A Time and Space Sharing Scheduler for Multiuser Parallel Machines

by

Geoffrey R. Gustafson
S.B., Computer Science and Engineering (1995)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

© Geoffrey R. Gustafson, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute
publicly paper and electronic copies of this thesis document
in whole or in part, and to grant others the right to do so.

Signature of Author .....
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by .....
Larry Rudolph
Visiting Research Scientist
Thesis Supervisor

Certified by .....
Arvind
Charles W. and Jennifer C. Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996 Eng.

LIBRARIES

A Time and Space Sharing Scheduler for Multiuser Parallel Machines

by

Geoffrey R. Gustafson

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A scheduler can be written to manage parallel jobs on a collection of processors running commercial operating systems. The Jump-StarT scheduler written for MIT's StarT architecture performs this function. The scheduler can accept or decline job requests, based on available resources and system load. The scheduler uses space sharing to map jobs to sets of processing elements, and time sharing to manage multiple threads on individual processors. Thread termination is detected and scheduling is adjusted accordingly. Users interact with jobs through a local proxy process that provides standard I/O stream connection to the parallel jobs, and which can be terminated to force job termination.

Thesis Supervisor: Larry Rudolph
Title: Visiting Research Scientist

Thesis Supervisor: Arvind
Title: Charles W. and Jennifer C. Johnson Professor of Computer Science and Engineering

Table of Contents

List of Figures

List of Tables

Chapter 1

The Jump-StarT Scheduler

A flexible, extendable, and efficient time and space sharing scheduler for managing parallel jobs can be built on top of existing operating systems. The design and implementation of the Jump-StarT scheduler for the StarT-Voyager and StarT-Jr parallel machines supports this thesis. This chapter introduces the responsibilities of the scheduler, outlines the goals that motivate the design of the scheduler, examines related work, and provides an overview of the remainder of the document.

1.1 Managing Parallel Jobs

The scheduler's role in the machine is essentially to manage parallel jobs. This includes receiving job requests, accepting or declining them, arranging for the execution of the accepted jobs, and then controlling when each job runs. The scheduler must recognize when jobs are finished, and provide users with ways to view job status and to force job termination if necessary.

Certain parameters are specified by the user when submitting a job, such as the number of processors required by the job. The scheduler uses these parameters to determine whether or not the new job can be accepted, given the status of the machine and information about the jobs that are already scheduled.

Once a job is accepted, the scheduler is responsible for selecting the processors on which to execute the job. The scheduler selects the processors by considering the load on the system, and the flexibility provided by time and space sharing.

The parallel machine can be shared in both space and time. Space sharing is accomplished by mapping jobs to a subset of processors. It allows small jobs to be spread out

among the available processors. Time sharing allows multiple job threads to run on the same processor, and is accomplished by switching between the threads at regular intervals.

When jobs are running, the scheduler is responsible for managing the time sharing on each processor. It directs the attention of each processor to one of its assigned job threads at a time. The job thread that is currently in control of the processor is the *hot thread*, and the other threads are *cold threads*. The scheduler decides when each thread should become hot, and how long it should remain hot.

The scheduling of job threads can be either coordinated or uncoordinated. Uncoordinated scheduling is inefficient because hot threads are forced to block when they try to communicate with cold threads. Gang scheduling is a coordinated method: all the threads in a job are hot at the same time. Thus, gang scheduling reduces the amount of communication blocking and improves CPU utilization [7].

1.2 Design Objectives

Several goals must be kept in mind in the design and implementation of the scheduler, in addition to accomplishing the purposes outlined above. The primary goal in designing the scheduler is flexibility. This includes making it easy to modify or replace the system's scheduling strategy. The scheduler code can also be made portable between various vendor-specific Unix operating systems. This is particularly useful in that the scheduler can be developed before details of the target operating system are known.

Another important goal is to provide an intuitive interface to the parallel machine. A reasonable model is to make the user interface as much like that of a normal Unix uniprocessor system as possible. The Jump-StarT scheduler provides support for standard input, output, and error streams to support this model. In the StarT systems, normal socket communication and NFS are available to the parallel jobs; no explicit support from the sched-

uler is necessary. To further enhance the familiar interface, each parallel job has an associated proxy process [11] running on the user's local machine. If this job is terminated by the user, the parallel job will be terminated as well.

Secondary goals are efficiency and scalability. To support the goal of efficiency, the parallel machine nodes should spend as little time doing scheduling as possible, so that the resources can be productive for user computation. Although targeted for a 32-node machine, the Jump-StarT scheduler was designed to be scalable beyond that level, given that certain expectations from the operating system could be met.

1.3 Background

Parallel job schedulers have typically been implemented with custom operating systems or with significant extensions to existing operating systems. The novelty of this thesis is in suggesting that a scheduler can be built on top of off-the-shelf operating systems.

Jump-StarT is the job scheduler for MIT's StarT-Voyager and StarT-Jr parallel machines. The StarT-Voyager parallel machine consists of commercial PowerPC 604 microprocessor nodes running AIX, connected by a fast Arctic network [3] for message passing. A special Network Interface Unit supports very low latency for message passing and also supports distributed shared memory: both SCOMA and NUMA. The StarT nodes are also connected to an Ethernet, which can be used for standard TCP/IP communication. StarT-Voyager has grown out of the StarT-NG project [5]. The StarT-Jr parallel machine is based on Pentium microprocessor nodes running Linux, and also uses the Arctic network. Jump-StarT can be compiled for AIX or Linux with no modifications.

1.3.1 System Assumptions

Jump-StarT is designed to run on a parallel machine and a single host machine. The host acts as the user's gateway to the parallel machine. Standard I/O between the user's

machine and the parallel machine is routed through the host, so a connection to the host is maintained throughout the entire execution of a parallel job. The host is a single point of failure for the system, but there is nothing special about the host machine itself. If it fails, it can be replaced by any Unix workstation. This is unlike the Cray T3D, which must have a Cray Y-MP to act as a host.

Jump-StarT assumes that each node of the parallel machine is running its own Unix operating system, so that scheduler modules can run on each node. Each node must have its own IP address, the basis for communication with the host. Each node must also have a unique node ID, between 0 and *n-1* on a *n*-processor machine, that can be communicated to the scheduler modules running on the node.

The Jump-StarT scheduler does not support job migration; each job thread executes to completion (or forced termination) on its assigned processing element. Some schedulers allow job threads to migrate to different processors over the course of their execution. A scheduler for networks of workstations at the University of Wisconsin is an example [13].

