose from up to thirty queues, each corresponding to a different combination of expected execution time, processor allocation, and memory demand, and (5) jobs may have to be terminated when the set of active queues (and hence the partitions) need to be changed, unless jobs can be preempted.

A more recent approach to scheduling parallel jobs is to allow the system to be dynamically partitioned to best satisfy the needs of jobs that have been submitted. Commonly, the user specifies the number of processors required for each job, and the system runs the job when enough processors are available (and the job is at the head of the queue). Although dynamic partitioning offers greater flexible than the previous approach, a different set of problems arise. First, when a job requiring many processors is submitted, the system must choose between waiting for a sufficient number of processors to become idle (which may result in starvation) or holding-off allocating processors to new jobs until the large job can be started (which may result in wasted resources). Second, processing resources can be wasted if sum of the processor requirements of the running jobs is less than the total number of processors. (We refer to this problem as *packing loss*.).

Up to now, commercial scheduling systems have failed to take into account a very important principle of parallel computing: in general, a job makes less efficient use of processing resources as its processor allocation increases (in the absence of memory effects) [Amd67, EZL89, GGK93]. The primary reason is that the fraction of time spent in the efficient, parallel phases of the computation decreases as the processor allocation increases, relative to the fraction of time spent in the inefficient sequential phases. This notion of *efficiency* is closely related to the *speedup* exhibited by the application on a given number of processors.[1] Thus, as the load increases, it is important to decrease average processor allocations, thereby making more efficient use of the processing resources of the system and allowing a higher load to be served [Sev89]. If a user provides a range of processor allocations that is acceptable for the job, the system can then allocate the job more processors in times of light load, and fewer processors as the load increases. As a side effect, it has been found that flexibility in processor allocations can greatly reduce packing losses, as leftover processors can be assigned to a newly-started job rather than being left idle.

An important facet of having the system choose allocation sizes is the way in which these sizes are determined. Given full knowledge of both the speedup characteristics and the service demand of each job, it is theoretically possible to determine the optimal processor allocation sizes with respect to any performance criteria, but doing so is typically extremely expensive. This, combined with the fact that obtaining perfect knowledge is difficult, has led to several efficient heuristics being proposed that make use of whatever limited or approximate information is available [GST91, CMV94]. Most recently, other resource requirements of parallel jobs have been taken into account [MZ95, Set95, PS96b, PS96a, ML94]. In particular, it has been found that the memory demands of jobs, which are often non-trivial in scientific computations, can greatly influence the scheduling decision if used in conjunction with speedup information. This issue is one of the major topics investigated in this thesis.

Recent observation of high-performance computing workloads has shown that jobs have highly variable service demands; in other words, many jobs have very short execution times, and a few jobs have very long execution times. One measure of system performance that users find important to minimize is the *mean response time*, which is the average time between job submission and job completion. The key to obtaining a good mean response time is to always run first the jobs that can complete soonest. As will be shown in this thesis, allowing jobs to be preempted is crucial to obtaining good response times given a high variability in service demands, especially if the demands of individual jobs are not known *a priori*.

Developing a parallel-job scheduling algorithm is clearly complex. Any such algorithm must choose both when to run a job and the set of processors to use for the job, taking into account:

- the overall load on the system, to determine an appropriate average processor allocation such that the system can sustain the given load,
- the speedup characteristics, to determine how processors should be allocated among different jobs,
- the memory, communication, and I/O requirements of jobs, to make most effective use of these

[1] Both these terms will be defined formally in Chapter 2.

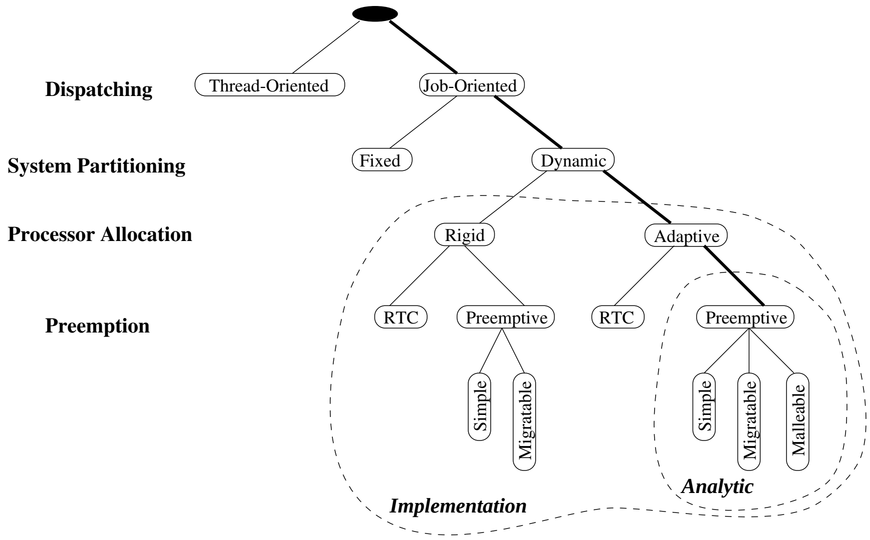


Figure 1.1: Classification of parallel-job scheduling disciplines

resources while ensuring that they are not overcommitted,

- the flexibility of the system in preempting, migrating, and varying processor allocations for jobs, as well as the costs of these actions, and
- the scheduling objective chosen by the organization.

1.1 Parallel-Job Scheduling Roadmap

The types of parallel-job schedulers that are considered in this thesis can be organized as shown in Figure 1.1. The first distinguishing feature is the type of dispatching that is used, namely thread-oriented or job-oriented. Concentrating on the latter, we can then consider the way in which the processors of the system are partitioned. In the *fixed* case, the partitions are defined by the system administrators, perhaps using a queue-based approach as described in the previous section. In the *dynamic* case, the processors form a global pool from which appropriately-sized partitions can be created. Given that there are rarely any hardware restrictions that prevent dynamic partitioning from being used and that it is more general than fixed partitioning, all disciplines considered in this thesis are dynamic in nature.

The next distinguishing feature is whether the user chooses processor allocations (termed *rigid* allocation) or the system does (termed *adaptive* allocation). Although rigid scheduling disciplines are most common today, adaptive disciplines provide significant benefits in terms of being capable of (1) adjusting processor allocations according to the load in the system and what is known about each job, and (2) reducing packing losses that can arise with rigid disciplines.

The last distinguishing feature is whether jobs must be run to completion (RTC) once activated or if they can be preempted to allow other jobs to be executed. If the latter is the case, then schedulers can assume increasing levels of flexibility (and implementation complexity), namely: (1) jobs must use the same set of processors each time they are activated (*simple* preemption), (2) threads of a job can be migrated to a different set of processors, while preserving the same allocation size (*migratable* preemption), and (3) not only can threads be migrated, but the job's processor allocation can also be changed during its execution (*malleable* preemption) [TG89, GTS91]. (Malleable preemption is only meaningful for adaptive disciplines.)

The class of disciplines studied in this thesis from an analytic perspective (Chapters 3 to 5) is restricted to those in the inner circle of the figure. The implementation portion of the thesis (Chapter 6), however, considers a wider range of disciplines, as illustrated by the outer circle.

Approaches to parallel-job scheduling research have evolved over time. Initially, it was assumed that users would specify processor allocation requirements [Ous82, MEB88]. When it was observed that this could significantly limit the performance of the system, focus shifted to adaptive disciplines, subject to minimum and maximum values provided by the user [Sev89, GST91, Sev94, NSS93, RSD$^+$94]. Minimums typically correspond to constraints due to memory and maximums to the point at which no additional speedup can be obtained. It was around this same time that it was quantitatively shown that parallel jobs would perform significantly (10-100%) worse if all the threads were not scheduled simultaneously [TG89, GTU91, FR92], thus driving research towards the gang-style strategies proposed much earlier by Ousterhout [Ous82].

It was subsequently observed that jobs typically found at high-performance computing centers tended to have highly variable service demands (in a statistical sense), resulting in a high mean response time when jobs were permitted to run until completion [CMV94]. To handle such workloads, preemption was incorporated into scheduling disciplines. Simple strategies appear to be effective at reducing response times, but it can be difficult to avoid having idle processors (due to packing losses) if jobs are not malleable. If jobs are malleable, then the scheduling strategy that is typically used as a baseline is *equipartition* [TG89], as it has often been shown to perform well under a wide variety of workloads [MVZ93]. In the ideal form of this discipline, processors are re-allocated evenly among all jobs available to run whenever a job arrives or departs from the system.

One of the next research steps is to consider the influence of other resource demands of jobs (namely memory and I/O) on the scheduling decision. Memory requirements have been considered by some researchers, while I/O has thus far received only limited attention [ML94].

1.2 Research Overview

In this section, the major contributions of the thesis are outlined. First, the characteristics of the systems and workloads being addressed are described, to place the contributions into context.

1.2.1 System Characteristics

The work described in this thesis is applicable to any multiprocessor system, although some benefits will vary depending on the size of the system and its architecture. In particular, larger-scale systems will exhibit greater improvements in throughput by using the disciplines described in this thesis, because in smaller systems, processor allocations to jobs will typically be relatively small and so will tend to lead to efficient use of resources under any scheduling discipline. Also, we distinguish between distributed-memory architectures and shared-memory architectures. In the latter, the allocation of memory can be decoupled from the allocation of processors, which can improve resource utilization and ultimately the sustainable throughput. Most of our proposed disciplines are intended for either class of system.

The experimental platform for the implementation portion of the research is a sixteen-processor network of workstations, having no physically shared memory and a high-latency network. There were two reasons for choosing this platform. First, it was the only parallel system readily available for extensive experimentation at the time when the implementation work was begun. Second, it supports a commercial load-sharing scheduling system upon which we could build our parallel scheduling disciplines. Given that this commercial software is currently available on many platforms, including the IBM SP-2, building our disciplines on top of this software increases the relevance of our implementation.

The small size of the experimental platform is not a significant factor, since we are primarily interested in demonstrating the feasibility of implementing parallel-job schedulers. All our disciplines are designed to support a configurable number of processors. The most significant drawback of this platform is that it is not sufficiently large to study the performance of different scheduling disciplines using large-scale workloads normally found in practice. Despite this limitation, we succeed in illustrating qualitative performance differences among our disciplines using this system.

1.2.2 Workload Characteristics

The types of workloads that are considered in this research are those that are primarily comprised of compute-intensive non-interactive parallel jobs. This allows the scheduler to employ job-oriented dispatching, while still making effective use of processors.

If an actual system were to be used for both interactive and non-interactive jobs, then a dynamic strategy could be used to divide the system into interactive and non-interactive partitions as a function of the load (e.g., to ensure that interactive jobs receive sufficient processing to assure acceptable response times). The interactive partition would typically be used for program development and debugging activities. (Since this partition will not normally employ job-oriented dispatching, however, performance debugging activities will typically need to be performed within the non-interactive partition.) The results presented in this thesis are directly applicable to the non-interactive partition of such an environment.

1.2.3 Overview of Contributions

This thesis examines three major topics.

1. The first topic is the importance of preemption in multiprocessor scheduling. It is shown that if the service demand of jobs is not known *a priori*, then preemption is necessary to obtain good response times, given the variability in service demands typically found in practice. As others have previously shown, equipartition performs very well under these conditions, but equipartition requires malleable jobs [LV90, CMV94]. It is shown how adaptive run-to-completion scheduling disciplines can be easily modified using only migratable preemption to obtain performance comparable to equipartition, namely up to three orders of magnitude improvement in mean response time in some circumstances over the original disciplines.

2. The second major topic is the impact of non-trivial memory demands of parallel applications on parallel-job scheduling. It is shown that an equi-allocation strategy, which allocates processors evenly among jobs selected to run, is nearly optimal if no knowledge of the speedup characteristics of jobs is available. Moreover, determining better processor allocation sizes based only on the memory requirements of each job is computationally expensive, requires precise information about the workload, and does not lead to significant performance improvements.

   As a side result of this analysis, it is also shown that equipartition is provably optimal for maximizing throughput for any given multiprogramming level[2] if no knowledge of the speedup characteristics of jobs is available. This provides a theoretical explanation for why equipartition has frequently been shown to perform well in past simulation studies.

   We then consider the case where complete knowledge of the speedup characteristics of jobs is available. In a theoretical sense, it is shown that the performance benefits of having this information can be arbitrarily great, although in practice the improvements will depend on the workload. We find that the greatest benefits arise when there exists a positive correlation (statistically) between memory requirements and speedup (i.e., large-sized jobs have better speedup than small-sized ones). This is significant because we believe that such correlations commonly exist in practice.

   The basic approach that we have taken in all this work is to first develop bounds on the maximum throughput that can be sustained for a given workload mixture. The practical benefit of developing these bounds is that it permits the performance of our scheduling disciplines to be effectively assessed. In particular, we found that a key factor affecting performance is the degree to which memory is utilized, which can be quantified using the bounds. We also show that the disciplines we propose in this thesis can, in general, sustain loads that are close to the maximums that are possible, as indicated by the bounds.

3. The third major topic is the implementation of job-oriented scheduling disciplines in the context of a real system. In the past, it has been rare for scheduling disciplines to be studied from both an analytic and an implementation perspective. As a result, analytic work is often criti-

[2] The multiprogramming level is the maximum number of jobs that are allowed to run simultaneously.

cized for not taking into account implementation issues. In this thesis, we first study scheduling disciplines analytically, in order to gain insight into the high-level policies that should be used, and then study the implementation of disciplines to demonstrate that they can be practically implemented and can lead to important performance benefits in real systems.

The implementation work is based on Platform Computing's Load Sharing Facility (LSF), using a scheduling extension application-programmer interface (API), allowing the disciplines to be utilized in any environment using a recent release of LSF. Partly based on our experience, Platform Computing is improving the design of the API to allow more efficient implementation of parallel job scheduling extensions.

The thesis is organized as follows:

Chapter 2: This chapter provides a detailed background of parallel-job scheduling. It begins by defining notation that will be used throughout the thesis, and formally defines metrics used to evaluate scheduling disciplines (e.g., sustainable throughput and mean response time). It then explores some of the characteristics of parallel applications found in practice, as they relate to the parameters used to model jobs.

Also, this chapter surveys prior scheduling results, first for sequential workloads and then for parallel workloads. The latter is organized as follows:

- optimality results for throughput and mean response time given full knowledge
- analytic results, structured according to the table presented in the roadmap; the influence of distributed memory on scheduling
- consideration of job memory requirements
- schedulers that have been implemented, both academic and commercial

Chapter 3: This chapter examines the need for preemption in parallel-job scheduling. This work arose from the observation that much of the research prior to 1994 assumed that jobs ran to completion, yet used mean response time as a measure of the performance of the system. It is first observed that the variability of the service-demand distributions typically found in high-performance computing centers can be very high. The work then goes on to demonstrate how previously-defined scheduling disciplines can be adapted for these types of workloads, and evaluates their performance for workloads having highly variable service demands. The basic structure of the chapter is as follows:

- introduction to the problem; evidence that service demands have a high degree of variability
- approach used to adapt existing disciplines, followed by the details of each of the disciplines chosen for evaluation

- evaluation methodology, including system model, workload model, and simulation details
- results of the evaluation

Chapter 4: This chapter examines the effects of workloads in which jobs have non-trivial memory demands, given that the speedup characteristics of individual jobs are not known. It develops bounds on the maximum achievable throughput. Based on the insight gained, three scheduling disciplines are proposed. The throughput bounds are then used to evaluate how these disciplines respond to increasing load. The basic structure of the chapter is similar to the previous one, except that considerable attention is devoted to the development of the bounds.

Chapter 5: This chapter examines the benefits of knowing the speedup characteristics of individual jobs when jobs have non-trivial memory demands. The purpose is to determine the conditions necessary to obtain greater throughput given speedup knowledge relative to the case where no such knowledge exists. After determining where such benefits exist, two scheduling disciplines are proposed and evaluated. The basic structure of the chapter is similar to the previous ones.

Chapter 6: This chapter describes the implementation of several scheduling disciplines in a practical context. As there have been so few implementations of parallel-job schedulers, especially ones that can be ported to numerous platforms, many of the disciplines described in this chapter are not tied to the work presented in previous chapters. Instead, they are intended to cover the range of environments that might be found in practice (as shown in Figure 1.1).

Chapter 7: This chapter summarizes the contributions of the thesis and presents the major conclusions that can be drawn from the work.

1.2.4 Overview of Disciplines

For convenience, the set of disciplines proposed and evaluated in this thesis is listed in Table 1.1, arranged according to the chapter in which each discipline appears. In each case, the basis for the design of the discipline (i.e., derivation) is summarized. Basically, the disciplines proposed in Chapter 3 are derived from combining preemption with previously-defined run-to-completion disciplines, while those proposed in Chapters 4 and 5 arise naturally from the development of bounds on the sustainable throughput from Chapter 4. Since the implementation work in Chapter 6 examines a wide range of disciplines, most do not correspond exactly to any particular discipline in the previous chapters; instead, features of previous disciplines appear where it is permitted by the constraints imposed on the type of scheduler being considered (e.g., if preemption is permitted). All disciplines are examined using either a synthetic workload or, in the case of the implementation work, a synthetic application. The studies that were used in parameterizing the workload are indicated in the last column of the table.

	Name	Derivation	Evaluation Methodology	Basis of Workload
Chap. 3	FB-PWS FB-ASP	Based on corresponding run-to-completion disciplines, PWS and ASP	Simulation	[NAS80] [Wu93] [CMV94]
Chap. 4	MPA-Basic MPA-Repl1 MPA-Pack	Based on the analysis of sustainable throughput bounds, assuming no speedup knowledge; disciplines make increasingly better use of memory to improve performance.	Simulation	[Wu93] [FN95]
Chap. 5	MPA-OCC MPA-EFF	Based on the analysis of sustainable throughput bounds, using speedup knowledge; two variations are considered.	Simulation	[FN95] [Hot96b]
Chap. 6	LSF-*	Based on above results. LSF-MALL-SUBSET is an implementation of MPA-EFF.	Implementation	[FN95] [Hot96b]

Table 1.1: Set of disciplines studied in this thesis.

The disciplines proposed in Chapters 3 and 4 are intended to be used for either distributed- or shared-memory systems. In Chapter 5, we focus on the shared-memory case, but we note that the natural choice of discipline for the distributed-memory case is MPA-EFF. Finally, the disciplines implemented in Chapter 6 are targeted specifically for distributed-memory systems (although they would serve as a good starting point for shared-memory systems).

Chapter 2

Background

This chapter presents the background for parallel-job scheduling that is relevant to the remainder of the thesis. Most of the information describes previously-published work, but the brief discussion of the Cornell Theory Center workload in Section 2.3.2 represents new results that have not been published elsewhere.

2.1 Notation

Let $\mathcal{J}$ denote a finite set of jobs to be scheduled. A job $i \in \mathcal{J}$ arrives in the system at time t_{a_i} and exits the system (after having been scheduled and executed) at time t_{e_i}, so its response time is given by $t_{e_i} - t_{a_i}$.

A job i is characterized by (1) its service demand, denoted by w_i, (2) its execution time on p processors, specified by a function $T_i(w, p)$, and (3) its memory requirement, denoted by m_i. The service demand of a job corresponds to the portion of the computation that is independent of the number of processors allocated to it. The memory requirement of a job is the amount of physical memory needed in order for the function $T_i(w, p)$ to accurately reflect the execution-time characteristics of the job. (Justification for using a value m_i that is independent of the number of processors allocated to the job is given in the chapters investigating the use of this parameter.)

The execution-time function $T_i(w, p)$ of a job i gives rise to two additional well-known, parallel job characteristics. The relative speedup function $S_i(w, p)$ is the ratio of the execution time for the job on a single processor relative to that on p processors:[1]

$$S_i(w, p) = \frac{T_i(w, 1)}{T_i(w, p)}$$

The relative efficiency function $E_i(w, p)$ reflects the efficiency with which a job utilizes processors

[1] It is also possible to consider absolute speedup, which considers the execution time of the best possible sequential implementation rather than that of the parallel job running on a single processor, but this is of lesser interest from a scheduling perspective, unless the user were to actually provide two versions of the application. For the most part, we refer to "relative speedup" as simply "speedup".

that have been allocated to it:

$$E_i(w,p) = \frac{S_i(w,p)}{p}$$

A job that possesses very good speedup characteristics will have a speedup function that is close to unit-linear in p, and an efficiency that is close to one.[2] In general, the efficiency of a job decreases as its processor allocation increases, a property which has significant impact on the design of parallel-job schedulers.

For the most part, we characterize the system as having P functionally equivalent processors and, for shared-memory systems, a total of M units of memory. We usually normalize M to one so that individual memory requirements m_i represent fractions of the total available memory.

Finally, we use the term *job* to refer to a request made by a user to run a given *application* with a given input. As such, an application is never scheduled; only jobs are.

2.2 Performance Metrics

There are several performance metrics that have been used to evaluate the performance of schedulers. The two most popular ones are *makespan* and *mean response time (MRT)*.

The makespan is the time required to complete a predefined set of jobs given that all jobs arrive at the same time (i.e., batch workloads). Formally, if t_{e_i} is the time at which job $i \in \mathcal{J}$ terminates and if all jobs arrive at time zero,

$$\text{makespan} = \max_{i \in \mathcal{J}} t_{e_i}$$

Closely related to the makespan, but more relevant to non-batch workloads, is the *sustainable throughput*. Formally, given an infinite sequence of jobs drawn in specified proportions from some mixture of job classes, the sustainable throughput is the maximum job arrival rate for which the expected response time is finite.

Both the makespan and sustainable throughput are very system-oriented, in that they reflect only the efficiency at which the system is being utilized. Users often have different performance objectives. A popular one is to minimize the mean response time. Formally, the mean response time is the expected value of $t_{e_i} - t_{a_i}$, which for a finite set of jobs $\mathcal{J}$ is:

$$\text{MRT} = \frac{1}{\|\mathcal{J}\|} \sum_{i \in \mathcal{J}} (t_{e_i} - t_{a_i})$$

Clearly, the mean response time is related to the sustainable throughput, in that if the arrival rate exceeds the sustainable throughput, then the mean response time will by definition be infinite. Thus,

[2] Although speedups greater than the number of processors, termed super-linear speedup, are sometimes reported, these nearly always suggest that some physical resource, namely memory or cache, is overloaded in the sequential case and not in the parallel case. In other cases, super-linear speedups occur because the order of computation is both significant in the computation and varies given different processor allocations.

in parallel-job scheduling, a key objective is to increase the sustainable throughput as the load increases.

Another metric that is sometimes considered is the *power*, which is defined as the ratio of the throughput to the mean response time [Kle79]. Kleinrock uses this ratio to determine a "proper" operating point in communication systems where a tradeoff exists between efficiency (utilization) and queueing delays or packet losses. In the context of an open system, as is typically considered in parallel job scheduling, the throughput is identical to the arrival rate (unless the system is saturated), and so examining power is equivalent to examining the mean response time.

In this thesis, we consider both sustainable throughput and mean response time. It has been our experience, however, that it is relatively straightforward to obtain good response times simply by running shorter jobs first (when they can be identified) and, as a result, understanding how the sustainable throughput can be increased is of greater interest.

2.3 Examination of Job Characteristics

2.3.1 Execution-Time Functions

To assess the performance of multiprocessor schedulers, it is necessary to describe or characterize the execution time of arriving jobs as a function of the number of processors allocated to them. Although one could use the exact execution-time function for each application, approximations are used for several reasons. First, getting exact data for a wide enough set of applications and problem sizes is very costly, as it would require that every job submitted in a production environment be re-run on varying processor allocations (consuming scarce computing resources). Second, relying exclusively on such exact data would limit the applicability of the results, especially if the system being studied is larger than the one on which measurements were taken. Finally, any form of analysis typically involves making simplifying assumptions, and in the context of parallel job scheduling, one such assumption is that simple models can be used to characterize the execution time of jobs.

Some of the models that have been used, from a scheduling perspective, are described next.

Parallelism Structure

The most detailed description of an application (next to the source code itself) is a graph depicting all the synchronization points. In a fine-grained application, these points represent data precedence relationships whereas in a coarse-grained application, these represent task precedence relationships. An example of a task precedence graph is given in Figure 2.1(a).

The primary use of the task graph is in the task-mapping problem, which is finding a mapping of the set of tasks to a restricted number of processors in order to minimize the overall execution time [NT93]. Sometimes, other costs are included in the problem, such as the communication overhead [SM94]. (As the task-mapping problem is often referred to as multiprocessor scheduling, it

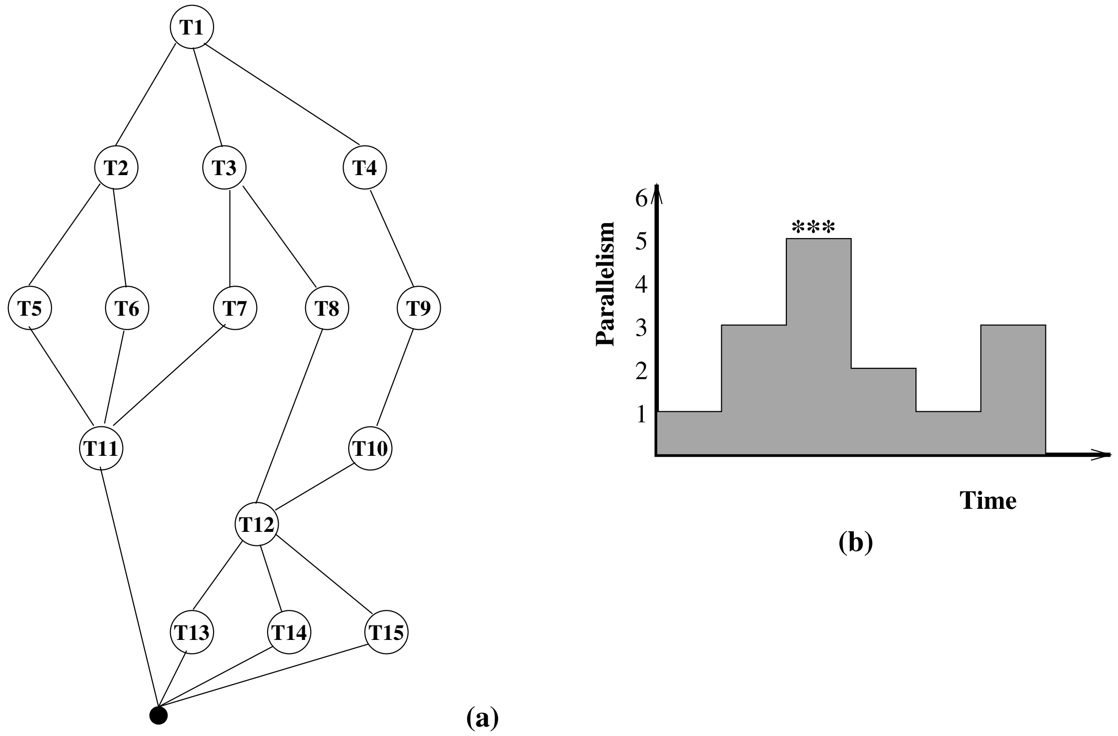


Figure 2.1: Example of (a) a task graph and (b) the corresponding parallelism profile. Each task in the task graph has an associated execution time (unit time for this example), which reveals itself in the parallelism profile.

should be noted that it is quite different from the type of scheduling studied in this thesis, where we are concerned with the scheduling of jobs rather than of the individual tasks within a job.)

Since a large number of applications exhibit very simple "fork-join" behaviour, where "forked" threads execute relatively independently of each other, a common simplification of the task graph is a fork-join (or barrier) graph. The application starts as a single thread, and repeatedly spawns (or releases) a number of threads to perform work in parallel, each time waiting until all threads are ready to join (or synchronize). Because of its regular structure, a fork-join graph is often much easier to analyze and to deal with than arbitrary task graphs.

A simplification of the precedence graph is a *parallelism profile*, which plots the degree of parallelism at each point in time given an unlimited number of processors, as illustrated in Figure 2.1(b). What is lost from the graph are the precedence relationships, so it is no longer possible to determine exactly how the application would behave given fewer processors than its maximum parallelism. In the example, if only 4 processors were available, then it is not clear if the period marked by asterisks ("*") would be extended in length, or if it could simply fill in a gap later on. From the task graph, we know that the latter is the case as task T_8 could be executed at the same time as T_{10} and T_{11} without increasing the overall length of the schedule.

From the parallelism structures, some key characteristics, such as the minimum, maximum, average parallelism, and possibly some higher moments (namely the variance) can be obtained. More interestingly, it is possible to derive theoretical bounds on the achievable speedup of applications. If A is the average parallelism, then the speedup $S(p)$ on p processors can be shown to be bounded as follows:

$$\frac{pA}{p+A-1} \leq S(p) \leq \min(A, p)$$

The upper bound is easy to see. In fact, the average parallelism, which is the area under the parallelism profile curve divided by the execution time, is the same as the speedup for an unlimited number of processors $S(\infty)$ [EZL89]. The lower bound is more difficult to derive, and relies on assumptions that are unrealistic in practice (namely, that overheads are non-existent).

Figure 2.2 illustrates these bounds. The curve $S(p) = p$ is the hardware bound, since it represents the maximum attainable speedup permitted by the hardware. The curve $S(p) = A$ is the software bound, since it represents the maximum attainable speedup permitted by the application. The actual speedup must lie somewhere within the area lying below these two bounds.

Parallelism Signature

In practice, many system-related factors can reduce the performance of the application beneath the theoretical value. A shared-memory parallel application, for instance, can suffer from (1) *hardware congestion*: many processors accessing memory remotely or accessing it in such a way as to cause heavy invalidation traffic, (2) *software contention*: many processors accessing a given lock variable to modify data atomically, and (3) *overhead*: extra software to support parallelism, such as thread creation and lock/barrier preamble (as opposed to delay), or the duplication of computation. These

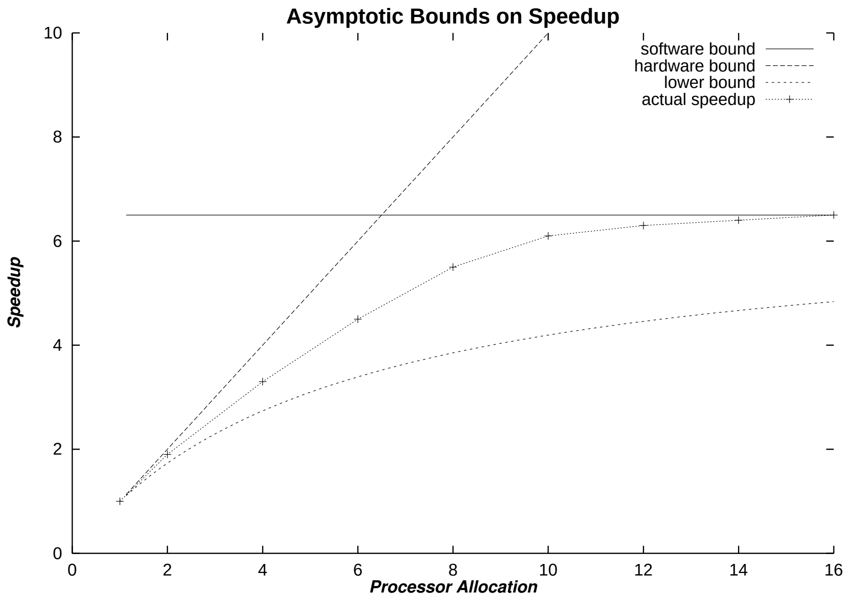


Figure 2.2: Asymptotic bounds on speedup assuming no overhead for communication or synchronization.

types of overheads are not explicitly indicated in the parallelism structure.

The *execution signature* of an application is an equation characterizing (i.e., approximating) its execution time as a function of the number of available processors. It can take into account both an application's parallelism structure as well as the system-related factors (e.g., contention, congestion, overhead) it will experience when running on a certain type of machine. Ideally, each phase of a computation is treated separately, but it is more often the case that a single signature is used for an entire computation.

Dowdy proposes a simple execution-rate function that leads to the execution-time function:

$$T(p) = C_1 + C_2/p$$

where C_1 and C_2 are constants obtained from empirical measurements [Dow90]. In effect, the first parameter represents the amount of work that is sequential, while the second represents the amount that is fully parallel. Dowdy's execution-time function can therefore be re-written in a form similar to Amdahl's Law [Amd67]:

$$T(w, p) = w(s + (1-s)/p) \tag{2.1}$$

where $w = C_1 + C_2$ is the service demand (or work), and $s = C_1/(C_1 + C_2)$ is the fraction of work that is sequential.

Although widely used, this characterization can be unrealistic because it indicates that increasing the number of processors always leads to a reduction in execution time (assuming $s < 1$). In reality, parallelism-related overheads can often negate the gains obtained from increased parallelism, leading to a slowdown after a certain point.

Sevcik proposes the following function that better captures overheads that limit the reduction in execution time [Sev94]:

$$T(w,p) = \frac{\phi(p)w}{p} + \alpha + \beta p \tag{2.2}$$

Conceptually, $\phi(p)$ corresponds to the multiplicative effect of load imbalance when work w is divided among the p processors, α corresponds to the per-processor overhead, and β corresponds to first-order contention effects. If the application has a significant amount of sequential work, then this must be taken into account in one of the existing parameters (in either α or $\phi(p)$). To simplify matters, the value of $\phi(p)$ is assumed to approach a constant value for larger p.

Although ϕ, α, and β can be chosen so that Sevcik's function approximates real execution times better than the Dowdy function, using curve-fitting approaches may cause these parameters to lose their intended meaning [Wu93]. The main benefit of Sevcik's function is that it implies a maximum parallelism—the degree of parallelism after which execution time increases rather than decreases—to be implicitly part of the execution-time function. For the Dowdy execution-time function, one should explicitly use a maximum parallelism value $pmax$ in conjunction with the parameter s, since the function may only be representative for $p < pmax$. In this thesis, both Dowdy's and Sevcik's functions are used.

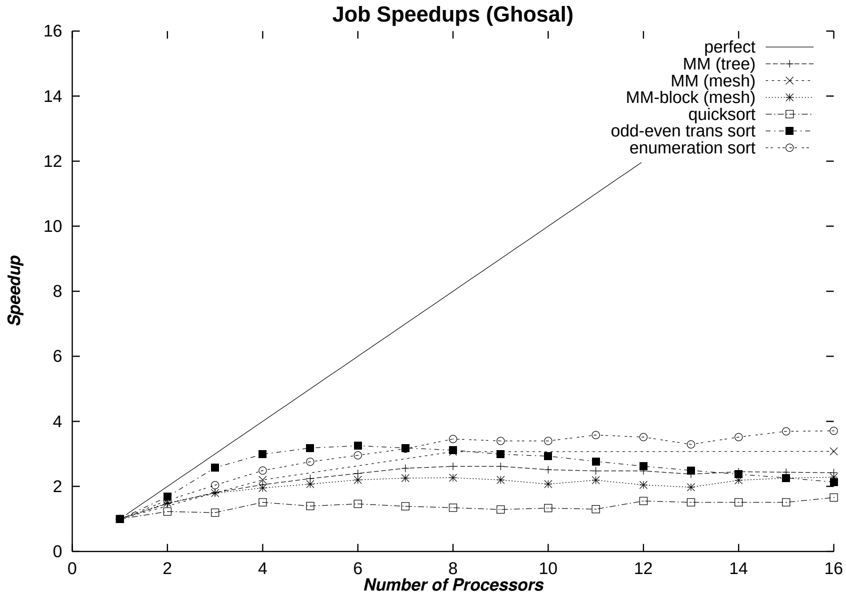
Finally, it should be noted that neither Sevcik's nor Dowdy's function captures the problem-size scalability of an application, in that changes to the problem size may cause ϕ, α, and β (for Sevcik's function) or s (for Dowdy's function) to change.

2.3.2 Workload Characteristics

Both Sevcik's function and Dowdy's function have been used to characterize the performance of applications in evaluating scheduling algorithms. However, most studies choose a range of execution-time function parameters that correspond to jobs having very poor speedup to ones having near-perfect speedup. Rarely are these parameters obtained from actual application runs for the reasons mentioned at the beginning of Section 2.3.1.

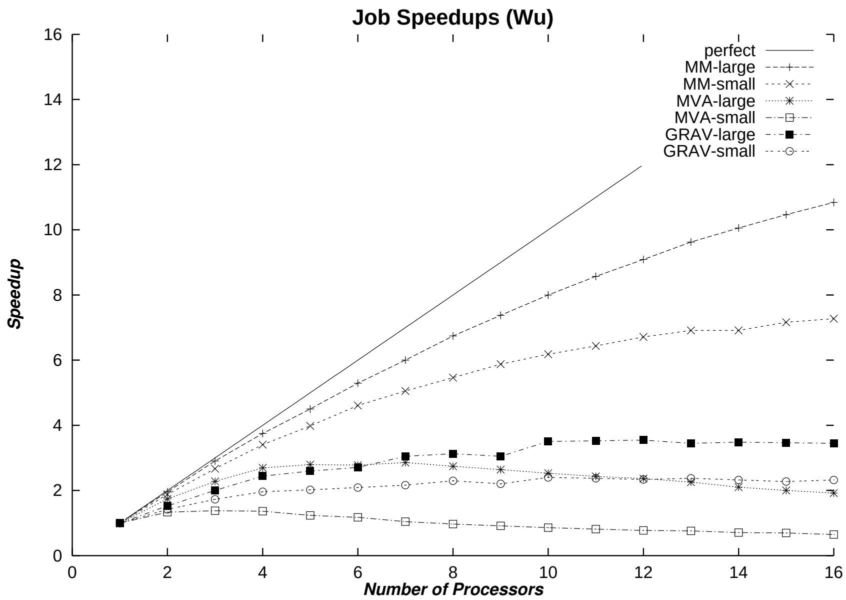
To guide the choice of such values, though, we examine speedup characteristics of applications that have been studied in the context of parallel-job scheduling. In Figures 2.3 and 2.4, we present the speedup curves of applications measured by Ghosal *et al.* on a 16-processor IMS T800 transputer [GST91] and by Wu on a 16-processor M88000-based system [Wu93], respectively. Below the graphs, we list the Dowdy and Sevcik function parameters that correspond to each application (using least-squared fit). In Ghosal *et al.*'s work, there were no jobs that possessed good speedup, while in Wu's work, a 400x400 matrix multiply demonstrated relatively good speedup right up to 16 processors.

More recently, Nguyen *et al.* studied the performance of a number of scientific applications on a 50-processor KSR-2 [NVZ96, Ngu96]. Their results are presented in Figure 2.5. These applications were either explicitly parallelized by the programmer or implicitly by a KAP or SUIF compiler. As can be seen, many of the explicitly-coded applications have very good speedup (with the exception



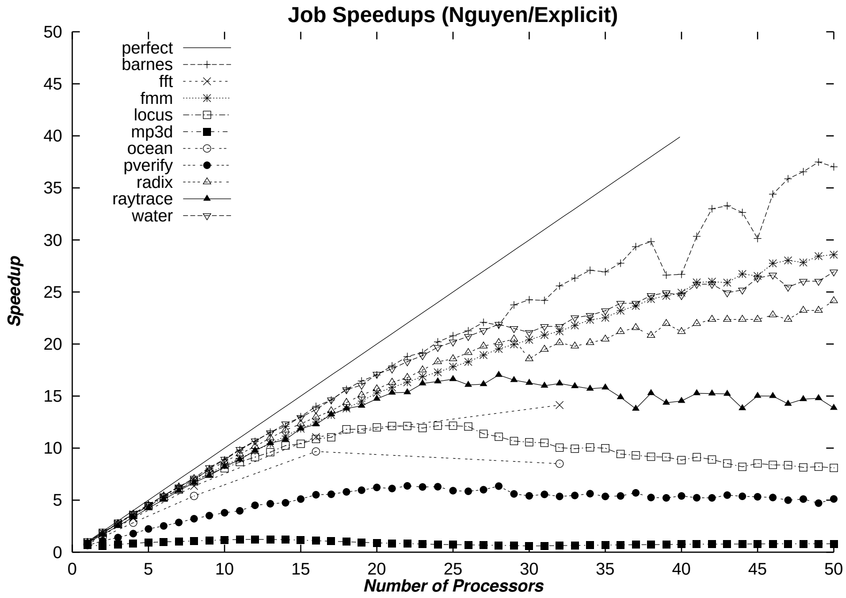
Application	**Dowdy Parms** (s, $pmax$)	**Sevcik Parms** (ϕ, α, β)
MM (tree)	0.307; 9	1.18; 15.9; 0.768
MM (mesh)	0.350; 8	1.00; 27.5; 0.636
MM-block (mesh)	0.261; 16	1.48; 5.50; 0.808
quicksort	0.659; 16	2.00; 0.00; 1.331
odd-even sort	0.138; 6	1.04; 0.0; 3.677
enum sort	0.216; 16	1.14; 15.0; 0.459

Figure 2.3: Speedup curves for applications studied by Ghosal *et al*. The table lists the parameters that can be used in both Dowdy and Sevcik functions to approximate the execution time for each application.



Application	Dowdy Parms (s, $pmax$)	Sevcik Parms (ϕ, α, β)
MM-large	0.027; 16	1.01; 3.39; 0.125
MM-small	0.072; 16	1.01; 1.13; 0.043
MVA-large	0.227; 16	1.10; 0.25; 1.772
MVA-small	0.371; 16	1.10; 3.22; 0.692
GRAV-large	0.199; 7	1.16; 9.66; 0.262
GRAV-small	0.596; 4	1.14; 3.11; 0.062

Figure 2.4: Speedup curves for applications studied by Wu. The table lists the parameters that can be used in both Dowdy and Sevcik functions to approximate the execution time for each application.



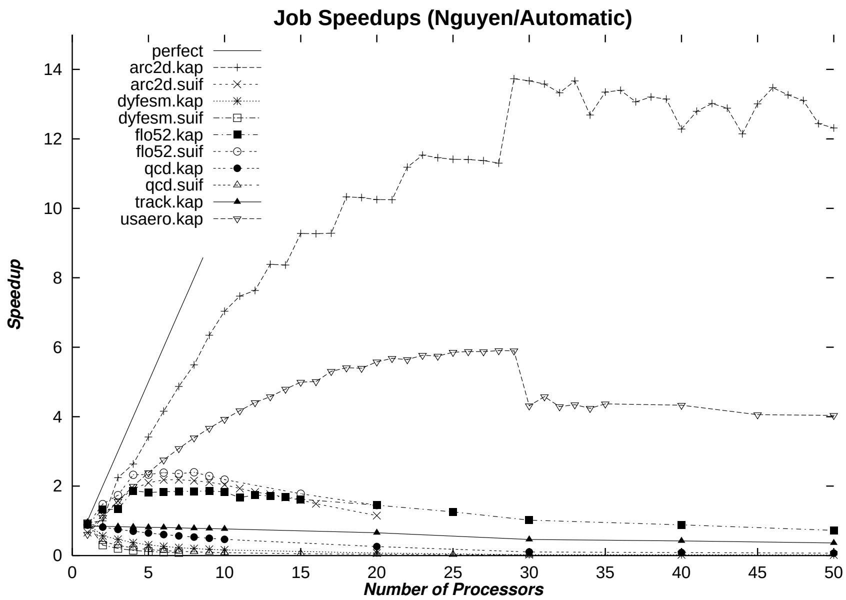


Figure 2.5: Speedup curves for applications studied by Nguyen *et al.* (separated into two graphs for clarity). Note that the scales of the two vertical axes differ by a factor of more than three.

of MP3D), in contrast to the automatically-parallized ones. If we use the Dowdy function to model these applications, then the s parameter ranges from 0.007 to 1.

Cornell Theory Center

The next source of workload data that we examine is from the Cornell Theory Center (CTC). Traces for all jobs submitted to CTC's SP-2 between June 18, 1995 to Oct 31, 1996, consisting of over 50 000 parallel jobs, were collected and made available by Hotovy [Hot96b, Hot96a]. These traces included, for each job, the submission, start, and termination times, aggregate node time, aggregate CPU time (divided between user and system time), and details as to the application name and user that submitted the job. The workload characteristics presented in this section are based on our analysis of these traces.

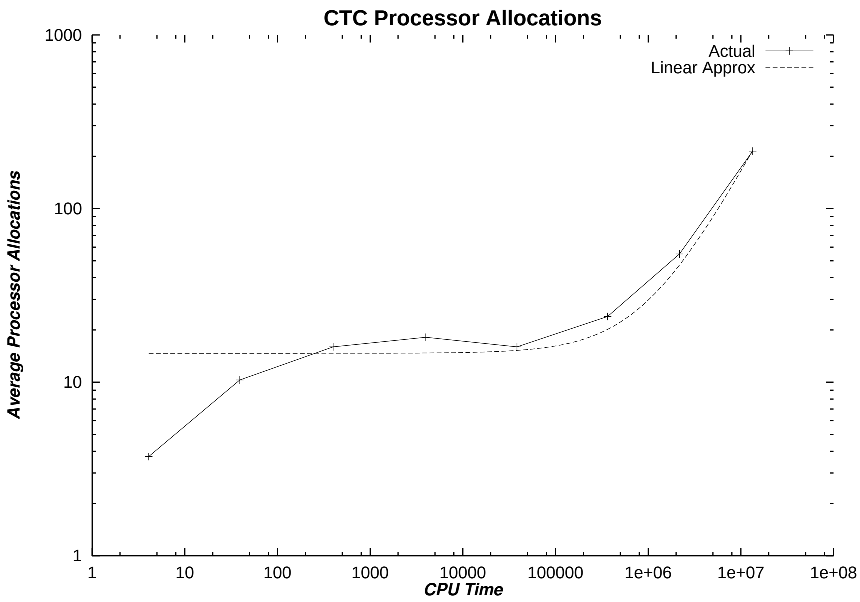
First, we observe that the aggregate CPU times for the jobs have a high degree of variability, having a coefficient of variation, which is the ratio of the standard deviation to the mean (see Section 2.4.1), that is close to five. The fact that there was an 18-hour wall-clock limit imposed on the execution time of jobs most likely led to lower variability than if there had not been such a restriction. Other parallel-computing centers have reported coefficients of variation even higher than this (ranging from 10 at NASA Ames [FN95] to between 30 and 70 at Cray YMP sites [CMV94]).

Next, we consider two aspects of this workload which are relevant to this thesis. In Figure 2.6(a), we plot the average processor allocation size for different ranges of CPU demands.[3] Although users could specify a minimum and maximum allocation size for each job, these two values were equal for the vast majority of requests. From the graph, it can be seen that average allocation size increases with CPU demand. Part of this can be attributed to the 18-hour limit, as large service demands can only be met with large processor allocations, but this trend also occurs for small CPU demands. Also shown in this graph is a linear approximation of the data points, which matches the data very closely except for low CPU demands. One possible reason for average processor allocations being low for small CPU demands is that the overhead of starting parallel threads is included in the CPU cost, thus pushing jobs running on large processor allocations into the next range of CPU times. For the remainder of the jobs, the average processor allocation increased, not surprisingly, by one for approximately every 18 hours of CPU time.

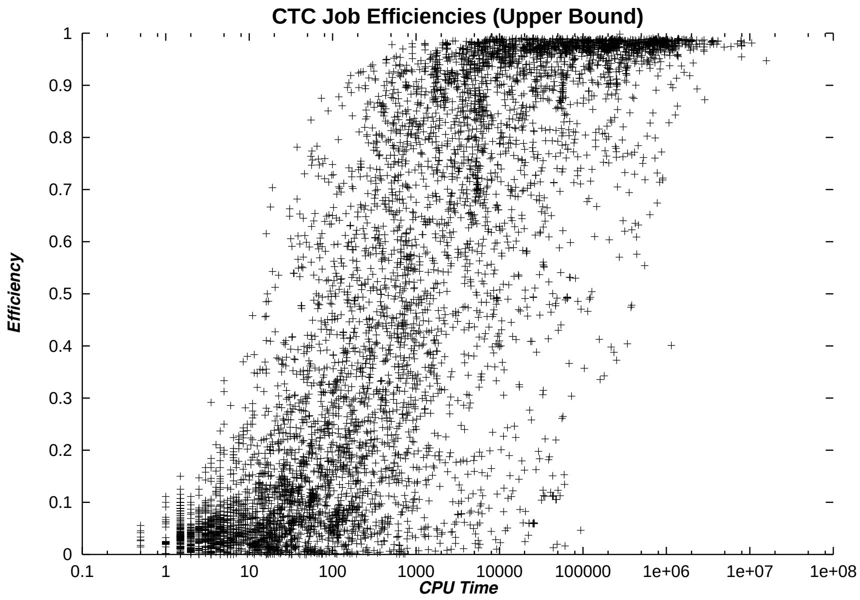
In Figure 2.6(b), we plot an upper bound on the efficiency of jobs as a function of their CPU demands (for clarity, we show only every ten jobs in order of efficiency value). Given that we have the CPU and wall-clock times of jobs, it is possible to determine how busy each job keeps the processors it is allocated. Since some of this CPU time might be associated with overhead, the ratio of CPU time to node time represents only an upper bound on efficiency.

As can be seen, long-running jobs tend to be more efficient than small ones, many having efficiencies very close to one (even for jobs having large processor allocations). Also, many of the

[3] Jobs were placed into buckets according to the $\log_{10}$ of their execution time (i.e., $<= 10$ secs in first bucket, > 10 secs but less than $<= 100$ secs in second bucket, etc.).



(a)



(b)

Figure 2.6: Data from the Cornell Theory Center showing (a) the average processor allocation as a function of execution time, and (b) the efficiency of jobs as a function of their execution time.

poor-efficiency jobs have efficiencies that are below the minimum possible (given the model that at least one processor is computing at any point in time). This implies that these jobs are either I/O-intensive, or are spending considerable amounts of time waiting for messages to propagate. What is most remarkable, however, is that significant clumping exists for both very efficient jobs and very inefficient jobs.

Generating Synthetic Workloads

Many of the performance results in this thesis are based on the generation of synthetic workloads, either to drive event-driven simulations or, for Chapter 6, to spawn synthetic jobs to the system. In this section, we briefly describe the generation of such workloads. (The specific details of each workload considered in the thesis are described at the point at which they are used.)

A synthetic workload is one in which the parameters of jobs are determined by a number of random variables (each drawn from a distinct statistical distribution):

Arrival Time The arrival times of jobs are normally determined by the *inter-arrival* time between consecutive jobs. The inter-arrival time distribution normally used is exponential, which has the well-known property of being memoryless. (In other words, the amount of time that has elapsed since the last job arrival does not give any indication as to how much longer one must wait until the next arrival.) This arrival pattern is known as a Poisson arrival process.

Service Demand The service demands of jobs are drawn from a statistical distribution that approximates that of actual workloads. The most common approach is to first measure or estimate the mean and the coefficient of variation of a workload, and then to use an exponential-class discipline, setting the distribution parameters so that the mean and variance of the distribution match those found empirically. In particular, if the coefficient of variation (CV) is less than one (i.e., the distribution has low variability), then an Erlang distribution, corresponding to the sum of two or more independent samples from an exponential distribution, is used; if the CV equals one, then a single sample from an exponential distribution is used; finally, if the CV is greater than one (i.e., the distribution has high variability), then a hyper-exponential distribution, which corresponds to choosing between two or more exponential distributions, is used.

Speedup Characteristics There is no widely-accepted method for the speedup of jobs to be pseudo-randomly generated. Sometimes, the workload is assumed to consist of a mixture of a small number of specific applications, the speedup characteristics of which are known. This approach, however, lacks flexibility and is not necessarily representative of real workloads (unless all jobs of a real workload are included). More commonly, a Dowdy function is used to model jobs, and the parameter s is drawn from some statistical distribution. One must be careful in choosing the distribution of s because the obvious choice, the uniform distribution, under-represents jobs having good speedup [BG96].

In this thesis, we typically define one or more speedup classes using either the Sevcik function, with specific values of α, β, and ϕ, or the Dowdy function, with specific values of s. For each job, we select a speedup class in specific proportions (e.g., 25% from one class, 75% from another). The combination of speedup class and service demand defines the speedup characteristics of the job.

Memory Requirements To date, there is very little empirical data on which to base distributions of memory requirements in large-scale systems. In one study, Setia considers three distinct distributions that are essentially based on uniform ones [Set95]. McCann and Zahorjan also use uniform distributions, but mention that binomial distributions were considered too [MZ95]. Lacking further data regarding the memory requirements of jobs in actual workloads, we also use uniform distributions in this thesis.

Thus, to generate a job in a synthetic workload, we select the inter-arrival time, service demand, and speedup class based on independent random variables. In Chapters 5 and 6, however, the choice of distribution used for selecting the speedup class depends on the memory requirements of the job, and so in this case, we select the memory requirement of a job before its speedup class.

2.4 Related Work

2.4.1 Review of Uniprocessor Scheduling Results

A scheduling problem can be defined by (1) the performance metric of interest, (2) the workload to be processed, (3) the knowledge about the workload that is available to the scheduler, and finally, (4) the scheduling constraints imposed by the system. The only performance metric that we consider here is the mean response time, but if preemption is constrained (e.g., causes non-negligible overhead), then a tradeoff might exist between minimizing the mean response time and maintaining a sufficiently-high sustainable throughput. The workload is generally modeled as just described, except that for uniprocessors, only the arrival pattern and service demand distribution are relevant.

