Implementing Multiprocessor Scheduling Disciplines

Eric W. Parsons and Kenneth C. Sevcik

Computer Systems Research Institute
University of Toronto, Canada

{eparsons,kcs}@cs.utoronto.ca

Abstract

An important issue in multiprogrammed multiprocessor systems is the scheduling of parallel jobs. Consequently, there has been a considerable amount of analytic research in this area recently. A frequent criticism, however, is that proposed disciplines that are studied analytically are rarely ever implemented and even more rarely incorporated into commercial scheduling software. In this paper, we seek to bridge this gap by describing how at least one commercial scheduling system, namely Platform Computing's Load Sharing Facility, can be extended to support a wide variety of new scheduling disciplines.

We then describe the design and implementation of a number of multiprocessor scheduling disciplines, each differing considerably in terms of the type of preemption that is assumed to be available and in terms of the flexibility allowed in allocating processors. In evaluating the performance of these disciplines, we find that preemption can significantly reduce overall response times, but that the performance of disciplines that must commit to allocations when a job is first activated can be significantly affected by transient loads.

1 Introduction

As large-scale multiprocessor systems become available to a growing user population, mechanisms to share such systems among users are becoming 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, systems may have been acquired exclusively for use by a small number of individuals, they are now being installed for the benefit of large user communities, making the efficient scheduling of these systems an important problem.

Although much analytic research has been done in this area, one of the frequent criticisms made is that proposed disciplines are rarely implemented and even more rarely ever become part of commercial scheduling systems. The commercial scheduling systems presently available, for the most part, only support run-to-completion (RTC) disciplines and have very little flexibility in adjusting processor allocations. These constraints can lead to both high response times and low system utilizations. On the other hand, most research results support the need for both preemption and mechanisms for adjusting processor allocations of jobs.

Given that a number of high-performance computing centers have begun to develop their own scheduling software [Hen95, Lif95, SCZL96, WMKS96], it is clear that existing commercial scheduling software is often inadequate. To support these centers, however, mechanisms to extend existing systems with external (customer-provided) policies are starting to become available in commercial software [SCZL96]. This allows new scheduling policies to be easily implemented, without having to re-implement much of the base functionality typically found in this type of software.

The primary objective of this paper is to help bridge the gap between some of the analytic research and practical implementations of scheduling disciplines. As such, we describe the implementation of a number of scheduling disciplines, involving various types of job preemption and processor allocation flexibility. Furthermore, we describe how different types of knowledge (e.g., amount of computational work or speedup characteristics) can be included in the design of these disciplines. A secondary objective of our work is to briefly examine the benefits preemption and knowledge may have on the performance of parallel scheduling disciplines.

The remainder of the paper is organized as follows. In the next section, we present motivation for the types of scheduling disciplines that we chose to implement. In Section 3, we describe Load Sharing Facility (LSF), the commercial software scheduling software on which we based our implementation. In Sections 4 and 5, we describe an extension library we have developed to facilitate the development of multiprocessor scheduling disciplines, followed by the set of disciplines we have implemented. Finally, we present our experimental results in Section 6 and our conclusions in Section 7.

2 Background

There have been many analytic studies done on parallel-job scheduling since it was first examined in the late eighties. Much of this work has led to three basic observations.

First, the performance of a system can be significantly degraded if a job is not given exclusive use of the processors on which it is running. Otherwise, the threads of a job may have to wait for significant amounts of time at synchronization points. This can either result in large context-switch overheads or wasted processor cycles. In general, a single thread is associated with each processor, an approach which is known as *coordinated* or *gang* scheduling [Ous82, FR92]. Sometimes, however, it is possible to multiplex threads of the same job on a reduced number of processors and still achieve good performance [MZ94]. (In the latter case, it is still assumed that only threads from a single job are simultaneously active on any given processor.)