The Jump-StarT scheduler treats all processing elements equally: a job's threads may be mapped to any subset of the machine's processors. An accepted job is given its requested number of processor threads and cannot obtain more during the course of execution. However, thread termination need not be coordinated. The scheduler reacts to each thread's termination independently.

The Jump-StarT scheduler does not currently consider resources other than processors in its scheduling decisions. Some schedulers consider usage of resources such as memory to influence the scheduling process [15].

1.3.2 Socket Communication

For the sake of portability, Jump-StarT uses Unix sockets and pipes for all communication. Pipes are unidirectional and can only be used to communicate with another process

on the local machine. Sockets are bidirectional and can communicate across the network. In Unix socket communication [4], both a client and server program allocate a socket, which can be described as an *endpoint* of communication. The server program binds its *rendezvous socket* to a port number that it chooses. When the client connects to the server, the server sees a connection request on its rendezvous socket. If it accepts the connection, the server receives a new *connection socket*. Many connection sockets can be established through the single rendezvous socket.

If the server tries to bind its socket to a port number that is already in use by another program on the same machine, the bind fails. The server is free to try another port number. However, to establish a connection, a client program must know the port number that the server has used. If the client is set to use a particular port number, but the server was unable to bind to that port, the client will be unable to connect. Jump-StarT solves this problem in a robust way.

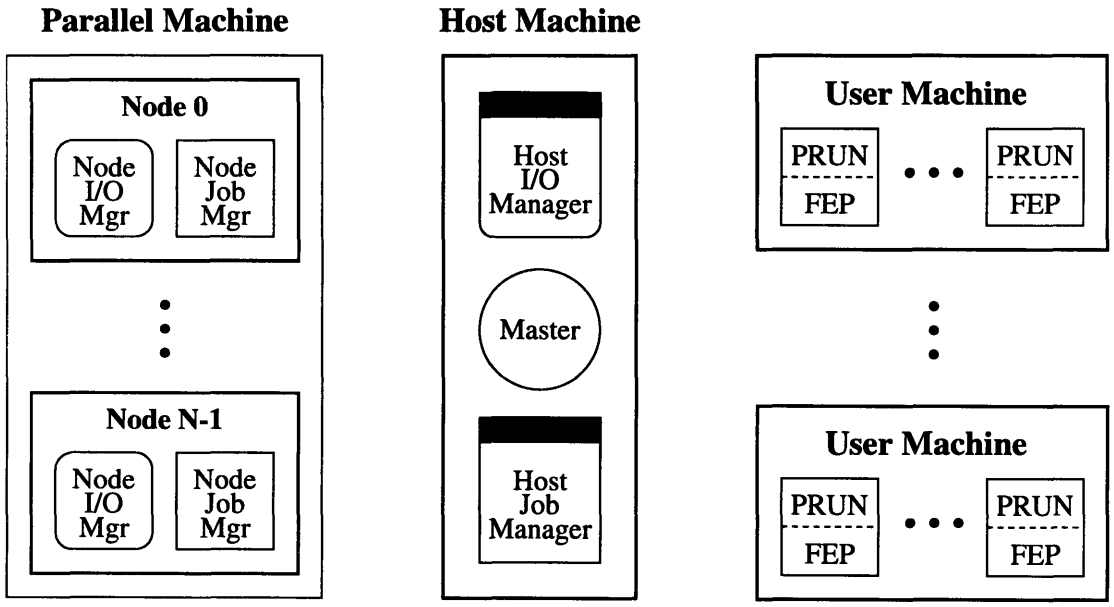

Figure 1.1: *Jump-StarT System.* Each thick box represents a single processor. Each shape within a box is a scheduler module running on the processor. Nodes are labeled with their node IDs. Users may be running other processes (not shown) on their local User Machines. Job threads (not shown) run along with the Node Managers on each node of the parallel machine.

1.4 Document Overview

The Jump-Start scheduler consists of program modules that run on (i) the nodes of the parallel machine, (ii) the single host machine, and (iii) local user workstations. The system is illustrated in Figure 1.1. The Host Job Manager, Node Job Managers, and PRUN program make up the job management system. The Host I/O Manager, Node I/O Managers, and Front End Processes make up the I/O system. The Master process provides global services to the other scheduler modules.

Chapter Two looks at the job management system and the design of the supporting modules. Chapter Three examines the design of the scheduler's I/O system and explains why it is necessary. Chapter Four covers some secondary issues about the scheduler design, such as socket port management. Chapter Five summarizes the results of the project. Appendix A provides an overview of implementation concerns.

Chapter 2

The Job Management System

The most important function of a parallel job scheduler is to create jobs and manage their use of system resources. The part of the scheduler that performs this function is the job management system. Section 2.1 examines various scheduling strategies and derives design constraints for a flexible job management system. Section 2.2 introduces a scheduling strategy based on job classes. Section 2.3 develops the design of Jump-StarT's job management modules from the design constraints. The remaining sections look at each module individually.

2.1 Design Constraints

The design of the job management system is driven primarily by the flexibility goal. Flexibility is needed to cover the range of scheduling strategies that can be implemented. This section considers several different scheduling strategies and their implications for the design of the system. A good survey of the spectrum of schedulers has been produced by Feitelson [8].

An extremely simple scheduling strategy can use space sharing without time sharing: only one job thread is assigned to a processor at a time. Such a system maintains a list of the free processors. Upon the arrival of a job request, the system simply checks the number of free processors. If there are enough, it assigns some processors to the job and removes them from the free list. Otherwise, the job is declined. Upon termination of a processor's job thread, the system returns the processor to the free list. The Loadleveler scheduler for the IBM SP2 uses simple space sharing. The EASY scheduler for the SP2 also uses simple space sharing, with backfilling [12].

A simple space sharing strategy can be implemented with a single centralized job manager. Scheduler software at each node only needs to execute job threads and inform the job manager when the current thread terminates.

The functionality of the system can be improved by allowing a job to be queued, if sufficient resources are not available for it to run when it is submitted. Such a system will still use purely centralized control. However, there are major disadvantages to having only one job thread running on a processor at a time. For example, the system can only efficiently support batch jobs. An interactive job running on such a system would have exclusive control of the processors to which it was assigned, but use only a small fraction of their computational power, thus wasting resources.

With time sharing, greater resource utilization can be achieved. Multiple interactive jobs can run for short, alternating intervals, providing each with the short response time and modest computational resources needed. Batch jobs can be mixed with interactive jobs, perhaps by alternating periods of good response time for the interactive jobs with periods of poor response time when the batch jobs would be given large time slices.