The information available to a uniprocessor scheduler can be classified as follows:

No Knowledge Jobs are indistinguishable to the scheduler, and nothing is known about the characteristics of the overall workload.

Workload Knowledge Certain characteristics of the workload, such as the number of job classes and the moments (e.g., mean, variance) of their service-demand distributions, are known (either exactly or approximately).

Individual Knowledge The service required by each individual job is known (either exactly or approximately) when the job arrives.

Finally, the most common system-imposed constraint considered in uniprocessor scheduling is the cost of preemption, which may range from zero (free preemption) to infinity (no preemption).

As mentioned before, the underlying approach in minimizing the mean response time for uniprocessor systems is to favour jobs having the smallest remaining service demand, preempting (if possible) longer jobs when shorter ones arrive. Thus, given exact knowledge of job service demands, the best non-preemptive discipline is shortest processing time first (SPT) and the best preemptive discipline (assuming preemption is free) is shortest remaining processing time first (SRPT). Even if service demands are known only approximately, the analogous disciplines shortest expected processing time (SEPT) and shortest expected remaining processing time (SERPT) can, depending on the quality of the approximations, help reduce the mean response time.[4]

Workload knowledge that is particularly useful is the variability in service demand of jobs; it is usually expressed in terms of the coefficient of variation (CV) which is the ratio of the standard deviation of the service demand distribution to its mean. Given this knowledge, it is possible to approximate SRPT by considering the expected remaining service time conditioned on the currently acquired processing. If $\mathrm{CV} < 1$, a first-come first-served (FCFS) discipline tends to give the best mean response time, since the job with the most acquired service is expected to be the closest to completion. On the other hand, if $\mathrm{CV} > 1$ then a multilevel feedback policy (FB), which (in its ideal form) always runs the job having the least-acquired processing time first, tends to perform best. The reason is that jobs that have acquired the least processing time have, for such distributions, the least expected remaining service time.[5] If no information is known about the workload, then the round-robin discipline (RR) offers a good compromise as it yields a mean response time that is insensitive to the service time distribution.

Figure 2.7 illustrates the way in which mean response times depend on the coefficient of variation of service times for FCFS, RR, and FB, along with the exact-knowledge disciplines SPT and SRPT.[6] The value of service time knowledge can be seen for SPT and SRPT (particularly for $\mathrm{CV} < 2$) and the value of preemption can be seen for RR, FB, and SRPT (for $\mathrm{CV} > 2$). It is interesting to note that the FB discipline achieves nearly as good performance as SRPT. The optimal choice of scheduling discipline under various conditions is summarized in the table below the graph.

2.4.2 Multiprocessor Scheduling

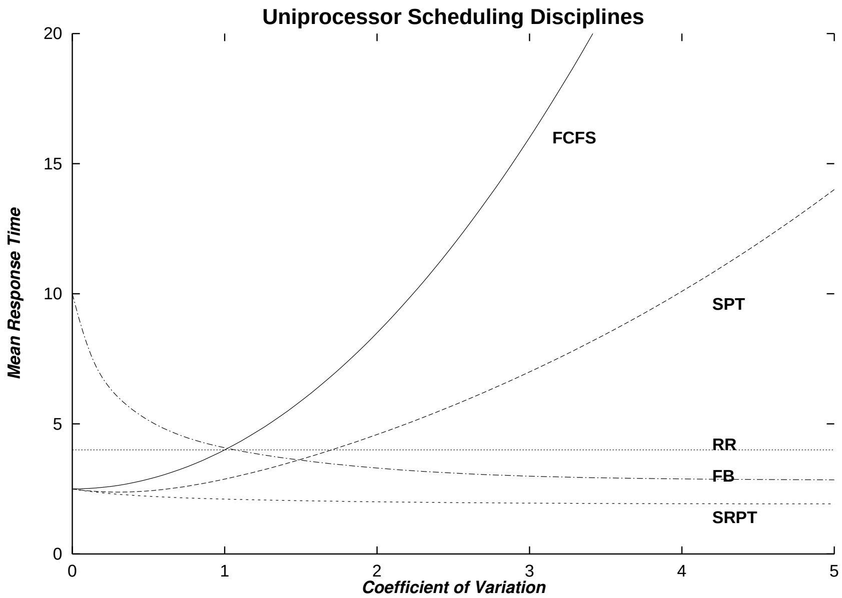
Theoretical Results for Multiprocessor Scheduling

We begin our examination of multiprocessor scheduling by briefly presenting some recent theoretical work on parallel-job scheduling. It has been known for a long time that exact solutions to many scheduling problems are NP-complete, so the usual goal is to find a good heuristic that has low com-

[4] Given the exact statistical distribution of the workload, SEPT is provably optimal while SERPT in general is not. For the special case of hyper-exponential distributions, however, SERPT is optimal [Sev72].

[5] The performance of FB actually depends on the service time distribution. It has been shown that for some distributions (e.g., a 3-point distribution with high CV), the performance of FB can be quite poor [Sch70]. For the hyper-exponential class of distributions, as are most often used, FB is markedly better than RR.

[6] All curves are relatively insensitive to the actual service-demand distribution except for FB; the curves shown here are for Erlang ($\mathrm{CV} < 1$), exponential ($\mathrm{CV} = 1$), or hyper-exponential ($\mathrm{CV} > 1$) distributions.



	Knowledge		
	None	Workload	Individual
Non-Preemptive	FCFS	FCFS	SPT
Preemptive	RR	FCFS (CV $<$ 1), FB (CV $>$ 1)	SRPT

Figure 2.7: Mean response time as a function of coefficient of variation for uniprocessor scheduling disciplines. (Adapted from Schrage [Sch70].)

putational complexity. But most of the research in this area has concentrated on off-line worst-case rather than on-line average-case results (the latter being of greater relevance to our work).

Early parallel-job scheduling research focussed on the makespan problem for non-adaptive jobs, but more recently interest has moved to minimizing the mean response time and considering adaptive jobs (where processor allocations can be chosen when the job is activated).[7] We consider the non-adaptive and adaptive cases separately.

Non-Adaptive Jobs Let $\mathcal{J}$ be the set of jobs to be scheduled, and assume that job i requires p_i processors and demands $t_i = T_i(w_i, p_i)$ units of service (where w_i is the amount of work associated with the job). We say that we are scheduling on a PRAM[8] if we do not place any restrictions on the choice of processors allocated to a job, and we say that we are scheduling on a line if the processors allocated to a job must be contiguous in a linear ordering of the processors. We can also schedule on a mesh, hypercube, or any other interconnection network, where the processor allocation must be contiguous in those forms.

Scheduling on a PRAM is a specific case of Garey and Graham's general result for scheduling jobs having multiple resources [Cof76]. By using a simple list-scheduling scheme, one can come within a factor of $r+1$ of the optimal makespan, where r is the number of resources (in this case just one—the processors). The jobs are placed in a list in any order, and whenever one or more processors become free, the first job in the list that fits is scheduled to run, giving us a worst-case makespan of at most twice the optimal.

Sleator later showed how scheduling on a line could be achieved within a factor of 2.5 times optimal [Sle80]. Scheduling on a line is the same as packing rectangles of size (p_i, t_i) in a strip of width P. Thus, minimizing the length of the strip needed is equivalent to finding the minimum makespan. The problem with Sleator's algorithm, and other similar algorithms, is that there is clearly some wasted space that can be easily used to reduce the schedule length. Sleator recognizes this fact, but explains that although taking advantage of this might improve the average-case behaviour, it has not been possible to prove that a better worst-case bound exists. What this means is that an algorithm which exhibits some provable worst-case behaviour might be interesting from a theoretical perspective, but may not be anywhere close to optimal with respect to average case behaviour.

The problem of scheduling jobs to minimize the mean response time has only recently been considered [TSWY94, LT94]. The heuristic proposed by Turek *et al.* is as follows. The set of jobs $\mathcal{J}$ are partitioned into disjoint subsets $\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_h$ such that job $i \in \mathcal{J}_k$ iff $2^{k-1} < t_i < 2^k$; effectively, the jobs are partitioned according to the log of their execution times. The jobs in each partition are then ordered according to increasing processor requirement and a schedule is constructed "shelf-by-shelf". On the first shelf, jobs are laid from the first partition until the next job no longer fits; the next shelf starts where the previous shelf left off. When all tasks in one partition have been exhausted,

[7] Research in this area tends to refer to such jobs as being *malleable*, conflicting with our use of the term.

[8] A parallel random access machine (PRAM) is a theoretical model of a ideal parallel machine having infinite global memory and identical processors.

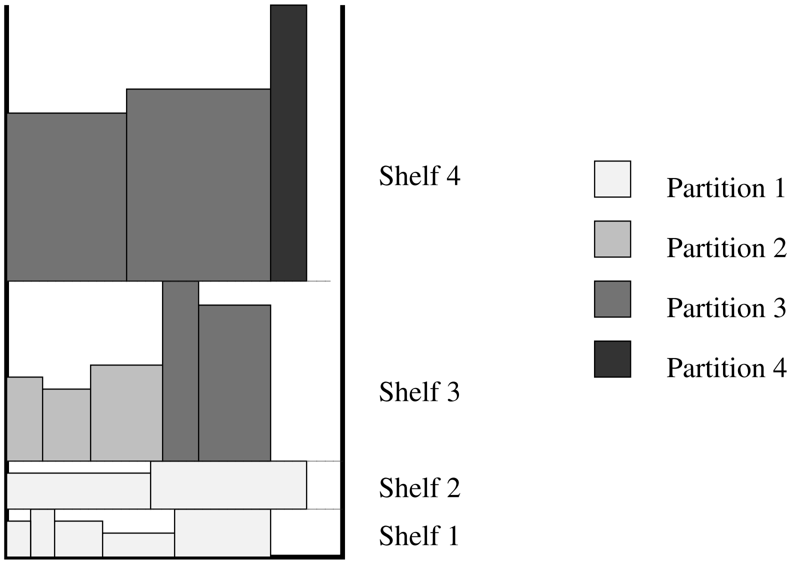


Figure 2.8: Turek *et al.*'s heuristic for minimizing mean response time

the algorithm continues with the next partition. As illustrated in Figure 2.8, the algorithm succeeds at placing the short tasks first and the longer ones later, as is needed to minimize response time.

The important fact in all this is that if the total height of a shelf is H_i and the number of tasks on a shelf is N_i, then the optimal ordering of shelves for minimizing the response time is the one which satisfies:

$$\frac{H_1}{N_1} \leq \frac{H_2}{N_2} \leq \cdots$$

The algorithm just described gets, in the authors' words, "close" to this ordering. A number of techniques exist to improve the solution from this point, including dropping jobs from a higher shelf to a lower one and combining shelves when possible [TSWY94].

In the original paper, it was proven that the average response time must be within a bound of 32 of the minimum, which was acknowledged to not be that encouraging. Since then, the authors have shown that new variants of the heuristic attain much better results [Wol94].

Adaptive Jobs As mentioned earlier, many studies have investigated adaptive scheduling disciplines, both to minimize the makespan [BB90, TWPY92, TWY92, LT94, TSWY94] and the mean response time [TLW$^+$94]. In all of these studies, an execution-time function $T_i(w_i, p)$ is associated with each job, i, but some of these place restrictions on the characteristics of $T_i(w_i, p)$.

First, consider the following extremes for the makespan. If the speedup of every job is perfect, then any packing of jobs that does not waste any space will give the same makespan; the area of the rectangle representing a job will always be the same regardless of the number of processors allocated. If the efficiency of every job is monotonically decreasing from one, and there are enough jobs to schedule, then the optimal makespan is the one which allocates only one processor to each job (minimizing the area), except for possibly those jobs at the end. Where the optimal makespan is more difficult to determine is when there are not enough jobs to just give one processor to each job or when the execution-time function is not monotonically decreasing.

Ludwig and Tiwari describe a method by which any algorithm intended for minimizing the makespan given non-adaptive jobs can be extended to the adaptive-job case [LT94]. The basic idea is to first find a processor allocation $\{p_1, p_2, \cdots, p_n\}$ that satisfies a certain minimization problem, and to then use a non-adaptive packing algorithm using $\overline{p}$. This results in an algorithm for which the worst-case behaviour is the same as that of the non-adaptive algorithm.

Turek *et al.* use a similar approach in studying the mean response time problem [TLW$^+$94]. First, they develop a heuristic for minimizing the mean response time given non-adaptive jobs; they then extend the heuristic to the adaptive case by choosing an appropriate set of processor allocations. The combined heuristic leads to a worst-case mean response time that is at most a factor of two more than the optimal.

Analytical Results for Multiprocessor Scheduling

Similar to the uniprocessor case, multiprocessor scheduling problems can be classified according to the amount of information available about jobs, the extent to which they preempt and reallocate processors among jobs, and the arrival pattern of jobs. In the multiprocessor case, an additional type of information that may be available is the speedup characteristics of jobs, information which can allow greatly improved scheduling. In this section, the only performance metric we consider is the mean response time.

The threads of a parallel job may be dispatched either in a *thread-oriented* or in a *job-oriented* manner. In the thread-oriented case, there is a single (logical) queue of individual threads. Any free processor takes a thread off the queue and executes it either to completion or for a specified quantum. The queue may be ordered by arrival time, expected service required, number of threads in each job, or some other criterion depending on how much is known about jobs. In the job-oriented case, processors are allocated to and perhaps preempted from jobs in groups. This approach makes it possible to exploit cache affinity, to support fine-grained parallelism, and to make an appropriate processor allocation to each job in light of what is known about its characteristics.

The styles of preemption that we will distinguish for job-oriented scheduling are:

Run-To-Completion (RTC) Some number of processors is assigned to a job when it is activated, and it retains exclusive use of all those processors until its service is completed.

Simple Preemption When a job is first activated, it is assigned a *set* of processors on which to run; the job may subsequently be repeatedly preempted and resumed until its service demand is satisfied, but must use the same set of processors each time it is activated (i.e., threads cannot be migrated).

Migratable Preemption When a job is first activated, it is assigned a *number* of processors. The job may then not only be preempted, and resumed, but its threads can also be migrated to a different set of processors. The job can never be allocated more processors than its initial allocation; in some cases, however, the threads of the job can be multiplexed on fewer processors, with some loss of performance.

Malleable Preemption The number of processors allocated to a job may be changed during its execution, either any time or at specific times during the computation. (Most scheduling research assumes the former, for simplicity.) There may be a significant overhead for the job to reconfigure itself to use the new number of processors effectively.

One factor that is important in parallel scheduling disciplines is *packing loss*, which is the degree to which processors (and correspondingly, memory) is under-utilized. This occurs primarily when there are one or more jobs waiting to be run, but the processor requirements of each of these jobs is larger than the number of processors currently available. Packing loss for processors is not a problem if malleable preemption is available because allocation sizes can be adjusted to ensure all processors are utilized.

Thread-Oriented Dispatching Early thread-oriented multiprocessor scheduling disciplines were simple extensions of the uniprocessor FCFS, RR, SPT, and SRPT ones [MEB88, LV90]. In FCFS and RR, threads are ordered according to the time at which they were placed on the queue, while in SPT and SRPT, threads are ordered according to increasing cumulative remaining service demand for the job. Implicit in most models used to evaluate thread-oriented disciplines is the assumption that a job's service demand is independent of the number of processors it is allocated. There is a certain amount of work that needs to be done, which has been spread amongst a number of threads. In this case, the disciplines yield performance that is in relative agreement with uniprocessor results. In particular, the importance of preemption increases with the variability in the service-demand distribution.

Because of the multiprocessor aspect, some anomalies can occasionally occur. In particular, RR will tend to give proportionately more processing time to jobs having a larger number of threads. RRJob avoids this by timeslicing equally among jobs as well as among the threads of a job [LV90].

Job-Oriented Dispatching

Rigid RTC Scheduling There has been relatively little analytic research conducted in rigid run-to-completion (RTC) scheduling disciplines, partly because of its relative simplicity. Maximizing sustainable throughput basically becomes a problem of minimizing packing losses. If execution-time knowledge is available, then the scheduler can reduce mean response times by favouring shorter jobs, similar to SPT in the uniprocessor domain, but doing so may lead to lower sustainable throughput as the shortest jobs may not be the ones that lead to the least packing loss.

Parallel-job scheduling software used today in production environments (see 2.4.2) is for the most part rigid RTC in nature.

Adaptive RTC Scheduling The fundamental issue in adaptive RTC scheduling is choosing an appropriate number of processors to allocate each job. Early work in this area established that a good processor allocation is one which corresponds to the smallest ratio of execution time to efficiency (i.e., the value at the knee of the execution time-efficiency curve), as this is a point that maximizes the ratio of benefit to cost [EZL89]. In absence of that information, the average parallelism offers a good alternative, as it gives similar performance guarantees as the value at the knee [EZL89].

It was shown shortly thereafter that the overall workload volume should be taken into account in the scheduling decision [Sev89]. By reducing the number of processors allocated to jobs as the system load increases, the mean response time can be greatly improved because jobs operate at a better efficiency point. In the limit, where the system load is near one, jobs should (it can be argued) be allocated no more than one processor since this leads to the highest possible system efficiency, assuming there are no other considerations such as large memory requirements. Conversely, under light overall load, each job can be allocated as many processors as it needs to attain its maximum execution rate. Multiprocessor disciplines must take the system load into account to avoid the problem of early saturation that may be caused by running jobs with too many processors.

A number of disciplines have been proposed since, each requiring varying amounts of information in its scheduling decision. Ghosal considers an interesting generalization of efficiency which uses the ratio of a job's speedup to an arbitrary cost function instead of to the number of processors. This ratio, called the *efficacy*, is used to determine a job's processor working set (*pws*) [GST91]. With the particular cost function chosen for use, $p/S(p)$, the *pws* can be shown to be equivalent to the number of processors at the knee of the execution time-efficiency curve. A number of static RTC disciplines were investigated assuming knowledge of a job's *pws*. The best of these (called FIFO+LA by Ghosal *et al.* but more commonly referred to by others as PWS) is a discipline which never leaves a processor idle when work is pending and which reduces partition sizes as the load increases.

A discipline that assumes (more realistically) that only the maximum parallelism is known is Adaptive Static Partitioning (ASP) [ST93]. In this discipline, an arriving job is allocated the lesser of the number of free processors and its maximum parallelism. When a job leaves, the pending jobs are all allocated an equal fraction of the freed processors. Through analytic models, it has been shown that ASP is superior to PWS for a certain 2-point service time distribution [ST93]. Subsequent simulation studies provide further evidence that a variant of ASP which bounds the maximum processor allocation to a job can perform at least as well as PWS [CMV94]. In our work, however, we have found that PWS tends to generally perform better than ASP (although these differences are minor).

Another discipline that takes only maximum parallelism into consideration is Adaptive Partitioning (AP) [RSD$^+$94]. This discipline varies its target partition size gradually in response to changing load. Our research shows that this discipline is not competitive against PWS or ASP, however. Other RTC disciplines that have been studied include A+&mM [Sev89] and AVG [LV90, CMV94].

Adding Preemption A significant problem with RTC disciplines is that they can lead to very high mean response times for workloads in which jobs have a high degree of variability in service demand (which is typical in actual workloads). Even if *a priori* knowledge of the service demand of jobs is available, response times will be high, as a long-running job that has been activated can delay subsequently-arriving short jobs until it has completed.

In simple preemptive schemes, each job is assigned to a set of processors when it is activated, but may be preempted at various points in time to allow other jobs to run on those processors. Ousterhout proposed the first such scheduling discipline, originally called coscheduling but now more commonly known as *gang* scheduling. In his approach, a matrix is defined in which each column corresponds to a processor and each row to a time slice. When a job arrives, a row is found containing enough uncommitted processors for the job; if no such row exists, a new row is created. The scheduler simply time-slices between the rows of the matrix [Ous82]. Using this approach, considerable packing losses can occur because rows can become inefficiently packed as jobs arrive to and depart from the system. Allowing migration to occur in addition to preemption can effectively reduce these packing losses [Ous82, LV90, MVZ93, PS95].

In Chapter 3, we consider the use of preemption in concert with adaptive scheduling disciplines. This provides the benefit of both improved sustainable throughput as the load increases and improved mean response times for workloads having highly variable service-demand distributions.

Adding Malleability Malleability is most flexible type of preemption, allowing the processor allocations of jobs to be changed arbitrarily during their execution. This can be achieved in two ways. First, if a job is initialized with many threads, then the operating system can vary the processor allocation by changing the number of threads running on each processor. Various techniques can be used to minimize the context-switch overheads associated with this approach, as long as threads do not synchronize too frequently [GTU91, MZ94]. (The principle difference between this kind of malleability and migration where threads may be multiplexed on a processor is that the former assumes that this can be implemented efficiently, which will not be true for workloads containing applications that synchronize frequently.) Second, the operating system can cooperate with the application when changing processor allocation so that the application can repartition and/or redistribute the data and change the number of active threads accordingly [TG89, GTS91]. This allows the application to continue to execute efficiently after a change in allocation, even if it synchronizes frequently [FR92].

The best-known discipline for malleable jobs that has been studied is commonly known as *equipartition*. In this discipline, the processors are reallocated equally among available jobs whenever a job arrives or departs from the system [MVZ93]. Equipartition has been shown to be effective over a wide range of workloads and a wide range of distributions in service demand [LV90, CMV94, PS95]. When system loads increase, allocations to jobs decrease allowing them to operate at a more favourable point on their efficiency curve. Also, because equipartition is effectively the analog of round-robin in uniprocessing, it is relatively insensitive to variability in service demand. But the overheads associated with repartitioning, both in terms of application structuring and run-time reorganization, have not yet been quantified in practice.

The equipartition discipline utilizes no information about applications (other than perhaps maximum parallelism if available). Brecht and Guha study the benefit of having service demand and speedup knowledge in improving mean response times. They found that only the availability of both types of information would lead to any significant improvement [BG96]. With respect to sustainable throughput, no discipline can do better than equipartition (in the absence of any memory constraints) since at heaviest load, each job will only receive a single processor.

Distributed-Memory Multiprocessor Considerations Most of the research described thus far has been focussed on uniform-memory architecture (UMA) multiprocessors, where the cost of accessing main memory locations is uniform across all processors. In distributed-memory systems, memory is distributed in such a way that each processor accesses "local" memory using independent data paths, thus allowing the system to be scaled to much larger sizes. The two basic classes of distributed-memory architectures are message-passing, sometimes referred to as no-remote memory access (NORMA), and shared memory, usually referred to as non-uniform memory access (NUMA).

An important consideration with these systems is that, for many inter-connection networks, the allocation of processors to a job should be, in some sense, "contiguous" within the network. For example, in a hypercube, a job should be allocated a sub-cube, and in a hierarchical ring network, the threads of a job should be placed within the same ring (and super-rings given sufficient parallelism). Finding a good partitioning scheme for hypercubes can be quite difficult, and several techniques have been proposed for this problem [CS87]. Brecht investigated this same issue in the context of NUMA systems using hierarchical inter-connection networks [Bre93b, Bre93a]. In his work, he showed that the need to schedule cooperating threads in relative proximity (with respect to the communication distance) to each other increased as disparity in communication costs between nearby and distant processors increased.

Apart from this, parallel-job scheduling results for distributed-memory systems are consistent with those that assume UMA architectures, namely that malleability can greatly improve mean response times, both in NORMA [MZ94, NSS93, Set93, DCDP90] and in NUMA systems [CDV$^+$94, CDD$^+$91]. The important issue is how malleability can be implemented, particularly in the NORMA case. McCann and Zahorjan assume that applications are designed for a virtual machine that matches the characteristics of the physical machine, which in their case is a mesh. As the system load increases, the threads of a job are multiplexed in such a way that the load on each processor remains evenly balanced [MZ94]. (They assume that multiplexing will not significantly influence performance, which as mentioned before may not be true for applications that synchronize frequently.) Through a rotation scheme, imbalances in processor allocation between jobs are evened out in the long run. Others assume that jobs can change their degree of parallelism to match changes in processor allocations. (Dussa *et al.* briefly discuss the implementation issues [DCDP90].)

It has also been observed that message-passing applications often have very low processor utilization due to synchronous messaging, particularly when a job has a high degree of load imbalance.

If this is indeed true, then fine-grained timesharing between several applications within a given partition can also become desirable [SST93].

For either class of distributed-memory systems, an important property of the design of a scheduling discipline is that it be able to scale well [FR90, ALL89, NW89]. Otherwise, these systems will never be capable of growing to their intended sizes.

Considering Memory Requirements of Jobs In contrast to processor allocation, consideration of memory has only recently gained attention in multiprocessor scheduling research. If jobs have large memory requirements, it may not be possible to fit all available jobs in the system, as dictated by equipartition. It is thus impossible for the system to attain the maximum level of efficiency (where each job receives a single processor); in this case, the scheduler must choose how to allocate processors in order to maximize the sustainable throughput, a topic which is investigated in depth in Chapters 4 and 5.

McCann and Zahorjan investigate how the minimum processor allocation requirements due to memory sizes affect scheduling in the context of the Intel Paragon [MZ95]. Their strategy is to balance the processing time received by each job as evenly as possible, with the goal of minimizing the mean processor allocation. Implicit in this goal is that lower mean processor allocations imply higher processor efficiencies, but they do not explicitly consider the use of speedup characteristics of individual jobs in allocating processors.

Peris *et al.* describe a technique to model the paging behaviour of parallel jobs when memory is constrained [PSN94]. They use this model to evaluate the performance of a real parallel job on a message-passing system as the number of nodes allocated to the job (and hence memory) is varied. Although the problem of scheduling jobs is not investigated, this work provides a method for modeling the effects of reducing the memory allocated to a job. One of their conclusions is that the performance of the parallel application they consider decreases dramatically as soon as paging begins due to insufficient memory, a conclusion that is also reached by Setia [Set95] and Burger *et al.* [BHMW94].

An actual implementation of a multiprocessor scheduler that takes into account the memory sizes of applications is that of the Tera MTA [AKK+95]. This scheduler employs an algorithm for swapping jobs in and out of memory that minimizes the time memory is unavailable for running jobs. Moreover, it allows a priority to be associated with each job in the form of an expected space-time demand value; when a job's actual space-time usage drops below the assigned value, its priority is increased.

Commercial Scheduling Software

There are two scheduling systems that dominate parallel-job scheduling, namely Network Queueing System (NQS) and LoadLeveler. NQS has been used on numerous platforms, including the Cray, Intel Paragon, Thinking Machine CM-5, and Kendall Square Research KSR. It basically operates by defining queues to which jobs can be submitted, each of which is typically configured to be associated with a specific subset of processors. Queues can be rotated over time to allow different types of jobs to be run at different times (e.g., a very large partition may be open only at night or on weekends), and certain queues may have priority over others (e.g., to favour jobs having short execution times). One of the problems that users find with NQS is that they must often choose from up to thirty queues, depending on the anticipated execution time, memory requirements, or processor allocation size. In parallel environments, NQS is typically configured to run jobs until completion.

LoadLeveler is a competing product most often used on the IBM SP series of systems. It is more adaptive in nature than NQS, in that users can specify minimum/maximum processor allocations, but again, jobs are run until completion. A significant problem that has been observed by users of the system is that jobs having large processor requirements are often starved from execution. If such a job has been in the system for a long period of time, LoadLeveler begins to reserve processors, leaving them idle to see if enough processors will become available; if not enough processors become available within a specific, user-settable amount of time, the reserved processors are returned to the free pool and the system tries to schedule the job later. (All this is due to the run-to-completion nature of the system.)

An extension to LoadLeveler that has recently become popular in parallel computing centers is EASY [Lif95, SCZL96]. This is still a rigid RTC scheduler, but uses execution-time information provided by the user to offer both greater predictability and better system utilization. When a user submits a job, the scheduler indicates immediately when that job will run; jobs that are subsequently submitted may only be run before this job if they do not delay the start of its execution (i.e., a gap exists in the schedule containing sufficient processors for sufficient time). More recent work by Gibbons shows how historical information can be used instead of information provided by the user [Gib96]. (He also describes an implementation of EASY on top of Platform Computing's Load Sharing Facility.)

2.4.3 Summary

Research on parallel-job scheduling is best summarized in a table similar to that presented in the introduction. Thread-oriented schedulers can be considered as rigid in nature because it is typically the case that the degree of parallelism is chosen by the user submitting the job. Adaptive scheduling disciplines can be categorized according to the type of information which they assume is available, which can include service demand, speedup characteristics, and memory requirements. All types of preemption (simple, migration, malleability) can be applied to all adaptive disciplines, but only simple and migratable preemption are meaningful for rigid disciplines. The names of many disciplines are shown in Table 2.1, along with a list of studies examining each one. The disciplines developed in this thesis are included in the table, and are highlighted in italic.

Identical (or near-identical) disciplines which have been named differently are shown with an equal (=) sign. Some disciplines, namely RRJob, have been defined sufficiently differently to belong to different categories in their different incarnations. Finally, some of the rigid schedulers do use service-demand information if available, but this distinction is not shown in the table.

	RIGID		ADAPTIVE			
	Thread-Oriented	***Job-Oriented***		***Work***	***Speedup***	***Memory***
RTC	FCFS [MEB88, LV90]	RTC [ZM90]	A+,A+&mM [Sev89, CMV94]	yes	min/max	no
	SNPF [MEB88]	PPJ=SP=FP [RSD$^+$94, SST93, ST93, NSS93]	ASP=AP=EPM [ST93, NSS93, RSD$^+$94, AS97]	no	pws	no
		NQS	PWS [GST91]	no	no	no
		LSF	Equal,IP,AP [RSD$^+$94, AS97]	no	no	no
		LoadLeveler	SDF [CMV94]	yes	no	no
		EASY [Lif95, Gib96]	AVG,Adapt-AVG [CMV94]	no	avg	no
			System [SST93]	no	no	no
			AEP [AS97]	no	no	no
			DIF [AS97]	yes	yes	no
			shelf [TWPY92, TWY92]	yes	yes	no
			SMART [TSWY94, TLW$^+$94]	yes	yes	no
		NPTS [LT94]	MPTS [LT94]	yes	yes	no
		LSF-RTC	***LSF-RTC-AD***	either	either	either
PREEMPTION						
SIMPLE		Cosched (matrix) [Ous82]				
		LSF-PREEMPT	***LSF-PREEMPT-AD***	either	either	either
MIGRATABLE	PSNPF,PSCDF [MEB88, LV90]	Cosched (other) [Ous82, LV90]	Round-Robin [ZM90]	no	no	no
	RRJob [LV90]	RRJob [MVZ93]	***FB-ASP***	no	no	no
	RRProcess [LV90]		***FB-PWS***	no	pws	no
		LSF-MIG	***LSF-MIG-AD***	either	either	either
MALLEABLE	(not applicable)		Equi/Dynamic Partition [TG89, LV90, DCDP90, MVZ93, AS97]	no	no	no
			Dynamic [ZM90, MVZ93]	no	no	no
			FOLD,EQUI [MZ94]	no	no	no
			EQ-AVG [CMV94]	no	yes	no
			W&E [BG96]	yes	yes	no
			BUDDY,EPOCH [MZ95]	no	no	yes
			MPA-Basic/Repl1/Pack	no	no	yes
			MPA-OCC/EFF	no	yes	yes
			LSF-MALL-AD	either	either	either

Table 2.1: Classification of many of the scheduling disciplines that have been proposed and evaluated in the literature. On the top, disciplines are first distinguished by whether it is the user (rigid) or the system (adaptive) that chooses processor allocation sizes. In the latter case, disciplines can be further distinguished by the information the they takes into account (service demand, speedup, memory requirements). Disciplines that are italicized correspond to new contributions of this thesis, and those that are prefixed by "LSF-" to disciplines that have been implemented on top of LSF.

2.5 Related Topics

In multiprocessor scheduling research to date, very simple models of applications have been used. The two mentioned in Section 2.3.1 are Dowdy's function, which does not take into account increases in overhead as the number of processors grows, and Sevcik's function, which deals with some of the deficiencies of the former but can still be difficult to apply. Recall that the reason for having such models is to express the relationship between processor allocation and efficiency, thereby providing the means to evaluate the performance of various scheduling disciplines. In this section, we review some of the results in application scalability that relate to the use of these functions.

Application Scalability

The central theme of scalability analysis is to understand how well parallel systems can scale up in size. A parallel system, in this context, refers to a specific combination of a hardware architecture and an application.

The first notable development in application scalability, known as Amdahl's Law, states that the fraction of sequential computation places an upper bound on the achievable speedup [Amd67]. The next notable development, usually attributed to Gustafson, was the observation that in many cases, one does not just increase the number of processors for a fixed problem size [Gus88]. Rather, an increase in the number of processors is often accompanied by an increase in problem size (either data or total computation or both). Since this observation was made, a large number of scalability metrics have been proposed and examined, including speedup [Amd67], scaled speedup [Gus88], sizeup [SG91], and isoefficiency [GGK93]. This work is surveyed by Kumar and Gupta [KG94].

Increasing the problem size with the number of processors permits the determination of *scaled speedup*. In one form of this metric, the problem size is scaled in such a way that the per-processor computation load remains fixed. As an example, consider matrix-vector multiplication which requires $O(n^2)$ multiplications on one processor or $O(n^2/p)$ on each of p processors, where n is the size of the matrix and vector to be multiplied. In order to keep the computational load constant, the problem size n must increase by a factor of $\sqrt{p}$ as p is increased from 1. In an alternative form of scaled speedup, the problem size is scaled in such a way that the per-processor memory requirement remains fixed [SN93, GGK93].

Another problem with using fixed speedup is that it has the unfortunate property of favouring slow processors. Consider a simple model of an application where threads are either communicating or computing. On slower processors, the time spent communicating will represent a smaller fraction of the total execution time, resulting in better speedup. In order to provide a fairer comparison, Gustafson proposed *sizeup* as the ratio of the size of problem that can be solved on a parallel computer to that on a uniprocessor within a fixed amount of time. This sizeup metric led to the development of the SLALOM benchmark [Gus92].

A more recent scalability metric involves determining the rate at which a problem size must increase in order to maintain a certain level of efficiency as the number of processors is increased. The function that characterizes the problem-size increase is called the *isoefficiency function* [GGK93]. This can be used to compare the performance of different applications on differently-sized systems.

The reason why these metrics are used in preference to (fixed) speedup is that more realistic conclusions can be drawn regarding the scalability of an application-system combination. Where fixed speedup assumes that users will always want to solve problems of the same size, other metrics measure how problem sizes can be increased as a function of available resources.

One issue that is commonly neglected in scalability analysis is that scaling a problem size in a way that is of practical value, particularly scientific problems, often entails changes to other parameters. For instance, it has been shown that for a Barnes-Hut algorithm simulating the evolution of galaxies, an increase in the number of objects (to improve the accuracy of the results) requires an increase in the number of time steps over the same time interval [SHG93]. This illustrates the fact that any given scientific application may not necessarily scale as well as traditional scalability analysis suggests.

Chapter 3

Migratable Preemption in Adaptive Scheduling

Prior to 1994, there had been numerous scheduling disciplines proposed for multiprogrammed multiprocessor systems, the evaluation of which had, for the most part, been based on workloads having a relatively low variability in the service demands of jobs. However, high performance computing centers have reported that variability in service demands can in fact be quite high. In a detailed study of the anticipated workload of its "Numerical Aerodynamic Simulation" (NAS) facility [NAS80], NASA specified a workload consisting of eight types of computational tasks with expected mean service demands differing by as much as a factor 3500. Assuming that the service demands within each class are exponentially distributed, the coefficient of variation of service demand (CV), which is the ratio of the standard deviation of service demand to its mean, in the overall workload is 7.23. This high degree of variability is further supported by more recent workload characterization studies of high-performance computing centers at NASA [FN95], Cornell [Hot96b] and other locations [Gib96]. In a related scheduling study, Chiang *et al.* report that the coefficient of variation observed on a weekly basis on the CM-5 at the University of Wisconsin ranges from 2.5 to 6, with 40% of them being above 4 [CMV94]. They also report that some measurements from Cray YMP sites range from 30 to 70 [CMV94, Ver94].

In this chapter, we consider two well-known adaptive run-to-completion (RTC) scheduling disciplines, showing how they behave for workloads having a highly variable service demands, and present ways in which they can be adapted using migratable preemption to better handle this condition. These enhancements make no additional assumptions about the information available at a job's arrival, other than what is required by the original discipline.

Focussing on coefficients of variation in the range of 5 to 70, our goals are (1) to compare the performance of the existing disciplines, and (2) to propose enhancements to these disciplines that make them perform better over this range of CV. For comparison purposes, we also consider the performance of an ideal form of equipartitioning (IEQ) in which each job receives an equal share of the processors [TG89, ZM90]. If the overhead of adapting to the altered allocation is neglected (hence "ideal" equipartition), then IEQ is known to perform very well for high values of CV.

The principle underlying the enhanced disciplines is that, in uniprocessor scheduling, the variability in service demand plays a large role in determining the best scheduling discipline when exact knowledge of service demands is absent. For CV > 1, a discipline that favours jobs that have the least acquired processing time can greatly reduce the mean response time (MRT) of the system. Our enhanced disciplines generalize the uniprocessor multilevel feedback (FB) approach to a multiprocessor setting.

The results in this chapter are based on three synthetic workloads that differ in the amount of speedup attained by jobs. In the first workload, all jobs have near-perfect speedup. In this case, the results are consistent with what is known from the study of uniprocessor systems. As the coefficient of variation of service demand increases, so does the mean response time for the run-to-completion disciplines. The enhanced disciplines have worse performance when CV < 1, but provide improved performance as CV increases beyond one. In the second workload, short-running jobs have very poor speedup, while long-running ones have relatively good speedup. With this workload, the enhanced disciplines are still superior, but to a lesser degree. Finally, we consider a workload derived from the NASA study mentioned above, using a speedup characterization that lies between the previous two extremes, in order to show how the disciplines would behave under a more realistic workload.

The structure of this chapter is as follows. We first describe the baseline run-to-completion disciplines, and present the modifications needed to make them preemptive assuming migration is available. In Section 3.2, we specify the workload model and describe the simulation experiments upon which we base our conclusions. The results of the simulation experiments are then presented and analyzed in Section 3.3, and conclusions are presented in Section 3.4.

3.1 Disciplines

The two disciplines that we study here are PWS [GST91] and ASP [ST93]. These two disciplines differ primarily in the amount of information given to the scheduler; in PWS, some characteristics of the speedup curve are known while in ASP only the maximum parallelism is known. We also experimented with other disciplines (in particular AVG [LV90] and AP [RSD$^+$94]), but we omit the results as these did not offer much in terms of additional insight.

3.1.1 Baseline Disciplines

The *processor working set* (or *pws*) of a job i is the minimum number of processors that maximizes the ratio of its speedup $S_i(w_i, p)$ to a cost function $C_i(w_i, p) = p/S_i(w_i, p)$. This cost function expresses the notion that the cost of a processor depends on how efficiently it is being utilized. Ghosal *et al.* explore several different disciplines that make use of the *pws*, and conclude that the following discipline (called FF+FIFO by them) performs best [GST91]:

PWS When a job arrives in the system, and there are free processors, it is allocated the lesser of the number of free processors and its *pws*. When a job leaves, the scheduler repeatedly examines the jobs in the queue and selects the first one whose *pws* fits in the available processors; if none fit, then the first job is given the remaining processors.

PWS as originally defined limits its search of the queue to an initial window of jobs, but in this chapter, we set no such limit in order to maximize the chance of finding a job for which the *pws* fits in the available processors.

The ASP discipline differs from PWS in that it spreads free processors evenly among all waiting jobs instead of allocating the first job as many processors as it requires:

ASP When a job arrives to the system, it is given the lesser of its maximum parallelism and the number of free processors. When a job is completed, the processors are allocated evenly (as is possible) among jobs that are waiting.

We assume that a job's maximum parallelism is the number or processors for which its speedup function is maximized. (If Dowdy's function is being used to model jobs, then a *pmax* value associated with each job is required.)

Finally, we define ideal equipartition (IEQ) as follows:

IEQ When a job arrival or departure occurs, the processors are dynamically reallocated to the current set of jobs in such a way that $(P \bmod \|\mathcal{J}_{\text{ready}}\|)$ jobs are allocated $\lfloor P/\|\mathcal{J}_{\text{ready}}\| + 1 \rfloor$ processors and the rest one less, where $\mathcal{J}_{\text{ready}}$ is the set of jobs in the system ready to run. Periodically, the scheduler rotates the jobs, placing the last job (in a run queue) at the front, thereby evening out any imbalances in processor allocation. As a result, if there are more jobs than processors, then all jobs will receive some fraction of the system's processing capacity quickly. Once again, a job is never allocated more processors than its maximum parallelism.

3.1.2 Migratable Feedback-Based Disciplines

The preemptive feedback-based scheduling disciplines that we derive from the run-to-completion ones all follow a similar pattern. When a job arrives to the system, it is configured for a certain number of processors that depends on the known characteristics of the job and the other jobs currently available for execution. At the start of each time slice, all active jobs are examined, and those having the least acquired processing time are scheduled to run. Acquired processing time is the number of processor-seconds consumed by the job so far, and thus can differ from the actual work accomplished by the computation since the latter is influenced by the job's speedup characteristics. The scheduler repeatedly selects the job with the next least acquired processing time and which fits within the remaining set of processors (as configured upon arrival), and schedules it to run.

The two variants of the preemptive disciplines that we propose are:

FB-PWS An arriving job is configured for a partition size of

$$\min\left\{pmax_i, \frac{\min\{pws_i, P\}}{R+\min\{pws_i, P\}} \times P\right\}$$

processors, where $pmax_i$ is the job's maximum parallelism and R is the sum of the processor allocations for all other jobs currently in the system. In general, each job is allocated a fraction of the P processors that corresponds to its share of the total anticipated number of processors allocated in the system. Under light load, a job will be allocated more processors than its *pws* and under heavy load it will be allocated fewer.

FB-ASP An arriving job is configured with $\lfloor P/\mathcal{J}_{\text{ready}} + 0.5\rfloor$ processors, except that if there are at least twice as many available processors as the computed partition size, then one more processor is given (to account for uneven partition sizes). The partition size computation differs from the standard ASP, as the latter divides the number of *free* processors by the number of *waiting* jobs; in FB-ASP, jobs are configured immediately upon arrival and thus do not wait in the same sense.

When there are fewer processors left than any of the remaining jobs' configurations, the scheduler runs the next job anyway using the remaining processors. For this, we assume that we have a thread scheduler that, at the very least, avoids blocking threads while they are holding locks and implements some form of cache-affinity scheduling. But when a job configured for an allocation of p processors is activated with only q ($q < p$) processors, its execution rate will be less than q/p times its full execution rate due to the mismatch of threads and processors. Gupta *et al.* studied the effect of combined thread scheduling features, including those just described, and showed that for a set of four applications, the processor utilization dropped by just under 9% over batch scheduling (see Figure 6 in Gupta *et al.* [GTU91]). Based on this result, we assume that a job that is running with fewer processors than its configuration progresses 9% slower than q/p times its full execution rate. Experimentation indicates, however, that these two disciplines, particularly FB-ASP, tolerate higher slowdown values reasonably well. When the slowdown value was increased to 100%, the increase in mean response times for FB-PWS, relative to a slowdown value of zero, ranged from 4% at low loads to 30% at high loads, while for FB-ASP, the increase was less than 2% throughout.

3.2 Definition of Model

We use discrete event simulation to evaluate the different scheduling disciplines. Input parameters to the simulator include the arrival rate, average service demand, coefficient of variation of the cumulative service demand, job speedup characteristics, and the scheduling discipline to be employed. The specification of and results from the various simulation runs are given in Section 3.3.

3.2.1 System Model

The system model consists of 100 functionally equivalent processors. No details of the interconnection network or the memory system are modeled. The cost of de-scheduling and re-scheduling a job in the preemptive disciplines is modeled explicitly, and is assumed to be 2.5% of the length of a timeslice (a conservative estimate). The cost of reallocating processors in IEQ is assumed to be zero.

Jobs are assumed to arrive according to a Poisson process, and have a service-demand distribution that is Erlang, exponential, or hyper-exponential, depending on the specified coefficient of variation.

3.2.2 Workload Model

In this work, we use Sevcik's execution-time function [Sev94] for characterizing the speedups of jobs. Recall from equation 2.2 that this function has the form:

$$T(w,p) = \phi\frac{w}{p} + \alpha + \beta p$$

where p is the number of processors allocated to the job and w is the amount of work (cumulative service demand). Our choice of values for ϕ, α, and β are based on the work by Wu [Wu93].

Given this execution-time function, a job's maximum parallelism is:

$$pmax_i = \begin{cases} \infty & \text{if } \beta = 0 \\ \sqrt{\frac{\phi w}{\beta}} & \text{otherwise} \end{cases}$$

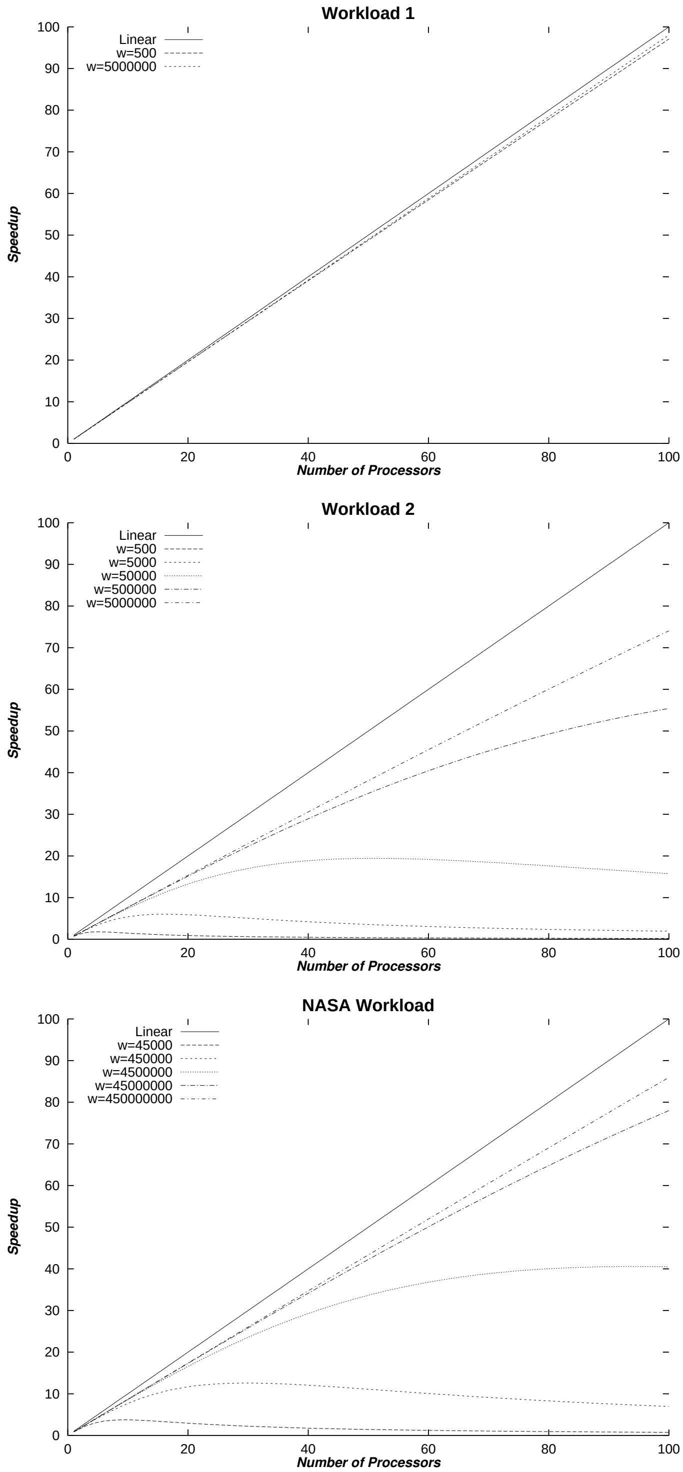
and its *pws* is

$$pws_i = \begin{cases} \frac{\phi w}{\alpha} & \text{if } \beta = 0 \\ \frac{\alpha - \sqrt{\alpha^2 + 12\beta\phi w}}{-6\beta} & \text{otherwise} \end{cases}$$

Two speedup characterizations were used to represent workloads having quite distinct parallelization overheads. In the first, all jobs, irrespective of size, have nearly perfect speedup, whereas in the second, short-running jobs experience poor speedup and long-running jobs experience relatively good speedup. Illustrated in Figure 3.1 are representative speedup curves for the two workloads for job sizes ranging from 500 to 5 000 000. The parameters used for characterizing the two workloads (as well the NASA workload) are shown in the table below the graphs.

We also experimented with other workloads and found that, qualitatively, their performance was between the two we chose. In particular, we considered mixed workloads in which jobs had varying values of ϕ, α, and β, as might be found in actual workloads. As such, we feel that the two workloads chosen are sufficient to explore the behaviour of the various scheduling disciplines.

We study a third specific workload, which is chosen to represent the NASA workload described earlier. The service-demand distribution is an 8-stage hyper-exponential distribution in which the



Workload	Mean CPU Demand	Sevcik Parameters
1	1000	$\phi = 1.02$, $\alpha = 0.05$, $\beta = 0.0$
2	1000	$\phi = 1.3$, $\alpha = 25$, $\beta = 25$
NASA	92371	$\phi = 1.15$, $\alpha = 1000$, $\beta = 600$

Figure 3.1: Representative speedup curves and Sevcik parameters for the workloads used in this chapter.

mean of each stage is set to the expected service demand of each corresponding workload component [NAS80]. In the absence of speedup information about the jobs, we chose to use a characterization that fell in between our endpoints as defined by workloads 1 and 2 (see Figure 3.1).

3.3 Analysis of Simulation Results

First, we examine the performance of the various scheduling disciplines under workloads 1 and 2 as a function of the coefficient of variation in service demand. Next, we examine the performance of all the disciplines under the NASA workload as a function of system load.

For the most part, a sufficient number of independent trials were done to obtain a 95% confidence interval that was within 5% of the mean for each data point in our simulation results. Because of the instability of distributions having high coefficient of variation, however, the data points for CV = 70 sometimes have a confidence interval greater than 5% of the mean for the higher system load values. Each trial had a warm-up period in which the first 20 jobs were discarded. A trial terminated when the subsequent 100 000 jobs to arrive left the system. The simulation results of a run are based only on the response times of these 100 000 jobs. To improve the stability of the results for the trials where CV = 30, we used twice as many jobs, and for those where CV = 70, we used four times as many.[1]

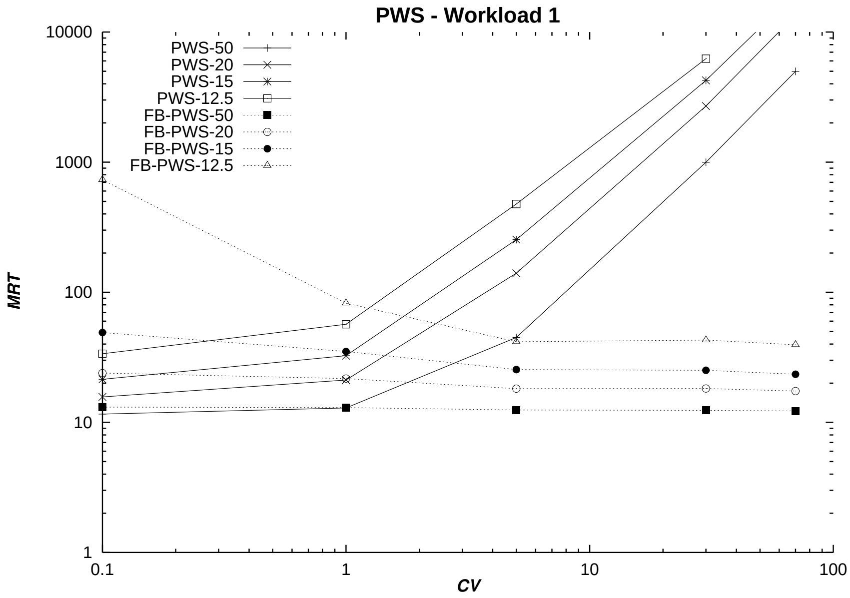
3.3.1 Workload 1

Figure 3.2 plots the performance of the original disciplines in comparison to their FB counterparts as a function of the CV. Curves are shown for each of four arrival rates for each discipline. The solid lines represent the non-FB disciplines, while the dotted lines represent their FB counterparts. The mean service required per job was 1000, so the mean interarrival times of 50, 20, 15, and 12.5 correspond to system loading factors of 20%, 50%, 67%, and 80%, respectively.

In either case, the FB variant outperforms the non-FB variant as the coefficient of variation increases beyond one. At high load and CV = 70, the response times of the non-FB variants of PWS and ASP are more than one hundred times worse than those of their FB counterparts. Consistent with results from uniprocessor scheduling, the FB variants have, in general, decreasing mean response time with increasing CV, while the opposite holds for the RTC disciplines.