Second, jobs generally make more efficient use of the processing resources given smaller processors allocations. As a result, providing the scheduler with some flexibility in allocating processors can significantly improve overall performance [GST91, Sev94, NSS93, RSD+94]. In most systems, users specify precisely the number of processors which should be allocated to each job, a practice that is known as *rigid* scheduling. In *adaptive* scheduling disciplines, the user specifies a minimum processor allocation, usually resulting from constraints due to memory, and a maximum, corresponding to the point after which no further processors are likely to be beneficial. In some cases, it may also be necessary to specify additional constraints on the allocation, such as being a power of two. If available, specific knowledge about jobs, such as amount of work or speedup characteristics, can further aid the scheduler in allocating processors in excess of minimum allocations.

In adaptive disciplines, jobs can be allocated a large number of processors at light loads, giving them good response times. As the load increases, however, average allocation sizes can be decreased so as to improve the efficiency at which the processors are utilized, and hence allowing a higher load to be sustained (i.e., a higher *sustainable throughput*). Also, adaptive disciplines can better utilize processors than rigid ones because, with the latter, processors are often left idle due to packing inefficiencies, while adaptive disciplines can adjust allocations to make use of all available processors.

The third observation is that workloads found in practice tend to have a very high degree of variability in the amount of computational work (also known as *service demand*) [CMV94, FN95, Gib96]. In other words, most jobs have very small service demands but a few jobs can run for a very long time. Run-to-completion (RTC) disciplines exhibit very high response times because once a long-running job is dispatched, short jobs must wait a considerable amount of time before processors become available. Preemption can significantly reduce the mean response times of these workloads relative to run-to-completion disciplines [PS95].

Unlike the sequential case, preemption of parallel jobs can be quite expensive and complex to support. Fortunately, results indicate that preemption does not need to be invoked frequently to be useful, since only long-running jobs ever need to be preempted. In this paper, we consider three distinct types of preemption, in increasing order of implementation complexity.

Simple In simple preemption, a job may be preempted but its threads may not be migrated to another processor. This type of preemption is the easiest to support (as threads need only be stopped), and may be the only type available on message-passing systems.

Migratable In migratable preemption, a job may be preempted and its threads migrated. Normally, this type of preemption can be easily supported in shared-memory systems, but ensuring that data accessed by each thread is also migrated appropriately can be difficult. In message-passing systems, operating-system support for migration is not usually provided, but checkpointing can often be employed instead.[1] For example, the Condor system provides a transparent checkpointing facility for parallel applications that use 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 run-time library re-establishes the necessary network connections and resumes the computation from the point at which the last checkpoint was taken. As such, using checkpointing to preempt a job is similar in cost to swapping, except that all kernel resources are relinquished.

Malleable In malleable preemption, the size of a job's processor allocation may be changed after it has begun execution, a feature that normally requires explicit support within the application.[2] In the *process control* approach, the application must be designed to to adapt dynamically to changes in processor allocation while it is running [TG89, GTS91, NVZ96]. As this type of support is uncommon, a simpler strategy may be to rely on application-level checkpointing, often used by long-running jobs to tolerate system failures. For these cases, it might be possible to modify the application so as to store checkpoints in a format that is independent of allocated processors, thus allowing the job to be subsequently restarted on a different number of processors.

A representative sample of coordinated scheduling disciplines that have been previously studied is presented in Figure 1, classified according to the type of preemption available and the flexibility in processor allocation (i.e., rigid versus adaptive). Adaptive disciplines are further categorized by the type of information they assume to be available, which can include service demand, speedup characteristics, and memory requirements.[3] All types of preemption (simple, migratable, malleable) can be applied to all adaptive disciplines, but only simple and migratable preemption are meaningful for rigid disciplines. The disciplines proposed in this paper are highlighted in italics. (A more complete version of this table can be found elsewhere [Par97].)