After adding this complexity to the system, it becomes desirable to let the nodes themselves take charge of scheduling their own job threads, so that the central job manager does not become a bottleneck. Local job managers at each node can assist in the scheduling process. For instance, job threads can be executed in an uncoordinated manner by local job managers, thus requiring no communication among the local job managers and little communication with the central job manager. But uncoordinated scheduling leads to inefficiencies when job threads try to communicate [10].

Efficiency can be improved by using a coordinated strategy such as gang scheduling. Gang scheduling [7, 10, 14] relies on the premise that on a time sharing parallel machine, there are sets of threads which communicate much more frequently with each other than

with threads outside the set. Thus it improves efficiency to ensure that every thread in such a set (or gang) is running at the same time on the various processors, so that communication delays are minimal. These sets can be determined by dynamic monitoring of communication [9], but a simple approximation, without the overhead of low-level monitoring, is to assume that the threads of a single job behave as such a set. Gang scheduling has been implemented for SGI IRIX systems [2] and the Tera MTA [1].

A coordinated scheduling strategy can either be fully distributed among local job managers, eliminating the central job manager, or use a hybrid of centralized and distributed techniques. A fully distributed method for implementing gang scheduling is Distributed Hierarchical Control [6], where the nodes are organized into a binary tree. Each node has its local job manager, but a node with children also has job management responsibility for the subtree rooted at the node. A job is assigned to a particular subtree big enough to run the job, based on system load across comparable subtrees. The node responsible for that subtree ensures that the threads of the job execute together by communicating with its child nodes.

Distributed Hierarchical Control completely removes the scheduling burden from the central job manager and distributes it across the nodes. It also makes use of communication between the nodes. This is advantageous on the StarT machines because the fast message passing layer can be used for the communication, although this makes the code less portable to other systems. The binary tree structure is useful for any strategy that needs to broadcast the same data to many nodes.

Another method of supporting gang scheduling returns more control to the central job manager, combining the use of centralized and distributed control. With this method, the central job manager would not simply give local job managers a thread to execute, but give them a scheduling pattern to follow. Assuming clock synchronization between the

nodes, by coordinating a starting time for the pattern, thread gangs could be scheduled together as desired. This does have the problem of making the central job manager more of a bottleneck, and thus reducing scalability. However, since the central job manager can reside on the host machine, this method reduces the computation done on the parallel nodes, leaving more CPU cycles for user job threads. Since the scheduling pattern given to each local job manager is different, there is no need to broadcast data, and therefore inter-node communication through a binary tree is unnecessary.

The Jump-StarT system is designed to allow a wide range of strategies to be explored. The extra communication paths necessary for Distributed Hierarchical Control have not been added, but could be without significant effort. The next section covers a particular way of designing a scheduling strategy. In Section 2.3, the design of the Jump-Start job management system will be explained. The various software modules implied by the above discussion will be given names and their roles and interaction will be clarified.

2.2 Job Class Scheduling Strategy

An interesting way to address the scheduling problem is to classify jobs based on significant characteristics like interactivity and priority. Each job is assigned to a particular class and the scheduler treats jobs of each class differently. This section examines the questions involved in scheduling based on job class, and offers Jump-StarT's answers.

2.2.1 Creating Job Classes

First, there must be a set of job classes to choose from. As an example, consider a set of two classes: batch and interactive jobs. The batch jobs require long time slices but can be delayed until off-peak hours to get most of their service. The interactive jobs are optimized for response time and must be given frequent, short time slices. These are important distinctions that the scheduler can use to affect the time and space sharing of the machine.

Jump-StarT allows job classes to be defined and configured by a system administrator. The information specified for each job class includes priority level and the required frequency, length, and regularity of time slices. The system administrator also decides how to divide the system up among the different job classes. For example, batch jobs could be given 60% of the machine and interactive jobs, 40%. This guarantees that under heavy load, the interactive jobs as a whole will be given 40% of the CPU time. If a particular class does not need its entire share of time, the remainder will be divided among the other job classes according to the same ratios.

2.2.2 Assigning Job Classes

Another issue surrounding job classes concerns the point in time at which a particular job will be assigned to a class. The user could select the job class at submission time. The scheduler could determine the job's class by monitoring its initial resource usage. Or the job class could even be dynamic, changing as the job's resource usage changes over time. Since Jump-StarT is to be portable it is simplest to allow the user to select the job class. If priority based classes are used, this might require limiting the classes available to particular users.

2.2.3 Job Parameters

In addition to the information associated with a job's class are other job-specific parameters used by the scheduler. Deciding which parameters should be part of the job class and which should be specified separately for each job is not a simple issue. This should be explored further when data about real jobs can be gathered.

Among the job-specific parameters are the required number of processors, total CPU time required per processor, and a deadline for the job. These and other parameters will be discussed in Section 2.4, where Jump-StarT's job submission program is explained.

2.2.4 Determining Acceptance and Assigning Nodes

The decision of whether to accept or decline a requested job should be based on knowledge of system load. If all the processors are to be treated equally in a pool of computational resources, then the processor nodes should be considered in order of increasing load expected during the time that the job will run. A practical way to enact this rule is to compute the amount of free processing time each node expects to have before the deadline of the requested job. If there are enough nodes with sufficient free processing time, the job should be accepted. Otherwise, it must be declined, perhaps with a suggestion of what the deadline would need to be for the job to be accepted.

The job should be assigned to particular nodes in the same way. Again, since each node is treated equally in the processor pool, the least loaded processors can be used. With the job class model, there are also issues to consider about compatibility between jobs. The job class ratios present the challenge or whether the ratios should be fulfilled primarily through time or space sharing.

Batch jobs need long time slices, which limits the response time of interactive jobs being time shared on the same processors. Interactive jobs can run more successfully with other interactive jobs on the same processors than with batch jobs. This indicates that space sharing between job classes is important. On the other hand, when a 32-node machine is available, users should be able to have 32 threads in a batch job and not just 60% of 32. There needs to be time sharing between job classes, but when possible it will be avoided.

2.2.5 Managing Running Jobs

A portable job scheduler running on existing Unix systems has two ways to control jobs: by stopping and restarting the job threads as their time slices end and begin, or by using the priority scheme of the underlying Unix scheduler. Each technique requires only the

knowledge of each of the job thread PIDs and permission to make the right system calls. This permission is available to the scheduler because it runs under root privileges.