Figure 3.3 shows the relative performance of the various disciplines at light and heavy loads in comparison to ideal equipartitioning. At light load, the performance difference between the FB-based disciplines and IEQ is very small (PWS and ASP are indistinguishable), and at heavy load, FB-PWS performs equally well as or better than IEQ for CV > 1. The reason why FB-ASP does not perform as well as FB-PWS in this workload is that long-running jobs receive the same share

[1] For high-variability two-stage hyper-exponential distributions, a very large fraction of samples is drawn from an exponential distribution having small mean and the remainder from one having a very large mean. For example, given a coefficient of variation of 70, only 0.01% of jobs are chosen from the latter (using the method proposed by Sevcik *et al.* [SLTZ77]). As such, a large number of samples points (e.g., 400 000) are needed for the sample mean and CV to be consistently close to that of the underlying distribution.



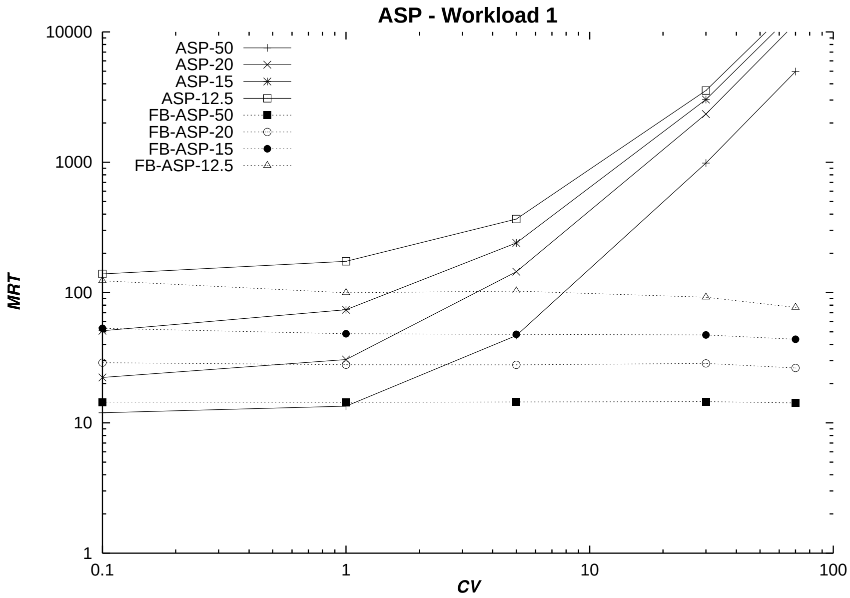


Figure 3.2: Performance of scheduling disciplines under workload 1.

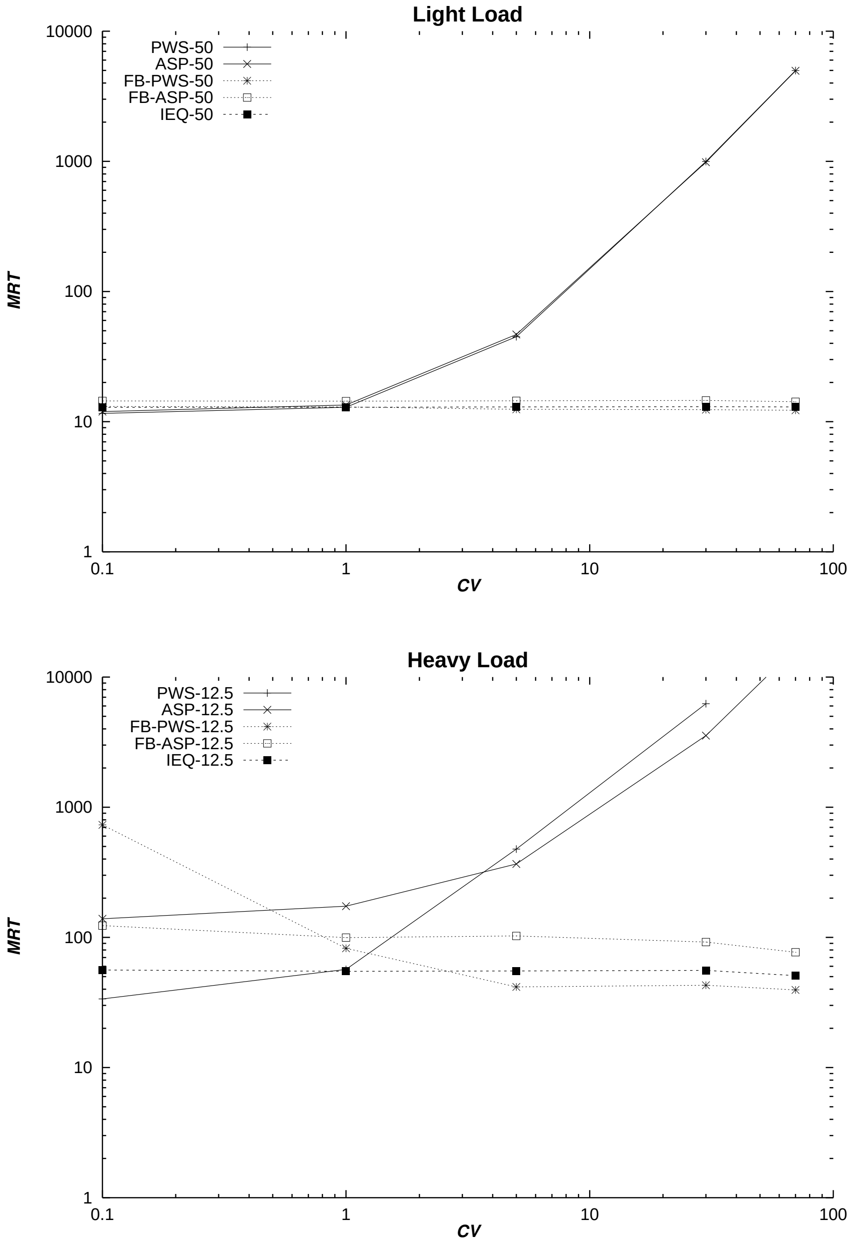


Figure 3.3: Relative performance of scheduling disciplines under workload 1 at light and heavy loads.

of processors as short-running jobs, leading to a lower processor utilization when the short-running jobs leave the system.

3.3.2 Workload 2

As workload 1 is studied in Figures 3.2 and 3.3, the corresponding graphs for workload 2 are shown in Figures 3.4 and 3.5.

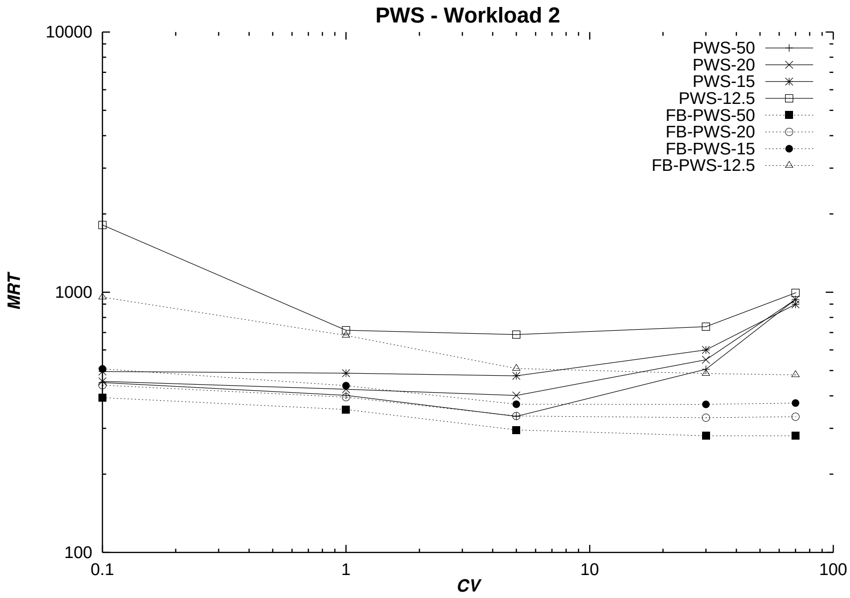
Recall that, in workload 2, long-running jobs (with $w \geq 500\,000$) attain nearly linear (but not unitary) speedup out to 100 processors, but the speedup for short-running jobs (with $w = 500$) reaches a maximum by the point at which five processors are assigned. Although the graphs display similar tendencies as with the first workload, one can observe a number of important differences, primarily due to the different speedup characteristics exhibited by differently sized jobs.

In the graph for PWS, the non-FB version shows a much smaller degradation for high CV as compared to that with workload 1. In fact, at high values of CV, a crossover takes place and mean response time is slightly lower at higher loads than at lower loads. The problem with RTC policies in general is that long jobs delay short jobs for the duration of their execution. What happens in PWS, however, is that long-running jobs tend to receive smaller and smaller partitions as load increases, reducing their negative impact on mean response time.

This behaviour stems from the fact that PWS allocates a job the lesser of its *pws* and the number of free processors. As the load increases, the pending queue gets larger, and processors freed by a departing job are immediately allocated to another. The size of the new partition is no greater than that of the departing job and, as a result, partition sizes tend to only get smaller as time goes on. Under light load, it is quite possible for a long-running job to arrive in a relatively quiet period and monopolize a large proportion of the processors for an extended period of time, but as load increases, this becomes less and less likely. The performance of PWS is poor at CV $= 0.1$ since all jobs are roughly the same size (and thus have the same *pws*); partition sizes never get a chance to decrease in size.

The FB variant of PWS, in this workload, does not offer as great an improvement as with the previous workload. At low loads, PWS has response times more than three times worse than FB-PWS for CV $= 70$. This ratio drops to about two under heavier system load. At lighter load, FB-PWS shows better performance because it can assign a job more processors than can PWS in order to make use of the entire machine.

ASP is quite different from PWS in this workload. Its response times still increase with CV for light loads to the point that, for CV $= 70$, the response times exceed those of higher loads. At higher system loads, response times are insensitive to CV. Since jobs are given what amounts to an equal fraction of processors, ASP behaves much like a round robin system would in this case. Nonetheless, FB-ASP performs better than ASP for all coefficient of variation values. (Note that the curve for FB-ASP at an arrival rate of 20 is very close to that for ASP at a rate of 25.) One reason for this is that ASP partitions processors freed by a departing job quite aggressively. For example, if a job which has 10 processors is completed, and three jobs are pending, the processors are partitioned as 3-3-4.



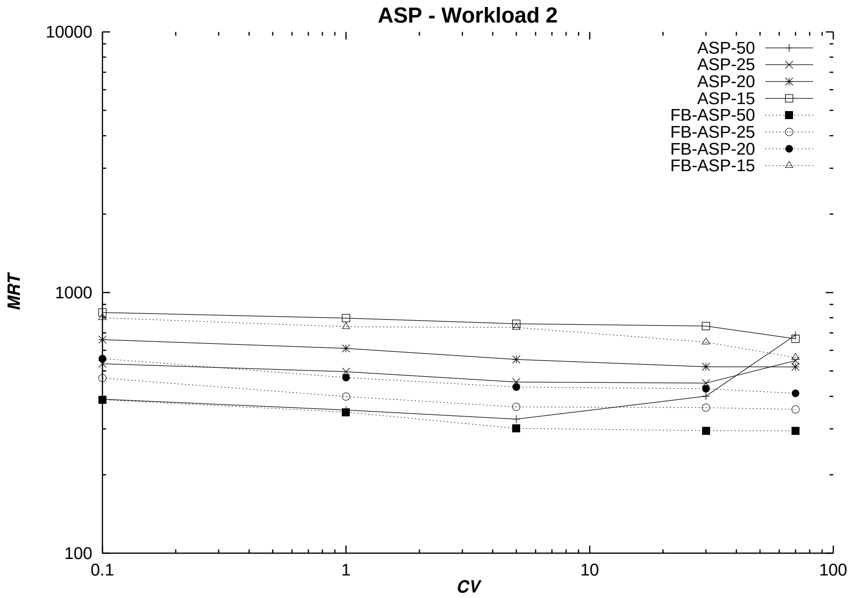


Figure 3.4: Performance of scheduling disciplines under workload 2.

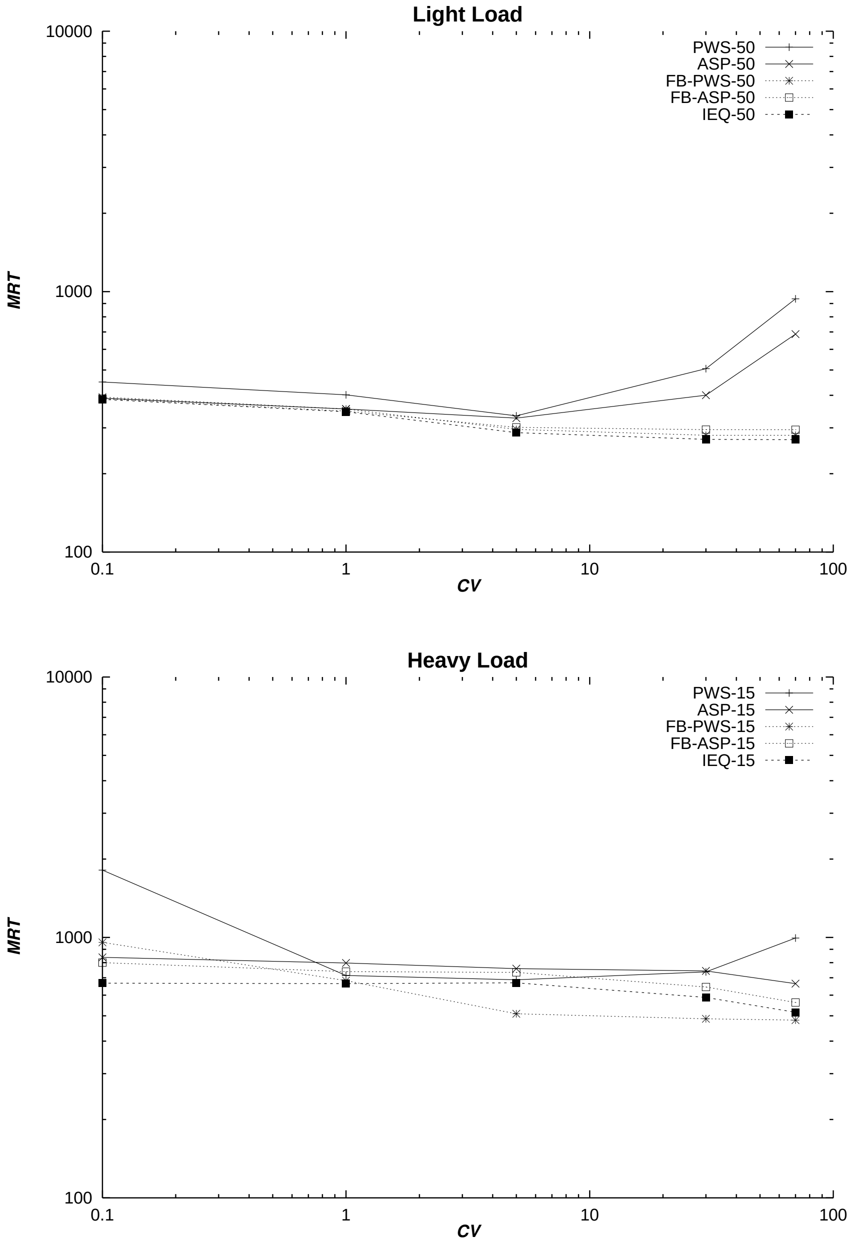


Figure 3.5: Relative performance of scheduling disciplines under workload 2 at light and heavy loads.

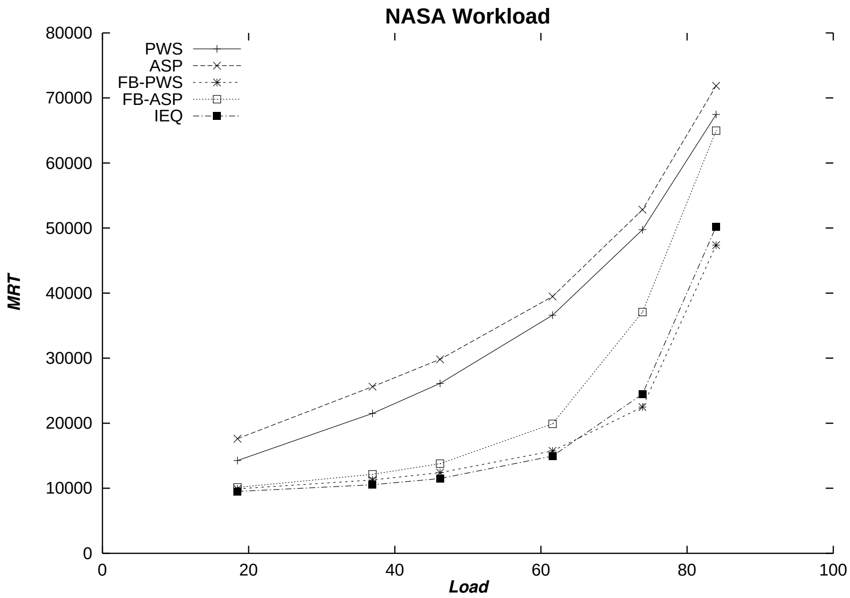


Figure 3.6: Performance of the scheduling disciplines under the NASA workload as a function of system load.

FB-ASP takes a more gradual approach, giving each arriving job a fraction of the processors based on the total number of jobs in the system. Thus, ASP often ends up not allocating enough processors to each job, leaving many processors idle.

The comparison between the scheduling disciplines in Figure 3.5 again shows little performance difference between the feedback-based disciplines and IEQ under light load, but again shows better performance by FB-PWS than IEQ as the load increases for CV > 1. One can observe that IEQ has a decreasing mean response time with increasing CV. This is due to the fact that short-running jobs are greatly restricted in the number of processors they can acquire (e.g., a job of size 500 has a maximum parallelism of 6), resulting in long-running jobs acquiring proportionately more processors than they would without such restrictions. Since long-running jobs have better speedup characteristics, the overall efficiency of the system increases, thus reducing the mean response time as CV increases.

3.3.3 NASA Workload

To obtain a better understanding of how the various disciplines would perform under a more realistic workload, we evaluated the disciplines under the NASA workload described earlier. Since this workload has a fixed coefficient of variation of 7.23, we plot the mean response time as a function of load in Figure 3.6.

As can be seen, FB-PWS and IEQ once again perform very well at all load levels. The benefit of preemption as used by the FB disciplines and IEQ is quite apparent, especially at lower load levels. At about 50% load, the mean response time for PWS is twice that of FB-PWS. Similarly, ASP has mean response time a little more than double that of FB-ASP at the same load level.

3.3.4 Considerations for Distributed Shared-Memory Systems

We are particularly interested in studying migratable feedback-based scheduling schemes because malleable schemes, such as equipartition, may either not yet be supported in systems or may currently be more expensive on distributed shared-memory systems. On such architectures, remote data accesses can be an order of magnitude more expensive than local requests. As a result, moving a thread from one processor to another, for load balancing purposes, can be quite costly. Similarly, reconfiguring a job to run on a different number of allocated processors might require substantial amounts of data movement. For instance, a matrix that has been stored in an interleaved fashion may need to be completely redistributed if the original processor allocation is not a multiple of the new one. In this section, we investigate the effects of such overheads on equipartition.

Recall that in our FB disciplines, we use leftover processors to run jobs that may have been configured for a greater partition size (to avoid having processors remain idle). In the centralized shared-memory architecture, the overhead of doing this was a small factor of 9%; in a distributed shared-memory system without malleable jobs, the overhead is likely to be much larger. We consider the case where a job runs half as fast as it would be expected to (i.e., a slowdown factor of 100%), which is much more conservative than the case where threads can be multiplexed (possibly unevenly) on the remaining processors. We also consider the case where the overhead for equipartitioning is non-zero. Since we do not have any previous work on which we can base a value for the overhead, we consider the cases where the repartitioning overhead is 0.25%, 0.50%, and 0.75% of the mean service demand for jobs.[2]

Figures 3.7 shows the effect of higher slowdown values for the NASA workload, now omitting the lines for the RTC disciplines. FB-ASP seems to be relatively unaffected by the increased slowdown. The degradation in performance relative to the previous model is 1.6% averaged over the range in loads shown. FB-PWS was affected by this change to a greater extent, ranging from 4% at low load to roughly 30% at high load. The reason for this is that FB-PWS allocates much larger partition sizes to long-running jobs than FB-ASP does, which causes these jobs to run more frequently in "leftover partitions" at half the speed. But IEQ is affected quite severely by repartitioning overhead, especially at the two higher levels. For an overhead of 0.75%, equipartitioning has a mean response time up to almost twice that of FB-PWS.

3.4 Conclusions

In this chapter, we have examined the sensitivity of various scheduling disciplines to highly variable service demands and proposed new preemptive disciplines that have much better performance characteristics when the coefficient of variation for the service-demand distribution is large. Our primary focus is where this coefficient of variation (CV) ranges from 5 to 70, as has been observed at various

[2]This is not intended to be representative of actual overheads, but merely a way to examine how overheads can affect IEQ.

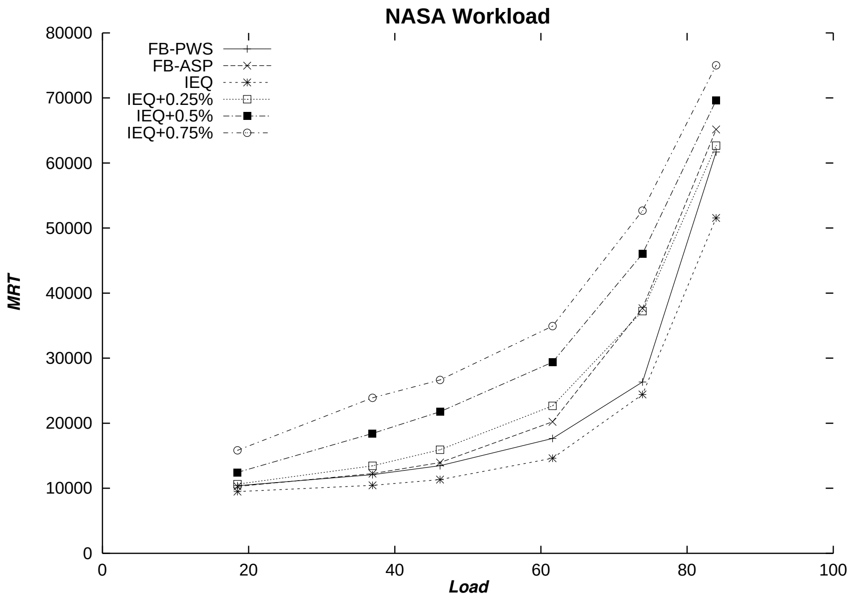


Figure 3.7: Effect of increasing scheduling overheads on the NASA workload, both for the feedback-based and the IEQ disciplines.

high performance computing centers.

When jobs have nearly perfect speedup characteristics (e.g., workload 1), the behaviour of the various disciplines are relatively consistent with the corresponding uniprocessor results. The performance of run-to-completion (RTC) disciplines degrades severely as the CV increases, while the performance of preemptive multilevel feedback (FB) disciplines improves. As expected, the RTC and FB disciplines yield comparable response times at CV $= 1$, but at CV $= 70$, their response times differ by two orders of magnitude.

When, on the other hand, short-running jobs have poor speedup and long-running ones good speedup (e.g., workload 2), there are a number of interesting differences. Because of the limit on the number of processors that can be allocated to poor-speedup jobs, the RTC disciplines become more round-robin in nature, displaying a fair amount of insensitivity to the CV at higher loads. The improvements gained by using FB variants of these is substantially reduced relative to the near-perfect speedup case, but even in this case, these disciplines led to at least a factor of four reduction in mean response times. In every workload we considered, the feedback-based disciplines (in particular FB-PWS) proved to be competitive with ideal equipartitioning (IEQ).

Our results can be summarized as follows:

- IEQ does very well in all our experiments with high CV. The corresponding practical discipline, EQ, is a good discipline to use if it can be implemented without incurring excessive overhead from (1) frequent preemptions, (2) loss of cache affinity when threads are moved from one processor to another, and (3) restructuring of jobs to adapt to changes in processor allocations. However, in large multiprocessors with physically distributed memory modules, coordinated placement of data and threads is critical, so overhead (3) is likely to be large in most cases today. In this case, the feedback-based disciplines may be more attractive than EQ disciplines due to their simplicity and high level of performance.

- If estimates of the *pws* for each job are available for use in scheduling, then FB-PWS does as well as IEQ at keeping response times low for high CV. This is somewhat surprising since FB-PWS must commit to a partition size for each job when it is activated, while IEQ does not.
- If only an estimate of each job's maximum parallelism is available for use in scheduling, then FB-ASP is the best rule to use, although additional knowledge can make a significant difference (as shown by FB-PWS).

Although the need for preemption is intuitive, most adaptive, non-malleable disciplines that have been proposed (see Figure 2.1) assumed that jobs should be run to completion. This can be partly attributed to the lack of workload studies describing service-demand distributions in production centers and the corresponding lack of evidence that preemption was necessary. The work by Chiang *et al.* was significant in respect [CMV94]. They first provided evidence of the high degree of variability in service demands at high-performance computing centers (i.e., coefficients of variation ranging from 5 to 70). Then, they showed that a simple way to reduce mean response times is to limit the number of processors that could be allocated to any given job, thereby reducing the chance that all processors are occupied by long-running jobs. For their workloads, they find that limiting allocations to 20% of the total number of processors to be best. This type of hard limit is impractical at lighter loads or if we take into consideration the memory requirements of jobs (as in the next two chapters). They also consider the benefits of limited (one-time) preemption, using malleable preemption. They observe some reduction in mean response time, but do not obtain as good performance as IEQ.

In this chapter, we have examined the benefits of preemption much more thoroughly. In particular, we have combined migratable preemption with two well-known adaptive (run-to-completion) disciplines, and compared the performance of these new disciplines against both the base disciplines and IEQ over a wide range of workloads. We have demonstrated clearly that, as the coefficient of variation in service demands increases, preemption becomes increasingly important. As such, all disciplines proposed in the remainder of the thesis are preemptive in nature.

Chapter 4

Memory-Constrained Scheduling without Speedup Knowledge

4.1 Introduction

We now turn our attention to another critical resource, namely the physical memory requirements of jobs. Past research in multiprocessor scheduling has tended to focus solely on the allocation of processors, even though such memory requirements might also constrain performance. In fact, with current technological trends, processors are less likely to be the bottleneck resource in the future than either memory or I/O. Even with the large memory capacity of new machines, we can expect the combined memory requirements of large-scale scientific applications to exceed capacity in multiprogramming environments [Ast93, AKK$^+$95].

In this chapter, we investigate the coordinated allocation of processors and memory in large-scale multiprocessors (which we term *memory-constrained scheduling*). We derive upper bounds on system throughput when both memory and processors are needed for the execution of each job. These bounds provide, for the first time, some theoretical basis for assessing the performance of memory-constrained scheduling disciplines. Although our primary objective is to minimize mean response time, understanding the throughput bounds is important for two reasons. First, it permits us to relate an arrival rate to the maximal sustainable throughput, enabling us to compare the performance of various disciplines under different workload conditions. Second, and more importantly, it provides new insight into how memory-constrained scheduling disciplines must respond to increases in the load in order to avoid saturation.

Our first result in Section 4.2 applies to non-memory-constrained scheduling when the workload speedup is convex upward (i.e., has monotonically decreasing slope). It shows that an equi-allocation strategy,[1] which we showed in the previous chapter to offer good response times given

[1] Although similar, we do not call this discipline equipartition because it must take into account memory requirements of applications; we use equi-allocation as a more generic term to refer to disciplines that strive to allocate processors equally among jobs.

sufficiently low overheads, is also optimal from a throughput perspective if no knowledge exists about the speedup characteristics of individual jobs and if memory is abundant.[2] Our subsequent result then shows that the same strategy can also offer near-maximum throughput for the memory-constrained case. We show that, although higher throughputs are theoretically feasible with more sophisticated schedulers, the gains that can be achieved are small and the computational costs high, making an equi-allocation strategy attractive.

Based on these results, we propose a set of memory-processor allocation (MPA) disciplines, the performance of which we evaluate. We simulate these disciplines under a variety of workloads and relate their throughput to the theoretical bounds we derive. We also demonstrate the importance of maximizing memory utilization, as memory packing losses can significantly restrict throughput.

In the next section, we present the derivation of the throughput bounds and examine their implications in greater detail. The simulation results are then presented in Section 4.3, and our conclusions in Section 4.4.

4.2 Bounds on the Throughput

In deriving the analytic bounds on throughput, we first examine the maximum sustainable throughput with respect to processors (assuming unbounded memory) and then do the same with respect to memory (assuming unbounded processors). For the latter, the distribution of memory requirements is significant in the derivation of the throughput bound, so we explicitly consider the case where this distribution is bounded by a minimum and a maximum value.

Each of the upper bounds on throughput due to processing and memory are expressed in terms of the average processor allocation (to be precisely defined shortly). As such, they can be combined to produce bounds of the form illustrated in Figure 4.1. As the average processor allocation increases, the upper bound due to processors decreases because the efficiency of jobs decreases. If all jobs require the same amount of memory, then the upper bound due to memory increases with the average processor allocation because jobs, on average, occupy memory for less time. For an arbitrary distribution of memory requirements, however, the memory bound is constant over all average processor allocations because extreme cases exist where the average processor allocation is irrelevant. The maximum system throughput is obtainable at the average processor allocation value for which the processor and memory bounds intersect.

4.2.1 Job Memory Requirements

In our model, each job i has a distinct memory requirement m_i, corresponding either to the amount of physical memory required by the job or, in a distributed-memory system, to its minimum processor

[2] In saying that no knowledge is known about the speedup characteristics of individual jobs, we imply that there is no statistical correlation between the memory requirements of jobs and their speedup characteristics.

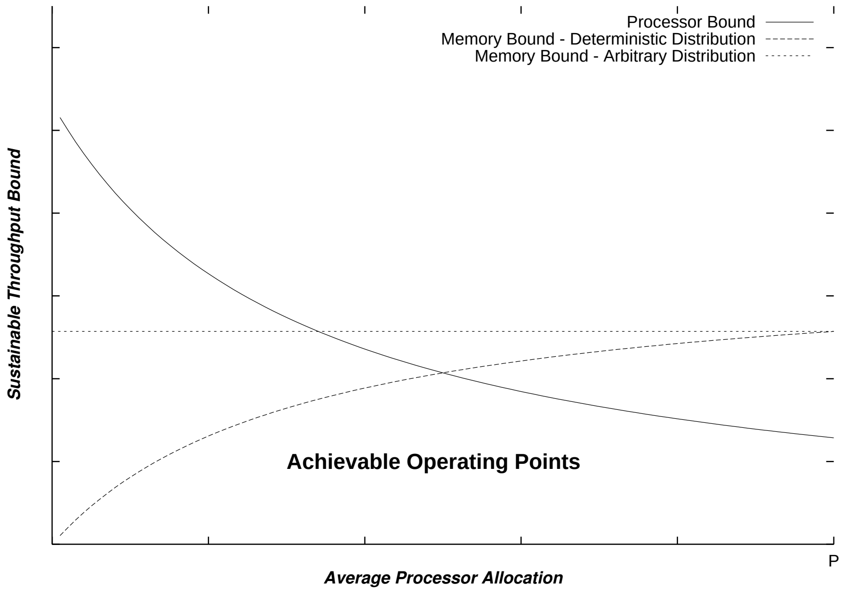


Figure 4.1: Example of upper bounds on throughput for the case where the average memory requirement is half of total memory. (The area beneath all three curves represents operating points at which the system is not saturated.)

allocation. We believe that using a constant value m_i for each job[3] is a realistic simplification. Recent studies of some parallel applications indicate that the performance of each decreases significantly if it does not have its entire data set available in memory [PSN94, BHMW94]; any non-negligible level of paging results in synchronization delays, similar to those that can occur with thread-oriented dispatching. This means that many parallel scientific applications will have their entire data set loaded into physical memory during computation, and m_i denotes the size of this data.

There are three factors that might affect the use of a constant value for job memory requirements. First, if an application uses dynamic memory allocation, then its memory requirements will vary over time. The approach taken in the Tera MTA is to reserve a certain amount of memory for dynamic memory allocation [AKK$^+$95], but leaving memory unused for this purpose can reduce the performance of the system, as will be shown in Section 4.3.3. Since the Tera MTA does not support paging, a job is immediately swapped out if there is insufficient memory to satisfy a memory allocation request, but in a more conventional system, it may be possible to rely on paging to tolerate transient memory overcommitments.

The second factor that might affect the use of a constant value m_i is that an application may go through several phases during its computation, each phase requiring different data structures to be resident in memory. In this case, it is possible to treat a job as several distinct sub-jobs, each having different memory requirements. However, this approach places ordering constraints among the sub-

[3]Using a constant value is only an issue in shared-memory systems, not distributed-memory systems where memory allocation increases linearly with processor allocation.

jobs, which is not explicitly considered in the analysis that follows. Alternatively, the system can reserve a certain amount of memory for small variations in memory requirements between phases. If memory requirements differ significantly between phases, then the scheduler might treat a phase transition as an opportunity to swap the job out of memory and schedule another one.

The third factor is that paging may become acceptable in parallel computing, perhaps through the aggressive use of prefetching. In this case, the execution time of a job will be a function of both the memory and processors allocated to a job. Moreover, a job's working set size may increase with the number of processors allocated to a job, which means that memory and processor allocation are not independent variables. Further work is required to establish the viability of paging for parallel computing and the effects it will have on execution times.

Since the prevalent assumption today is that paging is not suitable for parallel applications, and variability in memory requirements (either due to dynamic memory allocation or to phase transitions) have yet to be shown to be significant, we assume that the amount of memory associated with a job is constant over its lifetime. If, in the future, this assumption is no longer appropriate, then it will be necessary to characterize both the effects of paging on execution times and the variations of memory requirements that occur in parallel applications.

4.2.2 Processor Bound

Let there be a finite set of job classes $\mathcal{C}_T$ distinguished by an execution-time function, $T_\chi(w,p)$, representing the execution time on p processors of a job from class $\chi \in \mathcal{C}_T$ having service demand w. (In other words, for each job i, there exists a job class χ, $\chi \in \mathcal{C}_T$, such that $T_i(w_i,p) = T_\chi(w_i,p)$ for all p.) Let the fraction of jobs generated for each class be f_χ and let the statistical distribution of work for the jobs in class χ be $b_\chi(w)$. The average execution time of the workload on p processors is thus

$$T(p) = \sum_{\chi \in \mathcal{C}_T} f_\chi \int b_\chi(w) T_\chi(w,p)\, dw$$

We assume that all execution-time functions $T_\chi(w,p)$ are concave and monotonically decreasing; given that $T(p)$ is merely an integral over such functions, it itself is a concave, monotonically decreasing function.[4]

We now define the *workload speedup* to be $S(p) = T(1)/T(p)$, which will be a convex upward function, given the characteristics of $T(p)$. Jobs can thus have different speedup characteristics, but we assume that the scheduler cannot distinguish jobs by their class, either upon arrival or during their

[4] $T(p)$ can be differentiated with respect to p,

$$\frac{d}{dp}T(p) = \sum_{\chi \in \mathcal{C}_T} f_\chi \int b_\chi(w) \frac{\partial}{\partial p} T_\chi(w,p)\, dw$$

(and similarly for the second partial derivative); since all $T_\chi(w,p)$ have negative first derivative and positive second derivative, the same will be true of $T(p)$.

execution.[5]

A *processor allocation* assigns to each job some number of processors in the range $[0, P]$. Let the fraction of jobs to which the processor allocation assigns p_j processors be r_j. Since the scheduler cannot distinguish between job classes, the *average time-weighted processor allocation*, $\overline{p}$, is given by

$$\overline{p} = \frac{\sum_j p_j r_j T(p_j)}{\sum_j r_j T(p_j)} \tag{4.1}$$

(If the number of processors allocated to a job varies over time, then it can be treated as several separate job portions, each having a constant allocation.)

Proposition 4.1 *If $\overline{p}$ is the average time-weighted processor allocation, then the maximum throughput is bounded above due to processor availability by*

$$\frac{P}{\overline{p}T(\overline{p})} \tag{4.2}$$

where P is the number of processors in the system and $T(p)$ is the workload speedup.

Proof: First, observe that maximizing the system throughput corresponds to minimizing the average processor occupancy of the system, which is $\theta_p = \sum_j p_j r_j T(p_j)$. Our approach is to prove (by contradiction) that, if $\overline{p}$ is the average processor allocation, then θ_p is minimized when all jobs are allocated $\overline{p}$ processors.

Let there be an optimal processor allocation (with respect to processor occupancy) such that some jobs are allocated p_l processors while others are allocated p_m processors, $p_l < \overline{p} < p_m$. Such an allocation pair must exist unless all jobs are allocated exactly $\overline{p}$ processors.

Choose a fraction s_l such that

$$\frac{p_l s_l r_l T(p_l) + p_m(1-s_l) r_m T(p_m)}{s_l r_l T(p_l) + (1-s_l) r_m T(p_m)} = \overline{p} \tag{4.3}$$

Such a value of s_l must exist in the interval $[0, 1]$ by the intermediate value theorem of calculus. We now let a fraction s_l of jobs which were allocated p_l processors be allocated $\overline{p}$ processors, and the same for a fraction $s_m = (1 - s_l)$ of jobs which were allocated p_m processors, again assuming that the scheduler cannot distinguish between job classes. By equations (4.1) and (4.3), the average time-weighted processor allocation remains unchanged from this reallocation:

$$\begin{aligned}
&\frac{\sum_c r_c T(p_c) p_c - [p_l s_l r_l T(p_l) + p_m s_m r_m T(p_m)] + [\overline{p} s_l r_l T(\overline{p}) + \overline{p} s_m r_m T(\overline{p})]}{\sum_c r_c T(p_c) - [s_l r_l T(p_l) + s_m r_m T(p_m)] + [s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]} \\
&\quad = \frac{\overline{p}\sum_c r_c T(p_c) - \overline{p}[s_l r_l T(p_l) + s_m r_m T(p_m)] + \overline{p}[s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]}{\sum_c r_c T(p_c) - [s_l r_l T(p_l) + s_m r_m T(p_m)] + [s_l r_l T(\overline{p}) + s_m r_m T(\overline{p})]} = \overline{p}
\end{aligned}$$

[5] For example, we do not allow a scheduler to infer the class of a job by the amount of time it has executed thus far.

The contribution to the average processor occupancy of the jobs selected from the original allocation is $\theta_{p,1} = p_l s_l r_l T(p_l) + p_m s_m r_m T(p_m)$, which by (4.3) can be rewritten as $\theta_{p,1} = \overline{p}[s_l r_l T(p_l) + s_m r_m T(p_m)]$. When allocated $\overline{p}$ processors, these jobs contribute $\theta_{p,2} = \overline{p}(s_l r_l + s_m r_m)T(\overline{p})$ to the average processor occupancy. We show that the ratio $\theta_{p,1}/\theta_{p,2}$ is greater than one, which implies that the average processor occupancy under the new allocation is smaller than under the original one. This contradicts the presumption that the original processor allocation is optimal.

$$\begin{aligned}
\frac{\theta_{p,1}}{\theta_{p,2}} &= \frac{s_l r_l T(p_l) + s_m r_m T(p_m)}{(s_l r_l + s_m r_m)T(\overline{p})} \\
&= \frac{1}{s_l r_l + s_m r_m} S(\overline{p}) \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)}\right]
\end{aligned}$$

Since $S(p)$ is convex upward, its value at $\overline{p}$ is greater than the linear interpolation between the points $(p_l, S(p_l))$ and $(p_m, S(p_m))$ on the workload speedup curve:

$$\frac{\theta_{p,1}}{\theta_{p,2}} > \frac{1}{s_l r_l + s_m r_m} \left[S(p_l) + (\overline{p} - p_l)\left(\frac{S(p_m) - S(p_l)}{p_m - p_l}\right)\right] \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)}\right] \tag{4.4}$$

Given $T(p) = T(1)/S(p)$, we can rewrite equation (4.3) as:

$$\overline{p} = \frac{p_l s_l r_l S(p_m) + p_m s_m r_m S(p_l)}{s_l r_l S(p_m) + s_m r_m S(p_l)}$$

Substituting into (4.4) and performing some algebraic manipulation, we obtain $\theta_{p,1}/\theta_{p,2} > 1$:

$$\begin{aligned}
\frac{\theta_{p,1}}{\theta_{p,2}} &> \frac{1}{s_l r_l + s_m r_m} \left[S(p_l) + \left(\frac{(p_m - p_l) s_m r_m S(p_l)}{s_l r_l S(p_m) + s_m r_m S(p_l)}\right)\left(\frac{S(p_m) - S(p_l)}{p_m - p_l}\right)\right] \left[\frac{s_l r_l}{S(p_l)} + \frac{s_m r_m}{S(p_m)}\right] \\
&= \frac{S(p_l)}{s_l r_l + s_m r_m} \left[1 + \frac{s_m r_m [S(p_m) - S(p_l)]}{s_l r_l S(p_m) + s_m r_m S(p_l)}\right] \left[\frac{s_l r_l S(p_m) + s_m r_m S(p_l)}{S(p_l) S(p_m)}\right] \\
&= \frac{1}{s_l r_l + s_m r_m} \left[\frac{s_l r_l S(p_m) + s_m r_m S(p_l) + s_m r_m S(p_m) - s_m r_m S(p_l)}{S(p_m)}\right] = 1
\end{aligned}$$

If every job is allocated exactly $\overline{p}$ processors, then the average processor occupancy is $\overline{p}T(\overline{p})$. If there are P processor-time units available per unit time, then the maximum achievable throughput is $P/(\overline{p}T(\overline{p}))$. (This bound can be attained only if P is a multiple of $\overline{p}$.) □

This proof shows is that, if no information is known about the relative speedups of individual jobs and the workload speedup is convex upward, then an equi-allocation strategy for processors will maximize the sustainable throughput at heavy load. It has already been shown experimentally that equipartition yields good response times for a variety of workloads; Proposition 4.1 provides a theoretical basis for why it is also the best discipline to maximize throughput, enabling it to also yield good response times at high load.

4.2.3 Memory Bounds

Next, we address the corresponding bounds on throughput imposed by memory rather than by processors. While the processor bound depends only on the mean service demand, as captured in $T(p)$, the memory bound depends on the distribution of memory requirements. We start by addressing the simplest special case, where all jobs have the same memory requirement, $\hat{m}$.

Proposition 4.2 *If $\overline{p}$ is the average time-weighted processor allocation, and $\hat{m}$ is the amount of memory required by each job, then the maximum throughput is bounded above due to memory availability by*

$$\frac{M}{\hat{m}T(\overline{p})} \tag{4.5}$$

where M is the total amount of memory in the system and $T(p)$ is the workload speedup.

Proof Overview: The proof is similar to that for Proposition 4.1, except that we need to minimize the average memory occupancy instead of average processor occupancy. By equation (4.1), minimizing the average processor occupancy for a given average processor allocation, $\overline{p}$, is equivalent to minimizing the average execution time, which is equivalent to minimizing the average memory occupancy (since all jobs require the same amount of memory). In the last step of the proof, we have M memory-time units available per unit time and an average memory occupancy of $\hat{m}T(\overline{p})$, which constrains throughput to be at most $M/\hat{m}T(\overline{p})$. □

The realistic case, where jobs have different memory requirements, is more complex. Ignoring memory packing constraints, maximizing the throughput is equivalent to minimizing the average memory occupancy per job. In general, this is achieved by giving smaller jobs fewer processors and larger jobs more processors. Restricting memory requirements to the range m_L to m_U, where $0 \leq m_L \leq m_U \leq M$, permits us to represent a continuum of situations from that of arbitrary memory requirements ($m_L = 0$ and $m_U = M$) to that of identical memory requirements ($m_L = m_U$). Through experimentation, we found that, even for quite small values of m_L/M, the restriction that memory requirements lie in the interval $[m_L, m_U]$ meant that the bound on throughput was very close to that given by Proposition 4.2.

The next proposition shows how the throughput of a system S, in which jobs have arbitrary memory requirements in the interval $[m_L, m_U]$, cannot be any better than a related system S' in which all jobs require either m_L or m_U memory units. This proposition is useful because (1) it is relatively simple to compute the throughput bound for S', and (2) the bound applies to a wide range of distributions of memory requirements (although it is tight only in the case where all memory requirements are either m_L or m_U.)

First, assume that we can classify jobs according to their memory requirements, in addition to the previous classification based on execution-time function. Each memory class must have the same distribution of execution-time function classes because otherwise, the memory requirements of a job yields information regarding the relative speedup of a job that could be used by the scheduler. However, each memory class ω can account for a different fraction, g_ω, of the job arrivals. Thus, the fraction of jobs that belong to both execution-time function class χ and memory class ω is $f_\chi g_\omega$.

We can now define the *average work-weighted memory requirement*, $\overline{m}$, to be the amount of memory required by a job of class ω, m_ω, weighted by the fraction of jobs in that class: $\overline{m} = \sum_\omega g_\omega m_\omega$. If all jobs require the same amount of memory, $\hat{m}$, then $\overline{m} = \hat{m}$.

Proposition 4.3 *Let S be a system having a finite set of job classes, where the memory requirement corresponding to class ω is m_ω and in which the average work-weighted memory requirement is $\overline{m}$. Let m_L and m_U be the minimum and maximum memory requirement of jobs, respectively, in S. Then the throughput of this system for a particular average time-weighted processor allocation $\overline{p}$ is bounded above by that of another system S' having the same average work-weighted memory requirement of $\overline{m}$, but only two job classes, the memory requirements of which are m_L and m_U, respectively. Preserving the value of $\overline{m}$, the fraction of work, g_L, corresponding to the m_L class in S' is $g_L = (m_U - \overline{m})/(m_U - m_L)$.*

Proof: Let there be an optimal processor allocation with respect to memory occupancy for jobs in each memory class. Consider each job class ω in turn. We construct system S' by transforming the jobs of class ω, which have memory requirement m_ω, into jobs of either size m_L or of size m_U, without changing the values of $\overline{m}$ and $\overline{p}$.

Let a fraction $s_{L,\omega} = \frac{m_U - m_\omega}{m_U - m_L}$ of the jobs in memory class ω in S be of size m_L in S' and the remaining fraction, $s_{U,\omega} = 1 - s_{L,\omega}$, be of size m_U in S'. Let these jobs execute with the same number of processors in S', thus leaving the average time-weighted processor allocation unchanged. The average work-weighted memory requirement $\overline{m}$ of the jobs in S' also remains unchanged because the contribution of work-memory requirement is the same as before, as is the total amount of work in the system. Since the fraction of jobs generated for memory class ω is g_ω, the fraction of work-memory requirement in the original class is $m_\omega g_\omega$ and in the transformed class it is $s_{L,\omega} m_L g_\omega + s_{U,\omega} m_U g_\omega$, which are equal due to the definition of $s_{L,\omega}$. The average memory occupancy also remains the same by a similar argument as for the average work-memory requirement.

Thus, S' has the same average time-weighted processor allocation and the same average work-weighted memory requirement as S, but has an optimal processor allocation that is guaranteed to be no worse than that of S. From the transformation, the total fraction g_L of work associated with the m_L class is

$$g_L = \sum_\omega g_\omega \frac{m_U - m_\omega}{m_U - m_L} = \frac{m_U - \overline{m}}{m_U - m_L}$$

□

For a system with exactly two memory-requirement classes, the optimal processor allocation for each of these classes at a given value of $\overline{p}$ can be determined by minimizing the average memory occupancy. The optimization method is based on the following proposition (which has been generalized for an arbitrary number of memory classes):

Proposition 4.4 *For any system, the optimal processor allocation with respect to memory occupancy for a given average processor allocation ($\overline{p}$) is obtained when all jobs within any memory class are allocated the same number of processors.*

Proof: Let there be a processor allocation which minimizes the memory occupancy. We will show by contradiction that if all jobs in each memory class are not allocated the same number of processors, then an allocation with lower memory occupancy can be found.

Assume that the fraction of jobs allocated p_i processors in memory class ω is $r_{i,\omega}$. Since the fraction of jobs generated for memory class ω is g_ω, the average processor allocation is, by definition,

$$\overline{p} = \frac{\sum_\omega g_\omega \sum_i p_i r_{i,\omega} T(p_i)}{\sum_\omega g_\omega \sum_i r_{i,\omega} T(p_i)}$$

Now, assume that the jobs of some memory class ω are given varying processor allocations, and let this class' average time-weighted processor allocation be $\overline{p_\omega}$:

$$\overline{p_\omega} = \frac{\sum_i p_i r_{i,\omega} T(p_i)}{\sum_i r_{i,\omega} T(p_i)} \tag{4.6}$$

From the proof of Proposition 4.1, we know that the processor allocation that minimizes the average execution time of jobs in class ω is one which assigns all jobs $\overline{p_\omega}$ processors. Therefore, the processor allocations that are actually assigned to jobs in class ω will lead to an average execution time $\sum_i r_{i,\omega} T(p_i) = \gamma_\omega T(\overline{p_\omega})$, for some $\gamma_\omega > 1$.

Since $T(p)$ is monotonically decreasing with p, we can choose a value $\overline{q_\omega} < \overline{p_\omega}$ such that $T(\overline{q_\omega}) = \gamma_\omega T(\overline{p_\omega})$. If we allocate $\overline{q_\omega}$ to all jobs in class, then the average processor allocation $\overline{p}$ for the system will decrease because the denominator remains the same ($\sum_i r_{i,\omega} T(p_i) = T(\overline{q_\omega})$) while the numerator decreases (since from (4.6) we can see that if the left decreases and the denominator on the right remains the same, then the numerator must decrease).

The average memory occupancy, which is

$$\sum_\omega m_\omega g_\omega \sum_i r_{i,\omega} T(p_i)$$

remains the same after the adjustment, on the other hand, since $\sum_i r_{i,\omega} T(p_i) = T(\overline{q_\omega})$. Now, any increase to the processor allocation of some job in class ω, to return $\overline{p}$ to its original value, will cause the average memory occupancy to decrease (if $m_\omega > 0$), contradicting our initial presumption of optimality. $\square$

Determining the optimum processor allocation for a given value of $\overline{p}$ corresponds to the following problem:

$$\text{Minimize: } \theta_m(p_L, p_U) = m_L g_L T(p_L) + m_U(1 - g_L) T(p_U)$$

subject to the constraint that

$$\frac{g_L p_L T(p_L) + (1 - g_L) p_U T(p_U)}{g_L T(p_L) + (1 - g_L) T(p_U)} = \overline{p}$$

This problem has a real solution for p_L and p_U in the interval $[0, P]$. If we assume we have perfect packing of jobs into memory of size M, then our throughput bound is $M/\theta_m(p_L^*, p_U^*)$ where p_L^* and p_U^* are the optimal processor allocations for the two memory classes, such that the above constraint involving $\overline{p}$ is still satisfied.

4.2.4 Application of the Bounds

To illustrate the bounds established in the previous sections, we use a Dowdy-style execution-time function, since the Sevcik function does not meet our convexity conditions. The Dowdy-function parameter s that we use has been chosen somewhat arbitrarily to be 0.06 (corresponding to one application studied by Setia [ST93]); the actual value chosen has little importance since this is just an illustration.

By varying the total number of processors, P, in the system, however, we can use the same execution-time function to study the throughput bounds given both a good and a poor speedup. Intuitively, this corresponds to the application having good speedup when run on a small machine or for a large problem size, but poor speedup when run on a large machine or for a small problem size. The two values of P that we consider are $P = 100$, which leads to a maximum speedup of 15.0, and $P = 16$, which leads to a maximum speedup of 8.5.

In Figure 4.2, the throughput bounds are plotted as a function of the average time-weighted processor allocation for $P = 100$. In the first graph, the ratio of the average work-weighted memory requirement to total memory, $\overline{m}/M$, has been set to 0.25 while in the second, it has been set to 0.50, with no statistical correlation between memory and service demands.

The simple bounds on throughput obtained from Propositions 4.1 and 4.2 meet where $\overline{p}_o = \frac{\overline{m}}{M}P$, with a resulting throughput bound of $P/(\overline{p}_o T(\overline{p}_o))$. (In the case of the simple memory bound, we hereafter assume that $\hat{m} = \overline{m}$.) Also shown in the graph are the bounds obtained from Proposition 4.3, for values of $m_L/M = \{0, 0.1, 0.2\}$. For $m_L/M = 0.1$, the throughput bound for an arbitrary memory-requirement distribution in the interval $[m_L, M]$ is already quite close to that given by Proposition 4.2; as m_L/M increases, the throughput bound further approaches that in which all jobs have the same memory requirement. As a result, the throughput achievable for arbitrary distributions of memory requirements is only marginally higher than that achievable in the identical-memory-requirement case unless m_L is very small relative to M.