LoadLeveler is a commercial scheduling system designed primarily for the IBM SP-2 system. A recent extension to LoadLeveler that has become popular is EASY [Lif95, SCZL96]. This is a rigid RTC scheduler that 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 a time by which that job will be run; jobs that are subsequently submitted may be run before this job only if they do not delay the start of the previously-scheduled job's execution (i.e., a gap exists in the schedule containing enough processors for sufficient time).

The disciplines that we present in this paper have been implemented as extensions to another commercial scheduling system, called Load Sharing Facility (LSF). By building on top of 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. LSF runs on a large number of platforms, including the SP-2, SGI Challenge, SGI Origin, and HP Exemplar, making it an attractive vehicle for this type of scheduling research. Our work is based on LSF version 2.2a.

3 Load Sharing Facility

Although originally designed for load balancing in workstation clusters, LSF is now becoming popular for parallel job scheduling on multiprocessor systems. Of greatest relevance to this work is the batch subsystem.

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 currently 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 1). An administrator could define

[1] Although the costs of this approach may appear to be large, we have found that significant reductions in mean response times can be achieved with minimal impact on throughput, even with large checkpointing overheads.

[2] Malleable preemption is often termed dynamic partitioning in the literature, but we find it more convenient to treat it as a type of preemption.

[3] Some rigid schedulers do use service-demand information if available, but this distinction is not shown in this table.

	Rigid	Adaptive	*Work*	*Speedup*	*Memory*
RTC	RTC [ZM90]	A+,A+&mM [Sev89]	yes	min/max	no
	PPJ [RSD+94]	ASP [ST93]	no	pws	no
	NQS	PWS [GST91]	no	no	no
	LSF	Equal,IP [RSD+94]	no	no	no
	LoadLeveler	SDF [CMV94]	yes	no	no
	EASY [Lif95]	AVG,Adapt-AVG [CMV94]	no	avg	no
	LSF-RTC	***LSF-RTC-AD(SUBSET)***	either	either	either
Preemption					
simple	Cosched (matrix) [Ous82]				
	LSF-PREEMPT	***LSF-PREEMPT-AD(SUBSET)***	either	either	either
migratable	Cosched (other) [Ous82]	Round-Robin [ZM90]	no	no	no
	RRJob [MVZ93]	FB-ASP,FB-PWS	no	pws	no
	LSF-MIG	***LSF-MIG-AD(SUBSET)***	either	either	either
malleable		Equi/Dynamic Partition [TG89, MVZ93]	no	no	no
		FOLD,EQUI [MZ94]	no	no	no
	(not applicable)	W&E [BG96]	yes	yes	no
		BUDDY,EPOCH [MZ95]	no	no	yes
		MPA [PS96b, PS96a]	no	yes	yes
		LSF-MALL-AD(SUBSET)	either	either	either

Table 1: Representative set of disciplines that have been proposed and evaluated in the literature. Disciplines presented in this paper are italicized and have the prefix “LSF-”; for the adaptive ones, a regular and a “SUBSET” version are provided.