Stopping and starting the jobs has the advantage of yielding better control of time slices to the parallel scheduler. This is done by sending SIGSTOP and SIGCONT signals to each job thread. Using the priority levels has the advantage that Unix will allow other processes to run while the hot thread is blocked. This is accomplished with the `setpriority()` system call.

2.3 Job Control System Design

This section develops the design of the Jump-StarT scheduler's job control modules. A central job manager is the module which handles job requests, possibly communicating with the other managers to decide whether to decline or accept a certain job request. The local job managers are responsible for maintaining the time sharing of the local processor between threads. Depending on the scheduling strategy, the local job managers might merely obey a scheduling pattern set up by the central job manager, or take part in making the scheduling decisions themselves.

2.3.1 Job Thread Identification

A parallel machine has *n* processor nodes. From the point of view of the scheduler, each node has an identification number, called the *node ID*, ranging from 0 to *n-1*. This node ID is fixed and known by each node before the scheduler even starts up, as part of the node's configuration. This is necessary so that nodes can be distinguished by the *Master process*, introduced in Section 4.1.

Each parallel job has a unique *parallel job ID*, or PJID. The scheduler assigns one to each job when it is accepted, starting from PJID 1. The threads within a job all have the same PJID but are distinguished by their *rank*. The rank of each thread is also assigned by

the scheduler, and ranges from 0 to *n-1* for an *n*-thread job. Thus the PJID and rank uniquely specify a particular job thread on the machine. Each thread on a given node has a different PJID, and each thread in a given job has a different rank. Each thread also has a local Unix *process ID*, or PID, as on any Unix system. These PIDs are unique on a given node, but not across nodes.

2.3.2 Job Submission Interface

The most visible element of the job management system is its interface with users, especially for job submissions. The required parameters for a job request depend on the underlying scheduling strategy. Some possible parameters are the number of processors a job will need, its job class or priority, the amount of memory and swap space required for each thread, the amount of CPU time it expects to use on each processor, and a deadline by which the job should be completed.

Of all these parameters, only the number of processors is a necessity in Jump-StarT. Each accepted job is allocated its requested number of nodes, and a thread is executed on each node. Although some threads may terminate before the job completes, once begun a job is not allowed to create a new thread.

The amount of memory and swap space are not used in the current scheduling strategy. An improved strategy might use them to try to minimize cache interference and swapping through context switches. For example, a job using a lot of memory might be given longer time slices at longer intervals, to keep the job from spending all its time swapping in its data.

Specifying the amount of CPU time expected and a deadline allow the scheduler to accept or decline jobs based on load. If the scheduler accepts a job, it is indicating that it expects to be able to give the job the desired amount of CPU time before the deadline. Once the job is accepted, the scheduler will not accept other jobs that would keep the orig-

inal job from finishing on time. On the other hand, if the job is declined, the user needs to specify a later deadline or find another machine.

One problem with the simplicity of this behavior is that it treats all users equally. If Bill Gates walks in with a job that needs to run today, he might have to wait while an intern's Parallel Tetris game, already scheduled, has the machine fully loaded. There is also an issue when a job uses more than its requested CPU time. Different strategies might either continue to schedule the job or terminate it.

When a job request is submitted, it must include the filesystem pathname of the job's object file, and possibly command line arguments. All this job request information could be provided in several ways; for example, through command line parameters or by filling out a form in an X application. For simplicity, Jump-StarT uses a command line program, realizing that a fancier interface could always be provided as a front end. The command line program is called PRUN, for parallel run.

2.3.3 Job Control Modules

The PRUN program must submit its job request to a server to find out whether the job is accepted or declined. This server is the central job manager. The scheduler is designed to run in an environment where there is a host machine separate from the parallel machine. Thus it is useful to put the central job manager on the host machine, where its computation time does not interfere with parallel jobs. We refer to the central job manager as the *Host Job Manager*, and the local job managers as *Node Job Managers*.

Figure 2.1 illustrates the arrangement of the significant modules and connections in the job management system. In this example, one user executed a four processor job. Then one user executed a two processor job. A third user executed a two processor job followed by a one-processor job. Table 2.1 contains the Host Job Manager's processor allocation