These arbitrary-distribution bounds are for the most favourable distributions of memory requirement within the constraint that all memory requirements are in the interval $[m_L, M]$. For less ideal distributions, such as a uniform distribution, the throughput bounds are even closer to the bound of Proposition 4.2. Also, it should be noted that these curves are generated independently, in that for a given value of $\overline{p}$, the allocation needed to attain the processor bound (i.e., equi-allocation) may

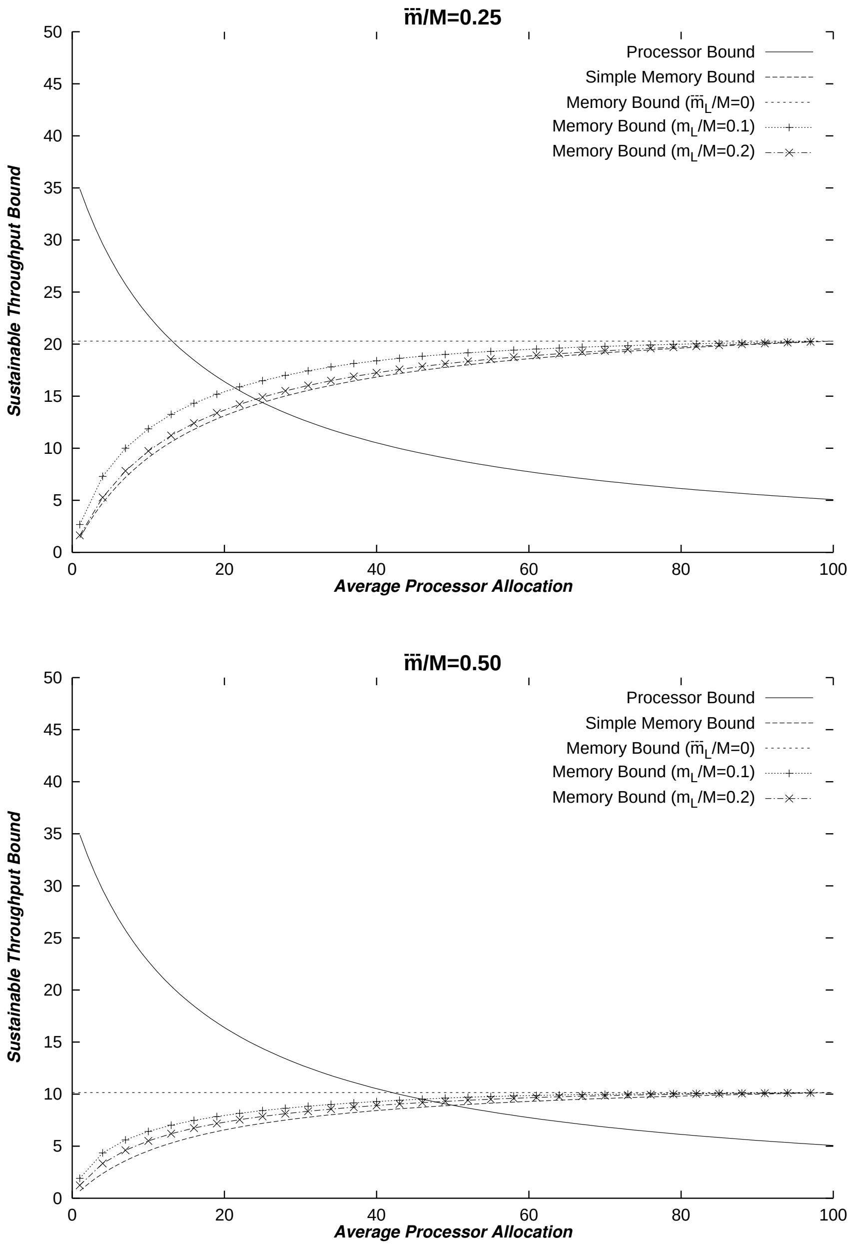


Figure 4.2: Throughput bounds for $P = 100$ (poor speedup).

differ from that needed to attain the memory bound. If the latter allocation were to be chosen, then this would have a negative impact on the processor bound, since we would no longer be using the equi-allocation strategy. As a result, the actual combined bound, where the same allocation must be used for both bounds, will be even closer to the case where all memory sizes are the same.

In the second graph in Figure 4.2, we can observe the effects of increasing $\overline{m}$. Since the slope of the processor bound decreases in absolute value with $\overline{p}$, the gains that can be achieved from sophisticated scheduling disciplines become smaller. In this particular situation, a relatively large range of average processor allocations, from about 50 to 65, will yield close to the same maximal throughput.

Consideration of Propositions 4.1 to 4.3 leads to three significant results:

1. *It is vital that memory requirements be considered in memory-constrained scheduling disciplines.* In non-memory-constrained scheduling disciplines, the average processor allocation must decrease as the load increases up to the point where, at the heaviest loads, each job receives exactly one processor. The graphs in Figure 4.2 show that, in memory-constrained scheduling, one must also reduce the average processor allocation as the load increases, but only up the point where the processor and memory throughput bounds meet. If the average processor allocation decreases beyond this point, then memory becomes the primary constraint on throughput. As can be seen in the graphs, allocating too many processors on average is bad, but allocating too few is worse, as the bound on throughput drops rapidly for small average processor allocations.

2. *When no information is known about speedups of jobs, an equi-allocation strategy is likely to be a good memory-constrained discipline.* From the proofs of Proposition 4.1 and 4.2, the point at which the processor and simple memory bounds meet corresponds to an equi-allocation scheduling discipline that attempts to always run $M/\overline{m}$ jobs at a time. We have found that determining a processor allocation that improves the throughput bound (as is shown to be possible by Proposition 4.3) is computationally expensive and highly dependent on the speedup characteristics of the workload. An equi-allocation strategy, on the other hand, is independent of the workload speedup function, leads to near-maximum throughput, and is already known empirically to yield good response times.

3. *Knowledge of the speedup characteristics of individual jobs is significantly more important in memory-constrained scheduling than in the non-memory-constrained case.* In non-memory-constrained scheduling, knowing speedup information can be helpful in minimizing the mean response time, but cannot improve the maximum throughput of the system. The reason is that maximizing efficiency is achieved by allocating each job one processor, regardless of the workload speedup. In memory-constrained scheduling, however, it is necessary to try to maximize the processor efficiency for a given average processor allocation. If the relative speedups of individual jobs were known, then the scheduler could better allocate processors (generally in a non-equi-allocation manner) to increase the overall efficiency, and thus the throughput. This topic is the focus of the next chapter.

Figure 4.3 presents curves corresponding to those in Figure 4.2, for a smaller value of P. As previously mentioned, this corresponds to the case where jobs exhibit relatively good speedup. Because the processor throughput bound is much flatter than for $P = 100$, sophisticated scheduling disciplines that try to optimize for memory throughput will yield only a more limited increase in terms of total system throughput. Notice that it is even more important than before to allocate sufficient processors to jobs on average, as the throughput bound due to memory drops more rapidly than the bound due to processors.

4.3 Scheduling Disciplines

In this section, we present a simple memory-constrained scheduling discipline that can be used for both shared-memory and distributed-memory architectures. We evaluate how well it performs as the load increases towards the throughput bounds. We then discuss how we can improve upon this discipline, in particular by improving the average memory utilization, which effectively increases the throughput bound (hence decreasing mean response times at high loads).

For the purposes of this simulation, we assume that jobs are malleable with no overhead. (We consider the non-malleable case in the context of the implementation in Chapter 6.) It has been observed that many workloads have sufficiently low arrival rates and long service demands in practice that repartitioning overheads would not be significant [CV96]; if this is not the case, then actual implementations of the proposed disciplines would have to be adapted to make scheduling decisions less frequently, only ever varying the processor allocation of long-running jobs (as in the disciplines described in Chapter 6).

Initially, we investigate the case where all jobs have the same speedup curve, and in Section 4.3.5, we comment on how different speedup curves influence the results.

4.3.1 Workload Model

As others have done in previous studies, we assume a uniform distribution of memory requirements with no correlation to service demand, focussing on values of $\overline{m}/M$ in the set $\{0.1, 0.25, 0.5\}$. The memory requirements are generated from the uniform distribution over the open interval $(0, \frac{2\overline{m}}{M})$. Based on informal evidence, we have found that workstations tend to have workloads in which a large fraction of jobs have small similarly-sized working sets. This is because an overwhelming majority of the jobs are invocations of roughly equally-sized system software commands (e.g., file-system operations). In a large-scale multiprocessor system, with batch submission of jobs, we have found that there is more variability in memory requirement because there is a greater variety of applications; this view is consistent with Feitelson and Nitzbergs' analysis of the NASA Ames workload [FN95]. We conclude that a uniform distribution is no less realistic than other distributions that we might assume.

Interarrival times in our system are drawn from an exponential distribution, and service demands from a hyper-exponential distribution of mean 1000 and coefficient of variation (CV) of 5 (a value

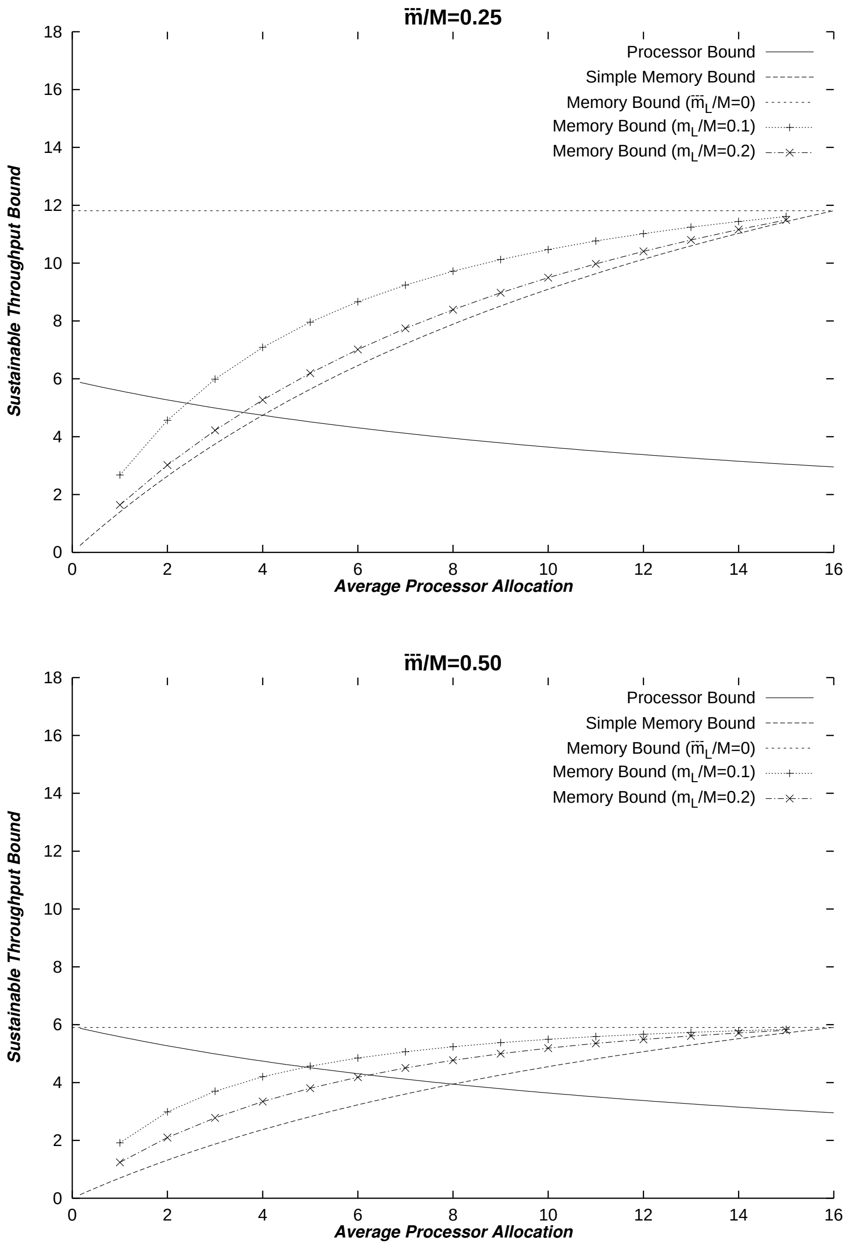


Figure 4.3: Throughput bounds for $P = 16$ (good speedup).

that is consistent with the Cornell Theory Center workload presented in Section 2.3.2). In this section, we use the same execution-time function and parameter as in the previous section; later we consider other execution-time functions. Also, we let $P = 100$ for these simulation experiments.

4.3.2 Scheduling Disciplines

As discussed in the previous chapter, we rely on favouring jobs having the least expected remaining service demand first; given the hyper-exponential service-demand distribution, this is equivalent to running the jobs having the least acquired processing time first.

In a uniform memory access time (UMA) multiprocessor system, the processor allocation can be done independently of a job's memory requirement, as all jobs share the same global memory. In a distributed-memory system, on the other hand, a job's memory requirement may place a lower bound on the number of processors that must be allocated to it. Our baseline memory-processor allocation (MPA) scheduling discipline deals with both cases as follows:

MPA-Basic At each job arrival or departure, or whenever the current quantum has expired, the scheduler re-assesses the jobs to run. It scans the jobs in order of increasing acquired service demand, identifying the ones that can fit in memory (i.e., first fit). In a shared-memory system, it then allocates each selected job the same number of processors. In a distributed-memory system, it first allocates each job its minimum processor allocation and then distributes any remaining processors in such a way as to equalize allocations as much as possible.

4.3.3 Simulation Results

Our MPA-Basic scheduling discipline was simulated using the model and workload parameters described above (Poisson arrival process, hyper-exponential service demand with mean 1000 and CV 5, and uniform memory requirement distribution). As in the experiments in Chapter 3, each trial had a warm-up period in which the first 20 jobs were discarded, and terminated when the subsequent 100 000 jobs had completed. A sufficient number of independent trials were conducted to obtain a 95% confidence interval that was within 5% of the mean.

The load factor on the system is defined to be the ratio between the arrival rate and the throughput bound obtained from Propositions 4.1 and 4.2: $X(\overline{m}) = M/(\overline{m}T(\frac{\overline{m}}{M}P))$. As discussed earlier, this bound does not represent an absolute upper bound on throughput, but rather a good approximation to this value. As such, it is possible for a discipline to handle load factors greater than one, although this did not occur in any of our experiments.

In Figure 4.4, we plot the mean response times of the disciplines for different memory requirements. As shown, smaller average memory requirements lead to a higher overall relative throughput. The reason is that as the average memory requirement decreases, memory is packed more efficiently by the first-fit algorithm. For $\overline{m} = 0.5$, our MPA-Basic discipline only manages to achieve about an 81.5% average memory usage, whereas for $\overline{m} = 0.1$, this value increases to about 94%.

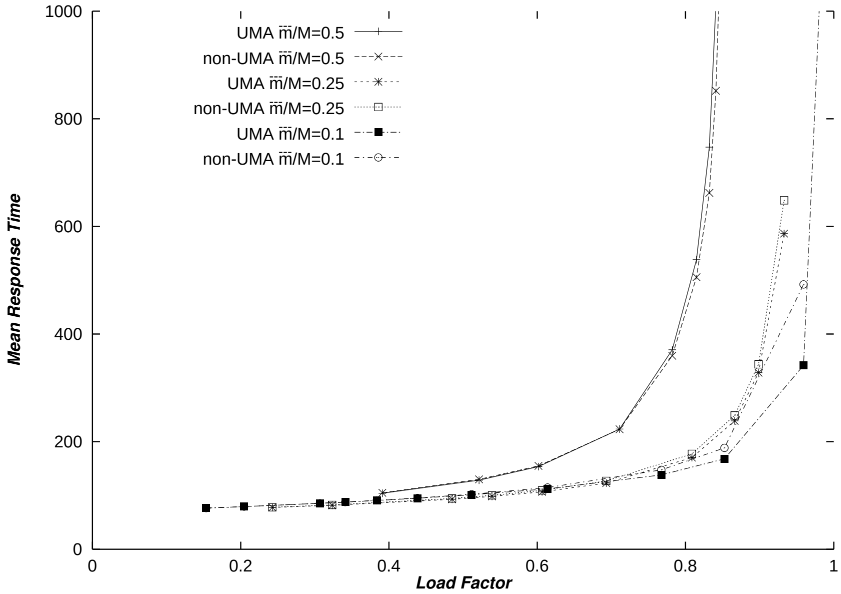


Figure 4.4: Performance of basic UMA and distributed-memory scheduling disciplines for $\overline{m} = \{0.1, 0.25, 0.5\}$.

A remarkable feature of Figure 4.4 is that the distributed-memory constraint ($p_i \geq m_i$) has only a very minor negative impact on performance. Nearly the same throughput is achieved with the distributed-memory constraint as without, and response times are only slightly higher. Although the constraint means that sometimes more processors are allocated to a job than necessary, this appears to be compensated for by the reduction in average memory occupancy. (Recall that, to minimize the average memory occupancy, one must in general give jobs with small memory requirements smaller processor allocations.)

It is possible to use the memory-packing efficiency to strengthen the throughput bound. If a scheduling discipline can use no more than a fraction s of the total system memory, on average, then the bound from Proposition 4.2 can be tightened to

$$\frac{sM}{\overline{m}T(\overline{p})}$$

If we set s to the memory usage observed in our experiments at saturation, then we find that the discipline comes very close to the modified bounds for all memory sizes.

This behaviour is illustrated in Figure 4.5. In this graph, the throughput of the system (which is equal to the arrival rate) is plotted against the average processor allocation exhibited by our discipline, for the case $\overline{m} = 0.25$. The throughput bounds are also plotted for comparison. As expected, the average processor allocation decreases as the load increases until it hits the processor bound. Memory packing losses prevents the discipline from approaching the memory bound. However, using the value of $s = 0.922$ that we observed in our experiments, our scheduling discipline reaches both the processor and the modified memory bound simultaneously.

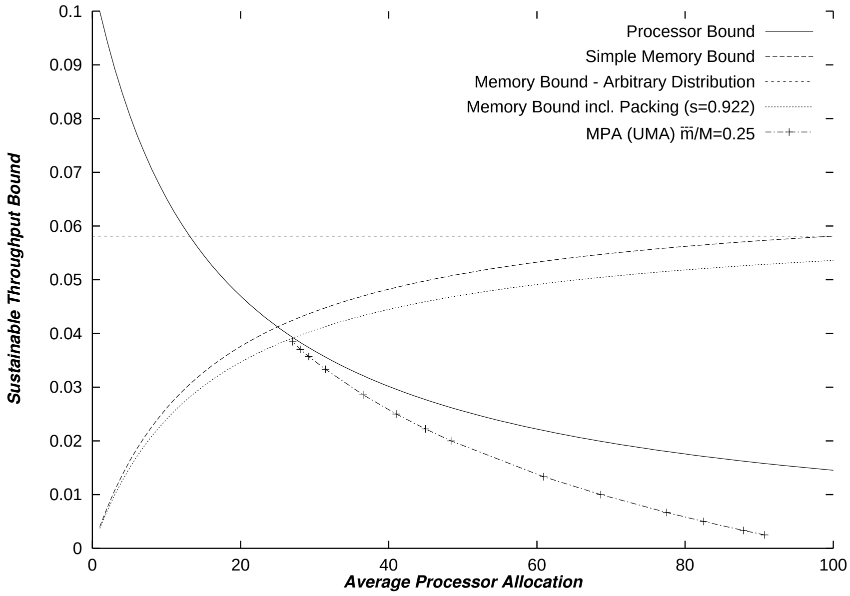


Figure 4.5: Average processor allocation as a function of load for $\overline{m} = 0.25$, overlaid on the throughput bound graph.

4.3.4 Improving the Disciplines

There are essentially two ways of improving the disciplines with respect to the memory throughput. Either the processor allocation can be optimized to reduce the average memory occupancy per job or the average memory utilization can be increased. The former is highly dependent on the workload speedup function involved, and requires considerable computation, so in this section, we concentrate on the latter approach.

We introduce two modifications to MPA-Basic:

MPA-Repl1 As in MPA-Basic, we scan the list of jobs and use first-fit to select the ones to run next. We then attempt to replace the last selected job (only) with another one not selected that achieves a higher memory utilization. The processors are allocated to the selected jobs in the same fashion as in MPA-Basic.

Since jobs with the least acquired processing time are still executed first, this modification does not noticeably degrade the average response times of jobs. But by increasing the average memory utilization, it results in a significant increase in the sustainable load for $\overline{m} = 0.5$, from a load factor of 0.85 to one of 0.89 in the UMA variant of MPA-Basic.

MPA-Pack Once again, we scan the list of jobs (call it R) and use first-fit to select the ones to run next (call this second list R_s). We then attempt to replace jobs in R_s by other jobs in R so as to improve memory utilization, subject to a constraint that a job can only be replaced by another if the latter has an acquired processing time within a factor F ($1 \leq F$) of the former (the choice of which we discuss shortly).

We use a branch-and-bound algorithm to traverse all possible combinations of jobs in R to find the one that leads to the best memory utilization while satisfying the constraint. More precisely, at each node of the decision tree, the partial candidate list of jobs $R_{s'}$ to use instead of R_s must (1) fit in available memory, and (2) have the property that the i^{th} job in $R_{s'}$ has an acquired processing time no greater than F times that of the i^{th} job in R_s (assuming both lists are sorted by acquired processing time).[6] Note that if $F = 1$, then this discipline behaves identically to MPA-Basic, as a job can only be replaced by itself. The processors are then allocated to the newly-selected jobs in the same fashion as in MPA-Basic.

Although more computationally expensive, the MPA-Pack discipline can achieve significantly higher memory utilization, up to 92.2% instead of 81.5% for MPA-Basic in the case of $\overline{m}/M = 0.50$. The constraint ensures that jobs which have close to the least acquired processing time are run first, thus maintaining low mean response times. The difficulty with this discipline, however, is in finding a good value for F, as the optimal value depends on the load on the system, and can be as low as 1 or as high as 40.

The performance of the MPA-Repl1 and MPA-Pack disciplines, assuming an UMA environment, is shown for $\overline{m}/M = 0.5$ in Figure 4.6. In the accompanying table, the maximum observed memory utilization is given for each of the disciplines, along with the predicted maximum sustainable load. In each case, the observed maximum throughput matches perfectly with the predicted value. MPA-Repl1 yields a significant improvement in achieved throughput, but MPA-Pack achieves a much better mean response time at high loads (e.g., about 54% improvement over MPA-Repl1 at a load factor of 89%).

4.3.5 Different Speedup Curves

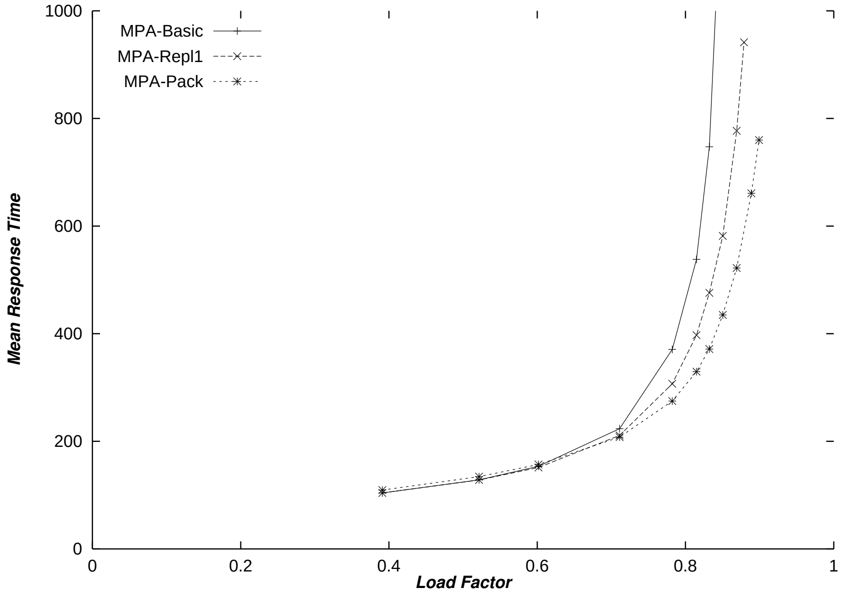
In practice, no system will have a workload in which all jobs have the same speedup curve. In fact, for any given job and problem size, speedup will often decline after a certain level of parallelism due to contention or overhead.

To investigate the effect of distinct speedup curves, we carried out simulation experiments to determine average response time as a function of load factor for MPA-Basic for the same average memory requirements as before. For this purpose, we used three Sevcik-style execution-time functions.

The speedup curves averaged over all jobs sizes for each of the three execution-time functions, which differ in ϕ, α, and β, are shown in Figure 4.7. These values were adjusted from those originally presented by Wu [Wu93] to be more meaningful in a 100-processor system given our average job lengths. Although only one curve exhibits slowdown on average, all jobs will exhibit slowdown given a sufficiently small amount of work w.

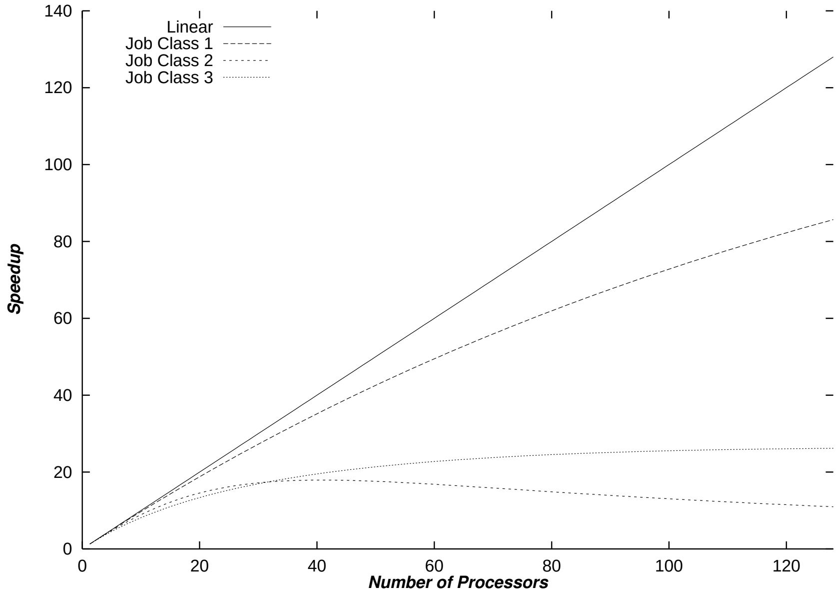
In Figure 4.8, we show the performance of the both the UMA and distributed-memory variants of the MPA-Basic scheduling disciplines, again for $\overline{m} = \{0.1, 0.25, 0.5\}$. There are two important

[6] If there are more jobs in $R_{s'}$ than in R_s, then only as many jobs in $R_{s'}$ as there are jobs in R_s need satisfy the criteria.



Discipline	*MemUsage*	*MaxLoad*
MPA-Basic	81.5%	0.85
MPA-Repl1	86.5%	0.89
MPA-Pack	92.2%	0.93

Figure 4.6: Performance of UMA scheduling disciplines with memory packing improvements for $\overline{m} = 0.5$.



Job Class	ϕ	α	β
1	1.01	3.4	0.0042
2	1.10	8.0	0.66
3	1.16	30.	0.050

Figure 4.7: Job-class speedup functions used for experiments where jobs have varying speedup characteristics.

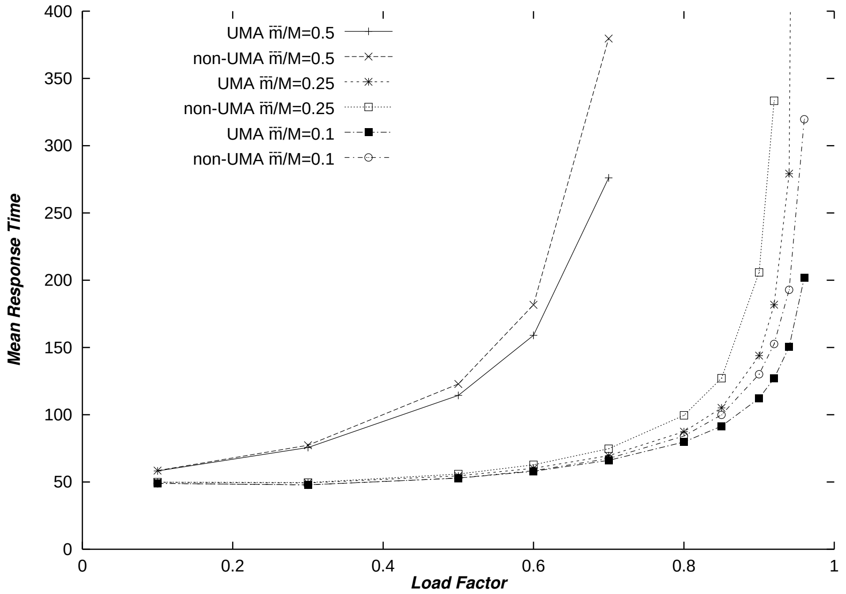


Figure 4.8: Performance of MPA-Basic with and without the distributed-memory constraint for $\overline{m} = \{0.1, 0.25, 0.5\}$ when jobs have varying speedup characteristics.

differences from the case where all jobs shared a common speedup function. First, in the case of $\overline{m} = 0.25$, the system saturates much more quickly than before. The reason is that imbalances in processor allocation have a much greater effect, since it is possible for a single poor-speedup job to be allocated all the processors in the system if it has a high memory requirement. This is the effect that was shown to occur in the proof of Proposition 4.1. Second, the distributed-memory constraint seems to impede performance more than before, because the minimum processor requirements lead to even more imbalance, producing an even greater loss in processor efficiency.

4.4 Conclusion

In this chapter, we investigated the coordinated allocation of processors and memory in the context of multiprocessor scheduling. By first establishing some bounds on the achievable system throughput, we were able to gain insight into how to design scheduling disciplines for this problem. Our significant observations include:

- It is very important that memory be considered in the scheduling decision, particularly in selecting jobs to run next. As the load increases, disciplines which can make better use of memory can sustain a higher load. Although increasing memory utilization will proportionally increase the bound on throughput due to memory, the overall benefit may be somewhat limited by the bound on throughput due to processing (as illustrated in Figure 4.5).

- If the workload speedup function is convex, but no information is known about the relative speedup characteristics of individual jobs, then an equi-allocation strategy favouring jobs with the least acquired processing time will yield good response times and achieve a throughput close to the maximum. If memory can be fully utilized, then the limit on maximum throughput will be

$$\frac{M}{\overline{m}T(\frac{\overline{m}}{M}P)}$$

- Knowing speedup information is more useful for improving performance (i.e., increasing sustainable throughput) in memory-constrained scheduling than in the non-memory-constrained case. In the latter, maximizing the sustainable throughput is achieved by allocating each job a single processor, an approach which is not possible in the memory-constrained case.

Most closely related to the results of this chapter is McCann and Zahorjan's work in memory-constrained scheduling [MZ95]. There are two aspects that make a direct comparison between their disciplines and the MPA-based ones difficult. First, their disciplines are designed specifically for distributed-memory systems, and it is not clear what approach should be taken to adapt them to the shared-memory case. Second, their disciplines do not explicitly describe how to handle job departures. In particular, each scheduling *cycle* involves running every job in the system in such a way that each job has the same processor occupancy within the cycle; given the variability of service demands found in practice, and the need to make the scheduling cycle large enough to keep overheads to an acceptable level, many (if not most) jobs will typically terminate during the cycle. Since the focus of the scheduling decision is at cycle boundaries, it is not clear how to best handle job departures.

The Tera MTA system also assumes that the aggregate memory requirement of the jobs available to run will exceed the capacity of the system [AKK$^+$95]. Their job-swapping algorithm is designed to minimize the amount of time that memory is unavailable for computation (since a job must be entirely in physical memory for its threads to be active). This job-swapping algorithm is also applicable to the type of scheduling described in this chapter since we assume that a job performs well only if its entire data image is in memory. The job selection strategy in the Tera scheduler is designed so that the acquired memory occupancy of a job relative to other jobs obeys a memory occupancy "demand" parameter for each job. This objective is very different from minimizing mean response time, preventing a direct comparison between the Tera job selection strategy and our multi-level feedback approach. Moreover, the issue of processor allocation in multithreaded architectures is so radically different from conventional multiprocessors that it too cannot be compared with MPA.

Chapter 5

Memory-Constrained Scheduling with Speedup Knowledge

5.1 Introduction

In the previous chapter, we investigated bounds on the achievable system throughput when nothing is known about the speedup characteristics of jobs and found that an equi-allocation strategy could yield excellent performance. But the no-knowledge assumption limits the applicability of the results to workloads exhibiting little or no correlation between memory requirements of jobs and their speedup characteristics. Actual workloads might not be so correlation-free.

One reason is that, for many scientific applications, there exists a clear relationship (i.e., correlation) between the "size" of the problem being solved and the service demand, memory requirement, and speedup characteristics of the job, as captured by various scalability models (e.g., [SN93, GGK93]). It is quite reasonable to also find such correlations in a diverse multiprocessor workload, particularly if a few applications were to dominate the workload.

In the first part of the chapter, we examine a wide spectrum of workloads, principally to understand the levels of performance improvement relative to an equi-allocation discipline that are possible given speedup knowledge and the types of workloads under which these improvements exist. We show that if no correlation exists between the memory requirement and speedup characteristics of jobs, then there is a moderate benefit in having speedup knowledge. If correlation does exist, then the potential improvement increases, theoretically to an arbitrarily large degree over that of an equi-allocation discipline.

In the second part of the chapter, we propose some scheduling disciplines that use speedup information in allocating processors and show that these disciplines perform very well compared to the equi-allocation discipline in the case where memory requirement and job speedup are correlated. Although these disciplines are only useful if speedup information is available, we do not consider this to be a problem. If the workload is dominated by a few important applications, it is relatively easy to measure the speedup characteristics of these applications directly and record the information for the

Workload Model Parameters	
f_c	fraction of work generated by jobs of class c ($\sum f_c = 1$)
m_c	memory requirement of jobs of class c ($m_c \leq M$)
$T_c(p)$	execution time of jobs of class c on p processors

Table 5.1: This table lists the parameters used in defining workloads. With respect to the fractions of work, a job from class c corresponds to $w_c = T_c(1)$ units of work.

scheduler. In other environments, it has been shown feasible to collect reasonably accurate speedup information of a job at run time with little user intervention [NVZ96].

In this chapter, we focus on shared-memory systems. In distributed-memory systems, the processor and memory occupancies associated with the execution of a job are essentially the same, differing only by a constant factor. Thus, the only logical way to use speedup information in this case is to allocate leftover processors to jobs which can consume work most efficiently, thereby minimizing both the processor and memory occupancies simultaneously. The shared-memory case, where memory allocation is not tied to processor allocation, is more challenging.

In the next section 5.2, we describe how we assess the throughput benefits of having speedup knowledge and give our analytic results. In Sections 5.3 and 5.4, we propose and evaluate our scheduling disciplines that make use of speedup information. We then present our conclusions in Section 5.5.

5.2 Throughput Analysis

We examine the difference in achievable throughput between an equi-allocation strategy having no information regarding the speedup characteristics of jobs and a non-equi-allocation strategy having complete information. Similar to previous chapters, our system model consists of a simple shared-memory multiprocessor having P processors and M units of memory (normalizing M to 1 for convenience). But for this analysis, we assume that all jobs belong to one of a number of classes $\mathcal{C}$ which are characterized by the parameters listed in Table 5.1.

To simplify the presentation, we let the throughput be the amount of work consumed per unit time. (A job from class c corresponds to $T_c(1)$ units of work.) This does not change the results, because the usual "job" throughput is simply $\sum f_c/T_c(1)$ times the "work" throughput, so maximizing one type of throughput is equivalent to maximizing the other.

5.2.1 General Case for Uncorrelated Workloads

To determine a bound on the maximum throughput of the system, we assume that all the job parameters described above are known. We also assume that there is a large queue of work (i.e., jobs) waiting to be consumed (such as would be the case in a heavily-loaded system) and require that the throughput (of work) in each class c be f_c times that of the overall throughput.

Classes:

$m_1 = 0.9$
$m_2 = 0.1$

Configurations:

$t_1 : C_1(100)$
$t_2 : C_1(50), C_2(50)$
$t_3 : C_2(10), C_2(10), \cdots, C_2(10)$

Linear Programming Problem:

Minimize: $z = t_1 + t_2 + t_3$
Subject to:

$$\begin{aligned} t_1 S_1(100) + t_2 S_1(50) &\geq f_1 W \\ t_2 S_2(50) + 10 t_3 S_2(10) &\geq f_2 W \end{aligned}$$

Figure 5.1: Predicting the maximum throughput of a workload given $P = 100$. The configurations are listed as class, followed by processor allocation. For this example, we used an equi-allocation strategy; in general, the second configuration would expand to all possible processor allocations, from $C_1(1), C_2(99)$ to $C_1(99), C_2(1)$.

To obtain the maximum overall throughput in the general case, we first enumerate all possible combinations of jobs that fit in memory and, for each combination, all possible processor allocations as permitted by the scheduling strategy of interest. (We term each of the possibilities a *configuration*; a simplified example is given in Figure 5.1.) We then specify a linear programming problem in which each configuration, j, is associated with a free variable t_j representing the time for which that configuration is executed in a schedule.[1] The amount of work from class c consumed per unit time, given a processor allocation p, is $T_c(1)/T_c(p) = S_c(p)$; therefore, running a configuration j for time t_j will consume $t_j S_c(p)$ jobs from class c for each job in that configuration (substituting the appropriate value of p for each job).

Let W be the amount of work to be consumed by the system; as such, the scheduler must consume $f_c W$ units of work from each class c. But to obtain the optimal solution, we relax the constraint to be that *at least* $f_c W$ units of work be consumed for each class c; if a configuration is run and there is no work left for a particular class, then any processors assigned to that class would be left idle. The objective function is simply $z = \sum_j t_j$, which should be minimized. As z gives the time needed to consume W units of work, the throughput is W/z. Since z is linear in W, the throughput is independent of the particular value chosen for W.

Recall that our goal is to investigate the throughput difference between a naive equi-allocation scheduler and one that possesses full knowledge of the characteristics of individual jobs. Clearly, the

[1]Note that we do not advocate the use of linear programming problems within a practical scheduler. In this section, the goal is to explore the limits on gains in performance from using application knowledge. More practical scheduling algorithms are proposed in Section 5.3.

linear program models the latter perfectly, producing the best possible throughput. Modeling a naive equi-allocation discipline is a little more difficult because, by favouring some configurations over others, the solution to a linear programming problem can implicitly make use of the speedup characteristics of jobs, even if within each configuration processors are allocated evenly among jobs. For example, consider the system from Figure 5.1, and assume that the large jobs have perfect speedup and the small ones very poor speedup. The optimal solution to the linear programming problem will have $t_2 = 0$. This does not model naive equi-allocation accurately as knowledge of speedup characteristics has been used in avoiding the second combination.

If there is no correlation between memory requirement and speedup characteristics, however, then the average execution time of a unit of work on p processors will be the same in all memory classes,[2] say $\overline{T}(p)$. In this case, the naive equi-allocation scheduler (called **naive equi**) can be modeled by aggregating all classes having the same memory requirement into a single class, having average execution time $\overline{T}(p)$. The full-knowledge scheduler (called **smart non-equi**) is still modeled by distinguishing between jobs having different speedup characteristics.

The relative performance of these two disciplines is shown for two two-memory-class workloads in Figure 5.2, assuming no correlation between memory requirement and speedup characteristics. In all graphs, we use Dowdy-style execution-time functions as these can be parameterized with a single parameter s (see equation (2.1)), allowing us to easily examine a range of workloads. The range of s values chosen for examination, from 0.001 to 0.2, correspond to realistic speedup curves for jobs that speedup very well (91 on 100 processors) and that speedup very poorly (5 on 100 processors), respectively.

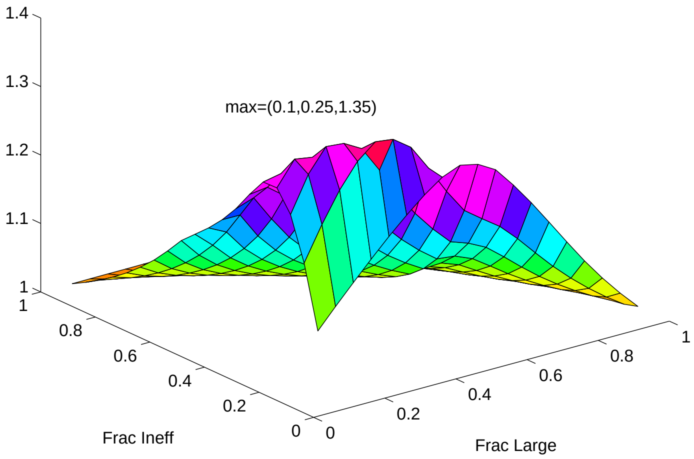
In the first example, the workload is composed of only some very large and some very small jobs. The graph shows that moderate benefit can be obtained from using speedup information, yielding up to a 35% increase in throughput for smart non-equi relative to naive equi. As the discrepancy in speedup curves decreases, the maximum performance difference between naive equi and smart non-equi decreases (e.g., 16% if $s_1 = s_3 = 0.01, s_2 = s_4 = 0.2$). In this first example, however, it is necessary for most of the work to be associated with small, inefficient jobs for a significant benefit to be obtained. This is contrary to our intuition regarding actual workloads, as we believe large jobs will generate the most work [FN95]. But if we decrease the size of large jobs, so that two large jobs can fit in memory together, we can observe increases in throughput across a wider variety of workloads. In particular, we can observe up to a 25% increase in throughput if most work is associated with large, efficient jobs.

In Figure 5.3, we show similar graphs for a three-memory-class workload. These graphs are for the case where only 20% of work is associated with poor speedup jobs, since our two-memory-class case showed that it was necessary to have a small fraction of work associated with poor-speedup jobs in order to obtain any performance improvement.[3]

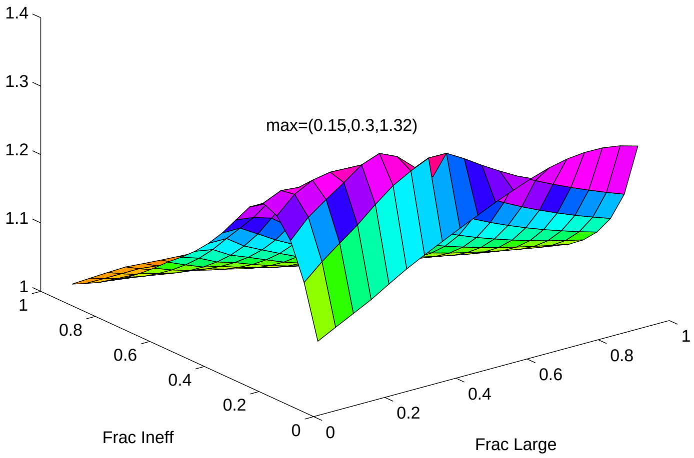
---

[2] A memory class ω corresponds to all job classes having identical memory requirement (i.e., for memory requirement m, $\omega_m = \{\bigcup c \in \mathcal{C} : m_c = m\}$).

[3] Because the total number of possible job configurations increases dramatically over the two-class case, it was neces-



Uncorrelated Workload 1(a)			
Class	m_c	s_c	f_c
1	0.9	0.001	$(1-g_\mathrm{I})\,g_\mathrm{L}$
2	0.9	0.2	$g_\mathrm{I}\,g_\mathrm{L}$
3	0.1	0.001	$(1-g_\mathrm{I})\,(1-g_\mathrm{L})$
4	0.1	0.2	$g_\mathrm{I}\,(1-g_\mathrm{L})$



Uncorrelated Workload 1(b)			
Class	m_c	s_c	f_c
1	0.5	0.001	$(1-g_\mathrm{I})\,g_\mathrm{L}$
2	0.5	0.2	$g_\mathrm{I}\,g_\mathrm{L}$
3	0.1	0.001	$(1-g_\mathrm{I})\,(1-g_\mathrm{L})$
4	0.1	0.2	$g_\mathrm{I}\,(1-g_\mathrm{L})$

Figure 5.2: Throughput of smart non-equi relative to naive equi for two job-classes in an uncorrelated workload. The lower axes correspond to the fraction of work associated with inefficient jobs, g_I, and that associated with large jobs, g_L. The maximum point is shown in (x,y,z) coordinates.

In the first graph, the potential benefit increases with the fraction of work associated with large-sized jobs, being relatively insensitive to the fraction of work associated with medium-sized ones. In the second graph, the range of potential benefit increases such that any choice of fractions can yield at least a 10% increase in performance. The reason is that large-sized jobs in the first graph can only run with small-sized jobs, while in the second, they can run with jobs of any size. This increases the opportunity for running inefficient large jobs with efficient ones.

5.2.2 Specific Case for Correlated Workloads

Next, we consider the case where there exists a correlation between execution-time function and memory requirement. As described earlier, it is not possible to model the naive equi discipline using the linear programming approach for the general case. It is possible, however, to investigate a special two-job-class case analytically.

Let there be two classes of jobs to be scheduled such that $m_1 + nm_2 = M, m_1 > M/2$. We assume that a naive equi scheduler will try to maximize the use of memory whenever possible, since this was shown to be important in the previous chapter. This is the reason, however, why naive equi will perform badly; if it could use the speedup characteristics of each job, the scheduler could trade off memory usage for processor efficiency. Given no knowledge of speedups, however, one must assume that maximizing memory usage at high loads is the best strategy.

There are only three possible combinations of jobs from which to choose: one class 1 job and n class 2 jobs; $l = \lfloor M/m_2 \rfloor$ class 2 jobs; or (only if needed) one class 1 job on its own. Again, let there be W units of work to be consumed. The fraction of work from class 1 jobs and class 2 jobs that can be consumed simultaneously can be computed by taking the ratio of the time to consume all work from class 2 jobs (n at a time) over that for all class 1 jobs:

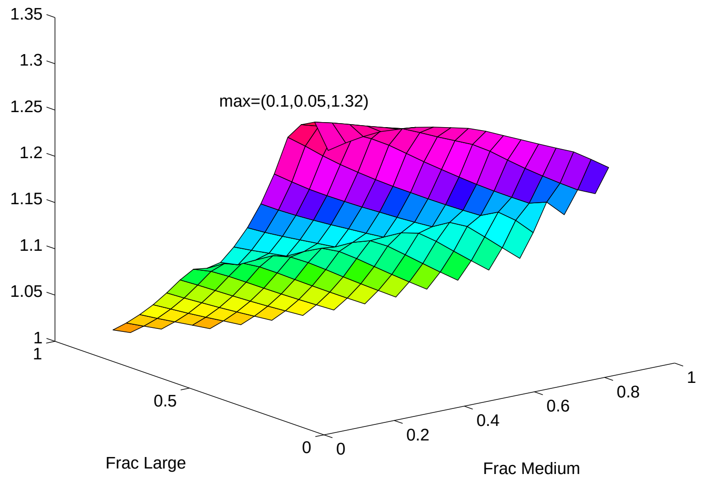
$$\gamma = \frac{f_2 W}{nS_2(P/(n+1))} \Bigg/ \frac{f_1 W}{S_1(P/(n+1))}$$

The time $\tau(W)$ required to consume W units of work is then:

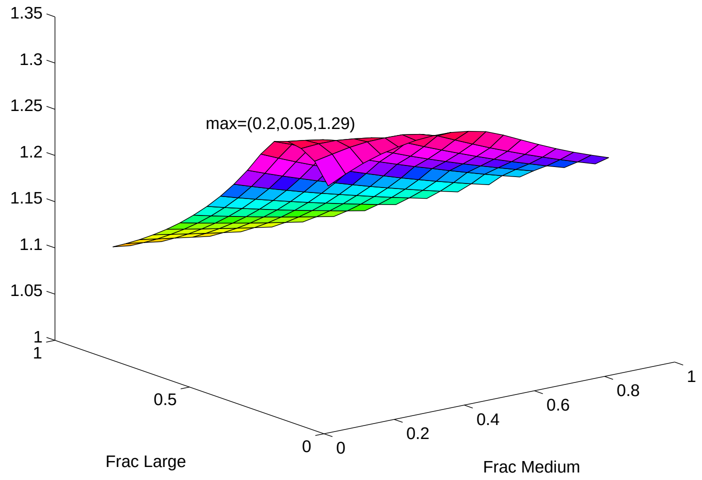
$$\tau(W) = \begin{cases} \gamma\left(\frac{f_1 W}{S_1(P/(n+1))}\right) + (1-\gamma)\left(\frac{f_1 W}{S_1(P)}\right) & \text{if } \gamma \le 1 \\ \left(\frac{f_1 W}{S_1(P/(n+1))}\right) + \left(1 - \frac{1}{\gamma}\right)\left(\frac{f_2 W}{lS_2(P/l)}\right) & \text{otherwise} \end{cases}$$

The first term in both cases is the time that class 1 work and class 2 work are consumed simultaneously (consuming all class 1 work if $\gamma \ge 1$). If $\gamma \le 1$, then the second term corresponds to leftover class 1 work. If $\gamma > 1$, then the second term corresponds to leftover class 2 work (for which we can fit l at a time in memory). This expression leads to a throughput of $W/\tau(W)$. Since $\tau(W)$ is linear in W, the throughput is again independent of W.

sary to limit the processor allocation choices to be multiples of four (other than 1 and P). We do not believe this significantly affected the observations and conclusions as compared to allowing all possible processor allocation choices.



Uncorrelated Workload 2(a)			
Class	m_c	s_c	f_c
1	0.9	0.001	$0.8g_{c1}$
2	0.9	0.2	$0.2g_{c1}$
3	0.25	0.001	$0.8g_{c2}$
4	0.25	0.2	$0.2g_{c2}$
5	0.1	0.001	$0.8(1-g_{c1}-g_{c2})$
6	0.1	0.2	$0.2(1-g_{c1}-g_{c2})$



Uncorrelated Workload 2(b)			
Class	m_c	s_c	f_c
1	0.5	0.001	$0.8g_L$
2	0.5	0.2	$0.2g_L$
3	0.25	0.001	$0.8g_M$
4	0.25	0.2	$0.2g_M$
5	0.1	0.001	$0.8(1-g_L-g_M)$
6	0.1	0.2	$0.2(1-g_L-g_M)$

Figure 5.3: Throughput of smart non-equi relative to naive equi for three job-classes in an uncorrelated workload. The lower axes represent the fraction of work associated with large, g_L, and medium jobs, g_M. For all cases, the fraction of work associated with poor-speedup jobs is 20%.

Given this, the performance of naive equi relative to smart non-equi is shown in Figure 5.4. In the first graph, we let $s_1 = 0.001$ and $s_2 = 0.2$. As can be seen, up to a 75% improvement in throughput can be obtained for workloads comprised mostly of large jobs. This corresponds to the type of workload that one might observe in a real system. The second graph shows that the potential performance improvement is larger in a case where the disparity in speedup behaviour between the two classes is extreme, but the improvement is never more than 100%. This limiting behaviour is captured by the following proposition.

Proposition 5.1 *Given a workload consisting of two job classes such that $m_1 + nm_2 = M, m_1 > M/2$, an optimal scheduler given full knowledge of the speedup characteristics of jobs may have a maximum throughput that is, in the limit, $(n+1)$ times that of a naive equi-allocation scheduler that seeks to maximize memory usage.*

Proof Overview Assume class 1 jobs have perfect speedup and class 2 jobs have no speedup. Let $m_2 \to 0, P = M/m_2$. Now, choose f_1 such that $f_1/S_1(P/(n+1)) = f_2/(nS_2(P/(n+1)))$ (i.e., $\gamma = 1$). Since $S_1(P/(n+1)) \to \infty$ and $S_2(P/(n+1)) = 1$ (i.e., a constant), $f_2 \to 0$ and $f_1 \to 1$. Thus, for naive equi, $\tau(W) = W/S_1(P/(n+1)) = W(n+1)/P$. Smart non-equi, on other hand, will schedule class 1 jobs on their own, leading to an execution time $\tau'(W) = f_1W/S_1(P) + f_2W/((M/m_2)S_2(1)) = W/S_1(P) = W/P$. □

The cases considered in this section represent an extreme situation where naive equi will perform very poorly. Later, in the context of our simulation studies, we study naive equi given workloads with realistic degrees of correlation, where it compares less poorly to the optimal schedule.

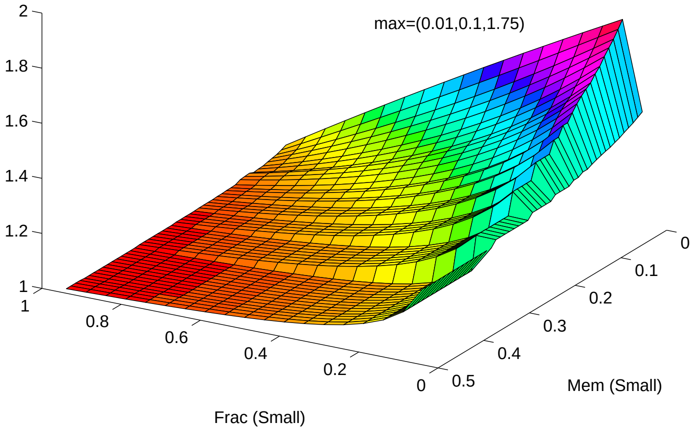
5.3 Scheduling Disciplines

We now turn our attention to developing practical scheduling disciplines that exploit knowledge of speedup characteristics to improve throughputs and mean response times.