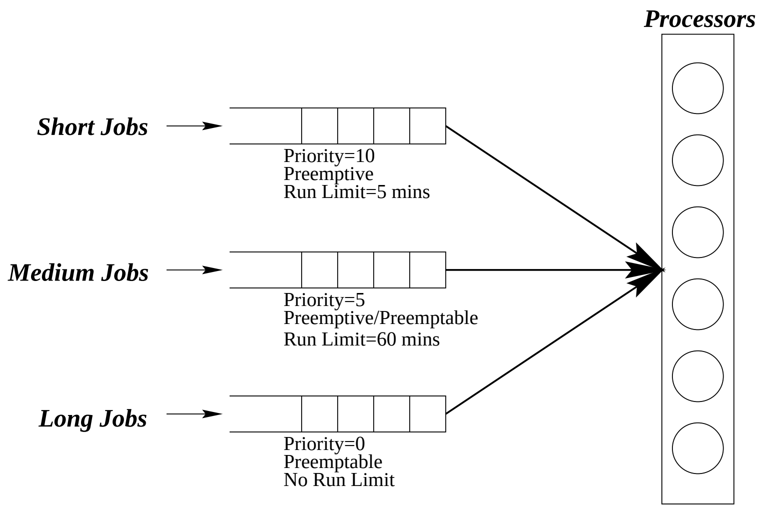

Figure 1: Example of a possible sequential-job queue configuration in LSF to favour short-running jobs. Jobs submitted to the short-job queue have the highest priority, followed by medium- and long-job queues. The queues are configured to be preemptable (allowing jobs in the queue to be preempted by 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.

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 any job that exceeds 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 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.

4 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. For this purpose, LSF provides an extensive application-programmer interface (API), allowing many aspects of job scheduling 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 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.[4]

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 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.

[4]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.

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,[5] 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 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 scheduling 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.

4.1 Job and System Information Cache

The Job and System Information Cache (JSIC) contains all the information about jobs, queues, and processors that are relevant to the scheduling disciplines that are part of the extension. Our data structures were 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).

In our library, 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 experimental reasons) to have distinct processor information objects associated with each queue. Using a global 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, but this was not necessarily the case on our system. 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. For each, the state of

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

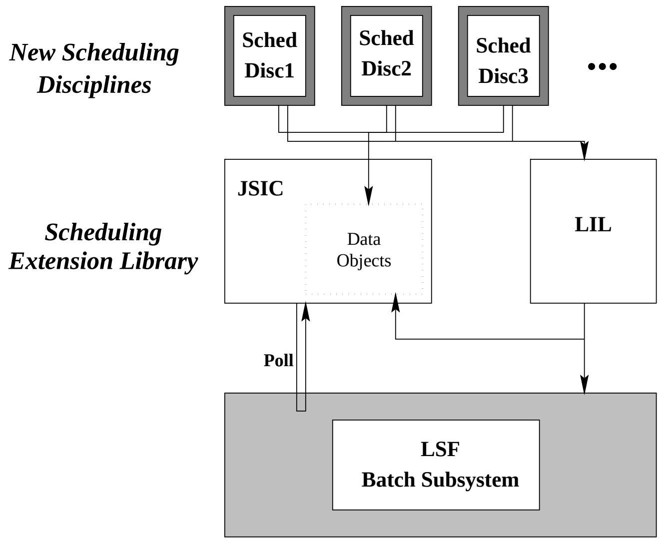

Figure 2: High-level design of scheduling extension extension library. As shown, the extension library supports multiple scheduling disciplines running concurrently within the same process.

the processor and a list of jobs running on the processor can be obtained.

4.2 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. As well, 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, such as physical memory, licenses, or swap space, required by the job 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 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, illustrated in Table 2, basically translate high-level scheduling operations into low-level LSF calls.

4.2.1 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

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 has previously been 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 pending state, allowing it to be subsequently restarted on a different set of processors.

Table 2: High-level scheduling functions provided by LSF Interaction Layer.

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 contrast, we assume that migratable and malleable preemption are implemented via a checkpointing facility, as described in Section 2. As a result, preempted jobs do not occupy any kernel resources, allowing any number of jobs to be in this state (assuming disk space for checkpointing is abundant).

To identify migratable jobs, we set an LSF flag in the submission request indicating that the job is re-runnable. To migrate such a job, we first send it a checkpoint signal (in our case, the `SIGUSR2` signal), and then send LSF a migrate request for 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 such a job to a queue that has been configured to not dispatch jobs prior to 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.

4.3 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 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 1); when the job has acquired a certain amount of processing time, the scheduling extension switches the job to the medium-priority queue, and after some more processing time, to the low-priority queue. In this way, 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 track the progress of jobs simply by examining the jobs in each of the three queues.