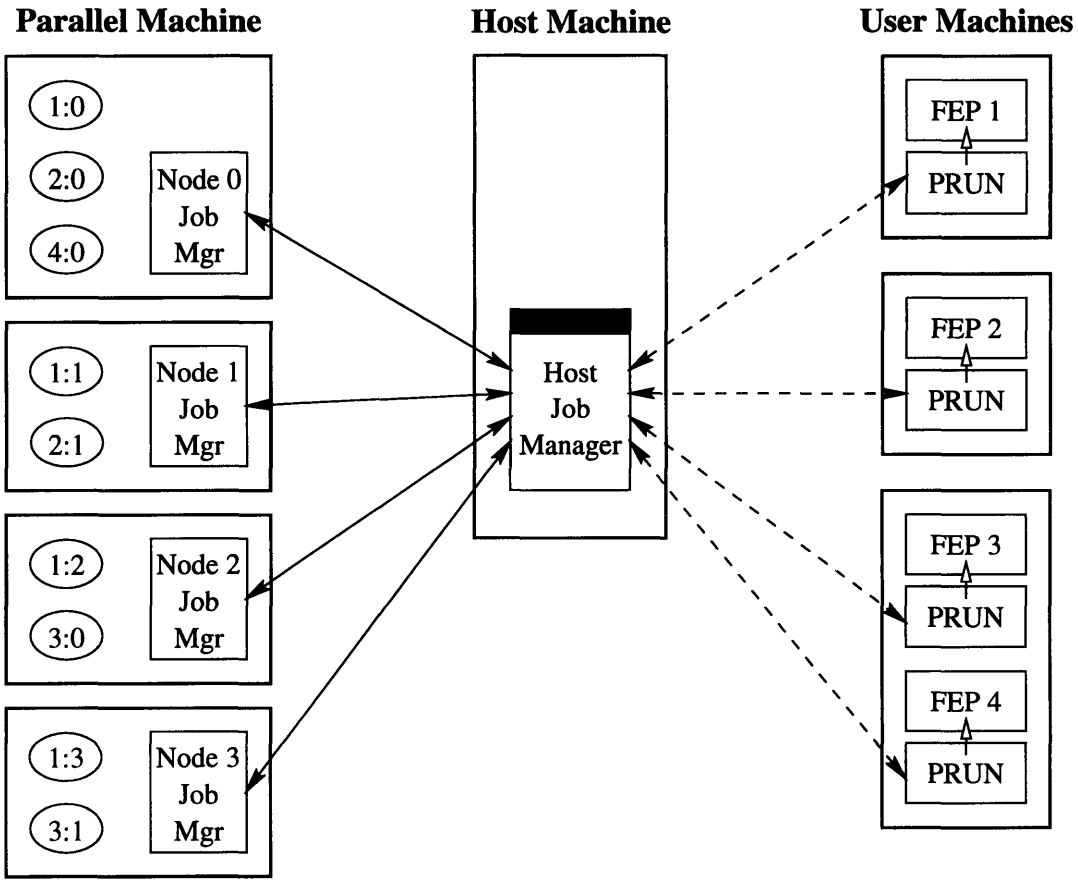

Figure 2.1: *Job Management System.* Solid lines with solid arrows denote permanent socket connections. Dashed lines denote temporary socket connections. Lines with hollow arrows denote transfer of execution. Thick boxes represent single processors. Processes running on the machines are denoted by labeled shapes inside the boxes. The ovals are parallel job threads, labeled with thread PJID and rank. FEPs are part of the I/O system discussed in Chapter Three, and are numbered with the PJID of the job they are associated with. Note that Node Job Managers maintain no connections to the job threads.

data for this example. Table 2.2 shows the data that Node Job Managers must maintain, which associates each parallel job thread with the actual Unix PID it is running under.

2.3.4 Scheduling Strategy Interface

To make it easy to modify the system's scheduling strategy, the strategy code is separated from the main control code by a modular interface. This interface is a set of strategy functions that must be written to implement a particular strategy. The strategy functions are event-driven: the Job Managers only call the strategy functions when certain events occur.

PJID	Node 0	Node 1	Node 2	Node 3
0	X	X	X	X
1	X	X		
2			X	X
3	X			

Table 2.1: *Processor Allocation.* These are the data that the Host Job Manager maintains for the example shown in Figure 2.1. Each parallel job is associated with the nodes on which its threads are running. This data reacts dynamically to the state of the job threads. If the thread on Node 2 for the job with PJID 0 terminates, the X will be removed from the corresponding cell.

Node 0 Associations			Node 2 Associations		
PJID	Rank	Unix PID	PJID	Rank	Unix PID
1	0	783	1	2	638
2	0	787	3	0	645
3	0	794			

Table 2.2: *Thread/Process Associations.* These are the Node Job Manager's associations between the logical PJID and thread rank of a parallel job thread, and the actual Unix PID of each thread on the local processor. Data shown corresponds to two of the nodes from the example in Figure 2.1. The specific PID values shown are unimportant but indicate the order in which the jobs were executed, because a typical Unix system assigns PIDs sequentially.

One strategy function must be invoked when a job request is received, which provides the strategy with the job request parameters. The strategy decides whether or not to accept the job by consulting its data structures, which it updates each time a strategy function is invoked.

When a job is accepted, another strategy function is invoked to assign the job to specific processing nodes. The strategy function returns a list of the nodes, after updating strategy data structures.

Other strategy functions are used to update the schedule. The schedule needs to be updated each time a job thread terminates, so that the strategy can take advantage of the decrease in load. A strategy function is provided to do this. Some strategies need to be updated at fixed time intervals, because they rely on aging data like the CPU time used by each job thread. Another strategy function is provided to update the schedule based on elapsed time.

Currently, the scheduler's strategy interface is not general enough; the interface as it stands is presented in Section 2.5. One lack of generality is that the strategy does not decide when the schedule will be updated; instead, the updates occur at preset intervals. The strategy functions should be able to set a timer that will cause the schedule to be updated at specified intervals.

A more general interface would support communication between the strategy code at the host and the nodes. This would allow strategy-specific data to be shared instead of only the data that the Job Managers always exchange. This communication can be provided in an event-driven manner. A function could be provided to send a strategy message from inside the strategy functions. This function would use of the underlying Job Manager socket connections. When a strategy message arrives at a Job Manager, the manager would call another strategy function to receive the message. Separate host and node versions of the functions would be provided.

At the node level, a strategy function should be provided to update data after thread termination. Another function would operate on a timer to control context switching. In this function, the strategy would communicate with the hardware to register a new thread as hot and the old thread as cold.

Part of the difficulty in keeping the scheduler modular is that the very parameters accepted for the job by PRUN are strategy-specific, as indicated earlier. So not only the

Host and Node Job Managers, but even the PRUN program must often be altered to change the strategy. To ameliorate this problem, strategy-specific job parameters, such as the job class, could be separated from the standard ones that the Host Job Manager always requires, such as a filesystem pathname for the job's executable object file.

2.4 PRUN Program Design

The PRUN program can be executed from any machine with permission to run parallel jobs. That is to say, when the PRUN program connects to the Host Job Manager, the IP address of the machine from which PRUN connected is known. The address is validated with a configuration file that specifies the machines authorized to initiate jobs.

The PRUN program takes a host of command line arguments for the various job parameters, shown in Table 2.3. After all the switches and their arguments, a filesystem path to an executable object file must be specified. This path must either be absolute, or relative to the current directory. That is, the PATH environment variable will not be searched to find the program. After the executable path there can follow command line arguments that will be given to the program. As always, wildcard characters are expanded

Switch	Argument	Meaning
-p	*n*	*n* is the number of processors the job requires
-m	*n*	*n* is the number of MB of RAM required per node
-s	*n*	*n* is the number of MB of swap space required per node
-t	*h*:*m* or *m*	*h* hours and *m* minutes of CPU time required per node
-d	*h*:*m* or *m*	job deadline is *h* hours and *m* minutes from now
-c	*s*	*s* is a string denoting the job class

Table 2.3: *PRUN Command Line Switches*. This table shows the command line switches which specify job parameters in the PRUN program. The argument letters *h*, *m*, and *n* denote integer arguments, and the letter s denotes a string argument. The megabyte values specified with the -m and -s switches are not used in the current scheduling strategy.

by the shell before the PRUN command gets them. Thus, they will be resolved with respect to the local file system, and not the parallel machine's file system.

Once the PRUN program gets all these parameters, it checks them in certain ways. For instance, if more processors are requested than exist in the system (a constant which is now compiled into the scheduler), PRUN will report the error. Assuming the parameters make sense and are complete, PRUN connects to the Host Job Manager and submits the job request. If the job is declined, it informs the user and exits. If the job is accepted, the PRUN program receives the PJID of the new parallel job from the Host Job Manager, and calls the `exec()` function to execute the *Front End Process* (see Chapter Three), passing the PJID as a parameter. Thus, the standard input, output, and error of the Front End Process are those of the replaced PRUN process.