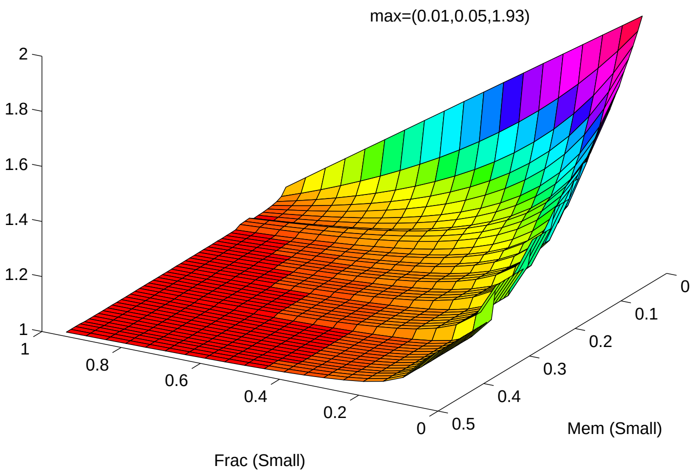
There are two essential steps that a processor-memory scheduler must perform. First, it must select jobs from those that are available to run during the next scheduling quantum. Again, since our goal is to minimize mean response time, we use the same principles as in previous chapters to perform this selection, namely favouring jobs with least-acquired processing time. Second, the scheduler must assign processors to the jobs that have been selected. The previous section showed that making poor processor allocation choices (given that job selection does not utilize speedup information) can lead to poor performance; in this section, we describe two scheduling disciplines that, by making better processor allocation choices, offer better performance when memory requirements and speedup are correlated.

5.3.1 Selecting Jobs

The basic strategy we use for job selection is to scan the list of jobs that are ready to run in order of increasing remaining service demand, choosing those that can fit in memory (i.e., first fit). In the



Correlated Workload 1(a)			
Class	m_c	s_c	f_c
1	$1-m_S$	0.001	$1-g_S$
2	m_S	0.2	g_S



Correlated Workload 1(b)			
Class	m_c	s_c	f_c
1	$1-m_S$	0.00001	$1-g_S$
2	m_S	0.99999	g_S

Figure 5.4: Throughput of smart non-equi relative to that of naive equi, given different memory requirements for small jobs, m_S, and different fractions work associated with small jobs, g_S.

previous chapter, we found that in order to maximize the sustainable throughput, it is necessary to improve the utilization of memory, particularly as the load on the system increases. In the context of this chapter, however, we found that a more aggressive memory packing algorithm was needed to maximize throughput.

For this purpose, we first select jobs for activation using the first-fit heuristic, but only commit to running a subset of these. We then use a subset-sum algorithm to find the set of remaining jobs that maximizes memory utilization of the remaining memory. Although the subset-sum problem is in general NP-complete, the size of problem with which we are concerned (i.e., up to 1000 jobs) can be quickly solved by branch-and-bound algorithms that have been proposed. The one that we use is presented elsewhere, and will be referred to simply as the *subset-sum algorithm* [MT90].

In our packing algorithm, job selection is increasingly based on improving memory utilization as the load increases; at heaviest load, remaining service demand is not considered at all, thus allowing the greatest freedom to maximize memory utilization. Given the nature of this algorithm, mean response times can be higher than those obtained using Repl1 and Pack variants described earlier (see Chapter 4), but higher throughputs can be achieved.

The job selection algorithm, invoked at the beginning of each quantum, is defined as follows:

- Let the load L be the number of jobs in the system, N_{ff} be the number of jobs selected by the first-fit algorithm, and δ be a tunable parameter that determines how aggressively the scheduler seeks to maximize memory usage as the load increases. The first N' jobs from first-fit are chosen for activation, where

$$N' = \left\lfloor N_{\mathrm{ff}} * \max(1 - \frac{L}{\delta N_{\mathrm{ff}}}, 0) \right\rfloor$$

  We choose $\delta = 100$, which means that the scheduler will gradually decrease the value of N' as the load increases, until L is 100 times greater than the number of jobs selected using first-fit (after which point $N' = 0$).

- Given the amount of memory remaining after the first N' jobs are chosen, the subset-sum algorithm is invoked to choose for activation a subset of the remaining jobs that maximizes the total memory usage.

An example of where this selection algorithm is beneficial is a system with a two-class workload where one class requires a small amount of memory and the other a large amount. With first-fit, a steady-state situation can arise where there is always a small number of small jobs at the beginning of the run queue. These small jobs are scheduled with enough processors to keep their number relatively steady; as a result, the large jobs never get a chance to run, a problem that neither the Repl1 or Pack variants can resolve. With the selection algorithm just described, however, if the load is sufficiently high (as indicated by a long queue of large jobs), the subset sum algorithm is invoked, allowing one of the large jobs to be selected. Thus, in the disciplines described next, we implicitly assume that this "Subset" variant is used in selecting jobs (but do not include it in the actual names of disciplines.)

5.3.2 Allocating Processors

Once the set of jobs to run has been chosen, the scheduler must allocate processors to these jobs. There are several approaches to making this allocation, two of which are described in this section.

The first approach is based on the bounds developed in the previous chapter, using average processor and memory occupancies (θ_p and θ_m, respectively) to determine the overall capacity of a system. Recall that given a set of jobs $\mathcal{J} = \{1, 2, \cdots, n\}$, such that job $i \in \mathcal{J}$ requires m_i units of memory, is allocated p_i processors, and has an execution-time function of $T_i(w_i, p)$,

$$\theta_m = \sum_{i \in \mathcal{J}} m_i T_i(w_i, p_i); \quad \theta_p = \sum_{i \in \mathcal{J}} p_i T_i(w_i, p_i)$$

In order to permit the highest possible system throughput (X), one must minimize the occupancies of memory and processors relative to the amounts available in the system, since throughput is bounded by:

$$X \leq \min(M/\theta_m, P/\theta_p) \tag{5.1}$$

Given a set $\mathcal{J}$, finding the processor allocations $\{p_1, \cdots, p_n\}$ that maximizes the throughput can be expressed as the following integer optimization problem:

$$\begin{array}{rll} \textbf{minimize}: & \max(\sum_{i \in \mathcal{J}} p_i T_i(w_i, p_i), \sum_{i \in \mathcal{J}} m_i T_i(w_i, p_i)) & \\ \textbf{subject to}: & \qquad\qquad \sum_{i \in \mathcal{J}} p_i & \leq P \end{array}$$

With our assumed Dowdy-style execution-time function (equation (2.1)), an increase in p_i for any job $i \in \mathcal{J}$, will increase the processor occupancy but decrease the memory occupancy. But given that we may not use more than P processors, any increase in allocation for one job must be balanced by a decrease for one or more others. Figure 5.5 illustrates the shape of the processor throughput and memory throughput surfaces for a three-job situation, assuming $p_1 + p_2 + p_3 = 100$. The memory throughput surface is always concave, as a low processor allocation will lead to a high memory occupancy for any given job. The processor throughput surface is always convex, as shifting a processor from a lower-efficiency job to a higher-efficiency one has greater effect when the initial allocation of the former is high.

Since service-demand distributions are highly variable, using a job's service demand in computing processor allocations that minimize occupancies can lead to extremely varied results. Thus, we perform the optimization using the same value of work for all jobs (e.g., 1). Our reasoning is that in each quantum, the scheduler will schedule one unit of work from each selected job; if a job reappears in a later quantum, then this corresponds to another unit of work from that job. Another option would be to perform the optimization using the expected fraction of work appearing in each class, but we did not wish to include this type of knowledge in our scheduler.

Thus, to find the point of maximum throughput, we use the following heuristic (which requires a few milliseconds of computation time):

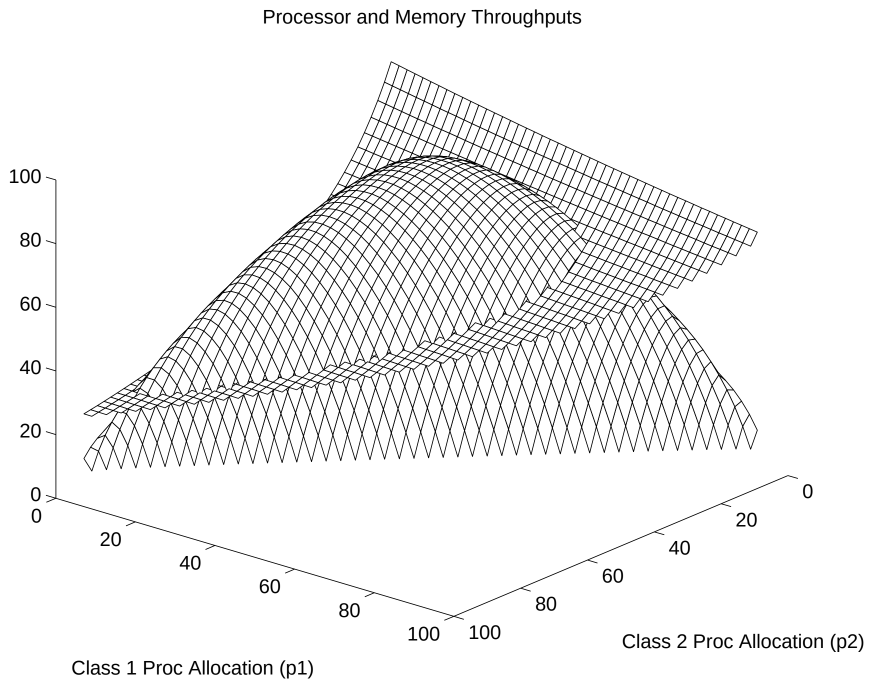


Figure 5.5: This graph illustrates memory and processor throughputs for a three-job situation. In this case, the maximum throughput occurs along the intersection of the concave memory throughput surface and the convex processor throughput surface. (We let $p_3 = 100 - p_1 - p_2$.)

- Determine the processor allocation that minimizes the memory occupancy. Using Lagrange multipliers, the critical points of:

$$\sum_{i \in \mathcal{J}} m_i T_i(1, p_i) + \lambda \sum_{i \in \mathcal{J}} p_i - P$$

  occur where $\lambda = \left(\sum_{i \in \mathcal{J}} \sqrt{m_i(1-s_i)/P} \right)^2$ and $p_i = \sqrt{m_i(1-s_i)/\lambda}$.

- Check if the processor occupancy is larger than memory occupancy. If so, choose that pair of jobs for which reallocating a processor from one to the other leads to the greatest increase in $\min(M/\theta_m, P/\theta_p)$. Repeat this step until improvements are no longer possible.

- Compare this allocation against equi-allocation; if equi is better (as happens on rare occasions), then use it instead.

The occupancy-based processor allocation discipline is thus as follows:

MPA-OCC Given the set of jobs selected to run, apply the heuristic to obtain the processor allocation that maximizes the throughput based on occupancies from equation (5.1).

The second approach is to try to maximize the efficiency of the system, which is equivalent to maximizing the amount of work consumed per unit time: $\sum_{i \in \mathcal{J}} 1/T_i(1, p_i)$. In this case, a simple

Sample Processor Allocations — 2 Job Classes			
	$s_1 = 0.001, s_2 = 0.2$ $m_1 = 0.8, m_2 = 0.1$	$s_1 = 0.2, s_2 = 0.001$ $m_1 = 0.8, m_2 = 0.1$	$s_1 = 0.1, s_2 = 0.2$ $m_1 = 0.5, m_2 = 0.25$
MPA-OCC	80-10-10	56-22-22	50-25-25
MPA-EFF	98-1-1	1-49-50	51-24-25
MPA-EQ	34-33-33	34-33-33	34-33-33

Table 5.2: This table illustrates allocation choices made by different disciplines, assuming one class 1 job runs with two class 2 jobs. The allocations are shown in order of the class 1 job followed by the two class 2 jobs.

greedy algorithm can be used where the next processor is given to the job which increases the work consumption the most per unit time (an algorithm which requires much less computation than the MPA-OCC heuristic). Formally, begin by setting $p_i = 1$ for every c. Choose a job j such that, for all jobs $k \neq j$,

$$\sum_{i \in \mathcal{J}, i \neq j} \frac{1}{T_i(1, p_i)} + \frac{1}{T_j(1, p_j + 1)} \geq \sum_{i \in \mathcal{J}, i \neq k} \frac{1}{T_i(1, p_i)} + \frac{1}{T_k(1, p_k + 1)}$$

and assign the next processor to j. Repeat this step until no processors are left. The efficiency-based processor allocation discipline is thus defined as follows:

MPA-EFF Given the set of jobs selected to run, apply the greedy algorithm to maximize the consumption of work per unit time.[4]

Finally, our baseline processor allocation discipline is similar to the MPA class of discipline from the previous chapter, except that we now use the subset-sum algorithm for job selection:

(Naive) MPA-EQ Given the set of jobs selected to run, allocate processors as evenly as possible.

To illustrate the difference in processor allocations made by each of these algorithms, Table 5.2 presents the resulting choices for the case when one class 1 job is running with two class 2 jobs, given various choices of s_c and m_c. In the case of MPA-EFF, allocating poor-speedup jobs a single processor while good-speedup jobs are available is better than not running the poor-speedup job at all, because it allows these jobs to run at full efficiency. (These poor-speedup jobs would otherwise have to be run on their own later on, at lower efficiency.)

5.4 Simulation Study

The performance of the disciplines described in the previous section has been examined by simulation under various workload conditions. First, we consider the workloads that were used as examples

[4] Incidently, this is the processor allocation strategy one would use in a distributed-memory environment to maximize the efficiency at which both processors and memory are utilized. In fact, MPA-OCC and MPA-EFF are equivalent to each other in the distributed-memory case.

in the analysis section. We find that there are minimal benefits from having speedup information in the case of the uncorrelated workloads from Section 5.2.1, but that there are great benefits in the case of the correlated workloads from Section 5.2.2. Then we consider a variety of correlated workloads that are likely to be more representative of real systems, where we can also find significant performance improvement.

As in the previous chapter, we assume that jobs are malleable with no overhead (for the same reasons previously stated). We also assume that the execution time of jobs can be accurately modeled by the Dowdy function used up to now.[5] Once again, the inter-arrival time distribution is assumed to be exponential, and the service-demand distribution hyper-exponential with coefficient of variation of five. For the most part, a sufficient number of independent trials were performed for each data point to obtain a 95% confidence interval that was within 5% of the mean. A trial terminated when the first 500 000 jobs that entered the system (after a short warm-up period) had departed. The mean response time for a trial was based only on these 500 000 jobs.

5.4.1 Uncorrelated Workloads

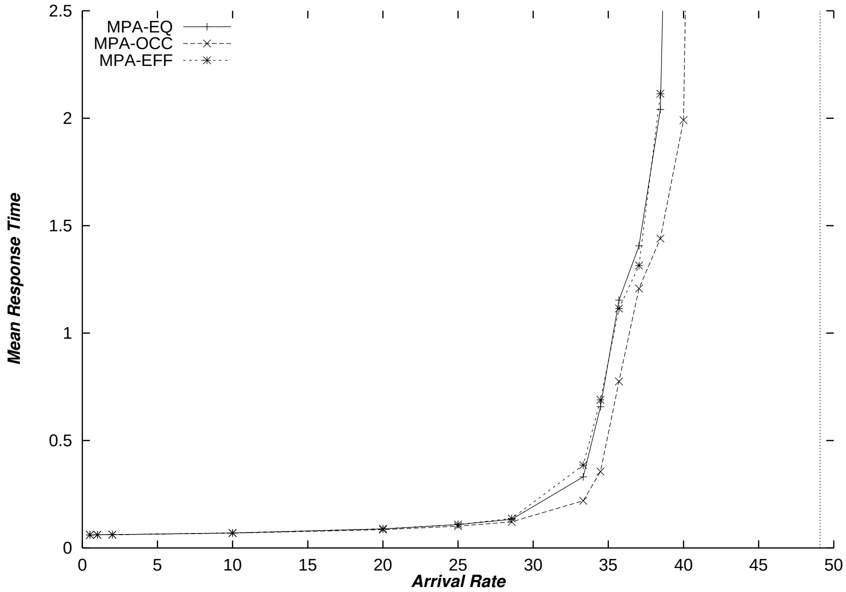
In Figure 5.6, we show the performance of the scheduling disciplines for the two-job-class systems studied in Figure 5.2. We chose parameters for the memory requirements, execution-time functions, and fractions such that performance gains were achievable.

In these workloads, the improvements in mean response times are minor, occurring only at very high loads. The MPA-EQ discipline saturates in both cases very close to the value predicted by our model (40.45 and 51.63, respectively); the MPA-OCC discipline saturates at an arrival rate that is 4% and 5%, respectively, greater than that of MPA-EQ. Improving the performance beyond what is shown has proven difficult, short of embedding the optimal solution in the scheduler and ensuring that different job configurations are selected in the correct proportions.

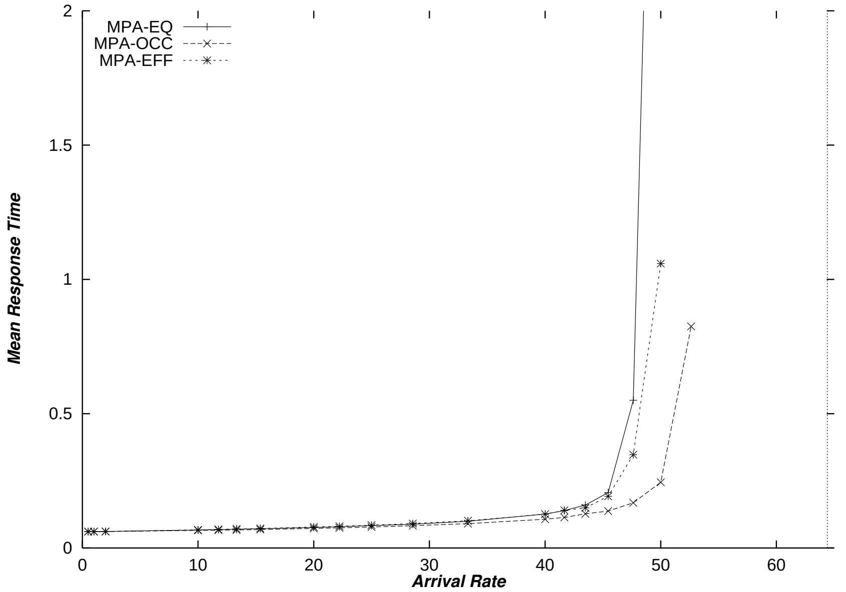
To illustrate the difficulty of achieving better performance, we consider the first of the two workloads ($m_1 = 0.9, m_2 = 0.1$). The optimal solution as given by the linear programming model is given in Table 5.3.

The maximum throughput is very sensitive to the way in which large inefficient jobs are scheduled; any deviation from the indicated values in the processor allocations for this class leads to significant decrease in achievable throughput. For example, using the processor allocation choices made by the MPA-OCC discipline for the above configurations (which only really differ in the third line where large jobs receive 75 processors and small ones 25) results in a maximum throughput that is 6.2% worse than the optimum. More importantly, increasing the fraction of work associated with large inefficient jobs to 25% (from 6.25%) results in a drastically different processor allocation in

[5] The major benefit of using the Sevcik function is in having a maximum parallelism value; since at heavier loads a job is unlikely to receive many more processors, this additional parameter is not crucial. Also, we found that in previous chapters, the choice of execution-time function did not significantly affect our results qualitatively, and so we focus on Dowdy-style functions.



Uncorrelated Workload 1(a)
($g_{\mathrm{I}} = 0.25, g_{\mathrm{L}} = 0.25$)



Uncorrelated Workload 1(b)
($g_{\mathrm{I}} = 0.25, g_{\mathrm{L}} = 0.25$)

Figure 5.6: Performance of scheduling disciplines for two-job-class uncorrelated workloads from Figure 5.2. The vertical line represents the maximum sustainable load, as predicted by our model. (If all jobs were assigned a single processor, then the average response time at light load would be one.)

Optimal Schedule for Two-Job-Class Problem				
time	lg ineff	lg eff	sm ineff	sm eff
$t_1 = 23.8\%$			10; 100	
$t_2 = 10.1\%$		1; 96	1; 4	
$t_3 = 44.5\%$	1; 35			1; 65
$t_4 = 21.6\%$	1; 74		1; 26	

Table 5.3: This table gives the optimal schedule for the workload studied in Figure 5.6. Time is in terms of a fraction of the total, and entries correspond to the number of jobs selected to run from the given class, followed by the total number of processors allocated to those jobs. Processors allocated to a class are distributed evenly among the jobs in that class.

the optimal case for the third configuration, giving more processors to the large inefficient jobs than to the small efficient ones.

In an attempt to improve the performance of the disciplines, we experimented with a different job selection strategy which avoided running inefficient jobs without efficient ones, and which avoided running more than one efficient job at any time. The intent was to maximize the efficiency of the system by always having an efficient job available. This change did not lead to any noticeable improvement, however, as efficient jobs were still being consumed too quickly for there to always be one available. (Note that the optimal solution runs inefficient jobs together 21% of the time.)

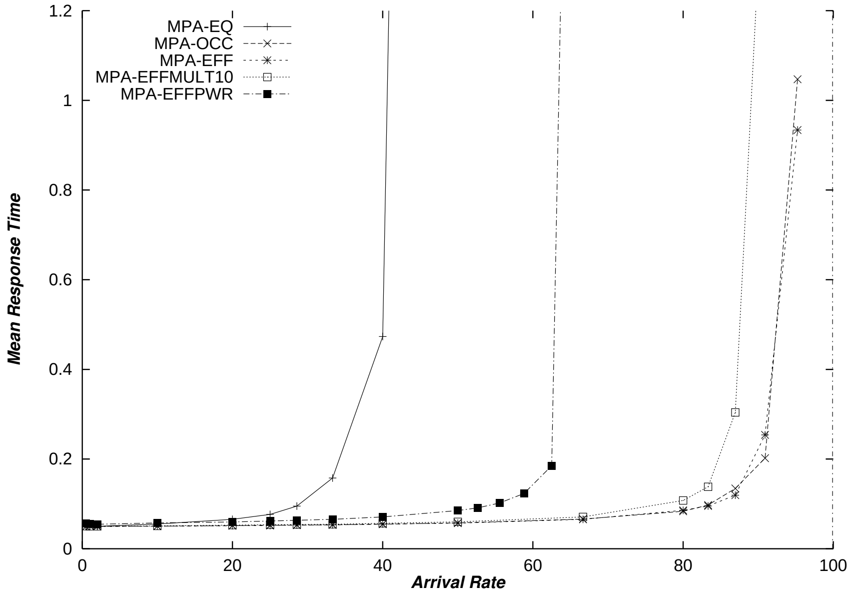
We conclude that for this uncorrelated workload, it is difficult to obtain better performance given only information about the jobs currently in the system. If there is a large backlog of jobs such that the fraction of work in each class is representative of the workload, then it would be possible (albeit expensive) to use a linear programming approach to find the best solution. If this is not the case, however, then any job selection and processor allocation strategies based on knowledge of jobs in the system are likely to be far from optimal.

5.4.2 Correlated Workloads

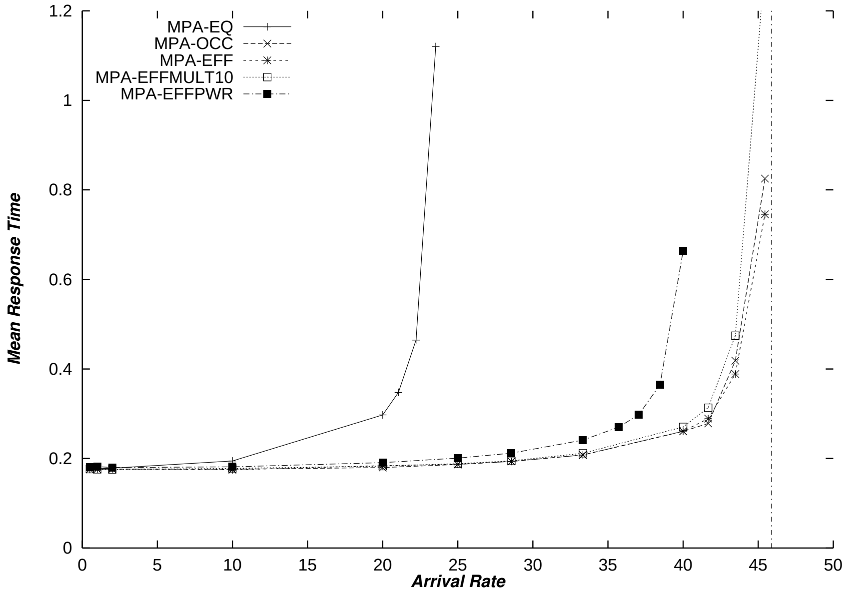
In Figure 5.7, we show the performance of the scheduling disciplines for the special-case two-job-class correlated workload where $m_1 = 0.9, m_2 = 0.1, s_1 = 0.001, s_2 = 0.2$. In the first graph, we set the fraction of work associated with large, efficient jobs to 94% and in the second to 83%.[6] As can be seen, in both cases, MPA-OCC and MPA-EFF reach the maximum possible load. MPA-EQ, on the other hand, only reaches about half the attainable load, as predicted by our earlier analysis in Section 5.2.2. (MPA-EFFMULT10 and MPA-EFFPWR represent disciplines similar to MPA-EFF, but in which processor allocations are constrained. The behaviours of these disciplines are discussed later, in Section 5.4.3.)

Next, we investigate the performance of the scheduling disciplines given more general models of correlation between memory requirements and speedup characteristics. As in the previous chapter,

[6]These fractions correspond to the best case for non-equi allocation and worst case for equi-allocation, respectively. They also happen to be in the range one might expect for actual workloads.



Correlated Workload 1(a)
($g_S = 0.94$)



Correlated Workload 1(a)
($g_S = 0.83$)

Figure 5.7: Performance of scheduling disciplines for two-job-class correlated workloads from Figure 5.4(a), using two different workload mixtures.

and for similar reasons, we draw the memory requirement from a uniform distribution. Although the analysis by Feitelson and Nitzberg [FN95] showed that there can exist statistical correlation between memory requirements and service demand, we did not find that such a correlation affected the results. We thus only present the case where memory requirements and efficiency are correlated.

The memory-speedup correlation is defined by a function $F : (0,1) \mapsto (0,1)$, used in the following manner:

$$\begin{aligned} m &= \mathrm{Unif}(0,1) \\ s &= \begin{cases} 0.2 & \text{with probability } F(m) \\ 0.001 & \text{otherwise} \end{cases} \end{aligned}$$

The performance of the various scheduling disciplines for our more general class of workloads is shown in Figures 5.8 and 5.9. The graphs inset in the performance graphs depict the correlation function F used. In general, as the memory requirement increases, the probability of a job being inefficient decreases.

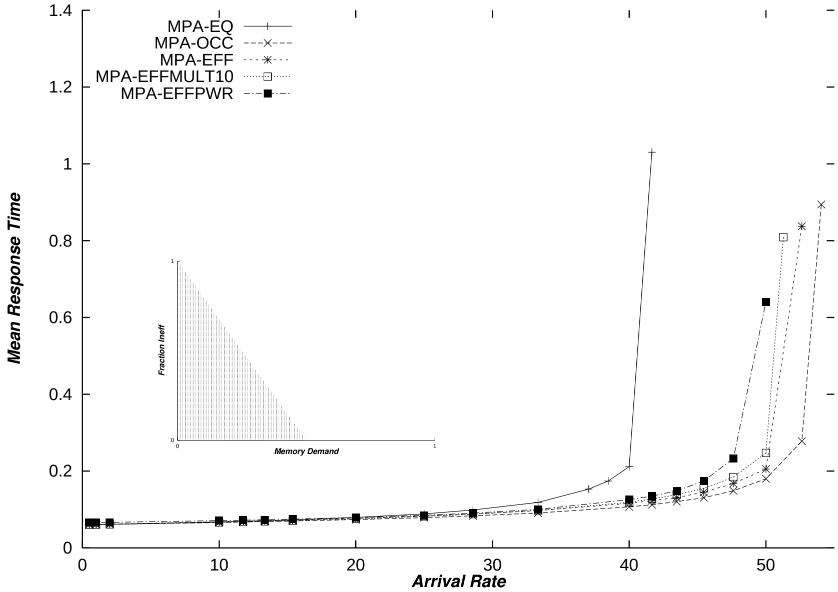
In all cases, the MPA-OCC discipline achieves the highest performance, closely followed by MPA-EFF. Also, the mean response times of MPA-OCC and MPA-EFF are very close to those of MPA-EQ before MPA-EQ leads to saturation; after this point, these disciplines continue to give good response times until they also cause saturation. To gain some sense of the performance of the disciplines relative to the maximum possible, we discretized the workloads into eight memory classes which we fed into the linear programming model. We found that MPA-OCC attained anywhere from 85% to 93% of the estimated maximum sustainable load.

In examining a variety workloads, the following trends were observed:

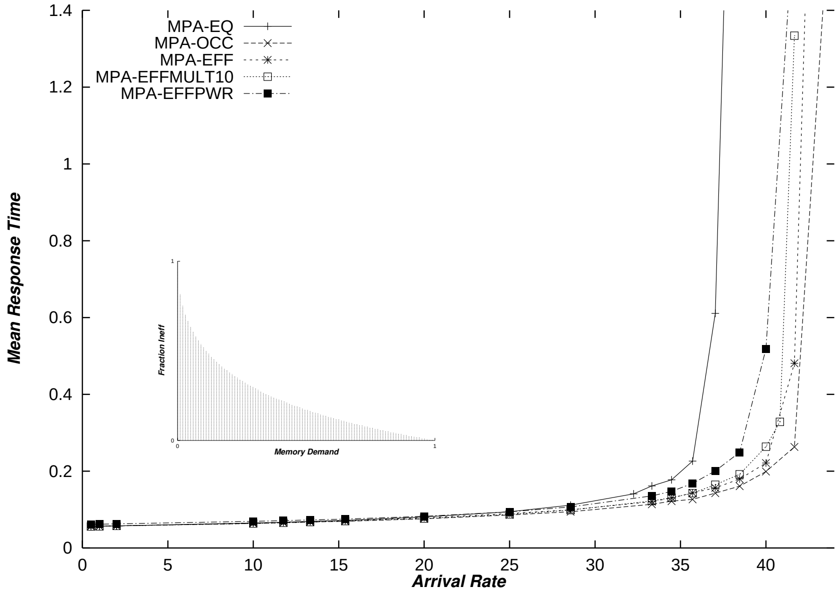
- Performance gains from having speedup information are small unless more than 50% of the work is associated with jobs having good speedup, as there will not be enough efficient jobs to run with inefficient ones.

- As the expected memory-work value (i.e., $\int xF(x)\,dx$, where F is the memory-speedup correlation function defined above) for poor-speedup jobs decreases, the improvement in throughput increases. The distributions in the graphs of Figures 5.8 and 5.9 have expected memory-work values for poor-speedup jobs of 0.042, 0.061, 0.083, 0.1458, in that order. This mirrors the pattern of improvement over the four distributions.

The graphs and results shown here are for a representative sample of all the workloads that we studied. Some other workloads exhibited higher performance difference between MPA-OCC and MPA-EQ and others less, but in no case studied did MPA-OCC or MPA-EFF perform worse than MPA-EQ.

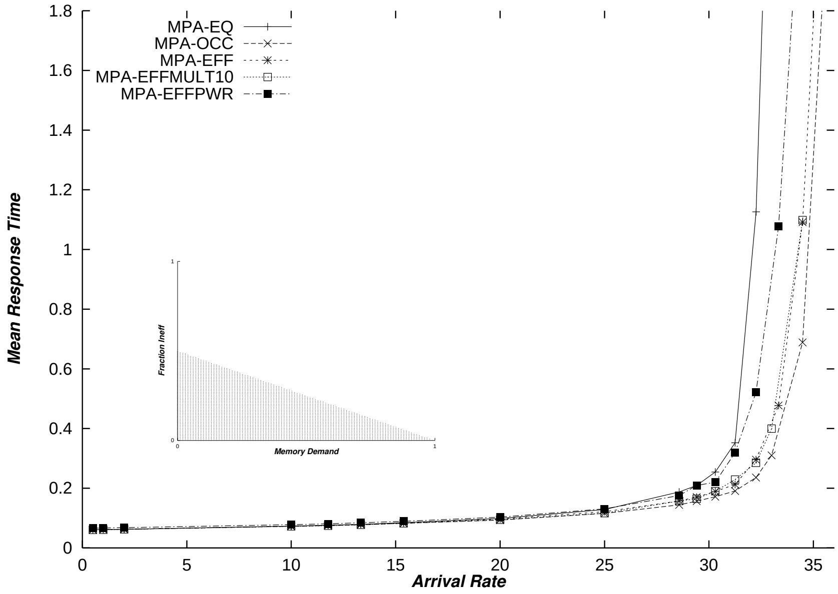


Correlated Workload 4(a)

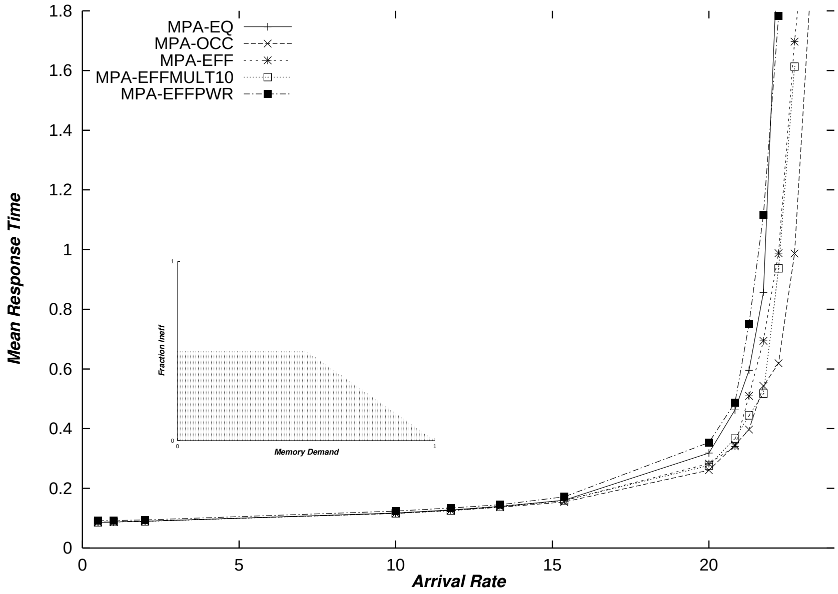


Correlated Workload 4(b)

Figure 5.8: Performance of the scheduling disciplines for general workloads having two speedup classes. The small inset graphs depict the correlation function F used to define the workload.



Correlated Workload 4(c)



Correlated Workload 4(d)

Figure 5.9: Performance of the scheduling disciplines for general workloads having two speedup classes (continued from previous figure).

5.4.3 Constraining Processor Allocations

For many types of systems and applications, certain processor allocations are more natural than others. For example, in the shared-memory multiprocessor system being developed at the University of Toronto [VBS+95], natural processor allocations are multiples of four or powers of two. Also, many applications are constrained to using specific allocations, such as an even number or power of two. For this reason, we investigated the effects of constraining processor allocations on the performance of the scheduling disciplines.

Given that MPA-EFF performed almost as well as MPA-OCC in most cases and it is simple to adapt, we chose it to be the basis for the constrained-allocation disciplines. The first constraint we studied was to limit allocations to be either a single processor, or, in one case, a multiple of four and, in the other case, a multiple of ten processors.[7] But as the performance for the multiple-of-four case was nearly the same as the no-constraint case, we only show the performance of the multiple-of-ten case. The performance of this discipline, labeled MPA-EFFMULT10, is shown in Figure 5.7 and Figures 5.8 and 5.9. As can be seen, this discipline performs nearly as well as MPA-EFF, implying that we can constrain allocations in the system in this manner and still obtain excellent performance.

The second constraint was to limit allocations to be powers of two. Note that due to the way in which the algorithm is implemented, only 96 of the 100 processors were ever utilized. As can be seen, this discipline performs noticeably worse than its unconstrained counterpart. The reason is that the discipline suffers from not being able to allocate as many processors to efficient jobs as MPA-EFF can if one efficient job is scheduled with one or more inefficient ones. Thus, this discipline may benefit from better job selection to avoid situations in which it cannot make good processor allocations.

5.5 Conclusion

In this chapter, we have investigated the benefit of having speedup knowledge of individual jobs in multiprocessor scheduling. Our approach was to first model the performance of two classes of disciplines, those that have no knowledge of the speedup characteristics of individual jobs and those that have full knowledge. By using these models, we were able to determine how much benefit existed from having speedup knowledge under various workload situations. We then proposed and evaluated two scheduling disciplines that make use of speedup knowledge, and showed that they perform well in the type of workloads likely to occur in real environments.

If memory requirements and speedup are uncorrelated, then there is a moderate benefit from using speedup information in increasing the sustainable throughput. Obtaining these improvements in practice might be difficult, however, given the nature of the optimal solution. (Any deviation can lead to large performance degradations.) When memory requirements and speedup are correlated,

[7] Our system is actually expected to have a power of two processors, in which case we would choose multiples of four and eight.

it becomes much easier to improve the performance in practice. In the cases we considered, we obtained anywhere from 85% to 100% of the maximum sustainable load. Although our primary interest was to minimize mean response times, we found that matching or surpassing the performance of our baseline scheduler, Naive MPA-EQ, at all load levels to be easy even if our processor allocations are chosen to maximize throughput.

This chapter considered the case where jobs have extremely different speedup characteristics. We have found that such differences are crucial for one to obtain the level of performance improvement described in this chapter. If most jobs share approximately the same speedup characteristics, then the benefits of using efficiency information naturally decreases.

Chapter 6

Implementation of LSF-Based Scheduling Extensions

One of the frequent criticisms made about parallel-job scheduling research is that proposed disciplines are rarely implemented and even more rarely ever become part of commercial scheduling systems. If we consider the commercial scheduling systems presently available, one would have to agree. Typically, these systems only support rigid run-to-completion disciplines, leading to high response times and low system efficiencies. As a result, processor utilization of only 70% and response times measured in hours (despite median job lengths of only minutes) are considered to be common [Hot96a].

Given the few available choices, high-performance computing centers have turned to implementing their own scheduling software to meet the needs of their users [Hen95, Lif95, SCZL96, WMKS96]. Commercial scheduling software companies have responded to this need by providing mechanisms allowing external (customer-provided) policies to be implemented on top of the existing software base [SCZL96].

In this section, the implementation of a variety of fully-functional scheduling disciplines is described. The primary objective of this work is to demonstrate that sophisticated parallel-job scheduling disciplines can be practically developed. To support this objective, the source code for each of our scheduling disciplines is included in the appendices. The disciplines investigated here span the entire range of possibilities, from rigid to adaptive disciplines, from run-to-completion to malleable preemption, and from no knowledge to both speedup and service-demand knowledge. A secondary objective of our work is to briefly examine the benefits adaptability, preemption, and knowledge may have on the performance of such disciplines.

Recently, Gibbons has shown that historical data can be used to predict the service demands of jobs [Gib96, Gib97], especially if different memory requirements correspond to different service demands (i.e., the memory requirements reflect the problem size). If a small number of applications represent a significant fraction of the workload, then it is possible to obtain speedup information for these jobs by direct measurement. Alternatively, speedup information can be relatively accurately measured during execution using multiprocessor monitoring facilities [NVZ96]; such information could then be used in the same way as historical service-demand information [Gib96, Gib97]. Although available speedup and service-demand information may only be approximate in practice, in this chapter, we consider the case where they are exact.

A scheduling system for a distributed or parallel multiprocessor involves user interfaces for job management, infrastructure for monitoring the state of the processors, and mechanisms for starting and signaling jobs remotely, all representing significant software development. The approach taken for this implementation work was to make use as much as possible of existing software in order to concentrate on the development of the scheduling disciplines. Given the close relationship between the University of Toronto and Platform Computing, the software we chose was this company's Load Sharing Facility (LSF). We found that we could make direct use of LSF for many aspects of job management, including the user interfaces for submitting and monitoring jobs, as well as the low-level mechanisms for starting, stopping, and resuming jobs. For our purposes, it was necessary to disable (or work around) LSF's internal scheduling policies in order to study our own.

6.1 Design of Scheduling Disciplines

6.1.1 Load Sharing Facility

The Load Sharing Facility (LSF) is a commercial distributed scheduling system, used to balance interactive jobs across the processors of the system and to schedule batch jobs. Of greatest relevance to this chapter is the batch scheduling system, which currently supports both sequential and parallel jobs.

Queues provide the basis for much of the control over the scheduling of jobs. Each queue is associated with a set of processors, a priority, and many other parameters not described here. By default, jobs are selected in FCFS order from the highest-priority non-empty queue and run until completion, but it is possible to configure queues so that higher-priority jobs preempt lower priority ones (a feature that is available only for the sequential-job case). The priority of a job is defined by the queue to which the job has been submitted.

To illustrate the use of queues, consider a policy where shorter jobs have higher priority than longer jobs (see Figure 6.1). An administrator could define several queues, each in turn corresponding to increasing service demand and having decreasing priority. If jobs are submitted to the correct queue, short jobs will be executed before long ones. Moreover, LSF can be configured to preempt lower priority jobs if higher priority ones arrive, giving short jobs still better responsiveness. To permit enforcement of the policy, LSF can be configured to terminate jobs that exceed the execution-time threshold defined for the queue.

The current version of LSF provides only limited support for parallel jobs. As part of submitting a job, a user can specify the number of processors required. When LSF finds a sufficient number of processors satisfying the resource constraints for the job, it spawns an application "master" process

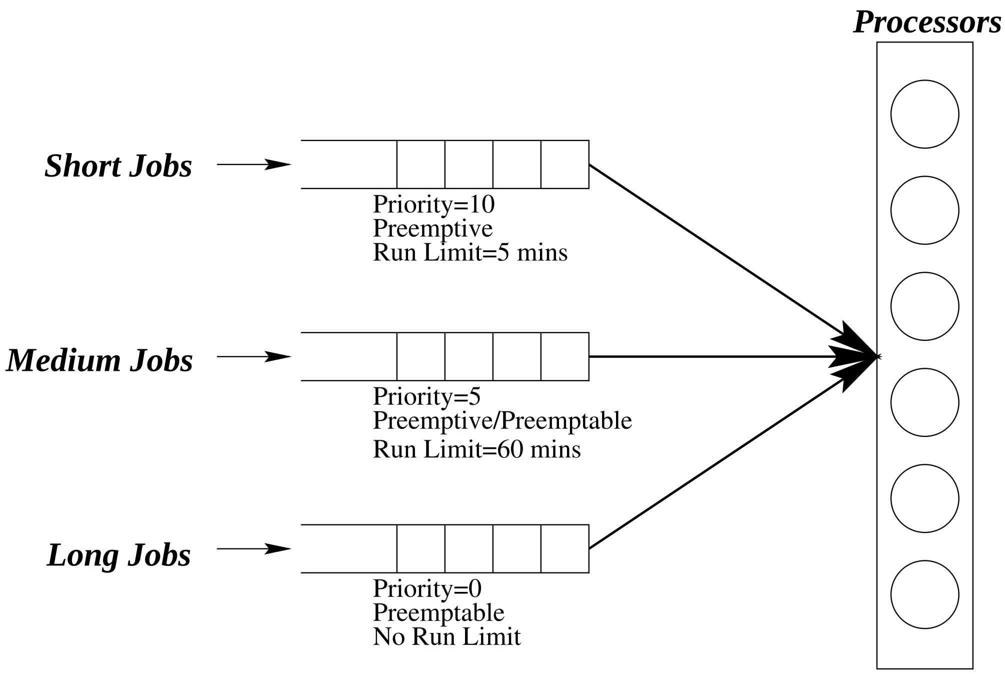


Figure 6.1: Example of a possible sequential-job queue configuration in LSF to favour short-running jobs. Jobs submitted to the short-job queue have the highest priority, followed by medium- and long-job queues. The queues are configured to be preemptable (allowing jobs in the queue to be preempted by higher-priority jobs) and preemptive (allowing jobs in the queue to preempt lower-priority jobs). Execution-time limits associated with each queue enforce the intended policy.

on one of the processors, passing to this process a list of processors. The master process can then use this list of processors to spawn a number of "slave" processes to perform the parallel computation. The slave processes are completely under the control of the master process, and as such, are not known to the LSF batch scheduling system. LSF does provide, however, a library that simplifies several distributed programming activities, such as spawning remote processes, propagating Unix signals, and managing terminal output.

6.1.2 Scheduling Extension Library

The ideal approach to developing new scheduling disciplines is one that does not require any LSF source code modifications, as this allows any existing users of LSF to experiment with the new disciplines. Fortunately, LSF provides an extensive application-programmer interface (API), allowing many aspects of job scheduling within LSF to be controlled. Our scheduling disciplines are implemented within a process distinct from LSF, and are thus called *scheduling extensions*.

The LSF API, however, is designed to implement LSF-related commands rather than scheduling extensions. As a result, the interfaces are very low level and can be quite complex to use. For example, to determine the accumulated run time for a job—information commonly required by a scheduler—the programmer must use a set of LSF routines to open the LSF event-logging file, process each log item in turn, and compute the time between each pair of suspend/resume events for the job. Since the event-logging file is typically several megabytes in size, requiring several seconds to process in its entirety, it is necessary to cache information whenever possible. Clearly, it is difficult for a scheduling extension to take care of such details and to obtain the information efficiently.

One of our research goals was thus to design a scheduling extension library that would provide simple and efficient access to information about jobs (e.g., processors currently used by a job), as well as to manipulate the state of jobs in the system (e.g., suspend or migrate a job). This functionality is logically divided into two components:

Job and System Information Cache (JSIC) This component contains a cache of system and job information obtained from LSF. It also allows a discipline to associate auxiliary, discipline-specific information with processors, queues, and jobs for its own book-keeping purposes.[1]

LSF Interaction Layer (LIL) This component provides a generic interface to all LSF-related activities. In particular, it updates the JSIC data structures by querying the LSF batch system and translates high-level parallel-job scheduling operations (e.g., suspend job) into the appropriate LSF-specific ones.

The basic designs of all our scheduling disciplines are quite similar. Each discipline is associated with a distinct set of LSF queues, which the discipline uses to manage its own set of jobs. All

[1] In future versions of LSF, it will be possible for information associated with jobs to be saved in log files so that it will not be lost in the event that the scheduler fails.

LSF jobs in this set of queues are assumed to be scheduled by the corresponding scheduling discipline. Normally, one LSF queue is designated as the submit queue, and other queues are used by the scheduling discipline as a function of a job's state. For example, pending jobs may be placed in one LSF queue, stopped jobs in another, and running jobs in a third. A scheduling discipline never explicitly dispatches or manipulates the processes of a job directly; rather, it implicitly requests LSF to perform such actions by switching jobs from one LSF queue to another. Continuing the same example, a pending queue would be configured so that LSF accepts jobs but never dispatches them, and a running queue would be configured so that LSF immediately dispatches any job in this queue on the processors specified for the job. In this way, a user submits a job to be scheduled by a particular discipline simply by specifying the appropriate LSF queue, and can track the progress of the job using all the standard LSF utilities.

Although it is possible for a scheduling discipline to contain internal job queues and data structures, we have found that this is rarely necessary because any state information that needs to be persistent can be encoded by the queue in which each job resides. This approach greatly simplifies the re-initialization of the scheduling extension in the event that the extension fails at some point, an important property of any production scheduling system.

Given our design, it is possible for several scheduling disciplines to coexist within the same extension process, a feature that is most useful in reducing overheads if different disciplines are being used in different partitions of the system. (For example, one partition could be used for production workloads while another could be used to experiment with a new scheduling discipline.) Retrieving system and job information from LSF can place significant load on the master processor,[2] imposing a limit on the number of extension processes that can be run concurrently. Since each scheduling discipline is associated with a different set of LSF queues, the set of processors associated with each discipline can be defined by assigning processors to the corresponding queues using the LSF queue administration tools. (Normally, each discipline uses a single queue for processor information.)

The extension library described here has also been used by Gibbons in studying a number of rigid scheduling disciplines, including two variants of EASY [Lif95, SCZL96, Gib96, Gib97]. One of the goals of Gibbons' work was to determine whether historical information about a job could be exploited in scheduling. He found that, for many workloads, historical information could provide up to 75% of the benefits of having perfect information. For the purpose of his work, Gibbons added an additional component to the extension library to gather, store, and analyze historical information about jobs. He then adapted the original EASY discipline to take into account this knowledge and showed how performance could be improved. The historical database and details of the scheduling disciplines studied by Gibbons are described elsewhere [Gib96, Gib97].

The high-level organization of the scheduling extension library (not including the historical database) is shown in Figure 6.2. The extension process contains the extension library and each of the disciplines configured for the system. The extension process mainline essentially sleeps until a schedul-

[2] LSF runs its batch scheduler on a single, centralized processor.

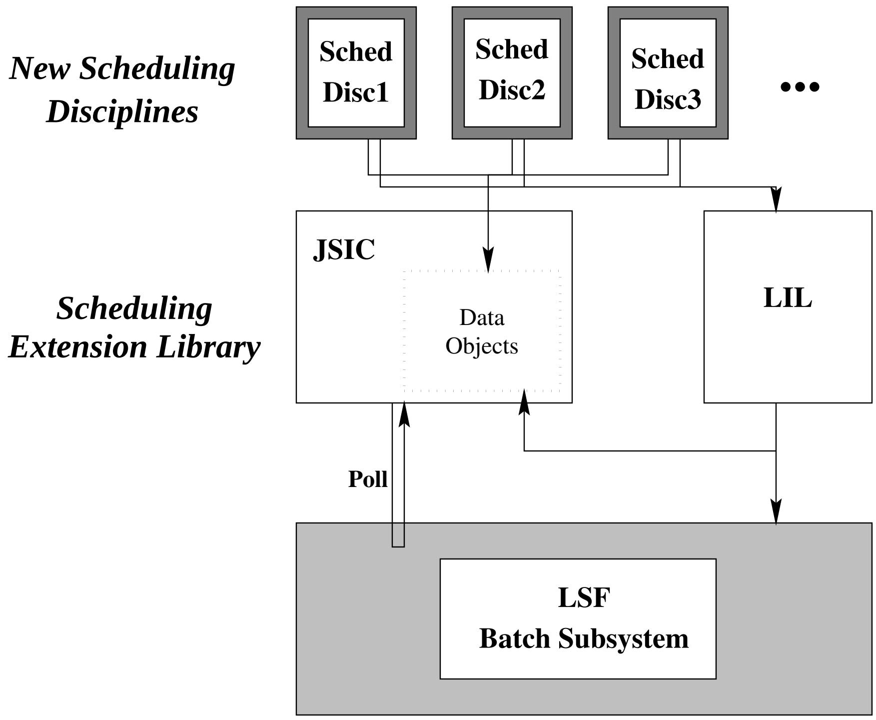


Figure 6.2: High-level design of scheduling extension extension library. As shown, the extension library supports multiple scheduling disciplines running concurrently within the same process.

ing event or a timeout (corresponding to the scheduling quantum) occurs. The mainline then prompts the LIL to update the JSIC and calls a designated method for each of the configured disciplines.

Next, we describe each component of the extension library in detail. Also, the header files for the JSIC and the LIL can be found in Appendices B and C.

Job and System Information Cache

The Job and System Information Cache (JSIC) contains all the information about jobs, queues, and processors that is relevant to the scheduling disciplines that are part of the extension. The layout of the data structures was designed taking into consideration the types of operations that we found to be most critical to the design of our scheduling disciplines:

- A scheduler must be able to scan sequentially through the jobs associated with a particular LSF queue. For each job, it must then be able to access in a simple manner any job-related information obtained from LSF (e.g., run times, processors on which a job is running, LSF job state).

- It must be able to scan the processors associated with any LSF queue and determine the state of each one of these (e.g., available or unavailable).

- Finally, a scheduler must be able to associate book-keeping information with either jobs or processors (e.g., the set of jobs running on a given processor).

The layout of our data structures is illustrated in Figure 6.3. First, information about each active job is stored in a `JobInfo` object. Pointers to instances of these objects are stored in a job hash table keyed by LSF job identifiers (`jobId`), allowing efficient lookup of individual jobs. Also, a list of job identifiers is maintained for each queue, permitting efficient scanning of jobs in any given queue (in the order submitted to LSF).

The information associated with a job is global, in that a single `JobInfo` object instance exists for each job. For processors, on the other hand, we found it convenient for our experimentation to have distinct processor information objects associated with each queue.[3] An approach similar to that for jobs would also be suitable if it is guaranteed that a processor is never associated with more than one discipline within an extension. Similar to jobs, processors associated with a queue can be scanned sequentially, or can be accessed through a hash table keyed on the processor name.

LSF Interaction Layer (LIL)

The most significant function of the LSF interaction layer is to update the JSIC data structures to reflect the current state of the system when prompted. Since LSF only supports a polling interface, however, the LIL must, for each update request, fetch all data from LSF and compare it to that which is currently stored in the JSIC. As part of this update, the JSIC must also process an event logging file, since certain types of information (e.g., total times pending, suspended, and running) are not provided directly by LSF. As such, the JSIC update code represents a large fraction of the total extension library code. (The extension library is approximately 1.5 KLOC.)