5 Parallel-Job Scheduling Disciplines

We now turn our attention to the parallel-job scheduling disciplines that we have implemented as LSF extensions. Important to the design of these disciplines are the costs associated with using LSF on our platform. 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 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.

If provided to the scheduler, service demand information is specified in terms of the amount of computation required on a single processor and speedup characteristics are specified in terms of the fraction of work that is sequential. 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 allocating processors. Since jobs can vary considerably in terms of their speedup characteristics, computing the remaining processing time will only be accurate if speedup information is available.

5.1 Run-to-Completion Disciplines

Next, we describe the run-to-completion disciplines. All three variants listed in Table 1 (i.e., LSF-RTC, LSF-RTC-AD, and LSF-RTC-ADSUBSET) are quite similar and, as such, are implemented in a single module of the scheduling extension. The LSF-RTC discipline 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.

The LSF system, and hence the JSIC, maintains jobs in order of arrival, so the default RTC 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 shortest processing time (SPT) discipline (again with skipping).

The LSF-RTC-AD discipline is very similar to the ASP discipline proposed by Setia *et al.* [ST93], 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 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.

If speedup information information is available, the scheduler allocates each leftover processor, in turn, to the job whose efficiency will be highest *after* the allocation. This approach minimizes both the processor and memory occupancy in a distributed-memory environment, leading to the highest possible sustainable throughput [PS96a].

The SUBSET variant seeks to improve the efficiency by which processors are utilized by applying an algorithm known as a subset-sum algorithm [MT90]. The basic principle is that, if jobs are selected so as to maximize the sum of minimum processor allocations, then that will lead to the fewest number of "excess" processors. Since we assume that jobs utilize processors more efficiently as allocations size decreases (down to the minimum allocation size), then reducing the excess processors allows the system to run at a higher overall efficiency.

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

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

(δ is a tunable parameter that determines how aggressively the scheduler seeks to minimize excess processors as the load increases; for our experiments, we chose $\delta = 5$.) Using any leftover processors and leftover jobs, the scheduler applies the subset-sum algorithm to select the set of jobs that minimizes "excess" processors. 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.

5.1.1 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.[6] 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.

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 a 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 than 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 is 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

[6] Packing losses occur when processors are left idle, either because there are an insufficient number to meet the minimum processor requirements of pending jobs or if only some of the processors required by stopped jobs are available.

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).

5.1.2 Migratable and Malleable Preemptive Disciplines

In contrast to the simple preemptive disciplines, the migratable and malleable ones assume that a job can be checkpointed and restarted at a later point in time. The primary difference between the two types is that, in the migratable case, jobs are always resumed with the same number of processors allocated when the job first started, whereas in the malleable case, a job can be restarted with a different number of processors.

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.

Similar to the run-to-completion case, SUBSET-variants of the adaptive disciplines have also been implemented.

6 Performance Results

The evaluation of the disciplines described in the previous section is primarily qualitative in nature. There are two reasons for this. First, experiments must be performed in real time rather than in simulated time, requiring a considerable amount of time to execute a relatively small number of jobs. Moreover, failures that can (and do) occur during the experiments can significantly influence the results, although such failures can be tolerated by the disciplines. Second, we intend our implementations to demonstrate the practicality of a 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).

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 (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.

An experiment consists 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[7] 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. 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.

6.1 Workload Model

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, a distribution whose median is 2985 seconds.[8] The parameters are consistent with measurements made over the past year at the Cornell Theory Center (scaled to 128 processors) [Hot96b, Hot96a]. The most significant difference is that the mean is about a quarter of that actually observed, which should not unduly affect results as it only magnifies scheduling overheads. (Recall that in the migratable and malleable preemption cases, we only preempt a job if it has run at least 10 minutes, since preemption requires at least one minute.) All disciplines received exactly the same sequence of jobs in any particular experiment, and in general, experiments required anywhere from 24 to 48 hours to complete.