2.5 Host Job Manager Design

There is a single Host Job Manager in the scheduler which runs on the host machine. It maintains several server sockets, for (i) PRUN connections, (ii) Node Job Manager connections, and (iii) a client connection to the *Host I/O Manager* (see Chapter Three). Upon initialization, the Host Job Manager considers the parallel machine to have no active processors. Thus if a PRUN request comes in, which of course would ask for at least one processor, it will decline the request. The Host Job Manager considers a node to be active only once a connection is established with its Node Job Manager.

Figure 2.2 shows the basic interface between the Host Job Manager and the scheduling strategy compiled into the Host Job Manager. The Host Job Manager calls upon the scheduling strategy is to decide whether or not each job will be accepted. If the strategy says to accept the job, the Host Job Manager assigns an a identifier to the job, its parallel job ID,

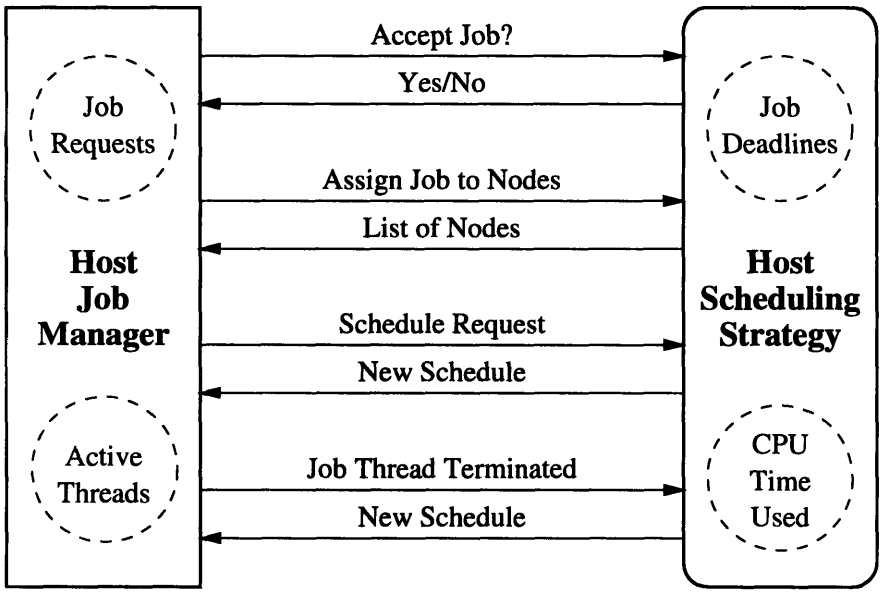

Figure 2.2: *Host Strategy Interface.* The Host Scheduling Strategy is a set of functions in the Host Job Manager which are called upon certain events. The arrows show the request and reply pairs that make up the interface. The Host Job Manager handles job requests and maintains data on the active threads in the system. The Host Scheduling Strategy tracks the job deadlines and CPU time used by each thread.

or PJID, and sends it to the PRUN process. Otherwise, the Host Job Manager informs the PRUN process that the job is declined.

Next, the Host Job Manager invokes another strategy function that decides which nodes will execute the threads of the new job. It receives a list of selected nodes, and sends job start-up information to the Node Job Manager on each selected node.

According to some strategy-specific rule, the Host Job Manager will at certain intervals, or based on certain messages from Node Job Managers, call a strategy function that allows the strategy to adjust the schedule. When a job thread terminates, the Host Job Manager is informed by a Node Job Manager, and calls another strategy function to adjust the schedule.

2.6 Node Job Manager Design

The Node Job Manager is present on every node of the parallel machine. It maintains one

client connection to the local Node I/O Manager and one to the Host Job Manager. Regardless of the scheduling strategy, the Node Job Manager's basic purpose is to manage the activity of job threads running on its node.

When the Node Job Manager starts up, it makes a connection to the Host Job Manager, and sends the local node ID. Then it waits to receive new job threads from the Host Job Manager. It also creates a connection to the local *Node I/O Manager*, which will be introduced in the following chapter.

For various technical reasons, the Node Job Manager does not directly create job threads, nor directly detect their termination. These functions are carried out by the Node I/O Manager. The Node Job Manager only interacts with the Node I/O Manager at the time of job thread creation and termination. The functions of the two Managers are distinct all throughout job thread execution, so they are separated into two modules.

The Node Job Manager is responsible for controlling each thread on the node. It maintains a correspondence between the parallel job ID (PJID) and the local Unix process ID (PID). The Node Job Manager may send signals to the process, set its priority in the local Unix scheduler, or do whatever is demanded by the scheduling strategy.

When a job thread terminates, the Node I/O Manager notifies the Node Job Manager. At this point, the Node Job Manager calls a scheduling strategy function to revise the schedule. It also passes on the PJID and rank of the job thread that terminated to the Host Job Manager, so that changes can be made to the global schedule.

Chapter 3

The I/O System

One of the main objectives of the scheduler is to provide an intuitive interface to the parallel machine, both for users and programmers. A good way to do this is to make the parallel computer seem as much like a normal Unix uniprocessor system as possible. One aspect of this is to ensure that I/O can be performed in traditional ways.

The purpose of any program is to produce output of some kind and communicate it in some way: a program that communicates nothing could just as well be left unexecuted. Many programs also accept input of some kind. This makes I/O services a significant concern of a parallel computer system. Section 3.1 shows why providing traditional I/O services is a burden of the scheduler. Section 3.2 develops the design of scheduler modules to support these I/O services. The remaining sections detail the behavior of the individual modules needed for this support.

3.1 Providing Standard I/O

Output from a program can be communicated in many ways, and input can come from many sources. Some examples of output destinations are tty terminals, X windows, files, pipes, and sockets. Similarly, some input sources are keyboards, mice, files, pipes, and sockets. To use most of these sources and destinations, a program must set up connections using a certain set of system calls.

However, Unix provides each process with basic input and output through the standard input, output, and error streams. Unix process management makes these streams available to each process automatically. Thus, these are a uniquely supported form of I/O and the scheduler should extend the support to the parallel realm if possible.

To see how this will be done, first consider a sequential program's standard input stream. By default, the input comes from the keyboard through the shell in which the program was executed. Alternatively, the user can redirect the standard input to a file source, from which the program reads data as if it were being typed at the keyboard. The input goes from a single source (keyboard or file) to a single destination.