To update the JSIC, the LIL performs the following three actions:

- It obtains the list of all active jobs in the system from LSF. Each job record returned by LSF contains some static information, such as the submit time, start time, resource requirements, as well as some dynamic information, such as the job status (e.g., running, stopped), processor set, and queue. All this information about each job is recorded in the JSIC.
- It opens the event-logging file, reads any new events that have occurred since the last update, and re-computes the pending time, aggregate processor run time, and wall-clock run time for each job. In addition, aggregate processor and wall-clock run times since the job was last resumed (termed residual run times) are computed.
- It obtains the list of processors associated with each queue and queries LSF for the status of each of these processors.

LSF provides a mechanism by which the resources required by the job, such as physical memory, licenses, or swap space, can be specified upon submission. In our extensions, we do not use the default set of resources to avoid having LSF make any scheduling decisions, but rather add a new

[3] By having a distinct object for each processor in each queue, it was possible to experiment with multiple disciplines using all processors in the system simultaneously. This is explained in greater detain in Section 6.2.

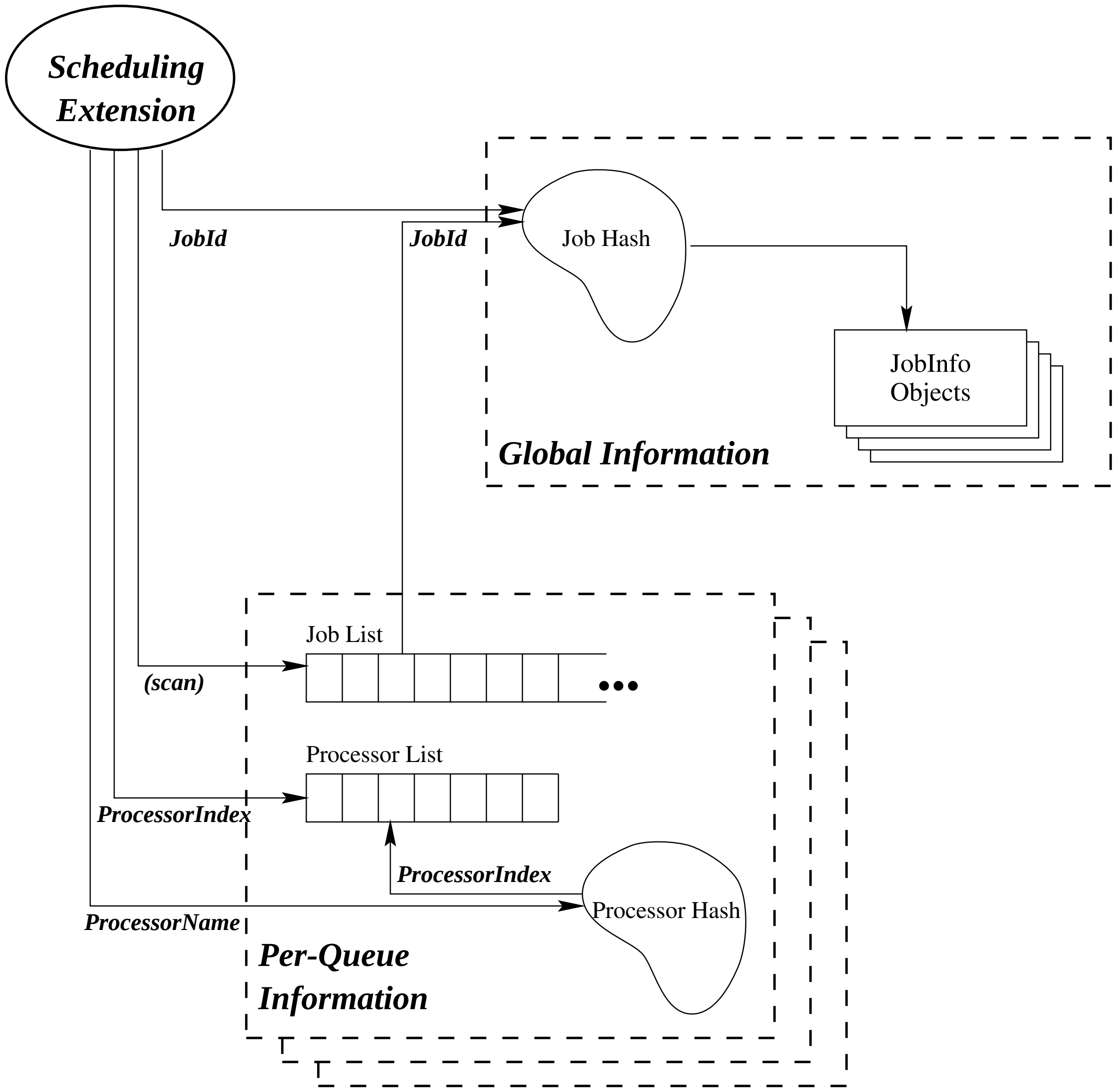


Figure 6.3: Data organization of Job and System Information Cache (JSIC).

OPERATION	DESCRIPTION
switch	This operation moves a job from one queue to another.
setProcessors	This operation defines the list of processors to be allocated to a job. LSF dispatches the job by creating a master process on the first processor in the list; as described before, the master process uses the list to spawn its slave processes.
suspend	This operation suspends a job. The processes of the job hold onto virtual resources they possess, but normally release any physical resources (e.g., physical memory).
resume	This operation resumes a job that is currently suspended.
migrate	This operation initiates the migration procedure for a job. It does not actually migrate the job, but rather places the job in a state that allows it to be restarted on a different set of processors.

Table 6.1: High-level scheduling functions provided by LSF Interaction Layer.

set of pseudo-resources that are used to pass parameters or information about a job, such as minimum and maximum processor allocations or service demand, directly to the scheduling extension. As part of the first action performed by the LIL update routine, this information is extracted from the pseudo-resource specifications and stored in the `JobInfo` structure.

The remaining LIL functions, presented in Table 6.1, basically translate high-level scheduling operations into low-level LSF calls. Since these operations affect the state (in LSF) of jobs, the effects of these operations on the JSIC must be considered. In one model, state changes resulting from the invocation of these operations are immediately reflected in the JSIC. This can pose some difficulty in the implementation of disciplines. For example, if a job is switched from one queue to another while scanning a list of jobs, the scan may need to be restarted from the beginning because the list is no longer the same. An alternative model is that the effects of these operations are only reflected in the JSIC the next time the scheduler is awakened. In this case, a discipline may have to locally keep track of modifications it may have made. Having experimented with both models, we find the latter to be simpler to use.

Preemption Considerations The LSF interaction layer makes certain assumptions about the way in which jobs can be preempted. For simple preemption, a job can be suspended by sending it a `SIGTSTP` signal, which is delivered to the master process; this process must then propagate the signal to its slaves (which is automated in the distributed programming library provided by LSF) to ensure that all processes belonging to the job are stopped. Similarly, a job can be resumed by sending it a `SIGCONT` signal.

In our extension library, job migration (without malleability) makes the assumption that a parallel job can be checkpointed, which for computationally-intensive parallel codes is quite realistic. For example, the Condor system now provides a transparent checkpointing facility for parallel applications using either MPI or PVM [PL96]. When a checkpoint is requested, the run-time library flushes any network communications and I/O and saves the images of each process involved in the computation to disk; when the job is restarted, the library re-establishes the necessary socket connections and resumes the computation from the point at which the last checkpoint was taken.

To allow jobs to be migrated in LSF, we first set a flag in the submission request indicating that the job is re-runnable. The migration is then performed by sending a checkpoint signal to the job (in our case, the `SIGUSR2` signal), and then requesting LSF to migrate the job. This would normally cause LSF to terminate the job (with a `SIGTERM` signal) and restart it on the set of processors specified (using the **setProcessors** interface). In most cases, however, we switch a job to a queue that has been configured to not dispatch jobs before submitting the migration request, causing the job to be simply terminated and requeued as a pending job.

The interface for changing the processor allocation of a malleable job is identical to that for migrating a job, the only difference being the way it is used. In the migratable case, the scheduling discipline always restarts a job using the same number of processors as in the initial allocation, while in the malleable case, any number of processors can be specified. In practice, checkpointing a malleable job is more complex than checkpointing a migratable one, since it is not possible to simply checkpoint process images. In this thesis, our goal is not to investigate such checkpointing issues (other than to note that malleability has been successfully implemented before [NVZ96]), but rather to investigate the benefits of such features if they were to be available.

A Simple Example

To illustrate how the extension library can be used to implement a discipline, consider a sequential-job, multi-level feedback discipline that degrades the priority of jobs as they acquire processing time. If the workload has a high degree of variability in service demands, as is typically the case even for batch sequential workloads, this approach will greatly improve response times without requiring users to specify the service demands of jobs in advance. For this discipline, we can use the same queue configuration as shown in Figure 6.1; we eliminate the run-time limits, however, as the scheduling discipline will automatically move jobs from higher-priority queues to lower-priority ones as they acquire processing time.

Users initially submit their jobs to the high-priority queue (labeled short jobs in Figure 6.1); when the job acquires, say, 120 seconds of processing time, the scheduling extension switches the job to the medium-priority queue, and after 300 seconds, to the low-priority queue. The pseudo-code for this scheduling extension is given in Figure 6.4.

In this example, the extension relies on the LSF batch system to dispatch, suspend, and resume jobs as a function of the jobs in each queue. Users can thus track the progress of jobs simply by examining the jobs in each of the three queues.

6.1.3 LSF-Based Parallel-Job Scheduling Disciplines

We now turn our attention to the parallel-job scheduling disciplines that have been implemented as LSF extensions. As mentioned in the introduction, a wide range of disciplines have been considered; for convenience, these are reviewed in Table 6.2.

proc *mlfb*(*hqueue*, *lqueue*, *threshold*)
  foreach *j* **in** *hqueue*
    if *j.cumRunTime* > *threshold* **then**
      switch(*j*, *lqueue*)
    endif
  endfor
endproc

mlfb(*highpri_queue*, *medpri_queue*, 120)
mlfb(*medpri_queue*, *lowpri_queue*, 300)

Figure 6.4: Multi-level feedback example for sequential jobs. *j.cumRunTime* is the cumulative run time of job j; also, the high-, medium-, and low-priority queues are labeled *highpri_queue*, *medpri_queue*, *lowpri_queue*, respectively.

	RIGID	**ADAPTIVE**
RTC	LSF-RTC	LSF-RTC-AD
PREEMPTION		
SIMPLE	LSF-PREEMPT	LSF-PREEMPT-AD
MIGRATABLE	LSF-MIG	LSF-MIG-AD
		LSF-MIG-ADSUBSET
MALLEABLE		LSF-MALL-AD
		LSF-MALL-ADSUBSET

Table 6.2: Range of LSF-based parallel-job scheduling implementations studied in this thesis. Both rigid and adaptive disciplines use service-demand information, if available, to run jobs having the least remaining service demand. Adaptive disciplines use speedup-related information, if available, in choosing processor allocations. All disciplines ensure that jobs are allocated at least their minimum processor allocation requirements (corresponding to the memory requirements of jobs).

There are some notable scheduling costs associated with using LSF on our platform, which is a network of workstations (i.e., distributed-memory system). It can take up to thirty seconds to dispatch a job once it is ready to run. Migratable or malleable preemption typically requires more than a minute to release the processors associated with a job; these processors are considered to be unavailable during this time. Finally, scheduling decisions are made at most once every five seconds to keep the load on the master (scheduling) processor to an acceptable level.

The disciplines described in this section all share a common job queue configuration. A pending queue is defined and configured to allow jobs to be submitted (i.e., *open*) but preventing any of these jobs from being dispatched automatically by LSF (i.e., *inactive*). A second queue, called the run queue, is used by the scheduler to start jobs. This queue is open, active, and possesses absolutely no load constraints. A scheduling extension uses this queue by first specifying the processors associated with a job (i.e., **setProcessors**) and then moving the job to this queue; given the queue configuration, LSF immediately dispatches jobs in this queue. Finally, a third queue, called the stopped queue, is defined to assist in migrating jobs. It too is configured to be open but inactive. When LSF is prompted to migrate a job in this queue, it terminates and requeues the job, preserving its job identifier. In all our disciplines, preempted jobs are left in this stopped queue to distinguish them from jobs that have not had a chance to run yet (in the pending queue).

Each job in our system is associated with a minimum, desired, and maximum processor allocation, the desired value lying between the minimum and maximum. Rigid disciplines use the desired value while adaptive disciplines are free to choose any allocation between the minimum and the maximum values. Speedup characteristics of a job are specified in terms of the fraction of work that is sequential. Finally, service demand is specified as the amount of computation time for the job if it were to run on a single processor.

All our disciplines are designed to use service-demand and/or speedup information if available. Basically, service-demand information is used to run jobs having the least remaining processing time (to minimize mean response times) and speedup information is used to favour efficient jobs in processor allocation. Since jobs can vary considerably in terms of their speedup characteristics, estimates of the remaining processing time will only be accurate if speedup information is available.

Run-to-Completion Disciplines

Next, we describe the run-to-completion disciplines. All three variants listed in Table 6.2 (i.e., LSF-RTC, LSF-RTC-AD, and LSF-RTC-ADSUBSET) are quite similar and, as such, are implemented in a single module. The source code for this module is provided in Appendix D.

The LSF-RTC discipline is the most straightforward, reflecting the type of discipline often used in practice today. It is defined as follows:

LSF-RTC Whenever a job arrives or departs, the scheduler repeatedly scans the pending queue until it finds the first job for which enough processors are available. It assigns processors to the job and switches the job to the run queue.

LSF, and hence the JSIC, maintains jobs in order of arrival, so the default discipline is FCFS (skipping any jobs at the head of the queue for which not enough processors are available). If service-demand information is provided to the scheduler, then jobs are scanned in order of increasing service demand, resulting in a SPT discipline (again with skipping).

The LSF-RTC-AD discipline is similar to the ASP discipline described in Chapter 3, except that jobs are selected for execution differently because the LSF-based disciplines take into account memory requirements of jobs (and hence cannot be called ASP).

LSF-RTC-AD Whenever a job arrives or departs, the scheduler repeatedly scans the pending queue, selecting the first job for which enough processors remain to satisfy the job's minimum processor requirements. When no more jobs fit, leftover processors are used to equalize processor allocations among selected jobs (i.e., giving processors to jobs having the smallest allocation). The scheduler then assigns processors to the selected jobs and switches these jobs to the run queue.

As in the disciplines presented in Chapter 5, we use speedup information not in selecting jobs, but in allocating leftover processors (i.e., available processors in excess of the sum of the minimum processor allocations of selected jobs). Given speedup information, the scheduler allocates each leftover processor, in turn, to the job whose efficiency will be highest *after* the allocation (as in the MPA-EFF discipline presented in Chapter 5). This approach minimizes both the processor and memory occupancy in a distributed-memory environment, leading to the highest possible sustainable throughput.

The SUBSET variant seeks to improve the efficiency by which processors and memory are utilized by incorporating the subset-sum memory packing algorithm that was introduced in Chapter 5.

LSF-RTC-ADSUBSET Let L be the number of jobs in the system and N_{ff} be the number of jobs selected by the first-fit algorithm used in LSF-RTC-AD. The scheduler only commits to running the first N' of these jobs, where

$$N' = \left\lfloor N_{\mathrm{ff}} * \max(1 - \frac{L}{\delta N_{\mathrm{ff}}}, 0) \right\rfloor$$

Using any leftover processors and leftover jobs, the scheduler applies the subset-sum algorithm to pack leftover processors as efficiently as possible (again using the minimum processor requirements for each job). The jobs chosen by the subset-sum algorithm are added to the list of jobs selected to run, and any remaining processors are allocated as in LSF-RTC-AD.

For these experiments, we chose $\delta = 5$ which is much lower than that used in the previous chapter. The reason is that higher values of δ did not lead to significant differences in the selection of jobs relative to the non-SUBSET variant, given the load at which we ran our experiments (queue lengths were rarely larger than 10).

Simple Preemptive Disciplines

In simple preemptive disciplines, jobs may be suspended but their processes may not be migrated. Since the resources used by jobs are not released when they are in a preempted state, however, one must be careful to not over-commit system resources. In our disciplines, this is achieved by ensuring that no more than a certain number of processes ever exist on any given processor. In a more sophisticated implementation, we might instead ensure that the swap space associated with each processor would never be overcommitted.

The two variants of the preemptive disciplines are quite different. In the rigid discipline, we allow a job to preempt another only if it possesses the same desired processor allocation. This is to minimize the possibility of packing losses that might occur if jobs were not aligned in this way. In the adaptive discipline, we found this approach to be problematic. Consider a long-running job, either arriving during an idle period or having a large minimum processor requirement, that is dispatched by the scheduler. Any subsequent jobs preempting this first one would be configured for a large allocation size, causing them, and hence the entire system, to run inefficiently. As a result, we do not attempt to reduce packing losses with the adaptive, simple preemptive discipline. The source code to both these disciplines is given in Appendices F and G, respectively.

LSF-PREEMPT Whenever a job arrives or departs or when a quantum expires, the scheduler re-evaluates the selection of jobs currently running. Available processors are first allocated in the same way as in LSF-RTC. Then, the scheduler determines if any running job should be preempted by a pending or stopped job, according to the following criteria:

1. A stopped job can only preempt a job running on the same set of processors as those for which it is configured. A pending job can preempt any running job that has the same desired processor allocation value.
2. If no service-demand information is available, the aggregate cumulative processor time of the pending or stopped job must be some fraction less than that of the running job (in our case, we use the value of 50%); otherwise, the service demand of the preempting job must be a (different) fraction less that that of the running job (in our case, we use the value of 10%).
3. The running job must have been running for at least a certain specified amount of time (one minute in our case, since suspension and resumption only consist of sending a Unix signal to all processes of the job).
4. The number of processes present on any processor cannot exceed a pre-specified number (in our case, five processes).

If several jobs can preempt a given running job, the one which has the least acquired aggregate processing time is chosen first if no service-demand knowledge is available, or the one with the shortest remaining service demand if service-demand knowledge is available.

Our adaptive, simple preemptive discipline uses a matrix approach to scheduling jobs, where each row of the matrix represents a different set of jobs to run and the columns the processors in the system. In Ousterhout's co-scheduling discipline, an incoming job would be placed in the first row of the matrix that has enough free processors for the job; if no such row exists, then a new one is created. In our approach, we use a more dynamic approach.

LSF-PREEMPT-AD Whenever the scheduler is awakened (due either to an arrival or departure or to a quantum expiry), the set of jobs currently running or stopped (i.e., preempted) are organized into the matrix just described, using the first row for those jobs that are running. Each row is then examined in turn. For each, the scheduler populates the uncommitted processors with the best pending, stopped, or running jobs. (If service-demand information is available, currently-stopped or running jobs may be preferable to a pending job; these jobs can switch rows if all processors being used by the job are uncommitted in the row currently being examined.) The scheduler also ensures that jobs that are currently running, but which have run for less than the minimum time since last being started or resumed, continue to run. If such jobs cannot be accommodated in the row being examined, then the scheduler skips to the next row.

Once the set of jobs that might be run in each row has been determined, the scheduler chooses the row that has the job having the least acquired processing time or, if service-demand information is available, the job having the shortest remaining service demand. Processors in the selected row available for pending jobs are distributed as before (i.e., equi-allocation if no speedup knowledge is available, or favouring efficient jobs if it is).

Migratable and Malleable Preemptive Disciplines

The migratable and malleable preemptive disciplines assume that a job can be checkpointed and restarted at a later point in time. The adaptive versions of these disciplines are quite similar, the only difference being that in the migratable case, jobs are always resumed with the same number of processors allocated when the job first started. So, we describe the migratable and malleable adaptive disciplines together.

LSF-MIG Whenever a job arrives or departs or when a quantum expires, the scheduler re-evaluates the selection of jobs currently running. First, currently-running jobs which have not run for at least a certain configurable amount of time (in our case, ten minutes, since migration and processor reconfiguration are relatively expensive) are allowed to continue running. Processors not used by these jobs are considered to be available for re-assignment. The scheduler then uses a first-fit algorithm to select the jobs from those remaining to run next, using a job's desired processor allocation. As before, if service-demand information is available, jobs are selected in order of least remaining service demand.

LSF-MIG-AD and LSF-MALL-AD Apart from their adaptiveness, these two disciplines are very similar to the LSF-MIG discipline. In the malleable version, the scheduler uses the same first-fit algorithm as in LSF-MIG to select jobs, except that it always uses a job's minimum processor allocation to determine if a job fits. Any leftover processors are then allocated as before, using an equi-allocation approach if no speedup information is available, and favouring efficient jobs otherwise. In the migratable version, the scheduler uses the size of a job's current processor allocation instead of its minimum if the job has already run (i.e., has been preempted) in the first-fit algorithm, and does not change the size of such a job's processor allocation if selected to run.

SUBSET-variants of the adaptive disciplines have also been implemented to improve the efficiency with which processors and memory are utilized. The modification is identical to the one used in LSF-RTC-ADSUBSET, and is not repeated here. All five of these disciplines have been implemented within a single module, the source code of which is presented in Appendix E.

6.2 Evaluation Methodology

In this section, we evaluate the relative performance of our disciplines to illustrate the benefits of various levels of preemption and allocation flexibility. The evaluation is intended to be primarily qualitative in nature for two reasons. First, experiments must be performed in real time rather than in simulated time, permitting only a small number of jobs to be run (relative to the types of experiments conducted in previous chapters). Also, the propensity for failures to occur during these tests on our experimental platform placed a significant limitation on the number of experiments that could be successfully run.[4] Our objective is to demonstrate the practicality of each class of discipline and to observe its performance in a real context, rather than to analyze its performance under a wide variety of conditions (for which a simulation would be more suitable).

The experimental platform for the implementation is a network of workstations (NOW), consisting of sixteen IBM 43P (133MHz, PowerPC 604) systems, connected by three independent networks (155 Mbps ATM, 100 Mbps Ethernet, 10 Mbps Ethernet). It is a fully distributed system, having relatively high messaging latencies. Although some shared-memory-based applications have been ported to this platform using distributed virtual shared memory (DSVM) [PBS96], we treat the system as a distributed-memory system.

To exercise the scheduling software, we use a parameterizable synthetic application designed to represent real applications. The basic reason for using a synthetic application is that it could be designed to not use any processing resources, yet behave in other respects (e.g., execution time, preemption) as a real parallel application. This is important in the context of our network of workstations, because the system is being actively used by a number of other researchers. Using real

[4]Since each experiment required 24 to 48 hours to run, more often than not, either one of the processors, the LSF batch system, or the file system failed in some manner during any given experiment, invalidating the data. This does not necessarily reflect the usability of the system, as the scheduler can tolerate node failures, but such failures do affect the performance data.

(compute-intensive) applications would have prevented the system from being used by others during the tests, or would have caused the tests to be inconclusive if jobs were run at low priority.

Each of our scheduling disciplines ensures that only a single one of its jobs is ever running on a given processor and that all processes associated with the job are running simultaneously. As such, the behaviour of our disciplines, when used in conjunction with our synthetic application, is identical to that of a dedicated system running compute-intensive applications. In fact, by associating a different set of queues with each discipline, each one configured to use all processors, it was possible to conduct several experiments concurrently. (The jobs submitted to each submit queue for the different disciplines were generated independently.)

The synthetic application possesses three important features. First, it can be easily parameterized with respect to speedup and service demand, allowing it to model a wide range of real applications. Second, it supports adaptive processor allocations using the standard mechanism provided by LSF. Finally, it can be checkpointed and restarted, to model both migratable and malleable jobs. The source code for this application is presented in Appendix A.

The experiments consist of submitting a sequence of jobs to the scheduler according to a Poisson arrival process, using an arrival rate that reflects a moderately-heavy load. A small initial number of these jobs (e.g., 200) are tagged for mean response time and makespan[5] measurements. Each experiment terminates only when all jobs in this initial set have left the system. To make the experiment more representative of large systems, we assume that each processor corresponds to eight processors in reality. Thus, all processor allocations are multiples of eight, and the minimum allocation is eight processors.[6] Scaling the number of processors in this way affects the synthetic application in determining the amount of time it should execute and the scheduling disciplines in determining the expected remaining service demand for a job.

Service demands for jobs are drawn from a hyper-exponential distribution, with mean of 8000 seconds (2.2 hours) and coefficient of variation (CV) of 4. This mean of the distribution is less than a quarter of that observed in the Cornell Theory Center workload (scaled to 128 processors), but our experiments would have required too much time had we used the higher value. Since such high-CV distributions are typically unstable over small sample sizes (in terms of moments of the distribution), it was necessary to repeatedly generate the initial sequence of 200 jobs until the CV was close to (in our case, within 25% of) the desired value. All disciplines received exactly the same sequence of jobs in any particular experiment, and in general, an experiment required anywhere from 24 to 48 hours to complete.

Minimum processor allocation sizes are uniformly chosen from one to sixteen processors, and

[5]We extend the definition of makespan to be the maximum completion time of the set of jobs under consideration, even though jobs can arrive at arbitrary points in time.

[6]This is similar to the MPA-EFFMULT discipline from the previous chapter, except in MPA-EFFMULT, allocations of a single processor were permitted whereas in this case they are not. Allowing jobs to have a processor allocation of one would have entailed allowing more than a single job to run at a time on a given processor, and it was felt that such a change would have deviated too significantly from an actual implementation.

maximum sizes are set at sixteen.[7] This distribution is the same as that which was used in the previous two chapters. The processor allocation size used for rigid disciplines is chosen from a uniform distribution between the minimum and the maximum processor allocations for the job.

Since it has been shown that performance benefits of knowing speedup information are only available if a large fraction of the total work in the workload (say 75%) has good speedup, and moreover, if larger-sized jobs tend to have better speedup than smaller-sized ones [PS96a], speedups are chosen in the following way. The probability of giving a job poor speedup will be calculated as a decreasing function of its desired processor allocation p:

$$\text{Prob[poor speedup]} = \begin{cases} 1 - \frac{p-1}{P} & 1 \leq p \leq \frac{P}{2} \\ 0 & \frac{P}{2} < p \end{cases}$$

So, jobs having a minimum processor allocation of one will have a 100% chance of having poor speedup, jobs having a minimum processor allocation of $P/2$ will have a 0% chance of having poor speedup with a linear relationship in between. For the purpose of these experiments, good speedup corresponds to a job for which 99.9% of the work is fully parallelizable, and a poor speedup to one for which only 90% of the work is fully parallelizable. (This approach is similar to the workload used in the results of Figure 5.8(b).)

6.3 Results and Lessons Learned

The performance results of all disciplines under the four knowledge cases (no knowledge, service-demand knowledge, speedup knowledge, or both) are given in Table 6.3 and summarized in Figures 6.5 and 6.6. As can be seen, the response times for the run-to-completion disciplines are much higher (by up to an order of magnitude) than those for the migratable or malleable preemptive disciplines. The simple preemptive, rigid discipline does not offer any advantages over the corresponding run-to-completion version. The reason is that there is insufficient flexibility in allowing a job to only preempt another that has the same desired processor requirement. The adaptive preemptive discipline is considerably better in this regard.

Adaptability appears to have the most positive effect for run-to-completion and malleable disciplines (see Figure 6.6). In the former case, makespans decreased by nearly 50% from the rigid to the adaptive variant using the subset-sum algorithm. To achieve this improvement, however, the mean response times generally increased because processor allocations tended to be smaller (leading to longer average run times). In the malleable case, adaptability resulted in smaller but noticeable decreases in makespans (5–10%). It should be noted that the opportunity for improvement is much lower than in the RTC case because the minimum makespan is 65412 seconds for this experiment (compared to actual observed makespans of approximately 78000 seconds).

[7] As mentioned before, having maximum processor allocation information is only useful at lighter loads, since at heavy loads, jobs seldom receive many more processors than their minimum allocation.

DISCIPLINE	NO KNOWLEDGE		SERVICE-DEMAND		SPEEDUP		BOTH	
	MRT	MAKESPAN	MRT	MAKESPAN	MRT	MAKESPAN	MRT	MAKESPAN
LSF-RTC	5853	147951	4040	140342	5279	130361	5627	143507
LSF-RTC-AD	10611	129093	8713	126531	8034	91003	8946	126917
LSF-RTC-ADSUBSET	8264	76637	8410	81767	8039	73324	8074	75340
LSF-PREEMPT	5793	145440	5039	143686	5280	130314	5028	143631
LSF-PREEMPT-AD	>2293	$>219105(2)$	1078	127204	2207	172768	821	111489
LSF-MIG	678	83985	662	81836	690	82214	660	82708
LSF-MIG-AD	769	88488	858	103876	784	86080	>1342	$>192031(1)$
LSF-MIG-ADSUBSET	770	90789	854	106065	769	85828	>1347	$>193772(1)$
LSF-MALL-AD	667	77534	632	78760	666	78215	650	78840
LSF-MALL-ADSUBSET	681	78537	680	79191	680	76481	644	78065

Table 6.3: Performance of LSF-based scheduling disciplines. In some trials, the discipline did not terminate within a reasonable amount of time; in these cases, a minimum bound on the mean response times is reported (indicated by a $>$) and the number of unfinished jobs is given in parenthesis. Units are in seconds.

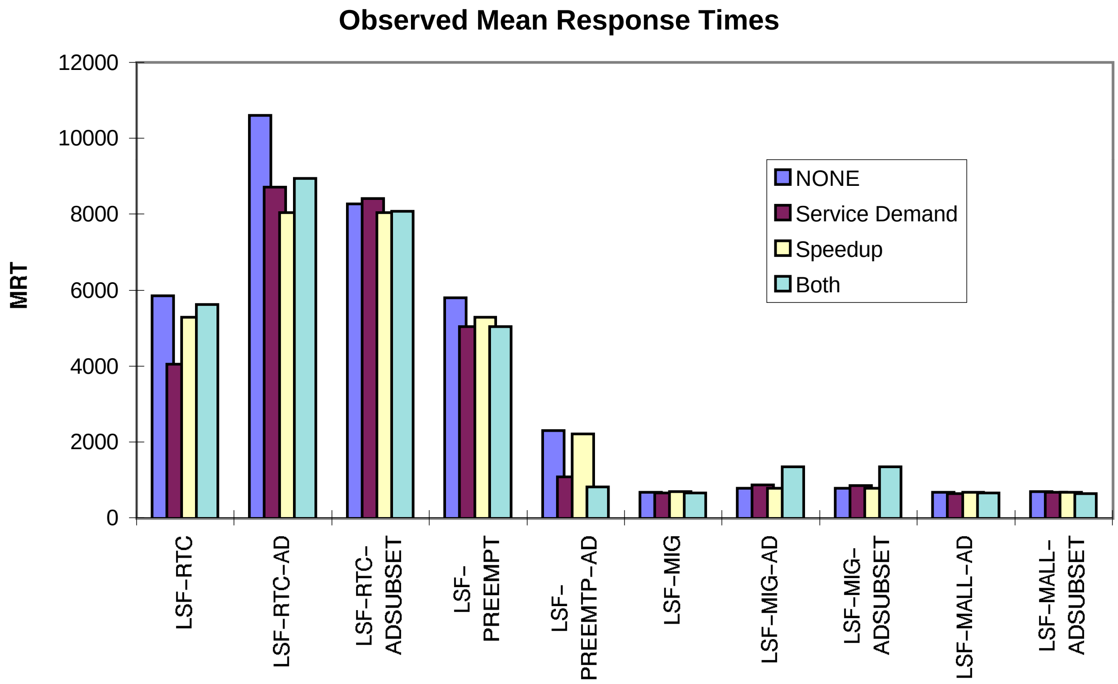


Figure 6.5: Observed mean response times for each discipline.

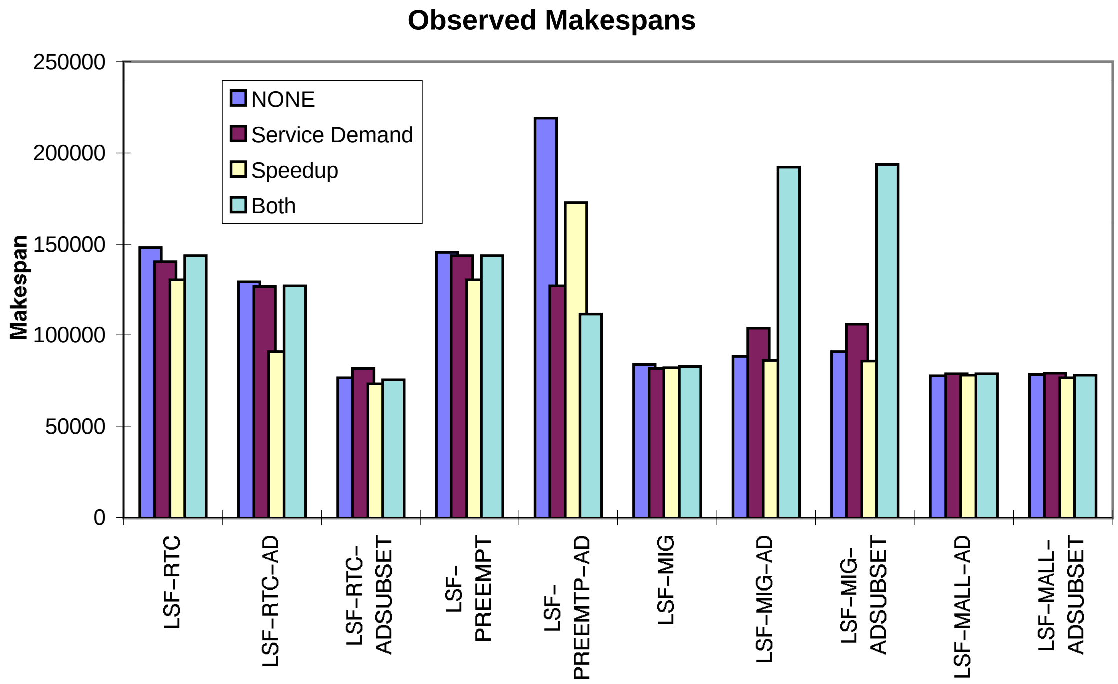


Figure 6.6: Observed makespans for each discipline.

Service-demand and speedup knowledge appear to be most effective when either the mean response time (for the former) or the makespan (for the latter) were large, but may not be as significant as one might expect. Service-demand knowledge had limited benefit in the run-to-completion disciplines because the high response times result from long-running jobs being activated, which the scheduler must do at some point. In the migratable and malleable preemptive disciplines, the multi-level feedback approach attained the majority of the benefits of having service demand information. Highlighting this difference, we often found queue lengths for run-to-completion disciplines to grow as high as 60 jobs, while for migratable or malleable disciplines, they were rarely larger than five.

Given our workload, we found speedup knowledge to be of limited benefit because poor-speedup jobs can rarely run efficiently. (To utilize processors efficiently, such a job must have a low minimum processor requirement, and must be started at the same time as a high-efficiency job; even in the best case, the maximum efficiency of a poor-speedup job will only be 58% given a minimum processor allocation of eight after scaling.) From the results, one can observe that service-demand knowledge can sometimes negate the benefits of having speedup knowledge as jobs having the least remaining service demand (rather than least acquired processing time) are given higher priority.

While performing our experiments, we monitored the behaviour of each of our schedulers, in order to further understand the performance results. Our observations can be summarized as follows:

- Jobs having large minimum processor requirements are often significantly delayed in run-to-completion disciplines. Since service demands have a high degree of variability, there is often at least one job running having a large service demand, making it difficult to ever schedule a job having large minimum processor requirement.

  This behaviour is illustrated in Figure 6.7. Even at light loads, it is quite likely for some processors to be occupied, preventing the dispatching of a job having a large processor requirement. Even the use of the SUBSET variant of the RTC disciplines cannot counteract this effect because it still requires all processors to be available at the time it makes its scheduling decision.

- Adaptive run-to-completion disciplines can lead to more variable makespans. In a 200-job workload, the makespan is dictated essentially by the long-running jobs in the system (e.g., in one of our experiments, one job had a sequential service demand of 265000 seconds, or almost 74 hours). The makespan of a rigid discipline will be relatively predictable because the execution time of these long jobs is set in advance. In the adaptive case, a scheduler may allocate such jobs small number of processors, which is good from an efficiency standpoint, but can lead to much longer makespans.

  Also, if long jobs are allocated few processors, which tends to occur in most adaptive disciplines as the load increases, these long jobs will occupy processors for longer periods of time (relative to the rigid case). This can make it even more difficult for jobs with large minimum processor requirements to ever find enough available processors.

  The conclusion is that run-to-completion disciplines are even more problematic than originally indicated. In Chapter 3, it was shown how high variability in service demands can lead to poor

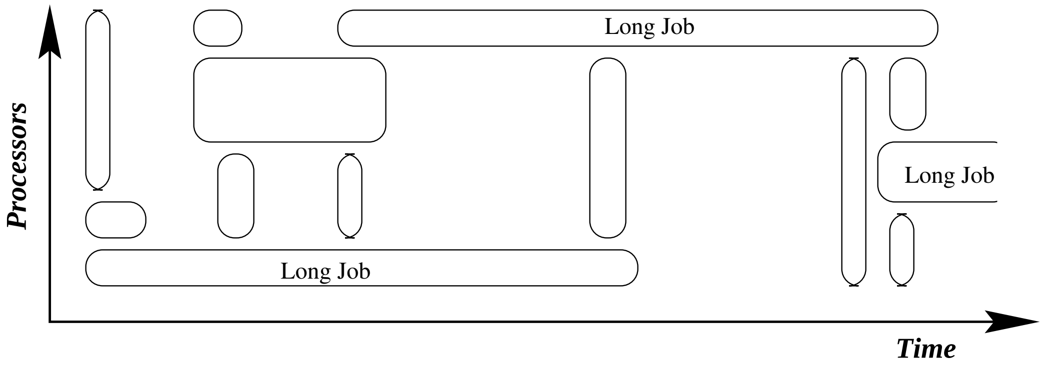


Figure 6.7: Effects of highly variable service demands on the ability for a run-to-completion scheduler to activate jobs having large minimum processor requirements. Because of the long-running jobs, the system rarely reaches a state where all processors are available, which is necessary to schedule a job having a large minimum processor requirement.

response times if memory is abundant; these observations show that highly variable service demands can also lead to starvation for jobs having large minimum processor requirements.

- Migratable disciplines can significantly reduce response times relative to RTC ones. However, adaptive versions of migratable disciplines can exhibit unpredictable completion times for long-running jobs, as a scheduler must commit to an allocation when a job is first activated. In some cases, the scheduler allocates a small number of processors to a long-running job, only to have other processors subsequently become available. In a production environment, this may encourage users submitting high service-demand jobs to specify a large minimum processor allocation simply to ensure that their jobs complete within a more desirable amount of time, but having a negative effect on the sustainable throughput.

  In other cases, long-running jobs were allocated a large number of processors, leading to potential starvation problems. (This was the cause of the large makespans in the full-knowledge LSF-MIGRATE-AD and LSF-MIGRATE-ADSUBSET experiments.) In order to resume such a job once stopped, the scheduler must be capable of preempting a sufficient number of running jobs to satisfy the stopped job's processor requirement. This can be difficult at high loads where jobs with small processor allocations are continuously being started, suspended, and resumed, since we only preempt jobs that have run at least ten minutes. In a real workload, we believe this problem will become less important as the ratio of the migration overhead to the mean service demand becomes smaller.

- From a user's perspective, malleable disciplines are most attractive. During periods of heavy load, the system allocates jobs a small number of processors, and as the load becomes lighter, long-running jobs receive more processors. Packing loss, where processors are left idle even though jobs could potentially use them, is not a problem because allocations of running jobs can be decreased to allow more jobs to fit in the system or increased to make use of all processors. Also, jobs rarely experience starvation because the scheduler does not commit itself to a processor allocation upon activating a job for the first time. As a result, adaptive malleable disciplines consistently performed best and have the highest potential for low response times and high throughputs (even if the cost of changing allocation size is as high as 10% of the scheduling quantum).

6.4 Conclusions

In this chapter, we present the design of parallel-job scheduling implementations, based on Platform Computing's Load Sharing Facility (LSF). We consider a wide range of disciplines, from run-to-completion to malleable preemptive ones, each with varying degrees of knowledge of job characteristics. Although these disciplines were designed for a network of workstations, they can be used on any distributed-memory multiprocessor system supporting LSF.

The primary objective of this work was to demonstrate the practicality of implementing parallel-job scheduling disciplines. By building on top of an existing commercial software package, we found that implementing new disciplines was relatively straightforward. Given the lack of maturity of parallel-job scheduling, the approach taken in this chapter of extending commercial scheduling software is a good one. Future work in this area, however, would be aided by the inclusion of the Job and System Information Cache (JSIC) and the corresponding update routines directly into the base scheduling software.

The secondary objective of this work was to study the behaviour of these disciplines in a more realistic environment and to illustrate the benefits of different types of preemption and knowledge. We found that preemption is crucial to obtaining good response times, supporting the results of Chapter 3. We believe that the most attractive discipline for today is a hybrid migratable/malleable discipline. Many long-running jobs in production environments already perform checkpointing to tolerate failures, and as mentioned before, technology exists to perform automatic checkpointing of many parallel jobs. Given that only long-running jobs ever need to be migrated or "malleated", disciplines that expect either of these two types of preemption are practical today. Although the majority of applications used today may support only migratable preemption, it is relatively simple to modify the adaptive migratable/malleable scheduling module presented in Appendix E to support both kinds of jobs. Using such a hybrid scheduling discipline would greatly benefit jobs that already support malleable preemption, and would further encourage application writers to support this kind of preemption in new applications.

Our observations suggest that further work could be done to better choose processor allocations given approximate speedup and service-demand knowledge about jobs in order to reduce the variability in completion times for any given job. One simple approach would be to base processor allocation decisions on the load over some interval in the past rather than on the instantaneous load, as in the disciplines described in this chapter. Such work, however, would lose its relevance if support for malleable jobs became even more prevalent in future systems, since the scheduler would not be committed to the initial processor allocation made for a job.

In production systems, there are often additional scheduling requirements in terms of job priorities in multi-class workloads. The LSF-based disciplines described in this chapter can be modified to take into account such requirements. For example, if certain job classes have absolute precedence over others, then the scheduler can simply favour those jobs in its job selection phase. If, instead, job classes need to be allocated certain fractions of computation time, then the scheduler can maintain a different queues for each job class and select jobs from each queue in appropriate proportions.

Another concern in production systems is the starvation of large-sized jobs at heavy loads. This is particularly a concern in run-to-completion disciplines because processors have to be left idle until enough are available for a large-sized job, a problem which does not arise in preemptive disciplines. In order to give large-sized jobs added responsiveness, it is possible to increase a job's priority the longer it waits for execution (as is done for processes in Unix).

Finally, many users find disciplines which predict when a job will complete, such as EASY, to be appealing. It is also possible to include such forms of prediction in preemptive schedulers, such as the ones described in this chapter, by computing in advance when jobs will be preempted and resumed. Ideally, such predictions would take into account future job arrivals, perhaps based on the recent past.

Chapter 7

Conclusions

Much has been learned about parallel-job scheduling since it was first studied in the late 1980's. Over that time, two basic principles emerge that guide the design of scheduling disciplines. First, for computationally-intensive workloads, threads of a job must typically be coscheduled in order to make efficient use of the processing resources. Otherwise, context-switching overheads caused by threads not being ready for synchronizations will become overwhelming. Second, adaptive scheduling disciplines are preferable to rigid ones, given that parallel jobs tend to utilize resources more efficiently as their processor allocation decreases. Adaptive disciplines can thus adapt to changing loads, offering jobs large processor allocations at light loads, and smaller allocations at heavier ones.

In this thesis, we add to this understanding of parallel-job scheduling in several respects. First, given the types of workloads found in practice, we show that preemption is necessary to obtain good response times. The extent to which preemption is needed depends primarily on the likelihood that long-running jobs will be allocated a large number of processors, given a particular workload and run-to-completion scheduling discipline. Second, we show that the memory requirements of jobs should be taken into account in the scheduling decision, as follows:

- As the load increases, it is vital to increasingly make better use of memory resources. This not only permits processors to be, on average, more efficiently utilized but also minimizes throughput limitations due to memory.

- For workloads in which larger-sized jobs tend to have better speedup characteristics, using speedup knowledge is key to sustaining high throughputs (and thus offering reasonable response times) if memory is limited. This point is particularly significant because speedup knowledge does not hold the same importance if memory is not limited.

- Preemption is even more necessary if memory is not abundant. Large memory requirements essentially decrease the average multiprogramming level (i.e., number of jobs that can run simultaneously), thus decreasing the chance that a job will find sufficient free memory in run-to-completion disciplines.

A current problem with parallel-job scheduling research is that very few results have found their way into commercial scheduling systems. Two reasons for this are that (1) few actual implementations exist, particularly ones that are portable to numerous platforms, and (2) the real benefits of using adaptive and preemptive scheduling disciplines have not been demonstrated in a practical context. In this thesis, we address both these issues by describing the implementation of a family of scheduling disciplines built on top of Load Sharing Facility (LSF), a commercial scheduling system presently used in a number of high-performance computing centers.

7.1 Summary of Results

7.1.1 Need for Preemption

In Chapter 3, we examined the importance of preemption in parallel-job scheduling. Apart from Ousterhout's work [Ous82], nearly all subsequent research in non-malleable parallel-job scheduling assumed that jobs must be run until completion. This was despite the fact that service demands being reported by high-performance computing centers have high degrees of variability, for which run-to-completion disciplines are known to lead to poor response times in uniprocessor systems.

In a large-scale multiprocessor, however, it is not clear that highly variable service demands will lead to poor response times. The reason is that a multiprocessor behaves essentially like a limited processing sharing server (in the queueing theory sense), allowing a certain number of jobs to be serviced concurrently. In these systems, run-to-completion (RTC) disciplines will only lead to poor response time if there is a high probability that all processors in the system are occupied with long-running jobs.[1]

To investigate this issue, we took two popular adaptive, run-to-completion scheduling disciplines and modified them with the assumption that (1) jobs could be migrated, and (2) the system could (at times) multiplex the threads of a job on fewer processors than its initial allocation, albeit with a loss in performance. We then considered a number of workloads, with varying arrival rates and speedup characteristics. As a baseline discipline, we used an ideal equipartition discipline in which there were no overheads associated with reallocating processors. Our major findings were:

- For workloads in which jobs have good speedups, run-to-completion disciplines have significantly worse performance than preemptive ones. For workloads in which the service demands have a coefficient of variation of 70, there can be up to three orders of magnitude difference in mean response times between the preemptive and non-preemptive disciplines.

- For workloads in which jobs have poor speedups, however, this difference in mean response times decreases from three orders of magnitude to a factor of four. The reason is that, in the

[1] For disciplines which required malleability, such as equipartition, preemption was typically assumed to be possible, but we have found that it is seldom used if memory is not limited. In a large system, say one having 100 processors, it would be necessary for there to be 100 outstanding long-running jobs for the lack of preemption adversely affect response times, a situation which rarely occurs.

poor-speedup case, many jobs had a maximum degree of parallelism that was quite low, so it is difficult for any job to monopolize the system.

- The performance of non-malleable disciplines can be comparable to that of malleable ones (in particular, equipartition). Also, depending on the costs of migrating processes versus reallocating processes (for the malleable case), the non-malleable disciplines can be preferable.

7.1.2 Memory-Constrained Scheduling

In Chapters 4 and 5, we considered the effects memory requirements may have on the scheduling decision. Many scientific applications have large memory requirements. Moreover, these requirements are continually increasing as the need to model physical systems in greater detail increases. Since paging has not proven to be effective for reducing the physical memory requirements of parallel versions of these applications, it is likely that the combined memory requirements of jobs queued in the system will exceed the memory capacity. In distributed-memory systems, memory requirements are stated as minimum processor allocations, since each memory module is local to a single processor module.

If the scheduler possesses no information about the speedup characteristics of jobs, then an equi-allocation discipline (i.e., one which allocates processors equally among jobs chosen to run) offers near-maximum throughput and good response times. Depending on the distribution of memory size requirements, however, memory packing losses can significantly degrade performance for two reasons. First, if memory is not effectively utilized, then the system is running fewer jobs together in the system, leading to larger average processor allocations. Since jobs typically exhibit decreased processor efficiency as processor allocation increases, the processors of the system will be less efficiently utilized. Second, it has been shown that the sustainable throughput for memory-constrained workloads is proportional to the available memory. If memory is less effectively used, then memory "effectively" available decreases, resulting in lower sustainable throughput.

As part of our analysis, we derive for the first time analytic bounds on the sustainable throughput when no information is known about the speedup characteristics of jobs (which also implies that no correlation exists between memory requirements and speedup characteristics). As a side-effect of these bounds, we find that an equi-allocation discipline is provably optimal for maximizing throughput at a given multiprogramming level. This result explains why equi-allocation has been shown to perform so well in many empirical and simulation studies.

Three different disciplines are proposed in Chapter 4, MPA-Basic, MPA-Repl1, and MPA-Pack, each making progressively better use of memory as the load increases, but each requiring somewhat more computation. In one experiment, these three disciplines achieved 80%, 85%, and 93%, respectively, of the maximum sustainable load.

If the speedup characteristics of jobs are known, however, then an equi-allocation strategy is no longer recommended if there exists a positive correlation between memory requirements and job efficiencies (i.e., large-sized jobs have higher efficiency than small-sized ones). In this case, it is necessary to favour efficient jobs in order to maximize the sustainable throughput (and hence maintain reasonable response times). In fact, it is theoretically possible to use speedup information to achieve an arbitrarily large performance improvement over an equi-allocation discipline, restricted only by the degree of correlation.

In contrast, we found that, if no such correlation exists, then speedup knowledge is of limited benefit in increasing the sustainable throughput. This confirms our previous result that an equi-allocation discipline will yield near-optimal performance, since having no speedup knowledge implies that memory requirements and speedup characteristics are not correlated. We anticipate, however, that many workloads will exhibit the type of correlation just described, since for many parallel applications, increasing the problem size simultaneously increases memory requirements and improves speedup characteristics.

For the case where speedup information is available, two additional disciplines are proposed, one based on equalizing processor and memory occupancies (MPA-OCC), for which a heuristic is provided, and another based on equalizing efficiencies (MPA-EFF), for which an exact algorithm is provided. Throughput gains over equi-allocation range from 0% when no correlation exists to 100% with high correlation for the workloads chosen for simulation.

7.1.3 Scheduling Implementations

The design of a family of parallel-job scheduling disciplines is presented in Chapter 6. These disciplines are implemented on top of Platform Computing's Load Sharing Facility (LSF), as scheduling extensions, allowing them to be used on any platform currently running LSF. It shows that many of the scheduling disciplines that have been proposed in prior literature or which have been presented in this thesis can be practically implemented.

The disciplines span the entire range of schedulers, from rigid to adaptive, and from run-to-completion to malleable preemption. Also, these schedulers can effectively utilize both speedup and service-demand knowledge of individual jobs, if the information is available. In evaluating the performance of these disciplines, we again demonstrated the benefit of preemption in reducing response times.

Some notable observations that we made regarding the behaviour of these disciplines are as follows:

- Starvation can occur in run-to-completion disciplines for jobs having large memory requirements (or large minimum processor allocation requirements in the distributed-memory system case). The reason is that the likelihood of there always being a long-running job running in the system is quite high, given the high variability in service-demand distributions. As a result, large-sized jobs rarely find enough memory (or processors) available in which to run.

- Disciplines that commit to a processor allocation for a job at the time when the job is activated (i.e., adaptive disciplines that are non-malleable) can lead to unpredictable response times for long-running jobs. In these disciplines, the processor allocation choice is often independent of the overall load on the system, but instead on transient load at the time when a job is dispatched. It is thus possible for a long-running job to arrive during a temporarily heavy period and to be allocated its minimum processor allocation, only to find itself taking a long time to complete while a number of other processors are idle. This type of unpredictability may encourage users to specify higher minimum processor allocation values in order to improve the predictability (defeating the benefit of adaptive scheduling disciplines).

- Malleable disciplines perform very well, even if the cost of processor allocation is quite high (in our case, 10% of the minimum time between reallocations).

7.2 Final Remarks and Future Work

In this thesis, a variety of approaches were used in studying the performance of parallel-job scheduling disciplines for multiprocessors. We used analytic models when possible and simulation either to confirm the results obtained from the models or to study situations that could not be modeled. Both approaches, however, evaluate performance "in the large". That is, they seek to determine the performance of a given discipline over an infinite horizon of job arrivals.

In our implementation work, it became evident that, although the choice of a particular scheduling discipline can lead to good long-term performance, short-term performance (e.g., that over 200 jobs) can fluctuate greatly. Particularly with non-malleable disciplines, it is possible for a job to be given a processor allocation that is inconsistent with the current overall load (as opposed to the transient load), leading to unpredictable response times. One avenue of future research in non-malleable scheduling is to reduce this type of unpredictability, which will inevitably frustrate users. (Users expect that a job submitted to a system at a given load will complete roughly in the same amount of time, from one submission to the next.)