Minimum processor allocation sizes are uniformly chosen from one to sixteen processors, and maximum sizes are set at sixteen.[9] This distribution is similar to those used in previous studies in this area [PS96a, MZ95, Set95]. 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.

It has been shown previously that performance benefits of knowing speedup information can only be obtained if a large fraction of the total work in the workload has good speedup, and moreover, if larger-sized jobs tend to have better speedup than smaller-sized ones [PS96a]. As such, we let 75% of the jobs have good speedup, where 99.9% of the work is perfectly parallelizable (corresponding to a speedup of 114 on 128 processors). Poor speedup jobs have a speedup of 6.4 on 8 processors and a speedup of 9.3 on 128 processors.[10]

6.2 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 3 and summarized in Figures 3 and 4. As can be seen, the response times for the run-to-completion disciplines are much higher (by up to an order of magnitude) than 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

[7]The makespan is the maximum completion time of the set of jobs under consideration, assuming that the first job arrives at time zero.

[8]The 25%, 50%, and 75% quantiles are 1230, 2985, and 6100 seconds, respectively.

[9]Note that maximum processor allocation information is only useful at lighter loads, since at heavy loads, jobs seldom receive many more processors than their minimum allocation.

[10]Such a two-speedup-class workload appears to be supported by data from the Cornell Theory Center if we examine the amount of CPU time consumed by each job relative to its elapsed time [Par97].

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 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.

effect for run-to-completion and malleable disciplines (see Figure 4). 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 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).

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 multilevel feedback approach achieved 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 the 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 can often experience significant delays 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 5. 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 a small number of processors, which is good from an efficiency standpoint, but can

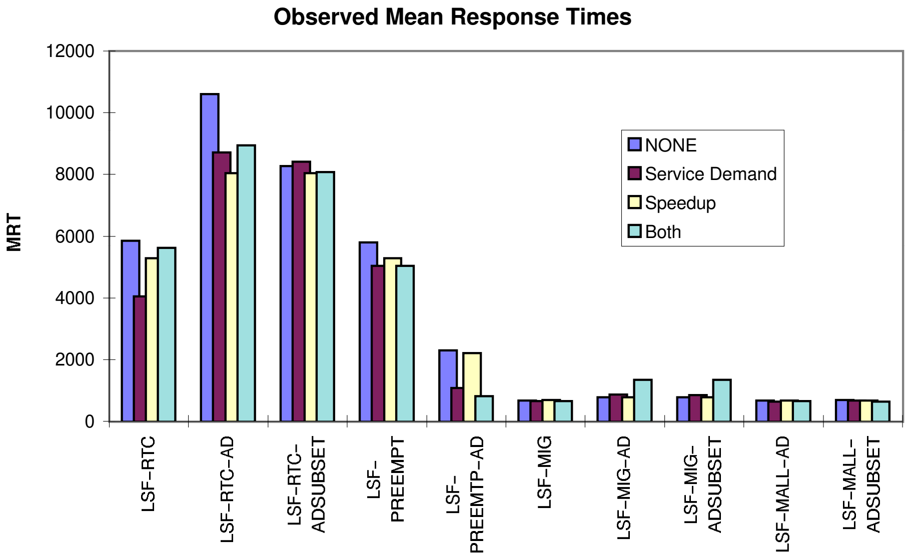

Figure 3: Observed mean response times for each discipline.