However, a parallel job has a number of threads running on different processor nodes. User input to such a job has a different meaning then it did in the single processor case. Here are three ways to deal with multiple destinations for the standard input. A user's input could be sent to all threads of a job. This would mean that each thread would see the exact same standard input stream over the course of execution. Another solution is that an arbitrary thread could be chosen as the input thread, through which all input would be exclusively directed. It would be the task of this thread to distribute the data as necessary. Finally, the scheduler could provide some way for the user to specify which thread certain input was intended for, allowing each thread to receive an independent input stream.

Now consider the opposite problem of mixing multiple threads with a single standard output stream. Once again there are three obvious choices. Analogous to broadcasting the input is to expect identical output streams. That is, each thread must produce the same output in the same order. This seems wasteful at best, because the output from all but one thread could be ignored. A second option is for one job thread to be dedicated as the output thread, and be the only one which could successfully use its standard output and standard error. Although this may be fine for standard output, it is awkward for standard error. Finally, every thread could be allowed to use its output streams if the output could be merged into one stream from the user's perspective.

Jump-StarT attempts to support all alternatives. Each thread is given independent standard input, output, and error streams. To determine which thread input is intended for, and

merge output from various threads, the input and output are tagged with a rank number unique to each thread in the job. An *n*-processor job has threads of rank 0 through *n-1*. A simple I/O filter can then provide the desired options. One example would be a filter that creates a window for each thread and removes the output tags, displaying the output from each thread in its own window. Input from any window could be invisibly tagged with the correct rank number associated with that window.

3.2 I/O Module Design Overview

The discussion of the I/O system behavior in the previous section led to the decision that each job thread would have independent standard input, output, and error streams. The user interacts with the parallel machine through the host, and expects input to the parallel job to come from a local keyboard, and output to go to a local screen. So something must be done to connect the I/O streams of each job thread to the local machine.

First, there must be some process running on the local machine, with its own standard input, output, and error. Since the user must run a program to submit a job request in the first place, that program can become the necessary I/O handling process, after a job has been accepted. The I/O handling process is known as the *Front End Process* (FEP). The FEP functions as a proxy for the parallel job: if it is terminated by the user, the parallel job should be terminated as well [11].

Since each job thread is an actual Unix process running under the Unix operating system of its node, it already has standard input, output, and error. The simplest way to connect the Front End Process to the I/O streams of the job threads might be to make socket connections for standard input, output, and error between the FEP and each thread. However, for a 32-node job, this would mean 96 socket connections to the FEP. Since file

descriptors are a limited resource to a Unix process (often only 64 are available), this is not a very scalable design.

Instead, the three streams can be multiplexed together on one socket connection, where each string being sent is tagged as either input, output, or error. Since this requires filtering the I/O streams before they leave the parallel node, there must be a scheduler module present on each node to perform this function. This module is known as the *Node I/O Manager*. Input and output from all threads on a node are filtered through the local Node I/O Manager.

Perhaps the obvious next step would be to connect an FEP to each Node I/O Manager on which its job has a thread running. This would mean each Node I/O Manager would have one socket connection for each thread on the node, and each FEP would have one socket connection for each thread in its job. This complexity in the FEP and Node I/O Manager can be removed by adding a central I/O manager, through which all communication flows: a sort of router for the standard input, output, and error of all jobs.

Jump-StarT uses a central I/O manager, which runs on the host machine. This *Host I/O Manager* maintains one connection to each Node I/O Manager, and one connection to each FEP. It routes input from an FEP to the correct Node I/O Manager, and output from a Node I/O Manager to the correct FEP. If the FEP terminates prematurely, this is observed by the Host I/O Manager, which receives an EOF on its connection to the FEP. Thus, this information is immediately available in a centralized place, which can orchestrate the termination of all the job threads. Figure 3.1 shows the modules and connections of importance to the I/O system.

In retrospect, there may not have been enough justification for adding the Host I/O Manager. The total number of socket connections between the Node I/O Manager layer and the FEP layer is greatly reduced, but all the remaining ones radiate from the single