In our work, we typically assumed that if some type of knowledge was available, that it was perfectly accurate. In practice, this is unlikely to be the case, particularly if historical data is used to predict characteristics of newly-submitted jobs. In the context of service demands, Gibbons showed that the use of estimates based on historical runs of an application allowed performance gains 75% as great as those obtainable with perfect knowledge of service demands. In terms of speedup characteristics, the degree to which errors will affect the performance of disciplines that use this information (e.g., MPA-OCC) is unclear. If the errors are small, they are unlikely to significantly influence results, because it is only necessary to distinguish between jobs that have large differences in speedup characteristics. If the errors are large, however, then one might only use speedup information for well-known applications, particularly those that dominate the workload.

There are many other related research topics of interest, some of which are:

- For the case where speedup information is available, the MPA-OCC and MPA-EFF disciplines use this information only in allocating processors. This makes it possible for two jobs to be selected for execution, both having poor speedup characteristics; in this case, any choice of processor allocation will lead to inefficient use of the system. Also using speedup information in selecting jobs helps avoid situations in which processors are poorly utilized.

- Scheduling research has up to now been focussed on computationally-intensive parallel applications, which make good use of the processors allocated. If all applications are I/O-intensive, however, then a simple Unix-like scheduler may perform well, because applications will not expect to be able to synchronize frequently. With hybrid workloads, the scheduler must carefully distinguish between I/O-intensive jobs and compute-intensive ones, or even identify long I/O phases within compute-intensive jobs.

- Increasingly, systems used in practice are not homogeneous. For example, the 512-node SP-2 at the Cornell Theory Center is comprised of several different types of nodes, having different memory and I/O capacities. Having heterogeneous processors, where some nodes are small-scale multiprocessors, would allow jobs that synchronize infrequently to use single-processor nodes, and allow jobs that synchronize more frequency to run on multiprocessor nodes. Much research remains to be done in this area.

Bibliography

[AKK$^+$95] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 19–44. Springer-Verlag, 1995.

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 49–60, 1989.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[AS97] Stergios V. Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, June 1997. To appear.

[Ast93] Greg Astfalk. Fundamentals and practicalities of MPP. *The Leading Edge*, 12:839–843, 907–911, 992–998, 1993.

[BB90] Krishna P. Belkhale and Prithviraj Banerjee. Approximate algorithms for the partitionable independent task scheduling problem. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume I, pages 72–75, 1990.

[BG96] Timothy B. Brecht and Kaushik Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27&28:519–539, 1996.

[BHMW94] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings Supercomputing '94*, pages 590–599, November 1994.

[Bre93a] Timothy Brecht. *Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors*. PhD thesis, Department of Computer Science, University of Toronto, December 1993.

[Bre93b] Timothy Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 1–18, 1993.

[CDD+91] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, 1991.

[CDV+94] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-VI)*, pages 12–24, 1994.

[CMV94] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 33–44, 1994.

[Cof76] E. G. Coffman, Jr., editor. *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, Inc., 1976.

[CS87] Ming-Syan Chen and Kang G. Shin. Processor allocation in an N-Cube multiprocessor using gray codes. *IEEE Transactions on Computers*, C-36(12):1396–1407, December 1987.

[CV96] Su-Hui Chiang and Mary Vernon. Dynamic vs. static quantum-based parallel processor allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 200–223. Springer-Verlag, 1996.

[DCDP90] K. Dussa, B. Carlson, L. Dowdy, and K-H. Park. Dynamic partitioning in a transputer environment. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 203–213, 1990.

[Dow90] Lawrence W. Dowdy. On the partitioning of multiprocessor systems. In *Performance 1990: An International Conference on Computers and Computer Networks*, pages 99–129, March 1990.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[FN95] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 337–360. Springer-Verlag, 1995.

[FR90] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, May 1990.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[GGK93] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[Gib96] Richard Gibbons. A historical application profiler for use by parallel schedulers. Master's thesis, Department of Computer Science, University of Toronto, 1996.

[Gib97] Richard Gibbons. A historical application profiler for use by parallel schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the Third Workshop on Job Scheduling Strategies for Parallel Processing*, 1997. To appear.

[GST91] Dipak Ghosal, Guiseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[GTS91] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making effective use of shared-memory multiprocessors: The process control approach. Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford University, July 1991.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Gus92] John L. Gustafson. The consequences of fixed time performance measurement. In *Proceedings of the 25th Hawaii Conference on System Sciences*, volume III, pages 113–124, 1992.

[Hen95] Robert L. Henderson. Job scheduling under the portable batch system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 279–294. Springer-Verlag, 1995.

[Hot96a] Steven Hotovy. Private communication, November 1996.

[Hot96b] Steven Hotovy. Workload evolution on the Cornell Theory Center IBM SP-2. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 27–40. Springer-Verlag, 1996.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., New York, 1991.

[KG94] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[Kle79] Leonard Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *Proceedings of the International Conference on Communications*, pages 43.1.1–43.1.10, June 1979.

[KLMS84] Richard M. Karp, Michael Luby, and A. Marchetti-Spaccamela. A probabilistic analysis of multidimensional bin packing problems. In *Proceedings of the Sixteenth Annual ACM Symposium on the Theory of Computing*, pages 289–298, 1984.

[Lif95] David A. Lifka. The ANL/IBM SP scheduling system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 295–303. Springer-Verlag, 1995.

[LT94] Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176, 1994.

[LV90] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 226–236, 1990.

[MEB88] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 104–113, May 1988.

[MEB91] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation*, 13(2):109–130, October 1991.

[ML92] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of Supercomputing '92*, pages 104–113, November 1992.

[ML94] Shikharesh Majumdar and Yiu Ming Leung. Characterization and management of I/O in multiprogrammed parallel systems. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 298–307, October 1994.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley & Sons, 1990.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[MZ94] Cathy McCann and John Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, 1994.

[MZ95] Cathy McCann and John Zahorjan. Scheduling memory constrained jobs on distributed memory parallel computers. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modelling of Computer Systems*, pages 208–219, 1995.

[NA91] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):57–61, March 1991.

[NAS80] Numerical aerodynamic simulator processing system. Technical Report PC320-02, NASA Ames Research Center, September 1980.

[Ngu96] Thu D. Nguyen. Private communication, 1996.

[NSS93] Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of Supercomputing '93*, pages 824–833, 1993.

[NT93] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, September 1993.

[NVZ96] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 175–199. Springer-Verlag, 1996.

[NW89] Lionel M. Ni and Ching-Farn E. Wu. Design tradeoffs for process scheduling in shared memory multiprocessor systems. *IEEE Transactions on Software Engineering*, 15(3):327–334, March 1989.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing (ICDCS)*, pages 22–30, October 1982.

[PBS96] Eric W. Parsons, Mats Brorrson, and Kenneth C. Sevcik. Modelling performance of distributed virtual shared memory systems for the next decade. Technical Report TR-353, Computer Systems Research Institute, University of Toronto, 1996.

[PL96] Jim Pruyne and Miron Livny. Managing checkpoints for parallel programs. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 140–154. Springer-Verlag, 1996.

[PS95] Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 127–145. Springer-Verlag, 1995.

[PS96a] Eric W. Parsons and Kenneth C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation*, 27&28:253–272, 1996.

[PS96b] Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 57–67, 1996.

[PS97] Eric W. Parsons and Kenneth C. Sevcik. Extending multiprocessor scheduling systems using queue-based mechanisms. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the Third Workshop on Job Scheduling Strategies for Parallel Processing*, 1997. To appear.

[PSN94] Vinod G. J. Peris, Mark S. Squillante, and Vijay K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, 1994.

[RSD+94] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning policies of multiprocessor systems. *Performance Evaluation*, 19:141–165, 1994.

[Sch70] Linus E. Schrage. Optimal scheduling rules for information systems. ICR Quarterly Report No. 26, Institute for Computer Research, University of Chicago, August 1970.

[SCZL96] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY-LoadLeveler API project. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 41–47. Springer-Verlag, 1996.

[Set93] Sanjeev Kumar Setia. *Scheduling on Multiprogrammed, Distributed Memory Parallel Computers*. PhD thesis, University of Maryland, 1993.

[Set95] Sanjeev K. Setia. The interaction between memory allocations and adaptive partitioning in message-passing multiprocessors. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 146–164. Springer-Verlag, 1995.

[Sev72] Kenneth C. Sevcik. Scheduling for minimum total loss using service time distributions. *Journal of the Assocation for Computing Machinery*, 21(1):66–75, January 1972.

[Sev89] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[Sev94] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[SG91] Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, 1991.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, July 1993.

[SL93] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[Sle80] Daniel D. K. D. B. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37–40, February 1980.

[SLTZ77] K. C. Sevcik, A. I. Levy, S. K. Tripathi, and J. L. Zahorjan. Improving approximations of aggregated queuing network subsystems. In K. M. Chandy and M. Reiser, editors, *Computer Performance*. North-Holland, 1977.

[SM94] S. Selvakumar and C. Siva Ram Murthy. Scheduling precedence constrained task graphs with non-negligible intertask communication onto multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):328–336, March 1994.

[SN93] Xian-He Sun and Lionel M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, Sept 1993.

[SSRV94] Anand Sivasubramanian, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 171–180, 1994.

[SST93] Sanjeev K. Setia, Mark S. Squillante, and Satish K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 158–170, 1993.

[ST93] Sanjeev Setia and Satish Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, January 1993.

[TG89] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[TLW+94] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. Scheduling parallelizable tasks to minimize average response time. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, 1994.

[TSWY94] John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 112–121, 1994.

[TTG92] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the benefits of cache-affinity scheduling in shared-memory multiprocessors. Technical Report CSL-TR-92-536, Stanford University, August 1992.

[TTG93] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: A summary. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 272–274, 1993.

[TWPY92] John Turek, Joel L. Wolf, Krishna R. Pattipati, and Philip S. Yu. Scheduling parallelizable tasks: Putting it all on the shelf. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, pages 225–236, 1992.

[TWY92] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

[VBS+95] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, Computer Systems Research Institute, University of Toronto, April 1995.

[Ver94] Mary K. Vernon. Private communication, September 1994.

[VZ91] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth Symposium on Operating System Principles (SOSP)*, pages 26–40, 1991.

[WMKS96] Michael Wan, Regan Moore, George Kremenek, and Ken Steube. A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 1162, pages 48–64. Springer-Verlag, 1996.

[Wol94] Joel Wolf. Private communication, June 1994.

[Wu93] Chee-Shong Wu. Processor scheduling in multiprogrammed shared memory NUMA multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, 1993. Also available as CSRI Technical Report 341.

[ZM90] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 214–225, 1990.

Appendix A

Sources for Workload Application

The following program is the synthetic parallel program used in the LSF-based experimentation described in Chapter 6. It supports all forms of preemption, namely simple (using SIGTSTP and SIGCONT), migratable, and malleable (the latter two using SIGUSR1 and SIGTERM), as described in Section 6.1.2.

```
/*
 * £Id: workload.C,v 1.12 1996/11/06 07:47:16 eparsons Exp £
 *
 * Created for the POW Project, Scheduling for Parallelism on Workstations,
 *         (c) 1996   Richard Gibbons, Eric Parsons
 *
 * Workload.c - A synthetic parallel job.  This job spawns a thread
 *    for each processor specified by the LSB_HOSTS environment
 *    variable.  It also allows various workload characteristics to be
 *    defined and supports different styles of preemption.
 */

#include <iostream.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <math.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/file.h>
#include <sys/time.h>
#include <sys/select.h>                          /* needed for LSF include files*/

extern "C" {
    int setitimer(int, struct itimerval *, struct itimerval *);
    int getitimer(int, struct itimerval *);

    int seteuid(uid_t);
}
```

```
#define MAXPROCS 1024
#define EXPANSION_FACTOR 8

#define DEFAULT_TIME 1
#define DEFAULT_DOWDY_FRAC 0.0

static void term_hdlr(int);
static void timeout(int);
static void timeout_wait(int);
static void chkpnt(int);
static void sigtstp(int);
static void sigcont(int);

static int alldone = 0;
static int verbose = 0;
static int error = 0;

static int myJobId = -1;
static char myhostname[255];
static char buf[255];

static int euid = -1, uid = -1;

// make SURE comp_time and res_comp_time are of same type
static int comp_time = DEFAULT_TIME, res_comp_time;
static struct itimerval comp_time_struct;
static struct itimerval wait_time_struct;
static struct itimerval null_time_struct;
static int sigcont_flag = 0;

static double dowdy_frac = DEFAULT_DOWDY_FRAC;

static struct sigaction handler;
static sigset_t nullmask;

static int slave = 0;
static int num_processes = 1;
static int rtask_ids[MAXPROCS];
static char *rtask_hosts[MAXPROCS];

void handleCheckPoint()
{
    sprintf(buf, "chkpt.%d", myJobId);
    int rtfd = open(buf, O_RDONLY, 0);
    if (rtfd>=0) {
        cout << myhostname << ": checkpoint file found\n";
        read(rtfd, &comp_time, sizeof(comp_time));
        close(rtfd);

#ifdef DEBUG_CHECKPOINT
        sprintf(buf, "tmp/job-%d", myJobId);
        int tmpfd=open(buf, O_RDWR | O_CREAT, S_IRUSR|S_IWUSR);
        lseek(tmpfd, 0, SEEK_END);

        sprintf(buf, "%d\n", comp_time);
        write(tmpfd, buf, strlen(buf));
        close(tmpfd);
    }
```

```
    else {
        cout << myhostname << ":  checkpoint not file found\n";
        sprintf(buf, "tmp/job-%d", myJobId);
        int tmpfd=open(buf, O_RDWR | O_CREAT, S_IRUSR|S_IWUSR);
        lseek(tmpfd, 0, SEEK_END);

        sprintf(buf, "%d (start)\n", comp_time);
        write(tmpfd, buf, strlen(buf));
        close(tmpfd);
#endif
    }

    if (comp_time<1) {
        cerr << myhostname << ":invalid comp time in checkpoint file!\n";
        unlink(buf);
        exit(-1);
    }
}

void
spawnChildren(int argc, char *argv[])
{
    num_processes = 1;
    rtask_hosts[0] = myhostname;

    char *env_hosts = getenv("LSB_HOSTS");
    if (env_hosts) {
        char *dup_env_hosts = new char[strlen(env_hosts)+1];
        char *this_host;

        strcpy(dup_env_hosts, env_hosts);

        this_host = strtok(dup_env_hosts, " \t");
        if (this_host==NULL || !strstr(myhostname, this_host)) {
            cout << myhostname
                << ":  master host must be first in LSB_HOSTS\n";
            exit(-1);
        }

        num_processes = 0;
        do {
            rtask_hosts[num_processes++] = this_host;
            this_host = strtok(0, " \t");
        } while (this_host);
    }

    char *newargv[argc+3];
    int xi;
    for (xi=0; xi<argc; xi++)
        if (strstr(argv[xi], "--"))
            break;
        else
            newargv[xi] = argv[xi];

    // augment parameters for slave processes
    char xparm1[128];
    char xparm2[128];
    sprintf(xparm1, "-p%d", num_processes);
```

```
    sprintf(xparm2, "-t%d", comp_time);                          // redefines previous setting
    newargv[xi] = xparm1;
    newargv[xi+1] = xparm2;

    for (; xi<argc; xi++)
        newargv[xi+2] = argv[xi];

    newargv[xi+2] = 0;

    seteuid(euid);
    if (ls_initrex(0, 0) < 0) {
        lsb_perror("lsb_init_rex");
        exit(-1);
    }
    seteuid(uid);

    // all exits hereafter should invoke SIGTERM
    sigemptyset(&nullmask);
    sigaddset(&nullmask, SIGUSR2);                               // too late for checkpoint
    handler.sa_handler = term_hdlr;
    handler.sa_flags = 0;
    handler.sa_mask = nullmask;
    sigaction(SIGTERM, &handler, NULL);

    // spawn off the processes on the other machines
    for (int i=0; i<num_processes; i++)
        rtask_ids[i] = -1;

    if (num_processes>1) {
        seteuid(euid);
        if (verbose) cout << "spawning:";
        for (int i=1; i<num_processes; i++) {
            if (verbose) cout << " " << rtask_hosts[i];
            if ((rtask_ids[i]=ls_rtask(rtask_hosts[i],
                                    newargv, REXF_USEPTY))<0) {
                cerr << "ls_rtask fails (" << rtask_hosts[i] << "): "
                    << ls_sysmsg() << "\n";
                error = 1;
                seteuid(uid);
                kill(getpid(), SIGTERM);
            }
        }
        seteuid(uid);
        if (verbose)
            cout << "\n";
    }
}

void
collectChildren()
{
    // remove checkpoint file, if it exists
    sprintf(buf, "chkpt.%d", myJobId);
    unlink(buf);

    // timeout for cleanup
    sigemptyset(&nullmask);
    handler.sa_handler = timeout_wait;
```

```
    handler.sa_flags = 0;
    handler.sa_mask = nullmask;
    sigaction(SIGALRM, &handler ,NULL);

    wait_time_struct.it_interval.tv_sec = 0;
    wait_time_struct.it_interval.tv_usec = 0;
    wait_time_struct.it_value.tv_sec = 10;
    wait_time_struct.it_value.tv_usec = 0;

    if (setitimer(ITIMER_REAL, &wait_time_struct, NULL)) {
        perror("setitimer");
        error = 1;
        kill(getpid(), SIGTERM);
    }

    int num_processes_left = num_processes-1;
    while (num_processes_left) {
        int tid;
        seteuid(euid);
        if ((tid=ls_rwait(0, 0, 0)) < 0) {
            cerr << "ls_rwait fails: " << ls_sysmsg() << "\n";
            error = 1;
            seteuid(uid);
            kill(getpid(), SIGTERM);
        }
        seteuid(uid);

        int gotflag = 0;
        for (int i=0; i<num_processes; i++)
            if (rtask_ids[i]==tid) {
                if (verbose)
                    cout << myhostname << ": joined " << i << "\n";
                rtask_ids[i] = -1;

                num_processes_left -= 1;
                gotflag = 1;
            }

        if (!gotflag)
            cerr << myhostname << ": ls_rwait failed to match "
                 << tid << "\n";
    }
}

int
main(int argc, char *argv[])
{
    euid = geteuid();
    uid = getuid();
    seteuid(uid);

    /* hostname is used to (1) print out error messages and (2) verify
    ** that first host in LSB_HOSTS corresponds to master */
    if (gethostname(myhostname, sizeof(myhostname))) {
        cerr << "Can't figure out the hostname!\n";
        exit(-1);
    }
```

```
/* if we are running under LSF, we use LSB_JOBID to generate a
** filename for checkpointing; the LSB_JOBID remains the same
** across job migrations (which is what we use to checkpoint) */
if (getenv("LSB_JOBID")) {
    myJobId = strtol(getenv("LSB_JOBID"), 0, 0);
    putenv("LSB_JOBID=-1");                    // slaves should have -ve JOBID
}

int c;
while ((c = getopt(argc, argv, "p:t:d:hv")) != -1) {
    switch(c) {
    case 't':                                  // compute time
        comp_time = strtol(optarg, 0, 0);
        if (comp_time < 1) {
            cerr << myhostname
                << ": comp time must be a positive integer ("
                << comp_time << ")\n";
            exit(-1);
        }
        break;

    case 'd':                                  // dowdy fraction
        dowdy_frac = strtod(optarg, 0);
        if (dowdy_frac<0) {
            cerr << myhostname
                << ": dowdy fraction must be positive ("
                << dowdy_frac << ")\n";
            exit(-1);
        }
        break;

    case 'v':                                  // verbose mode
        verbose = 1;
        break;

    case 'p':                                  // number of processes, used by slaves
        slave = 1;
        num_processes = strtol(optarg, 0, 0);
        break;

    case 'h':
    default:
        cerr << "Usage: " << argv[0] << " <options>\n";
        cerr << "options:\n";
        cerr << " -h : help\n";
        cerr << " -t : time to run in seconds\n";
        cerr << " -d : dowdy fraction for speedup\n";
        cerr << " -v : verbose\n";
        cerr << "Default: " << argv[0]
            << " -t" << DEFAULT_TIME << " -p1\n";
        exit(-1);
        break;
    }
}

// block some signals before we do anything serious
int mask=sigblock(sigmask(SIGUSR1)|sigmask(SIGUSR2)
                  |sigmask(SIGTSTP)|sigmask(SIGCONT));
```

```
if (!slave) {
    // deal with checkpointing issues
    handleCheckPoint();

    // spawn child processes; note comments in procedure re: exits
    // all subsequent exits should be kill(getpid(), SIGTERM);
    spawnChildren(argc,argv);
}

// compute run time based on speedup function and initialize timeout
int run_time
    = int(ceil(comp_time
            * (dowdy_frac+(1-dowdy_frac)/num_processes/EXPANSION_FACTOR)));

comp_time_struct.it_interval.tv_sec = 0;
comp_time_struct.it_interval.tv_usec = 0;
comp_time_struct.it_value.tv_sec = run_time;
comp_time_struct.it_value.tv_usec = 0;

if (verbose) {
    cout << myhostname << ":  numprocs=" << num_processes
        << ", comp_time=" << comp_time << ", dowdy_frac=" << dowdy_frac
        << "\n";
    cout << myhostname << ":  running for "
        << comp_time_struct.it_value.tv_sec << " secs\n";
}

//
// Define signal handlers
//

// timout occurred, indicating end of computation
sigemptyset(&nullmask);
sigaddset(&nullmask, SIGUSR2);                  // no more need for checkpoint
handler.sa_handler = timeout;
handler.sa_flags = 0;
handler.sa_mask = nullmask;
sigaction(SIGVTALRM, &handler ,NULL);
sigaction(SIGALRM, &handler ,NULL);

// sent by LSF when child process exits
sigemptyset(&nullmask);
handler.sa_handler = SIG_IGN;
handler.sa_flags = 0;
handler.sa_mask = nullmask;
sigaction(SIGUSR1, &handler, NULL);

// user-directed checkpoint
sigemptyset(&nullmask);
sigaddset(&nullmask, SIGTERM);                  // wait until checkpoint complete
sigaddset(&nullmask, SIGTSTP);                  // wait until checkpoint complete
handler.sa_handler = chkpnt;
handler.sa_flags = 0;
handler.sa_mask = nullmask;
sigaction(SIGUSR2, &handler, NULL);

// process stopped; must use TSTP to handle real alarms correctly
```

```
    sigemptyset(&nullmask);
    sigaddset(&nullmask, SIGUSR2);                    // no checkpoints when stopped
    handler.sa_handler = sigtstp;
    handler.sa_flags = 0;
    handler.sa_mask = nullmask;
    sigaction(SIGTSTP, &handler, NULL);

    // process resumed (slaves only)
    if (slave) {
        sigemptyset(&nullmask);
        sigaddset(&nullmask, SIGUSR2);                // no checkpoints when stopped
        handler.sa_handler = sigcont;
        handler.sa_flags = 0;
        handler.sa_mask = nullmask;
        sigaction(SIGCONT, &handler, NULL);
    }

    // restore sigmask now that we're ready to start
    sigsetmask(mask);

#undef SPIN
#ifdef SPIN
    if (setitimer(ITIMER_VIRTUAL, &comp_time_struct, NULL))
#else
    if (setitimer(ITIMER_REAL, &comp_time_struct, NULL))
#endif
    {
        perror("setitimer");
        error = 1;
        kill(getpid(), SIGTERM);
    }

    alldone = 0;
    while (!alldone) {
#ifdef SPIN
        ;
#else
        pause();
#endif
    }

    if (!slave) {
        // clean up children
        collectChildren();
    }

    // if we get here, it means we terminated normally
    if (verbose)
        cout << myhostname << ": completed normally\n";

    return 0;
}


// upon SIGTERM, propagate signal to remote slaves and exit
void term_hdlr(int val)
{
    if (verbose||error)
```

```
        cout << myhostname << ": received SIGTERM"
             << error ? " (internally)\n" : "\n";

    if (!slave)
        /* propagate signal to remote slaves */
        seteuid(euid);
        for (int i=1; i<num_processes; i++)
            if (rtask_ids[i]>=0) {
                if (ls_rkill(rtask_ids[i], SIGTERM)<0)
                    cerr << myhostname <<
                        ": ls_rkill(rtask #" << i << ") -> " << ls_sysmsg()
                         << "\n";
                rtask_ids[i] = -1;
            }
        seteuid(uid);

    exit(val);
}

// upon SIGVTALRM or SIGALRM, terminate program
void timeout(int val)
{
    if (verbose)
        cout << myhostname << ": received SIGVTALRM or SIGALRM\n";

    // no longer allow checkpoints
    sigemptyset(&nullmask);
    handler.sa_handler = SIG_IGN;
    handler.sa_flags = 0;
    handler.sa_mask = nullmask;
    sigaction(SIGUSR2, &handler, NULL);

    // set flag on which main body loops
    alldone = 1;
}

// this timeout indicates that remote processes haven't completed in time
void timeout_wait(int val)
{
    cerr << myhostname << ": timed out while waiting for remote processes\n";
    error = 1;
    kill(getpid(), SIGTERM);
}

// used to checkpoint process for later resumption
void chkpnt(int val)
{
    if (verbose)
        cout << myhostname << ": received SIGUSR2\n";

    if (slave)                                          // slaves don't do anything
        return;

    // determine how much time is left
#ifdef SPIN
    if (getitimer(ITIMER_VIRTUAL, &comp_time_struct))
        perror("getitimer");
#else
```

```
    if (getitimer(ITIMER_REAL, &comp_time_struct))
        perror("getitimer");
#endif

    // create checkpoint file
    sprintf(buf, "chkpt.%d", myJobId);
    int fd=open(buf, O_CREAT | O_RDWR | O_TRUNC, 0644);
    if (fd>=0) {
        res_comp_time
            = long(ceil(comp_time_struct.it_value.tv_sec /
                    (dowdy_frac+(1-dowdy_frac)/num_processes/EXPANSION_FACTOR)));

        write(fd, &res_comp_time, sizeof(res_comp_time));
        sprintf(buf, "\n%d\n", res_comp_time);
        write(fd, buf, strlen(buf));
        close(fd);
    }
}

// invoked when a stop signal is received
void sigtstp(int val)
{
    if (verbose)
        cout << myhostname << ":  received SIGTSTP\n";

    null_time_struct.it_interval.tv_sec = 0;
    null_time_struct.it_interval.tv_usec = 0;
    null_time_struct.it_value.tv_sec = 0;
    null_time_struct.it_value.tv_usec = 0;

#ifdef SPIN
    if (getitimer(ITIMER_VIRTUAL, &comp_time_struct))
        perror("getitimer");
#else
    if (setitimer(ITIMER_REAL, &null_time_struct, &comp_time_struct))
        perror("setitimer");
#endif

    if (verbose)
        cout << myhostname << ":  time left = "
             << comp_time_struct.it_value.tv_sec << "\n";

    if (!slave) {
        /* propagate SIGTSTP signal to remote slaves
        ** LSF might do this, but won't harm anything */
        seteuid(euid);
        for (int i=1; i<num_processes; i++)
            if (rtask_ids[i]>=0)
                ls_rkill(rtask_ids[i], SIGTSTP);
        seteuid(uid);

        // set up default action for SIGTSTP
        sigset_t mask;
        sigemptyset(&mask);
        sigaddset(&mask, SIGTSTP);
        sigprocmask(SIG_UNBLOCK, &mask, NULL);

        sigemptyset(&nullmask);
```

```
        handler.sa_handler = SIG_DFL;
        handler.sa_flags = 0;
        handler.sa_mask = nullmask;
        sigaction(SIGTSTP, &handler, NULL);

        // initiate normal SIGTSTP
        kill(getpid(), SIGTSTP);

        // restore proper action for SIGTSTP
        sigemptyset(&nullmask);
        handler.sa_handler = sigtstp;
        handler.sa_flags = 0;
        handler.sa_mask = nullmask;
        sigaction(SIGTSTP, &handler, NULL);

        /* propagate SIGCONT to remote slaves
        ** LSF might do this, but won't harm anything */
        seteuid(euid);
        for (int i=1; i<num_processes; i++)
            if (rtask_ids[i]>=0)
                ls_rkill(rtask_ids[i], SIGCONT);
        seteuid(geteuid());
    }
    else {
        // if slave, wait until process has received SIGCONT
        // note: can't actually stop slave, because process must be active
        //         for ls_rkill to work!
        sigcont_flag = 0;
        while (!sigcont_flag)
            pause();
    }

    if (verbose)
        cout << myhostname << ":  resuming for "
             << comp_time_struct.it_value.tv_sec << "\n";

#ifndef SPIN
    if (setitimer(ITIMER_REAL, &comp_time_struct, NULL)) {
        perror("setitimer");
        error = 1;
        kill(getpid(), SIGTERM);
    }
#endif
}

// this is really only for child processes; see notes in SIGTSTP
void sigcont(int val)
{
    if (verbose)
        cout << myhostname << ":  received SIGCONT\n";

    sigcont_flag = 1;
}
```

Appendix B

Source Headers for Job and System Information Cache

The following header file is for the job and system information cache (JSIC). The major data structures used by scheduling disciplines are `JobInfo` and `QueueInfo`.

```
/*
** £Id: Sched.H,v 1.31 1996/11/16 17:35:32 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**          (c) 1996   Richard Gibbons, Eric Parsons
*/

#ifndef SCHED_INCLUDE
#define SCHED_INCLUDE

#include <time.h>
#include <sys/param.h>
#include <assert.h>
#include <gdbm.h>

#define MAXPROCS 1024                          // max number of procs in system

#define INFINITE_FLOAT (float) (0x7fffffff)

extern char *cmdname;                          // pointer to process command name
extern int debug;                              // debugging variable

struct JobInfo;
typedef JobInfo *JobInfoPtr;

struct QueueInfo;
typedef QueueInfo *QueueInfoPtr;

struct JobInfo {
      int jobId;
      int cpuDemand;
      int numProcs;
```

```
int minProcs, maxProcs;
double dowdy;                                   // dowdy parameter (portion seq)

/* Values described in the man page for lsb_readjobinfo(). They
** are unused in any of our current schedulers */
char *user;
int status;
int reasons;                                    // pending or suspend reasons of job
int subreasons;                                 // load reasons of the pending job

float cpuFactor;
int nIdx;
float *loadSched;                               // stop scheduling new jobs if over
float *loadStop;                                // stop jobs if over this load

time_t submitTime;
time_t startTime;                               // Time job was actually started
time_t endTime;

time_t beginTime;
time_t termTime;

int sigValue;
time_t chkpntPeriod;
char *chkpntDir;

char *projectName;
char *command;

char *dependCond;                               // used by EASY
int dependJobId;                                // used by EASY

/* Hosts currently configured for the job
** NOTE: LSF expects numAskedHosts==numProcs for some commands
*/
int numAskedHosts;
char **askedHosts;

/* Resource string, which we convert to resource values
** We store its value to detect any changes that might occur */
char *resReq;

/* Queue to which this job belongs – needed by MANY disciplines */
QueueInfoPtr qip;

/* Standard resource limits: -1 ==> No limit. */
int cpuLimit;
int fileLimit;
int dataLimit;
int stackLimit;
int coreLimit;
int memLimit;
int runLimit;

/* Last event recorded for job */
enum EventType {SUBMITTED,SUSPENDED,RUNNING,MIGRATING,PENDPOSTMIG};
EventType lastEventType;
time_t lastEventTime;
```

```
    int lastNumProcs;
    int submitLogFlag;                        // indicates that submit record found

    /* Computed cumulative and residual times for each job mode
    ** res[..]Time refers to time since job was last started */
    time_t cumPendTime, resPendTime;
    time_t cumRunTime, resRunTime;
    time_t cumAggRunTime, resAggRunTime;      // aggregate time (all procs)
    time_t cumSuspTime, resSuspTime;

    /* Variable indicating how long job has not been found in openjobs;
    ** we keep jobs longer than normal due to possible race conditions */
    int cache;

    /* WHAT FOLLOWS ARE SCHEDULER-SPECIFIC
    ** PROPER APPROACH WOULD BE TO HAVE A DATA POINTER WHICH EACH
    ** SCHEDULER COULD FILL IN, BUT WE HAVEN'T DONE THAT YET */

    /*******
    ** EASY
    *******/

    /* The current scheduled start time, if there is one. */
    time_t currSchedStart;

    /* This variable is used to record the original run limit, to
    ** allow us to change the run limit value (so that LSF won't kill
    ** off the job), but still keep track of the original run limit.  */
    int originalRunLimit;

    /******************
    ** MOST DISCIPLINES
    ******************/

    int mark;                                 // used to mark jobs during iterations
    int pos;                                  // position in hierarchy (LPADAPTFF)

    JobInfo(int _jobId)
        : jobId(_jobId), cpuDemand(0), numProcs(0), minProcs(0), maxProcs(0),
          dowdy(0.0), user(0), cpuFactor(0.0), nIdx(0), loadSched(0),
          loadStop(0), submitTime(0), startTime(0), endTime(0), beginTime(0),
          termTime(0), sigValue(-1), chkpntPeriod(0), chkpntDir(0),
          projectName(0), command(0), dependCond(0), dependJobId(0),
          numAskedHosts(0), askedHosts(0), resReq(0), qip(0), cpuLimit(-1),
          fileLimit(-1), dataLimit(-1), stackLimit(-1), coreLimit(-1),
          memLimit(-1), runLimit(-1), lastEventType(SUBMITTED),
          lastEventTime(0), lastNumProcs(0), submitLogFlag(0), cumPendTime(0),
          resPendTime(0), cumRunTime(0), resRunTime(0), cumAggRunTime(0),
          resAggRunTime(0), cumSuspTime(0), resSuspTime(0), cache(0),
          currSchedStart(0), originalRunLimit(-1) {}

    ~JobInfo(void);

    void print();
};

/* A generic LL implementation */
template <class E>
```

```
class DLL {
    E *first, *last;

public:
    DLL() : first(0), last(0) {}

    E *getFirst() { return first; }
    E *getLast() { return last; }
    E *getNext(E *pos) { return pos->next; }
    E *getPrev(E *pos) { return pos->prev; }

    void remove(E *pos);
    void addNext(E *pos, E *item);
    void addPrev(E *pos, E *item);
};

/* This class is used to keep track of all jobs in system. It
** consists of a linked-list to allow all jobs to be scanned (needed
** for garbage collection) and a hash to allow jobs to be efficiently
** found. */
class JobList {
private:
    struct JobHashElem;

    struct JobListElem {
        JobListElem *next, *prev;
        JobInfoPtr const jip;
        JobHashElem *jhe;                    // to quickly delete hash elem

        JobListElem(JobInfoPtr _jip)
            : next(0), prev(0), jip(_jip) {}
    };

    struct JobHashElem {
        JobHashElem *next, *prev;
        const int jobId;
        JobListElem *const jle;

        JobHashElem(int _jobId, JobListElem *_jle)
            : next(0), prev(0), jobId(_jobId), jle(_jle) {}
    };

    /* A linked-list of jobs */
    DLL<JobListElem> jobs;

    /* A very simple hash table implementation */
    enum {JOBHASHSIZE=513};
    DLL<JobHashElem> jHashTable[JOBHASHSIZE];
    const unsigned int jhash(int jobId) {
        return jobId*17 % JOBHASHSIZE;
    }

    void removeJob(int jobId);
    void removeJob(JobListElem *jle);

public:
    JobList() {}
    ~JobList() {assert(0);}
```

```
    typedef JobListElem *JobRef;
    JobInfoPtr getJip(JobRef jref) {return jref->jip;}

    void addJob(JobInfoPtr jip);
    JobInfoPtr findJob(int jobId);
    void scavengeJobs();

    JobRef getFirstJob();
    JobRef getNextJob(JobRef pos);
    JobRef getPrevJob(JobRef pos);
    JobRef getLastJob();
};

extern JobList TheJobList;

/* This class is used to record information about the hosts (differs
** from queue to queue) */
struct HostInfo {
    enum {MAXSTOPPEDJOBS=20,MAXRUNNINGJOBS=20};

    char *hname;

    enum Mode {AVAIL,BUSY,UNAVAIL,INVAL};
    Mode mode;

    /* The following entries are managed by the scheduling disciplines
    ** (should probably be a void * to an object defined by given
    ** discipline) */
    enum SchedMode {IDLE, STOPPED, RUNNING, STOPPING, STARTING};
    SchedMode schedMode;

    int open;

    int numRunningJobs, numStoppedJobs;
    JobInfoPtr runningJobs[MAXRUNNINGJOBS];
    JobInfoPtr stoppedJobs[MAXSTOPPEDJOBS];

    HostInfo() : hname(0), mode(INVAL) {}
};

typedef HostInfo *HostInfoPtr;

struct QueueInfo {
    char *qname;

    /* JobIds of jobs currently in this queue.  This list is
    ** reconstructed every time we update the cache because it's
    ** cheap. */
    enum {MIN_JOBS=256};
    int maxJobs, numJobs;
    int *jobIds;

    /* This variable is to avoid having to reread the event file from the
    ** start because the job arrival is read from the event file before
    ** being read from the queue */
    int lastSubmitTime;
```

```
void expandJobs() {
    maxJobs = jobIds ? maxJobs*2 : MIN_JOBS;
    int *tmpJobIds = new int[maxJobs];
    if (jobIds) {
        memcpy(jobIds, tmpJobIds, numJobs*sizeof(jobIds[0]));
        delete [] jobIds;
    }
    jobIds = tmpJobIds;
}

void shrinkJobs() {
    /* do nothing for now */
}

void resetJobIds() {numJobs = 0;}
void addJobId(int jobId) {
    if (numJobs>=maxJobs)
        expandJobs();
    assert(numJobs<maxJobs);
    jobIds[numJobs++] = jobId;
}

/* Hosts associated with this queue; if the list changes, we
** recreate the list in its entirety */
int pollHosts;
int numHosts;
HostInfo *hosts;

/* A very simple hash table implementation */
struct HostHashElem {
    HostHashElem *next, *prev;
    const int hidx;

    HostHashElem(int _hidx)
        : next(0), prev(0), hidx(_hidx) {}
};

enum {HOSTHASHSIZE=29};
DLL<HostHashElem> hHashTable[HOSTHASHSIZE];
const unsigned int hhash(char *hname) {
    int len=strlen(hname);
    unsigned int val=hname[0];
    for (int i=1; i<len; i++) val *= hname[i];
    return val % HOSTHASHSIZE;
}

HostInfo *findHost(char *hname) {
    const int hval=hhash(hname);
    HostHashElem *hhe=hHashTable[hval].getFirst();
    while (hhe && strcmp(hosts[hhe->hidx].hname, hname))
        hhe = hHashTable[hval].getNext(hhe);

    if (hhe) return &hosts[hhe->hidx];
    else return 0;
}

void purgeHosts() {
    if (hosts) {
```

```
            for (int i=0; i<numHosts; i++)
                delete [] hosts[i].hname;
            delete [] hosts;
        }

        for (int i=0; i<HOSTHASHSIZE; i++) {
            HostHashElem *hhe;
            while ((hhe=hHashTable[i].getFirst()) ≠ 0)
                hHashTable[i].remove(hhe);
        }

        numHosts = 0;
        hosts = 0;
    }

    void defineHost(int hidx, char *hname) {
        assert(hidx<numHosts);
        assert(hosts[hidx].hname == 0);

        hosts[hidx].hname = new char[strlen(hname)+1];
        strcpy(hosts[hidx].hname, hname);

        const int hval=hhash(hname);
        HostHashElem *hhe = new HostHashElem(hidx);
        hHashTable[hval].addPrev(0, hhe);
    }

    QueueInfo(char *_qname, int _getHosts) :
        maxJobs(0), numJobs(0), jobIds(0), lastSubmitTime(0), pollHosts(0),
        numHosts(0), hosts(0) {
        qname = new char[strlen(_qname)+1];
        strcpy(qname, _qname);
    }
};

class QueueList {
private:
    struct QueueListElem {
        QueueListElem *next, *prev;
        QueueInfoPtr const qip;

        QueueListElem(QueueInfoPtr _qip)
            : qip(_qip) {}
    };

    struct QueueHashElem {
        QueueHashElem *next, *prev;
        QueueListElem *const qle;

        QueueHashElem(QueueListElem *_qle)
            : qle(_qle) {}
    };

    DLL<QueueListElem> queues;

    enum {QUEUEHASHSIZE=513};
    DLL<QueueHashElem> qHashTable[QUEUEHASHSIZE];
    int qhash(char *qname) {
```

```
        int len=strlen(qname);
        int val=qname[0];
        for (int i=1; i<len; i++) val *= qname[i];
        return val % QUEUEHASHSIZE;
    }

public:
    QueueList() {}
    ~QueueList() {assert(0);}

    typedef QueueListElem *QueueRef;
    QueueInfoPtr getQip(QueueRef qref) {return qref->qip;}

    void addQueue(QueueInfoPtr qip);
    QueueInfoPtr findQueue(char *qname);

    QueueRef getFirstQueue();
    QueueRef getNextQueue(QueueRef pos);
    QueueRef getPrevQueue(QueueRef pos);
    QueueRef getLastQueue();
};

extern QueueList TheQueueList;

typedef char *CharPtr;

/* This is wierd, I know!  But if you do this, it ensures that the
** tempate functions are generated for every class that uses the
** template. */
#include "Templates.C"

static inline int min(int x, int y) {return x<y ? x : y;}
static inline int max(int x, int y) {return x>y ? x : y;}
static inline long min(long x, long y) {return x<y ? x : y;}
static inline long max(long x, long y) {return x>y ? x : y;}
static inline float min(float x, float y) {return x<y ? x : y;}
static inline float max(float x, float y) {return x>y ? x : y;}
static inline double min(double x, double y) {return x<y ? x : y;}
static inline double max(double x, double y) {return x>y ? x : y;}

#endif
```

Appendix C

Source Headers for LSF Interaction Layer

The following header file is for the LSF Interaction Layer (LIL).

```
/*
** £Id: Sched.H,v 1.31 1996/11/16 17:35:32 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**         (c) 1996  Richard Gibbons, Eric Parsons
*/

#ifndef LSFINT_INCLUDE
#define LSFINT_INCLUDE

#include "Sched.H"

class LSFInteraction
{
    void updateJobInfo(JobInfoPtr jip, struct jobInfoEnt *newJobInfo);
    void updateJobHist();
    int checkJobHist();

    void updateJobs();
    void updateJobsInQueue(QueueInfoPtr qip);

    void updateAllQueueHosts();
    void updateQueueHosts(QueueInfoPtr, int, struct hostInfoEnt *);

public:
    void init();

    void updateAllQueues();

    void bswitch(int jobId, char *qname);
    void bstop(int jobId);
    void bresume(int jobId);
    void bkill(int jobId, int signo);
    void bmig(int jobId);
```

```
    int setHosts(int jobId, int numHosts, char **hosts);

    int waitUntilEvent(int timeout);
};

extern LSFInteraction TheLSFInteraction;

#endif
```

Appendix D

Sources for Run-To-Completion Disciplines

The following code implements the LSF-RTC family of disciplines. The `adaptiveFlag` variable passed to the `reschedGeneric()` routine determines if the discipline is rigid or adaptive. Similarly, the `subsetFlag` variable determines if the subset-sum algorithm is applied to improve the utilization of processors.

Header File

```
/*
** £Id: LSFRTC.H,v 1.1 1997/01/03 20:25:20 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**         (c) 1997 Eric Parsons
*/

#ifndef LSFRTC_INCLUDE
#define LSFRTC_INCLUDE

#include "Sched.H"

class LSFRTC {
protected:
    QueueInfoPtr ppending, prunning;

    int numAvail;

    virtual void updateHosts();
    virtual void reschedGeneric(int adaptiveFlag, int subsetFlag);

public:
    virtual void init() = 0;
    virtual void resched() = 0;
};
```

```
class LSFRTCRigid : LSFRTC {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(0,0);}
};

class LSFRTCAd : LSFRTC {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,0);}
};

class LSFRTCAdSubset : LSFRTC {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,1);}
};

extern LSFRTCRigid TheLSFRTCRigid;
extern LSFRTCAd TheLSFRTCAd;
extern LSFRTCAdSubset TheLSFRTCAdSubset;
#endif
```

Source File

```
/*
** £Id: LSFRTC.C,v 1.1 1997/01/03 20:25:18 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**         (c) 1997 Eric Parsons
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "Sched.H"
#include "LSFInt.H"
#include "LSFRTC.H"
#include "Subset.H"

void
LSFRTC::updateHosts()
{
    /* initially assume all hosts are idle */
    for (int i=0; i<prunning->numHosts; i++)
        prunning->hosts[i].schedMode = HostInfo::IDLE;

    /* scan all jobs in run queue, and mark hosts as busy */
    for (int i=0; i<prunning->numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning->jobIds[i]);
        assert(jip);

        for (int j=0; j<jip->numAskedHosts; j++) {
            HostInfo *hip = prunning->findHost(jip->askedHosts[j]);
            assert(hip);
```

```
                assert(hip→schedMode==HostInfo::IDLE);
                hip→schedMode = HostInfo::RUNNING;
            }
    }

    /∗ determine how many hosts are available for new jobs ∗/
    numAvail = 0;
    for (int i=0; i<prunning→numHosts; i++)
        if (prunning→hosts[i].mode == HostInfo::AVAIL
            && prunning→hosts[i].schedMode == HostInfo::IDLE) numAvail++;
}

void
LSFRTC::reschedGeneric(int adaptiveFlag, int subsetFlag)
{
    static int numSelJobs;                    // number of candidate jobs
    static JobInfoPtr selJobs[MAXPROCS];      // candidate set of jobs to run next
    static int selMark[MAXPROCS];             // marker indicating which will be run
    static int selAlloc[MAXPROCS];            // processor allocations for each job
    static double selDowdy[MAXPROCS];         // dowdy values for each job

    /∗ determine which hosts are available ∗/
    updateHosts();

    /∗ Unmark all jobs in pending queue. Marking is used to indicate
    ∗∗ which jobs have been selected to run; we postpone scheduling
    ∗∗ until the end, since we can only choose allocations once we
    ∗∗ know which jobs to run ∗/
    for (int i=0; i<ppending→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
        jip→mark = 0;
    }

    numSelJobs = 0;
    while (1) {
        /∗ If cpuDemand information is available, choose shortest job first ∗/
        JobInfoPtr thejip = 0;
        int thenp = 0;

        for (int i=0; i<ppending→numJobs; i++) {
            JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
            assert(jip);

            int np = adaptiveFlag ? jip→minProcs : jip→numProcs;

            if (jip→mark == 0
                && np ≤ numAvail
                && (thejip==0 || jip→cpuDemand < thejip→cpuDemand)) {
                thejip = jip;
                thenp = np;
            }
        }

        if (thejip==0)
            break;

        /∗ Mark job, and record details ∗/
        thejip→mark = 1;
```

```
        selJobs[numSelJobs] = thejip;
        selMark[numSelJobs] = 0;
        selAlloc[numSelJobs] = thenp;
        selDowdy[numSelJobs] = thejip→dowdy;
        numSelJobs += 1;
    }

    if (numSelJobs==0)
        /* No scheduling to do! */
        return;

    int numSelFF = 0;                          // number of jobs selected by first-fit
    int numSelProcs = 0;                       // min number of procs needed by these
    for (int i=0; i<numSelJobs; i++)
        if (numSelProcs+selAlloc[i] ≤ numAvail) {
            selMark[i] = 1;
            numSelFF += 1;
            numSelProcs += selAlloc[i];
        }

    if (subsetFlag) {
        /* We pack jobs more efficienty using the subset-sum algorithm */
        int numSelChop
            = int(floor(numSelFF*max(0.0, 1.0-numSelJobs/(16*numSelFF))));
        assert(numSelChop ≤ numSelFF);

        /* recompute variables with respect to chopped FF list*/
        numSelFF = 0;
        numSelProcs = 0;
        for (int i=0; i<numSelJobs; i++)
            if (numSelFF<numSelChop && selMark[i]) {
                numSelProcs += selAlloc[i];
                numSelFF += 1;
            }
            else
                selMark[i] = 0;

        int numPackProcs = numAvail - numSelProcs;
        assert(numPackProcs≥0);

        /* compute best packing for remaining procs using remaining jobs */
        int numPackJobs = 0;
        int sumPackProcs = 0;
        for (int i=0; i<numSelJobs; i++) {
            if (selMark[i] || selAlloc[i]>numPackProcs) continue;

            /* In what follows, the w[] array is specific to the
            ** subset-sum implementation, which is described in
            ** Knapsack Problems: Algorithms and Computer
            ** Implementations, Martello & Thoth, 1990.  Note that the
            ** w[] array starts at 1 */

            if (selAlloc[i] == numPackProcs) {
                /* If we have a job that requires all processors, subset-sum
                ** algorithm will choose this one first, so make it easy */
                numPackJobs = 1;
                sumPackProcs = selAlloc[i];
                w[1].wt = selAlloc[i];
```

```
                w[1].sel = 1;
                w[1].data = (void *)i;
                break;
            }
            else {
                /* Otherwise, add job to list */
                numPackJobs += 1;
                sumPackProcs += selAlloc[i];
                w[numPackJobs].wt = selAlloc[i];
                w[numPackJobs].rnd = numPackJobs;
                w[numPackJobs].sel = 0;
                w[numPackJobs].data = (void *)i;
            }
        }

        if (numPackJobs>0) {
            if (sumPackProcs>numPackProcs)
                /* Subset-sum algorithm only works if the sum of the
                ** minimum processor allocation exceeds what is available */
                subset(numPackProcs, numPackJobs);
            else
                /* If this is not the case, then just choose all jobs */
                for (int i=1; i<numPackJobs+1; i++)
                    w[i].sel = 1;

            /* Fold back subset-sum results into sel*[] arrays */
            for (int i=1; i<numPackJobs+1; i++)
                if (w[i].sel) {
                    assert(selMark[int(w[i].data)]==0);
                    assert(w[i].wt==selAlloc[int(w[i].data)]);

                    selMark[int(w[i].data)] = 1;
                    numSelProcs += w[i].wt;
                }
            assert(numSelProcs ≤ numAvail);
        }
    }

    /* Now, allocate remaining processors to jobs s.t. efficiency is
    ** maximized.
    ** Note: T(p) = w(d+(1-d)/p)
    **       S(p) = 1/(d+(1-d)/p)
    **       E(p) = 1/(dp + (1-d)) = 1/(1+d(p-1))
    **       (but since we want to test E(p+1), we get 1/(1+dp) */
    if (adaptiveFlag)
        while (numSelProcs < numAvail) {
            int minidx = -1;

            for (int i=0; i<numSelJobs; i++) {
                if (!selMark[i]) continue;

                if (minidx<0
                    || (selDowdy[i]>0.0 && selDowdy[minidx]==0.0)
                    || (selDowdy[i]>0.0 && selDowdy[minidx]>0.0
                        && (1.0/(1 + selDowdy[i]*selAlloc[i]*EXPFACT)
                            > 1.0/(1 + selDowdy[minidx]*selAlloc[minidx]*EXPFACT)))
                    || (selDowdy[i]==0.0 && selDowdy[minidx]==0.0
                        && selAlloc[i]<selAlloc[minidx]))
```

```
                minidx = i;
            }

            if (minidx<0)
                break;

            selAlloc[minidx]++;
            numSelProcs += 1;
        }

    /* Finally, assign processors and start jobs */
    for (int i=0; i<numSelJobs; i++) {
        if (!selMark[i])
            continue;

        JobInfoPtr jip = selJobs[i];
        int hcnt = selAlloc[i];
        char **hosts = new CharPtr[hcnt];

        int hidx = 0;
        while (hcnt>0) {
            while (prunning→hosts[hidx].mode ≠ HostInfo::AVAIL
                    || prunning→hosts[hidx].schedMode ≠ HostInfo::IDLE)
                hidx++;

            prunning→hosts[hidx].schedMode = HostInfo::RUNNING;

            hosts[hcnt-1] = prunning→hosts[hidx].hname;

            hcnt--;
            numAvail--;
        }

        if (debug) {
            if (!adaptiveFlag)
                printf("Running job %d (procs=%d,cpuDemand=%d,dowdy=%f) on:",
                        jip→jobId, jip→numProcs, jip→cpuDemand, jip→dowdy);
            else
                printf("Running job %d (procs=<%d,%d,%d>,cpuDemand=%d,dowdy=%f)
on:",
                        jip→jobId, jip→minProcs, jip→numProcs, jip→maxProcs,
                        jip→cpuDemand, jip→dowdy);

            for (int j=0; j<selAlloc[i]; j++)
                printf(" %s", hosts[j]);
            printf("\n");
        }

        /* assign hosts and schedule jobs */
        TheLSFInteraction.setHosts(jip→jobId, selAlloc[i], hosts);
        TheLSFInteraction.bswitch(jip→jobId, prunning→qname);

        delete [] hosts;
    }
}

void
LSFRTCRigid::init()
```

```
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-pending", 0);
    prunning = new QueueInfo("par-running", 1);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
}

void
LSFRTCAd::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-ad-pending", 0);
    prunning = new QueueInfo("par-ad-running", 1);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
}

void
LSFRTCAdSubset::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-adpack-pending", 0);
    prunning = new QueueInfo("par-adpack-running", 1);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
}

LSFRTCRigid TheLSFRTCRigid;
LSFRTCAd TheLSFRTCAd;
LSFRTCAdSubset TheLSFRTCAdSubset;
```

Appendix E

Sources for Rigid Simple Preemptive Discipline

The following code implements the LSF-PREEMPT discipline. To minimize packing losses, it only allows a job to preempt another if it possesses the same desired processor allocation value.