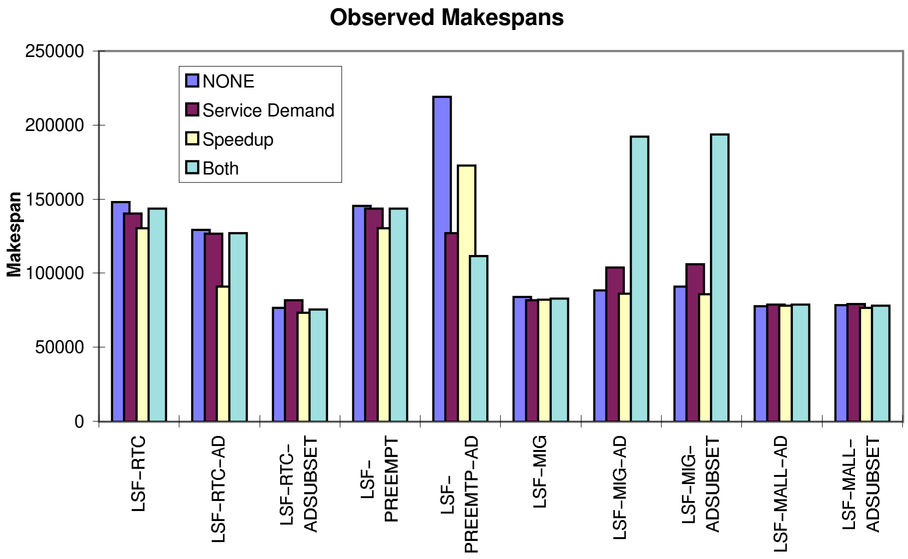

Figure 4: Observed makespans for each discipline.

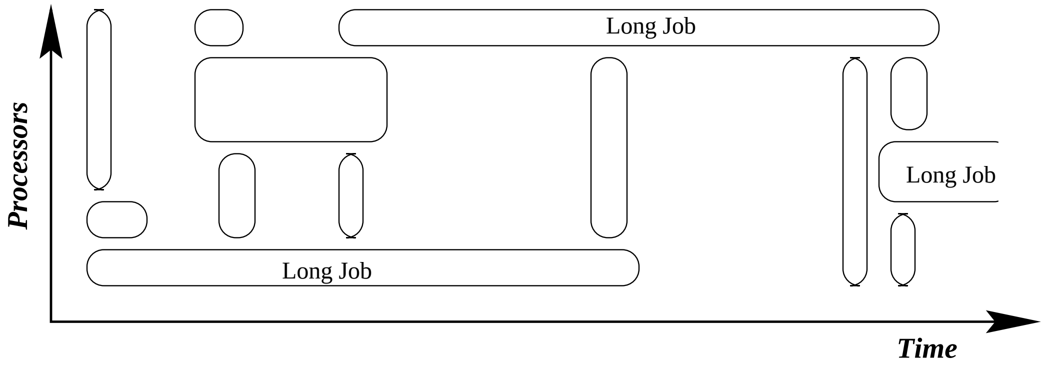

Figure 5: Effects of highly variable service demands on the ability for a run-to-completion scheduler to activate jobs having large minimum processor requirements. Because of the long-running jobs, the system rarely reaches a state where all processors are available, which is necessary to schedule a job having a large minimum processor requirement.

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. It has previously been shown how high variability in service demands can lead to poor 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 long-running jobs, 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. Unused processors arising from imperfect packing are never a problem, allowing a high level of utilization to be achieved. 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 given a 10% re-allocation overhead).

7 Conclusions

In this paper, 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 implemented on 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 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. 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 our adaptive migratable/malleable scheduling module 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 suggests 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. In particular, better decisions may be made by taking into consideration average load over some period of time rather than instantaneous load. Such improvements would be most relevant for simple and migratable preemption, since in this case, the scheduler must commit to a processor allocation for a job when the job is first started.

Acknowledgements

The network of workstations used for this study is part of a cooperative project between the University of Toronto and the Center for Advanced Studies at the IBM Toronto Development Lab. The research in this paper was supported by the Information Technology Research Council of Ontario, the Natural Sciences and Engineering Council of Canada, and Northern Telecom.

References

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

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Par97] Eric W. Parsons. *Using Knowledge of Job Characteristics in Multiprogrammed Multiprocessor Scheduling*. PhD thesis, Department of Computer Science, University of Toronto, 1997.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.