Header File

```
/*
** £Id: LSFPRig.H,v 1.1 1997/01/03 20:25:17 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**          (c) 1997 Eric Parsons
*/

#ifndef LSFPRIG_INCLUDE
#define LSFPRIG_INCLUDE

#include "Sched.H"

class LSFPreemptRigid {
protected:
    QueueInfoPtr ppending, prunning, pstopped;

    int numAvail;

    int numScheduledJobs;
    JobInfoPtr scheduledJobs[MAXPROCS];

    virtual void updateHosts();
    virtual void switchScheduledJobs();
    virtual int compRST(JobInfoPtr, JobInfoPtr);

public:
    virtual void init();
    virtual void resched();
};
```

```
extern LSFPreemptRigid TheLSFPreemptRigid;
#endif
```

Source File

```
/*
** £Id: LSFPRig.C,v 1.1 1997/01/03 20:25:15 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**        (c) 1997 Eric Parsons
*/

#include <stdio.h>

#include "Sched.H"
#include "LSFInt.H"
#include "LSFPRig.H"

const int MINRUNTIME = 120;
const int MINSWITCHTIME = 120;

void
LSFPreemptRigid::updateHosts()
{
    /* initially assume all hosts are idle */
    for (int i=0; i<prunning→numHosts; i++) {
        prunning→hosts[i].schedMode = HostInfo::IDLE;
        prunning→hosts[i].numRunningJobs = 0;
        prunning→hosts[i].numStoppedJobs = 0;
    }

    /* scan all jobs in stopped queue, and mark hosts appropriately */
    for (int i=0; i<pstopped→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
        assert(jip);

        for (int j=0; j<jip→numAskedHosts; j++) {
            HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
            assert(hip);

            hip→stoppedJobs[hip→numStoppedJobs++] = jip;
            if (jip→lastEventType ≠ JobInfo::SUSPENDED)
                /* job has not suspended yet (LSF is slow?) */
                hip→schedMode = HostInfo::STOPPING;
            else
                hip→schedMode = HostInfo::STOPPED;
        }
    }

    /* scan all jobs in run queue, and mark hosts as busy */
    for (int i=0; i<prunning→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning→jobIds[i]);
        assert(jip);

        for (int j=0; j<jip→numAskedHosts; j++) {
            HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
```

```
                assert(hip);

                assert(hip→schedMode == HostInfo::IDLE
                        || hip→schedMode == HostInfo::STOPPED);
                hip→runningJobs[hip→numRunningJobs++] = jip;
                hip→schedMode = HostInfo::RUNNING;
            }
        }

        /* determine how many unconstrained hosts are available for new jobs */
        numAvail = 0;
        for (int i=0; i<prunning→numHosts; i++)
            if (prunning→hosts[i].mode == HostInfo::AVAIL
                && prunning→hosts[i].schedMode == HostInfo::IDLE)
                numAvail++;
}

void
LSFPreemptRigid::switchScheduledJobs()
{
    int xi = 0;
    while (xi < numScheduledJobs) {
        JobInfoPtr jip = scheduledJobs[xi];

        /* mark job as being scheduled (to avoid being rescheduled) */
        jip→mark = 1;

        /* check if all hosts are available */
        int ready = 1;
        for (int j=0; j<jip→numAskedHosts; j++) {
            HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);

            if (hip→mode == HostInfo::AVAIL
                && hip→schedMode == HostInfo::IDLE)
                numAvail--;

            assert(hip→schedMode ≠ HostInfo::RUNNING
                    && hip→schedMode ≠ HostInfo::STARTING);
            if (hip→schedMode == HostInfo::STOPPING)
                ready = 0;
            else
                hip→schedMode = HostInfo::STARTING;
        }

        if (ready) {
            if (jip→qip == pstopped)
                TheLSFInteraction.bresume(jip→jobId);
            TheLSFInteraction.bswitch(jip→jobId, prunning→qname);

            scheduledJobs[xi] = scheduledJobs[--numScheduledJobs];
        }
        else {
            jip→cache = 0;                              // so job doesn't go away
            xi += 1;
        }
    }

    assert(numAvail ≥ 0);
```

```
}

int
LSFPreemptRigid::compRST(JobInfoPtr jip1, JobInfoPtr jip2)
{
    long rst1 = 0, rst2 = 0;

    if (jip1→cpuDemand>0 && jip2→cpuDemand>0) {
        rst1 = jip1→cpuDemand - (jip1→cumAggRunTime+jip1→resAggRunTime);
        if (rst1<0) rst1 = 0;

        rst2 = jip2→cpuDemand - (jip2→cumAggRunTime+jip2→resAggRunTime);
        if (rst2<0) rst2 = 0;

        if (rst1+MINSWITCHTIME < rst2) return -2;
        if (rst1 < rst2) return -1;

        if (rst1 > rst2+MINSWITCHTIME) return 2;
        if (rst1 > rst2) return 1;
        if (rst1 == rst2) return 0;
        assert(0);                                  // should never get here
    }

    double acq1 = jip1→cumAggRunTime+jip1→resAggRunTime;
    double acq2 = jip2→cumAggRunTime+jip2→resAggRunTime;
    if (acq1 < 0.9*acq2 && acq2-acq1 > MINSWITCHTIME) return -2;
    if (acq1 < acq2) return -1;

    if (0.9*acq1 > acq2 && acq1-acq2 > MINSWITCHTIME) return 2;
    if (acq1 > acq2) return 1;
    if (acq1 == acq2) return 0;
    assert(0);
}

void
LSFPreemptRigid::resched()
{
    /* determine which hosts are available */
    updateHosts();

    /* Unmark all jobs in all queue queues. Marking is used to
    ** indicate which jobs have been selected to run; we postpone
    ** scheduling until the end, since we can only choose allocations
    ** once we know which jobs to run */
    for (int i=0; i<ppending→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
        jip→mark = 0;
    }
    for (int i=0; i<prunning→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning→jobIds[i]);
        jip→mark = 0;
    }
    for (int i=0; i<pstopped→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
        jip→mark = 0;
    }

    /* Now try to switch scheduled jobs. All scheduled jobs are also
```

```
** marked by this routine. */
switchScheduledJobs();

while (1) {
    JobInfoPtr thejip = 0;
    JobInfoPtr thepreemptjip = 0;

    for (int i=0; i<ppending→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
        JobInfoPtr preemptjip = 0;
        int preemptnumstopped = 0;

        /* don't schedule jobs twice */
        if (jip→mark)
            continue;

        /* check if there are enough idle processors to run job */
        int runFlag = jip→numProcs ≤ numAvail;

        if (runFlag == 0) {
            /* otherwise, check if job can run on a partition used
            ** by a stopped job (but which has not yet been
            ** scheduled to restart) */
            for (int j=0; j<pstopped→numJobs; j++) {
                JobInfoPtr jip2 = TheJobList.findJob(pstopped→jobIds[j]);

                HostInfoPtr roothip = prunning→findHost(jip2→askedHosts[0]);
                if (roothip→schedMode ≠ HostInfo::STOPPED)
                    /* another job must currently be running,
                    ** starting, or stopping */
                    continue;

                if (roothip→numStoppedJobs ≥ 5)
                    /* too many jobs on partition */
                    continue;

                /* go ahead if job has same number of processors */
                if (jip→numProcs == jip2→numAskedHosts
                    && (preemptjip==0
                        || roothip→numStoppedJobs < preemptnumstopped)) {
                    preemptjip = jip2;
                    preemptnumstopped = roothip→numStoppedJobs;
                }
            }
        }

        /* now check if job can preempt a currently-running job */
        if (runFlag == 0) {
            for (int j=0; j<prunning→numJobs; j++) {
                JobInfoPtr jip2 = TheJobList.findJob(prunning→jobIds[j]);

                HostInfoPtr roothip = prunning→findHost(jip2→askedHosts[0]);
                if (roothip→schedMode ≠ HostInfo::RUNNING)
                    /* job must now be stopping or starting */
                    continue;

                if (roothip→numStoppedJobs + roothip→numRunningJobs ≥ 5)
                    /* too many jobs on partition */
```

```
                continue;

            if (jip2→resRunTime < MINRUNTIME)
                /* job has not been running long enough yet */
                continue;

            /* go ahead if job has same number of processors */
            if (jip→numProcs == jip2→numAskedHosts
                && (compRST(jip, jip2)<-1)
                && (preemptjip==0
                    || roothip→numStoppedJobs<preemptnumstopped)) {
                preemptjip = jip2;
                preemptnumstopped = roothip→numStoppedJobs;
            }
        }
    }

    /* select job if it is the best one */
    if ((runFlag || preemptjip ≠ 0)
        && (thejip==0 || compRST(jip, thejip)<0)) {
        thejip = jip;
        thepreemptjip = preemptjip;
    }
}


/* check if a stopped job is better than this one! this will
** only be the case if service-demand information is available */
for (int i=0; i<pstopped→numJobs; i++) {
    JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
    JobInfoPtr preemptjip = 0;

    /* don't schedule jobs twice */
    if (jip→mark)
        continue;

    HostInfoPtr roothip = prunning→findHost(jip→askedHosts[0]);
    if (roothip→schedMode == HostInfo::STARTING
        || roothip→schedMode == HostInfo::STOPPING)
        /* hosts are already busy */
        continue;

    /* check if job should preempt a currently-running job */
    if (roothip→schedMode == HostInfo::RUNNING) {
        JobInfoPtr runjip = roothip→runningJobs[0];

        if (runjip→resRunTime < MINRUNTIME)
            /* job hasn't run long enough */
            continue;

        if (!(compRST(jip, runjip) < -1))
            /* not enough of a difference */
            continue;

        preemptjip = runjip;
    }

    /* select job if it is the best one */
```

```
        if (thejip==0 || compRST(jip, thejip)<0) {
            thejip = jip;
            thepreemptjip = preemptjip;
        }
    }

    /* give up if no jobs can be scheduled */
    if (thejip == 0)
        break;

    /* mark job to avoid rescheduling */
    thejip->mark = 1;

    // Case 1: no preemption, just start or resume job
    if (!thepreemptjip) {
        if (thejip->qip == ppending) {

            int hcnt = thejip->numProcs;
            char **hosts = new CharPtr[hcnt];

            assert(hcnt <= numAvail);
            numAvail -= hcnt;

            int hidx = 0;
            while (hcnt) {
                while (prunning->hosts[hidx].mode != HostInfo::AVAIL
                        || prunning->hosts[hidx].schedMode != HostInfo::IDLE)
                    hidx++;

                prunning->hosts[hidx].schedMode = HostInfo::STARTING;

                hosts[hcnt-1] = prunning->hosts[hidx].hname;

                hcnt--;
                numAvail--;
            }

            if (debug) {
                printf("Running job %d (procs=%d,cpuDemand=%d) on:",
                        thejip->jobId, thejip->numProcs, thejip->cpuDemand);
                for (int cnt=0; cnt<thejip->numProcs; cnt++)
                    printf(" %s", hosts[cnt]);
                printf("\n");
            }

            TheLSFInteraction.setHosts(thejip->jobId, thejip->numProcs, hosts);
            delete [] hosts;
        }
        else {
            for (int i=0; i<thejip->numAskedHosts; i++) {
                HostInfoPtr hip = prunning->findHost(thejip->askedHosts[i]);

                assert(hip->schedMode == HostInfo::STOPPED);
                hip->schedMode = HostInfo::STARTING;
            }

            TheLSFInteraction.bresume(thejip->jobId);
```

```
        if (debug)
            printf("Resuming job %d.", thejip→jobId);
    }

    TheLSFInteraction.bswitch(thejip→jobId, prunning→qname);
}

// Case 2: preempt running job
if (thepreemptjip && thepreemptjip→qip == prunning) {

    printf("Preempting job %d.\n", thepreemptjip→jobId);

    for (int i=0; i<thepreemptjip→numAskedHosts; i++) {
        HostInfoPtr hip = prunning→findHost(thepreemptjip→askedHosts[i]);
        assert(hip→schedMode == HostInfo::RUNNING);
        hip→schedMode = HostInfo::STOPPING;
    }

    TheLSFInteraction.bswitch(thepreemptjip→jobId, pstopped→qname);
    TheLSFInteraction.bstop(thepreemptjip→jobId);

    if (thejip→qip == ppending) {
        char **hosts = new CharPtr[thepreemptjip→numAskedHosts];
        for (int i=0; i<thepreemptjip→numAskedHosts; i++)
            hosts[i] = thepreemptjip→askedHosts[i];

        if (debug) {
            printf("Running job %d (procs=%d,cpuDemand=%d) on:",
                    thejip→jobId, thejip→numProcs, thejip→cpuDemand);
            for (int cnt=0; cnt<thejip→numProcs; cnt++)
                printf(" %s", hosts[cnt]);
            printf("\n");
        }

        TheLSFInteraction.setHosts(thejip→jobId, thejip→numProcs, hosts);
        delete [] hosts;
    }

    scheduledJobs[numScheduledJobs++] = thejip;
    thejip→cache = 0;
}

// Case 3: run instead of stopped job
if (thepreemptjip && thepreemptjip→qip == pstopped) {

    printf("Leaving job %d\n", thepreemptjip→jobId);

    assert(thejip→qip == ppending);

    char **hosts = new CharPtr[thepreemptjip→numAskedHosts];
    for (int i=0; i<thepreemptjip→numAskedHosts; i++) {
        hosts[i] = thepreemptjip→askedHosts[i];

        HostInfoPtr hip = prunning→findHost(thepreemptjip→askedHosts[i]);

        assert(hip→schedMode == HostInfo::STOPPED);
        hip→schedMode = HostInfo::STARTING;
    }
```

```
            if (debug) {
                printf("Running job %d (procs=%d,cpuDemand=%d) on:",
                       thejip→jobId, thejip→numProcs, thejip→cpuDemand);
                for (int cnt=0; cnt<thejip→numProcs; cnt++)
                    printf(" %s", hosts[cnt]);
                printf("\n");
            }

            TheLSFInteraction.setHosts(thejip→jobId, thejip→numProcs, hosts);
            delete [] hosts;

            TheLSFInteraction.bswitch(thejip→jobId, prunning→qname);
        }
    }
}

void
LSFPreemptRigid::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-preempt-pending", 0);
    prunning = new QueueInfo("par-preempt-running", 1);
    pstopped = new QueueInfo("par-preempt-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

LSFPreemptRigid TheLSFPreemptRigid;
```

Appendix F

Sources for Adaptable Simple Preemptive Disciplines

The following code implements the LSF-PREEMPT-AD discipline. It is the most complex of all the implementations, as the overlapping of processor allocations for jobs requires considerable book-keeping.

Header File

```
/*
** £Id: LSFPAd.H,v 1.1 1997/01/03 20:25:12 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**          (c) 1997 Eric Parsons
*/

#ifndef LSFPAD_INCLUDE
#define LSFPAD_INCLUDE

#include "Sched.H"

class LSFPreemptAd {
    int keeplevel0;
    int numlevels;

    QueueInfoPtr ppending, prunning, pstopped;

    int numScheduledJobs;
    JobInfoPtr scheduledJobs[MAXPROCS];

    virtual void updateHosts();
    virtual void switchScheduledJobs();
    virtual int compRST(JobInfoPtr, JobInfoPtr);

public:
    virtual void init();
```

```
    virtual void resched();
};

extern LSFPreemptAd TheLSFPreemptAd;
#endif
```

Source File

```
/*
** £Id: LSFPAd.C,v 1.1 1997/01/03 20:25:11 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**         (c) 1997 Eric Parsons
*/

#include <stdio.h>

#include "Sched.H"
#include "LSFInt.H"
#include "LSFPAd.H"

/* In this discipline, we basically use the stoppedJobs matrix defined
** in the HostInfo class. The first row of the stoppedJobs matrix
** actually corresponds to running jobs; subsequent rows are for
** stopped jobs. In contrast to other disciplines, all processor
** slots used by a job are recorded in the same row (since this is a
** matrix-based discipline). */

const int MINRUNTIME = 120;
const int MINSWITCHTIME = 120;

void
LSFPreemptAd::updateHosts()
{
    /* initially assume all hosts are idle */
    for (int i=0; i<prunning->numHosts; i++) {
        prunning->hosts[i].schedMode = HostInfo::IDLE;

        for (int j=0; j<HostInfo::MAXSTOPPEDJOBS; j++)
            prunning->hosts[i].stoppedJobs[j] = 0;
    }

    numlevels = 0;
    keeplevel0 = 0;

    /* Record running jobs in first row of stoppedJobs matrix */
    for (int i=0; i<prunning->numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning->jobIds[i]);
        assert(jip);

        jip->pos = 0;
        for (int j=0; j<jip->numAskedHosts; j++) {
            HostInfoPtr hip = prunning->findHost(jip->askedHosts[j]);

            hip->stoppedJobs[0] = jip;
            hip->schedMode = HostInfo::RUNNING;
        }
```

```
        numlevels = 1;
    }

    for (int i=0; i<pstopped→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
        assert(jip);

        if (jip→lastEventType ≠ JobInfo::SUSPENDED) {
            /* job has not suspended yet (LSF is slow?) */
            /* place in first row of stoppedJobs matrix */

            jip→pos = 0;
            for (int j=0; j<jip→numAskedHosts; j++) {
                HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);

                hip→stoppedJobs[0] = jip;
                hip→schedMode=HostInfo::STOPPING;
            }
            keeplevel0 = 1;
            if (numlevels == 0) numlevels = 1;
        }
        else {
            /* this job is really stopped; find slot in stoppedJobs matrix */

            int pos=0, found;
            do {
                pos += 1;
                assert(pos<HostInfo::MAXSTOPPEDJOBS);

                found = 1;
                for (int j=0; j<jip→numAskedHosts; j++) {
                    HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);

                    if (hip→stoppedJobs[pos] ≠ 0) {
                        found = 0;
                        break;
                    }
                }
            } while (!found);


            jip→pos = pos;
            for (int j=0; j<jip→numAskedHosts; j++) {
                HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);

                hip→stoppedJobs[pos] = jip;
                if (hip→schedMode==HostInfo::IDLE)
                    hip→schedMode=HostInfo::STOPPED;
            }

            numlevels = pos+1;
        }
    }
}

void
LSFPreemptAd::switchScheduledJobs()
{
```

```
    keeplevel0 = numScheduledJobs>0;

    int xi = 0;
    while (xi < numScheduledJobs) {
        JobInfoPtr jip = scheduledJobs[xi];

        /* check if all hosts are available */
        int ready = 1;
        for (int j=0; j<jip→numAskedHosts; j++) {
            HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);

            assert(hip→schedMode ≠ HostInfo::RUNNING);
            if (hip→schedMode == HostInfo::STOPPING) {
                /* we essentially have a conflict here, because both the
                ** stopping job and the starting job should be placed in
                ** row zero; we avoid any problems by only considering row
                ** zero if there are any jobs in switchScheduledJobs */
                ready = 0;
                break;
            }
            else {
                hip→schedMode = HostInfo::STARTING;
                hip→stoppedJobs[0] = jip;
                hip→stoppedJobs[jip→pos] = 0;
                if (numlevels==0) numlevels=1;
            }
        }

        /* if so, run job */
        if (ready) {
            if (jip→qip == pstopped) TheLSFInteraction.bresume(jip→jobId);
            TheLSFInteraction.bswitch(jip→jobId, prunning→qname);

            scheduledJobs[xi] = scheduledJobs[--numScheduledJobs];
        }
        else {
            jip→cache = 0;                              // so job doesn't go away
            xi += 1;
        }
    }
}

int
LSFPreemptAd::compRST(JobInfoPtr jip1, JobInfoPtr jip2)
{
    long rst1 = 0, rst2 = 0;

    if (jip1→cpuDemand>0 && jip2→cpuDemand>0) {
        rst1 = jip1→cpuDemand - (jip1→cumAggRunTime+jip1→resAggRunTime);
        if (rst1<0) rst1 = 0;

        rst2 = jip2→cpuDemand - (jip2→cumAggRunTime+jip2→resAggRunTime);
        if (rst2<0) rst2 = 0;

        if (rst1+MINSWITCHTIME < rst2) return -2;
        if (rst1 < rst2) return -1;

        if (rst1 > rst2+MINSWITCHTIME) return 2;
```

```
        if (rst1 > rst2) return 1;
        if (rst1 == rst2) return 0;
        assert(0);                                         // should never get here
    }

    double acq1 = jip1→cumAggRunTime+jip1→resAggRunTime;
    double acq2 = jip2→cumAggRunTime+jip2→resAggRunTime;
    if (acq1 < 0.9∗acq2 && acq2-acq1 > MINSWITCHTIME) return -2;
    if (acq1 < acq2) return -1;

    if (0.9∗acq1 > acq2 && acq1-acq2 > MINSWITCHTIME) return 2;
    if (acq1 > acq2) return 1;
    if (acq1 == acq2) return 0;
    assert(0);
}

void
LSFPreemptAd::resched()
{
    int numSelJobs = 0;                                    // number of candidate jobs
    int numSelOpen = 0;                       // sum of minProcs for sel pending jobs
    int numProcsOpen = 0;                         // total free procs for pending jobs
    static JobInfoPtr selJobs[MAXPROCS];          // candidate set of jobs to run next
    static int selMark[MAXPROCS];                // marker indicating which will be run
    static int selAlloc[MAXPROCS];               // processor allocations for each job
    static double selDowdy[MAXPROCS];                      // dowdy values for each job

    /∗ determine which hosts are available ∗/
    updateHosts();

    /∗ Now try to switch scheduled jobs.  All scheduled jobs are also
    ∗∗ marked by this routine. ∗/
    switchScheduledJobs();

    int minlevel = -1;                                     // the current ”best” level
    JobInfoPtr minjip = 0;                                 // the ”best” job in minlevel

    if (numlevels<5) numlevels += 1;
    for (int level=0; level<numlevels; level++) {
        int numSelJobs2 = 0;
        int numSelOpen2 = 0;
        int numProcsOpen2 = 0;
        static JobInfoPtr selJobs2[MAXPROCS];
        static int selMark2[MAXPROCS];
        static int selAlloc2[MAXPROCS];
        static double selDowdy2[MAXPROCS];

        JobInfoPtr minjip2 = 0;

        /∗ skip all but the first level if jobs stopping or starting ∗/
        if (keeplevel0 && level>0)
            break;

        /∗ mark jobs so that we don’t schedule them twice ∗/
        for (int i=0; i<ppending→numJobs; i++) {
            JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
            jip→mark = 0;
            jip→pos = -1;
```

```
}
for (int i=0; i<prunning→numJobs; i++) {
    JobInfoPtr jip = TheJobList.findJob(prunning→jobIds[i]);
    jip→mark = 0;
}
for (int i=0; i<pstopped→numJobs; i++) {
    JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
    jip→mark = 0;
}

/* but we mark any leftover scheduledJobs */
for (int i=0; i<numScheduledJobs; i++)
    scheduledJobs[i]→mark = 1;

/* now count number of processor slots available at this level */
for (int j=0; j<prunning→numHosts; j++) {
    int mpl = 0;                                        // multiprogramming level
    for (int k=0; k<numlevels; k++)
        if (prunning→hosts[j].stoppedJobs[k] ≠ 0)
            mpl += 1;

    prunning→hosts[j].open = 0;
    if (prunning→hosts[j].stoppedJobs[level] == 0) {
        prunning→hosts[j].open = 1;

        if (mpl<5 && prunning→hosts[j].mode==HostInfo::AVAIL) {
            numProcsOpen2++;
            prunning→hosts[j].open = 2;                 // open for new jobs!!
        }
    }
    else {
        /* we use same loop to mark jobs and to find job having
        ** the best remaining service time, both at this level */
        JobInfoPtr jip = prunning→hosts[j].stoppedJobs[level];

        jip→mark = 1;
        if (minjip2==0 || compRST(jip, minjip2)<0)
            minjip2 = jip;
    }
}

/* check running jobs that MUST keep on running; only relevant
** if level>0 since all running jobs are at level 0; but if
** level>0, there cannot be any jobs in scheduledJobs, so
** stoppedJobs matrix will not be messed up*/
int levelok = 1;
if (level>0) {
    for (int j=0; j<prunning→numJobs; j++) {
        JobInfoPtr jip = TheJobList.findJob(prunning→jobIds[j]);

        if (jip→resRunTime<MINRUNTIME) {
            for (int k=0; k<jip→numAskedHosts; k++) {
                if (!prunning→findHost(jip→askedHosts[k])→open) {
                    levelok = 0;
                    goto checkrunend;
                }
            }
```

```
                    /* reserve processors at this level */
                    int nopen = 0;
                    for (int k=0; k<jip→numAskedHosts; k++) {
                        HostInfoPtr hip = prunning→findHost(jip→askedHosts[k]);
                        if (hip→open == 2) nopen += 1;
                        hip→open = 0;
                    }

                    selJobs2[numSelJobs2] = jip;
                    selDowdy2[numSelJobs2] = jip→dowdy;
                    selAlloc2[numSelJobs2] = jip→numAskedHosts;
                    numSelJobs2 += 1;
                    numSelOpen2 += nopen;
                    jip→mark = 1;

                    assert(numSelOpen2 ≤ numProcsOpen2);
                }
            }
        }
    checkrunend:
        if (!levelok)
            continue;

        /* schedule remaining jobs in available processor slots */
        while (1) {
            JobInfoPtr thejip = 0;
            int thenopen = 0;

            /* check pending jobs */
            for (int i=0; i<ppending→numJobs; i++) {
                JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);

                /* do not schedule job twice */
                if (jip→mark)
                    continue;

                if (jip→minProcs + numSelOpen2 ≤ numProcsOpen2
                    && (thejip==0 || compRST(jip, thejip)<0)){
                    thejip = jip;
                    thenopen = jip→minProcs;
                }
            }

            /* check stopped/running jobs */
            for (int qndx=0; qndx<1; qndx++) {
                QueueInfoPtr qip = qndx==0 ? pstopped : prunning;

                for (int i=0; i<qip→numJobs; i++) {
                    JobInfoPtr jip = TheJobList.findJob(qip→jobIds[i]);

                    /* do not schedule job twice */
                    if (jip→mark)
                        continue;

                    int runok = 1;
                    int nopen = 0;
                    for (int j=0; j<jip→numAskedHosts; j++) {
                        HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
```

```
                    if (!hip→open) {
                        runok = 0;
                        break;
                    }
                    else if (hip→open == 2) nopen += 1;
                }

                if (runok
                    && nopen + numSelOpen2 ≤ numProcsOpen2
                    && (thejip==0 || compRST(jip, thejip)<0)) {
                    thejip = jip;
                    thenopen = nopen;
                }
            }
        }

        if (thejip == 0)
            break;

        selJobs2[numSelJobs2] = thejip;
        selDowdy2[numSelJobs2] = thejip→dowdy;
        selAlloc2[numSelJobs2] = (thejip→qip == ppending
                                  ? thejip→minProcs
                                  : thejip→numAskedHosts);
        numSelJobs2 += 1;
        numSelOpen2 += thenopen;
        thejip→mark = 1;

        /* reserve processors at this level */
        if (thejip→qip ≠ ppending)
            for (int j=0; j<thejip→numAskedHosts; j++)
                prunning→findHost(thejip→askedHosts[j])→open = 0;

        if (minjip2==0
            || ((minjip2→pos == 0 && thejip→pos ≠ 0
                 && compRST(thejip, minjip2)<-1))
            || ((minjip2→pos == 0 && thejip→pos == 0
                 && compRST(thejip, minjip2)<0))
            || ((minjip2→pos ≠ 0
                 && compRST(thejip, minjip2)<0)))
            minjip2 = thejip;
    }

    if (minjip2==0)
        continue;

    if (minlevel < 0
        || ((minjip→pos == 0 && minjip2→pos ≠ 0
             && compRST(minjip2, minjip)<-1))
        || ((minjip→pos == 0 && minjip2→pos == 0
             && compRST(minjip2, minjip)<0))
        || ((minjip→pos ≠ 0
             && compRST(minjip2, minjip)<0))) {
        minlevel = level;
        minjip = minjip2;

        selJobs = selJobs2;
        selDowdy = selDowdy2;
```

```
            selAlloc = selAlloc2;
            numSelJobs = numSelJobs2;
            numSelOpen = numSelOpen2;
            numProcsOpen = numProcsOpen2;
        }
    }

    /* Now, allocate remaining processors to jobs s.t. efficiency is
    ** maximized.
    ** Note: T(p) = w(d+(1-d)/p)
    **       S(p) = 1/(d+(1-d)/p)
    **       E(p) = 1/(dp + (1-d)) = 1/(1+d(p-1))
    **       (but since we want to test E(p+1), we get 1/(1+dp) */
    while (numSelOpen < numProcsOpen) {
        int minidx = -1;
        for (int i=0; i<numSelJobs; i++) {
            if (selJobs[i]->qip == ppending
                && (minidx<0
                    || (selDowdy[i]>0.0 && selDowdy[minidx]==0.0)
                    || (selDowdy[i]>0.0 && selDowdy[minidx]>0.0
                        && (1.0/(1 + selDowdy[i]*selAlloc[i]*EXPFACT)
                            > 1.0/(1 + selDowdy[minidx]*selAlloc[minidx]*EXPFACT)))
                    || (selDowdy[i]==0.0 && selDowdy[minidx]==0.0
                        && selAlloc[i]<selAlloc[minidx])))
                minidx = i;
        }

        if (minidx<0)
            break;

        selAlloc[minidx]++;
        numSelOpen += 1;
    }

    /* Recompute processors that are open */
    for (int j=0; j<prunning->numHosts; j++) {
        int mpl = 0;                                    // multiprogramming level
        for (int k=0; k<numlevels; k++)
            if (prunning->hosts[j].stoppedJobs[k] != 0)
                mpl += 1;

        if (prunning->hosts[j].stoppedJobs[minlevel]==0
            && prunning->hosts[j].mode==HostInfo::AVAIL)
            if (mpl<5) prunning->hosts[j].open = 2;
            else prunning->hosts[j].open = 1;
        else
            prunning->hosts[j].open = 0;
    }

    if (minlevel != 0) {
        /* preempt running jobs that have not been selected to run;
        ** note that there are no starting jobs in stopped or pending
        ** queue to mess us up because otherwise, keeplevel0 would
        ** have been set */
        for (int i=0; i<prunning->numJobs; i++) {
            JobInfoPtr jip = TheJobList.findJob(prunning->jobIds[i]);
            assert(jip);
```

```
                int stillrunning=0;
                for (int j=0; j<numSelJobs; j++)
                    if (selJobs[j] == jip) {
                        stillrunning=1;
                        break;
                    }

                if (!stillrunning) {
                    for (int j=0; j<jip→numAskedHosts; j++) {
                        HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
                        hip→schedMode = HostInfo::STOPPING;
                    }

                    TheLSFInteraction.bswitch(jip→jobId, pstopped→qname);
                    TheLSFInteraction.bstop(jip→jobId);
                }
            }
        }

        for (int i=0; i<numSelJobs; i++) {
            /∗ mark processors being used by stopped or running jobs in selJobs;
            ∗∗ also resume those jobs that are currently stopped ∗/
            JobInfoPtr jip = selJobs[i];

            if (jip→qip ≠ ppending) {
                for (int j=0; j<jip→numAskedHosts; j++) {
                    HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
                    hip→open = 0;
                }
            }

            if (jip→qip == pstopped) {
                /∗ check if job can be restarted at this point ∗/
                int switchok = 1;
                for (int j=0; j<jip→numAskedHosts; j++) {
                    HostInfoPtr hip = prunning→findHost(jip→askedHosts[j]);
                    if (hip→schedMode == HostInfo::STOPPING)
                        switchok = 0;
                    else
                        hip→schedMode = HostInfo::STARTING;
                }

                if (switchok) {
                    TheLSFInteraction.bresume(jip→jobId);
                    TheLSFInteraction.bswitch(jip→jobId, prunning→qname);
                }
                else {
                    scheduledJobs[numScheduledJobs++] = jip;
                    jip→cache = 0;
                }
            }
        }

        /∗ Finally, deal with pending jobs ∗/
        for (int i=0; i<numSelJobs; i++) {
            JobInfoPtr jip = selJobs[i];

            if (jip→qip == ppending) {
```

```
            int hcnt=selAlloc[i];
            char **hosts = new CharPtr[hcnt];

            int hidx = 0;
            int switchok = 1;
            while (hcnt>0) {
                while (prunning->hosts[hidx].open != 2)
                    hidx++;

                assert(prunning->hosts[hidx].schedMode != HostInfo::STARTING
                        && prunning->hosts[hidx].schedMode != HostInfo::RUNNING);
                prunning->hosts[hidx].open = 0;
                if (prunning->hosts[hidx].schedMode == HostInfo::STOPPING)
                    switchok = 0;
                else
                    prunning->hosts[hidx].schedMode = HostInfo::STARTING;

                hosts[hcnt-1] = prunning->hosts[hidx].hname;

                hcnt--;
            }

            if (debug) {
                printf("Running job %d (procs=<%d,%d,%d>,cpuDemand=%d,dowdy=%f)
on:",
                        jip->jobId, jip->minProcs, jip->numProcs, jip->maxProcs,
                        jip->cpuDemand, float(jip->dowdy));
                for (int j=0; j<selAlloc[i]; j++)
                    printf(" %s", hosts[j]);
                printf("\n");
            }

            TheLSFInteraction.setHosts(jip->jobId, selAlloc[i], hosts);
            if (switchok)
                TheLSFInteraction.bswitch(jip->jobId, prunning->qname);
            else {
                scheduledJobs[numScheduledJobs++] = jip;
                jip->cache = 0;
            }

            delete [] hosts;
        }
    }
}

void
LSFPreemptAd::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-ad-preempt-pending", 0);
    prunning = new QueueInfo("par-ad-preempt-running", 1);
    pstopped = new QueueInfo("par-ad-preempt-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}
```

```
LSFPreemptAd TheLSFPreemptAd;
```

Appendix G

Sources for Migratable and Malleable Disciplines

The following code implements the LSF-MIGRATE and LSF-MALLEATE family of disciplines. The `adaptiveFlag` variable passed to the `reschedGeneric()` routine determines if the discipline is rigid or adaptive; the `malleableFlag` indicates if jobs are to be treated as being malleable; finally, the `subsetFlag` variable determines if the subset-sum algorithm is applied to improve the utilization of processors. To implement a hybrid discipline that supports both migratable and malleable jobs, the `malleableFlag` would be changed to be specific to a job rather than the discipline.

Header File

```
/*
** £Id: LSFMM.H,v 1.1 1997/01/03 20:25:11 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**        (c) 1997 Eric Parsons
*/

#ifndef LSFMM_INCLUDE
#define LSFMM_INCLUDE

#include "Sched.H"

class LSFMM {
protected:
    QueueInfoPtr ppending, prunning, pstopped;

    int numScheduledJobs;
    JobInfoPtr scheduledJobs[MAXPROCS];
    char **scheduledHosts[MAXPROCS];
    int numScheduledHosts[MAXPROCS];

    int numAvail;
```

```
    virtual void updateHosts();
    virtual void switchScheduledJobs();
    virtual int compRST(JobInfoPtr, JobInfoPtr);
    virtual void reschedGeneric(int, int, int);

public:
    virtual void init() = 0;
    virtual void resched() = 0;
};

class LSFMig : LSFMM {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(0,0,0);}
};

class LSFMigAd : LSFMM {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,0,0);}
};

class LSFMigAdSubset : LSFMM {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,0,1);}
};

class LSFMallAd : LSFMM {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,1,0);}
};

class LSFMallAdSubset : LSFMM {
public:
    virtual void init();
    virtual void resched() {reschedGeneric(1,1,1);}
};

extern LSFMig TheLSFMig;
extern LSFMigAd TheLSFMigAd;
extern LSFMigAdSubset TheLSFMigAdSubset;
extern LSFMallAd TheLSFMallAd;
extern LSFMallAdSubset TheLSFMallAdSubset;
#endif
```

Source File

```
/*
** £Id: LSFMM.C,v 1.1 1997/01/03 20:25:09 eparsons Exp £
**
** Created for the POW Project, Scheduling for Parallelism on Workstations,
**          (c) 1997 Eric Parsons
*/
```

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <math.h>

#include "Sched.H"
#include "LSFInt.H"
#include "LSFMM.H"
#include "Subset.H"

const int MINRUNTIME = 600;
const int MINSWITCHTIME = 600;

void
LSFMM::updateHosts()
{
    /* initially assume all hosts are idle */
    for (int i=0; i<prunning->numHosts; i++)
        prunning->hosts[i].schedMode = HostInfo::IDLE;

    /* scan all jobs in stopped queue, and mark hosts appropriately */
    for (int i=0; i<pstopped->numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(pstopped->jobIds[i]);
        assert(jip);

        if (jip->lastEventType != JobInfo::PENDPOSTMIG) {
            /* job has not suspended yet (LSF is slow?) */

            if (debug)
                printf("Job %d still migrating!\n",jip->jobId);

            for (int j=0; j<jip->numAskedHosts; j++) {
                HostInfoPtr hip = prunning->findHost(jip->askedHosts[j]);
                assert(hip);

                assert(hip->schedMode==HostInfo::IDLE);
                hip->schedMode = HostInfo::STOPPING;
            }
        }
    }

    /* scan all jobs in run queue, and mark hosts as busy */
    for (int i=0; i<prunning->numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning->jobIds[i]);
        assert(jip);

        for (int j=0; j<jip->numAskedHosts; j++) {
            HostInfo *hip = prunning->findHost(jip->askedHosts[j]);
            assert(hip);

            assert(hip->schedMode==HostInfo::IDLE);
            hip->schedMode = HostInfo::RUNNING;
        }
    }

    /* determine how many hosts are available for new jobs */
    numAvail = 0;
    for (int i=0; i<prunning->numHosts; i++)
```

```
        if (prunning→hosts[i].mode == HostInfo::AVAIL
            && (prunning→hosts[i].schedMode == HostInfo::IDLE
                || prunning→hosts[i].schedMode == HostInfo::STOPPING))
            numAvail++;
}

void
LSFMM::switchScheduledJobs()
{
    int xi = 0;
    while (xi < numScheduledJobs) {
        JobInfoPtr jip = scheduledJobs[xi];
        assert(jip);

        /* first, some consistency checks */
        assert(jip→lastEventType == JobInfo::SUBMITTED
               || jip→lastEventType == JobInfo::MIGRATING
               || jip→lastEventType == JobInfo::PENDPOSTMIG);

        /* mark job as being scheduled (to avoid being rescheduled) */
        jip→mark = 1;

        /* check if all hosts are available */
        int ready = 1;
        if (ready)
            for (int j=0; j<numScheduledHosts[xi]; j++) {
                HostInfoPtr hip = prunning→findHost(scheduledHosts[xi][j]);

                if (hip→mode == HostInfo::AVAIL
                    && (hip→schedMode == HostInfo::IDLE
                        || hip→schedMode == HostInfo::STOPPING))
                    numAvail--;

                if (hip→schedMode == HostInfo::STOPPING)
                    ready = 0;
                hip→schedMode = HostInfo::STARTING;
            }

        if (ready && jip→lastEventType ≠ JobInfo::MIGRATING) {
            TheLSFInteraction.setHosts(jip→jobId, numScheduledHosts[xi], scheduledHosts[xi]);
            TheLSFInteraction.bswitch(jip→jobId, prunning→qname);
            delete [] scheduledHosts[xi];

            numScheduledJobs -= 1;
            scheduledJobs[xi] = scheduledJobs[numScheduledJobs];
            scheduledHosts[xi] = scheduledHosts[numScheduledJobs];
            numScheduledHosts[xi] = numScheduledHosts[numScheduledJobs];
        }
        else {
            jip→cache = 0;                                  // so job doesn't go away
            xi += 1;
        }
    }

    assert(numAvail ≥ 0);
}

int
```

```
LSFMM::compRST(JobInfoPtr jip1, JobInfoPtr jip2)
{
    long rst1 = 0, rst2 = 0;

    if (jip1→cpuDemand>0 && jip2→cpuDemand>0) {
        rst1 = jip1→cpuDemand - (jip1→cumAggRunTime+jip1→resAggRunTime);
        if (rst1<0) rst1 = 0;

        rst2 = jip2→cpuDemand - (jip2→cumAggRunTime+jip2→resAggRunTime);
        if (rst2<0) rst2 = 0;

        if (rst1+MINSWITCHTIME < rst2) return -2;
        if (rst1 < rst2) return -1;

        if (rst1 > rst2+MINSWITCHTIME) return 2;
        if (rst1 > rst2) return 1;
        if (rst1 == rst2) return 0;
        assert(0);                                          // should never get here
    }

    double acq1 = jip1→cumAggRunTime+jip1→resAggRunTime;
    double acq2 = jip2→cumAggRunTime+jip2→resAggRunTime;
    if (acq1 < 0.9∗acq2 && acq2-acq1 > MINSWITCHTIME) return -2;
    if (acq1 < acq2) return -1;

    if (0.9∗acq1 > acq2 && acq1-acq2 > MINSWITCHTIME) return 2;
    if (acq1 > acq2) return 1;
    if (acq1 == acq2) return 0;
    assert(0);
}

void
LSFMM::reschedGeneric(int adaptiveFlag, int malleableFlag, int subsetFlag)
{
    static int numSelJobs;                          // number of candidate jobs
    static JobInfoPtr selJobs[MAXPROCS];            // candidate set of jobs to run next
    static int selMark[MAXPROCS];                   // marker indicating which will be run
    static int selAlloc[MAXPROCS];                  // processor allocations for each job
    static double selDowdy[MAXPROCS];               // dowdy values for each job

    static int numProcsRunning;                     // number of procs used by running jobs

    /∗ determine which hosts are available ∗/
    updateHosts();

    /∗ Unmark all jobs in all queue queues.  Marking is used to
    ∗∗ indicate which jobs have been selected to run; we postpone
    ∗∗ scheduling until the end, since we can only choose allocations
    ∗∗ once we know which jobs to run ∗/
    for (int i=0; i<ppending→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(ppending→jobIds[i]);
        jip→mark = 0;
    }
    for (int i=0; i<prunning→numJobs; i++) {
        JobInfoPtr jip = TheJobList.findJob(prunning→jobIds[i]);
        jip→mark = 0;
    }
    for (int i=0; i<pstopped→numJobs; i++) {
```

```
        JobInfoPtr jip = TheJobList.findJob(pstopped→jobIds[i]);
        jip→mark = 0;
    }

    /* Now try to switch scheduled jobs.  All scheduled jobs are also
    ** marked by this routine. */
    switchScheduledJobs();

    numSelJobs = 0;
    numProcsRunning = 0;
    while (1) {
        JobInfoPtr thejip = 0;

        QueueInfoPtr queues[3] = {ppending, pstopped, prunning};
        for (int qnum=0; qnum<3; qnum++) {
            QueueInfoPtr qip = queues[qnum];
            assert(qip);

            for (int i=0; i<qip→numJobs; i++) {
                JobInfoPtr jip = TheJobList.findJob(qip→jobIds[i]);

                /* don't schedule jobs twice */
                if (jip→mark)
                    continue;

                /* skip jobs in the run queue that haven't run long enough */
                if (qip == prunning && jip→resRunTime < MINRUNTIME)
                    continue;

                if (thejip==0
                    || (thejip→qip ≠ prunning && compRST(jip, thejip)<0)
                    || (thejip→qip == prunning && compRST(jip, thejip)<-1))
                    thejip = jip;
            }
        }

        if (thejip==0)
            break;

        /* Mark job, and record details */
        thejip→mark = 1;
        selJobs[numSelJobs] = thejip;
        selMark[numSelJobs] = 0;
        if (adaptiveFlag)
            if (thejip→qip == ppending || malleableFlag)
                selAlloc[numSelJobs] = thejip→minProcs;
            else
                selAlloc[numSelJobs] = thejip→numAskedHosts;
        else
            selAlloc[numSelJobs] = thejip→numProcs;
        selDowdy[numSelJobs] = thejip→dowdy;
        numSelJobs += 1;

        if (thejip→qip == prunning)
            for (int j=0; j<thejip→numAskedHosts; j++) {
                HostInfoPtr hip = prunning→findHost(thejip→askedHosts[j]);
                if (hip→mode == HostInfo::AVAIL)
                    numAvail += 1;
```

```
        }
}

/* determine which jobs to run */
int numSelFF = 0;
int numSelProcs = 0;
for (int i=0; i<numSelJobs; i++)
    if (numSelProcs+selAlloc[i]≤numAvail) {
        selMark[i] = 1;
        numSelFF += 1;
        numSelProcs += selAlloc[i];
    }

if (subsetFlag) {
    /* We pack jobs more efficienty using the subset-sum algorithm */
    int numSelChop
        = int(floor(numSelFF*max(0.0, 1.0-numSelJobs/(16*numSelFF))));
    assert(numSelChop ≤ numSelFF);

    /* recompute variables with respect to chopped FF list*/
    numSelFF = 0;
    numSelProcs = 0;
    for (int i=0; i<numSelJobs; i++)
        if (numSelFF<numSelChop && selMark[i]) {
            numSelFF += 1;
            numSelProcs += selAlloc[i];
        }
        else
            selMark[i] = 0;

    int numPackProcs = numAvail - numSelProcs;

    /* compute best packing for remaining procs using remaining jobs */
    int numPackJobs = 0;
    int sumPackProcs = 0;
    for (int i=0; i<numSelJobs; i++) {
        if (selMark[i] || selAlloc[i]>numPackProcs) continue;

        /* In what follows, the w[] array is specific to the
        ** subset-sum implementation, which is described in
        ** Knapsack Problems: Algorithms and Computer
        ** Implementations, Martello & Thoth, 1990.  Note that the
        ** w[] array starts at 1 */

        if (selAlloc[i] == numPackProcs) {
            /* If we have a job that requires all processors, subset-sum
            ** algorithm will choose this one first, so make it easy */
            numPackJobs = 1;
            sumPackProcs = selAlloc[i];
            w[1].wt = selAlloc[i];
            w[1].sel = 1;
            w[1].data = (void *)i;
            break;
        }
        else {
            /* Otherwise, add job to list */
            numPackJobs += 1;
            sumPackProcs += selAlloc[i];
```

```
                w[numPackJobs].wt = selAlloc[i];
                w[numPackJobs].rnd = numPackJobs;
                w[numPackJobs].sel = 0;
                w[numPackJobs].data = (void *)i;
            }
        }

        if (numPackJobs>0) {
            if (sumPackProcs>numPackProcs)
                /* Subset-sum algorithm only works if the sum of the
                ** minimum processor allocation exceeds what is available */
                subset(numPackProcs, numPackJobs);
            else
                /* If this is not the case, then just choose all jobs */
                for (int i=1; i<numPackJobs+1; i++)
                    w[i].sel = 1;

            /* Fold back subset-sum results into sel*[] arrays */
            for (int i=1; i<numPackJobs+1; i++)
                if (w[i].sel) {
                    assert(selMark[int(w[i].data)]==0);
                    assert(w[i].wt==selAlloc[int(w[i].data)]);

                    selMark[int(w[i].data)] = 1;
                    numSelProcs += w[i].wt;
                }
        }
    }

    /* Now, allocate remaining processors to jobs s.t. efficiency is
    ** maximized.
    ** Note: T(p) = w(d+(1-d)/p)
    **       S(p) = 1/(d+(1-d)/p)
    **       E(p) = 1/(dp + (1-d)) = 1/(1+d(p-1))
    **       (but since we want to test E(p+1), we get 1/(1+dp) */
    if (adaptiveFlag)
        while (numSelProcs < numAvail) {
            int minidx = -1;

            for (int i=0; i<numSelJobs; i++) {
                if (!selMark[i])
                    continue;

                if (selJobs[i]→qip ≠ ppending && !malleableFlag)
                    continue;

                if (minidx<0
                    || (selDowdy[i]>0.0 && selDowdy[minidx]==0.0)
                    || (selDowdy[i]>0.0 && selDowdy[minidx]>0.0
                        && (1.0/(1 + selDowdy[i]*selAlloc[i]*EXPFACT)
                            > 1.0/(1 + selDowdy[minidx]*selAlloc[minidx]*EXPFACT)))
                    || (selDowdy[i]==0.0 && selDowdy[minidx]==0.0
                        && selAlloc[i]<selAlloc[minidx]))
                    minidx = i;
            }

            if (minidx<0)
                break;
```

```
            selAlloc[minidx]++;
            numSelProcs += 1;
        }

    /* Preempt running jobs that have not been selected to run or
    ** whose allocations might change! */
    for (int i=0; i<numSelJobs; i++) {
        JobInfoPtr jip = selJobs[i];

        if (selJobs[i]->qip == prunning
            && (selMark[i]==0 || selAlloc[i] != selJobs[i]->numAskedHosts)) {

            printf("Preempting job %d.\n", jip->jobId);

            for (int j=0; j<jip->numAskedHosts; j++) {
                HostInfoPtr hip = prunning->findHost(jip->askedHosts[j]);

                assert(hip->schedMode == HostInfo::RUNNING);
                hip->schedMode = HostInfo::STOPPING;
            }

            TheLSFInteraction.bkill(jip->jobId, SIGUSR2);
            TheLSFInteraction.bswitch(jip->jobId, pstopped->qname);
            TheLSFInteraction.bmig(jip->jobId);
        }
    }

    /* Schedule jobs (and run if possible) */
    for (int i=0; i<numSelJobs; i++) {
        if (!selMark[i])
            continue;

        if (selJobs[i]->qip == prunning
            && selAlloc[i] == selJobs[i]->numAskedHosts)
            continue;

        JobInfoPtr jip = selJobs[i];
        int hcnt = selAlloc[i];
        char **hosts = new CharPtr[hcnt];

        int ready = 1;
        int hidx = 0;
        while (hcnt>0) {
            while (prunning->hosts[hidx].mode != HostInfo::AVAIL
                    || (prunning->hosts[hidx].schedMode != HostInfo::IDLE
                        && prunning->hosts[hidx].schedMode != HostInfo::STOPPING))
                hidx++;

            if (prunning->hosts[hidx].schedMode == HostInfo::STOPPING)
                ready = 0;

            prunning->hosts[hidx].schedMode = HostInfo::STARTING;

            hosts[hcnt-1] = prunning->hosts[hidx].hname;

            hcnt--;
```

```
            numAvail--;
        }

        if (debug) {
            if (!adaptiveFlag)
                printf("%s job %d (procs=%d,cpuDemand=%d,dowdy=%f) on:",
                        ready ? "Scheduling" : "Running",
                        jip→jobId, jip→numProcs, jip→cpuDemand, jip→dowdy);
            else
                printf("%s job %d (procs=<%d,%d,%d>,cpuDemand=%d,dowdy=%f) on:",
                        ready ? "Scheduling" : "Running",
                        jip→jobId, jip→minProcs, jip→numProcs, jip→maxProcs,
                        jip→cpuDemand, jip→dowdy);

            for (int j=0; j<selAlloc[i]; j++)
                printf(" %s", hosts[j]);
            printf("\n");
        }

        if (!ready
            || jip→qip == prunning
            || (jip→qip == pstopped
                && jip→lastEventType ≠ JobInfo::PENDPOSTMIG)) {
            /* postpone start until all procs available or job migrated */
            selJobs[i]→cache = 0;
            scheduledJobs[numScheduledJobs] = selJobs[i];
            scheduledHosts[numScheduledJobs] = hosts;
            numScheduledHosts[numScheduledJobs] = selAlloc[i];
            numScheduledJobs += 1;

            hosts = 0;
        }
        else {
            /* job can be started immediately */
            TheLSFInteraction.setHosts(jip→jobId, selAlloc[i], hosts);
            TheLSFInteraction.bswitch(jip→jobId, prunning→qname);

            delete [] hosts;
        }
    }
}

void
LSFMig::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-migrate-pending", 0);
    prunning = new QueueInfo("par-migrate-running", 1);
    pstopped = new QueueInfo("par-migrate-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

void
LSFMigAd::init()
{
```

```
    /* register queues for this discipline */
    ppending = new QueueInfo("par-ad-migrate-pending", 0);
    prunning = new QueueInfo("par-ad-migrate-running", 1);
    pstopped = new QueueInfo("par-ad-migrate-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

void
LSFMigAdSubset::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-adpack-migrate-pending", 0);
    prunning = new QueueInfo("par-adpack-migrate-running", 1);
    pstopped = new QueueInfo("par-adpack-migrate-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

void
LSFMallAd::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-ad-malleate-pending", 0);
    prunning = new QueueInfo("par-ad-malleate-running", 1);
    pstopped = new QueueInfo("par-ad-malleate-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

void
LSFMallAdSubset::init()
{
    /* register queues for this discipline */
    ppending = new QueueInfo("par-adpack-malleate-pending", 0);
    prunning = new QueueInfo("par-adpack-malleate-running", 1);
    pstopped = new QueueInfo("par-adpack-malleate-stopped", 0);

    TheQueueList.addQueue(ppending);
    TheQueueList.addQueue(prunning);
    TheQueueList.addQueue(pstopped);
}

LSFMig TheLSFMig;
LSFMigAd TheLSFMigAd;
LSFMigAdSubset TheLSFMigAdSubset;
LSFMallAd TheLSFMallAd;
LSFMallAdSubset TheLSFMallAdSubset;
```