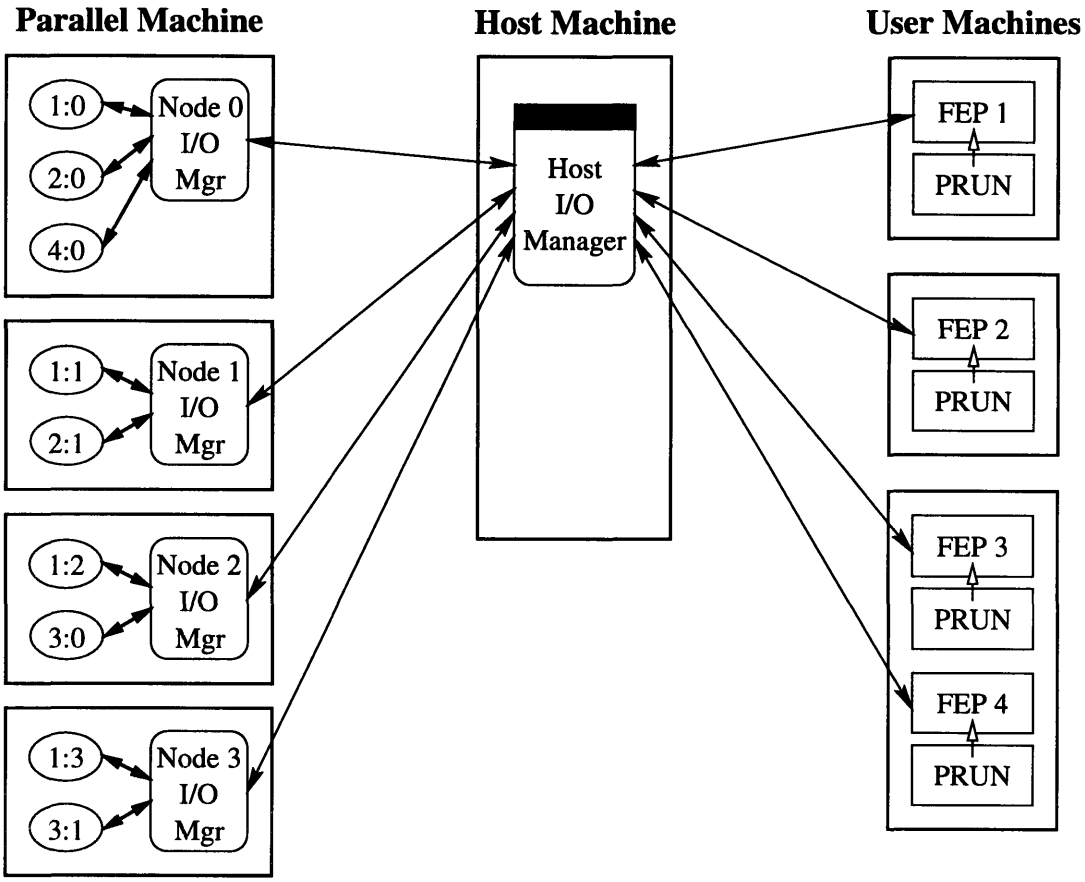

Figure 3.1: *I/O System.* Thin lines with solid arrows represent socket connections. Thick lines with solid arrows represent three one-way pipe connections. Lines with hollow arrows represent transfer of execution. Thick boxes represent single processors. Significant processes running on the machines are denoted by labeled shapes inside the boxes. The ovals are parallel job threads, labeled with thread PJID and rank.

Host I/O Manager. Thus the Host I/O Manager is a bottleneck. Since I/O is expected to be infrequent, it may not be a computational bottleneck, but again the file descriptor limit may become a barrier to scalability. If the host machine runs a Unix that allows many file descriptors per process, this would not be a problem. The AIX machine used for most of the development work for this thesis, for instance, allowed 2000 descriptors per process. Moreover, a Unix system can be reconfigured to support more descriptors. Thus, our choice should work fine.

3.3 Front End Process

The Front End Process is provided as the user's basic tty interface to a parallel job, and as a proxy for the job. Keyboard input to the FEP is interpreted as standard input to the parallel job. The standard output and standard error of the parallel job are displayed to the user through the corresponding streams of the FEP. If the user terminates the FEP, the parallel job will be terminated. When the parallel job completes execution normally, the FEP exits as well.

3.3.1 User I/O Interface

The Front End Process is executed as a result of a successful job request. The user submits a job request with the PRUN program, and if the job is accepted by the scheduler, the PRUN process executes the Front End Process. The FEP maintains a single socket connection, as a client of the Host I/O Manager. All communication about signals and standard input, output, and error, occurs through this connection.

If any of the parallel job threads send output through standard output or standard error, the output is routed through the system to the FEP. The type (normal or error output) is specified as well as the rank of the thread it came from. Each thread in a job has a unique rank: a number from 0 to *n-1* in a job with *n* threads. After receiving output, the FEP displays on the proper stream (standard output or standard error) the rank, followed by a colon, a newline character, and the output text. Thus output from different threads can be distinguished by the user. Figure 3.2 shows an example of FEP output.

This output strategy was designed with the idea that FEP output could be piped to an output filter program that would display the output in any way the user would prefer. For instance, such a filter could open a window for each thread's output and separate them itself without displaying the rank tags.

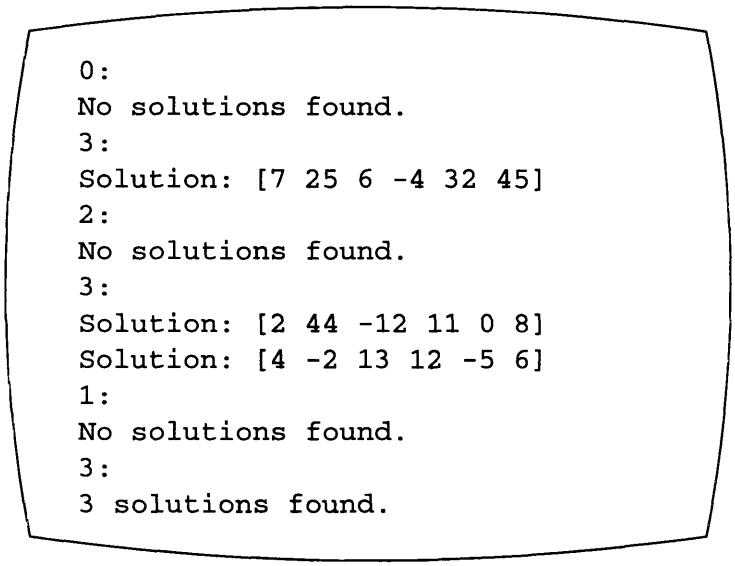

Figure 3.2: *Sample FEP Output.* This is an example of FEP output from a fictional program which finds row vector solutions to some matrix problem. The solution space is divided up between four threads, with ranks 0 to 3. Each thread reports any solutions it finds along the way and then reports the number of solutions found. All three solutions are found by the thread of rank 3. When output is received from the same thread, the rank tag is not specified again, as shown in the second set of output produced by thread 3.

Similarly, the user can give keyboard input through the FEP. The user should type the rank of the thread that the input should be sent to, followed by a colon, and the desired input string. The input will be routed to the proper thread and placed in its standard input stream.

In the current implementation, the input will be sent when the user hits Enter, and any further lines of input will need to be tagged again with a rank. This means the C library functions like `gets()` and `scanf()` will work as normal in a parallel job, but character-based functions like `getchar()` will not. When the user types '`a`', that character will not reach the parallel job thread until the user finishes the line of text.

Finer granularity of input presents problems, but could be provided if necessary. The system could pass along every character as it is typed. This would require a new way of specifying which thread input is intended for. Perhaps a special keypress would let the

user change the target thread. For instance, pressing Ctrl-A, 2, Enter might reset the input destination to the thread of rank two. From then on, each character typed would be sent to thread two until another Ctrl-A was reached. However, with this fine granularity, a substantial number of messages would be generated within the scheduler. If many users were running interactive jobs like this at once, it could severely impact performance.

3.3.2 Manual Job Termination

The FEP acts as a proxy for the actual parallel job. When the last job thread terminates, the FEP is notified and execution ends, returning the user to the shell prompt like any command-line program. Similarly, if the FEP is terminated, the actual parallel job will be terminated as well.

In the current implementation, there are three ways that the FEP's termination can lead to the termination of the parallel jobs. The first case is that the user gives the FEP the interrupt keystroke (usually Ctrl-C). This causes a SIGINT signal to be sent to the FEP. When the FEP receives that signal, it will exit and the system will send a SIGINT to each of the parallel job threads, so they can handle the signal as desired. If the parallel job threads choose to ignore the signal, they will be killed (with SIGKILL) after a certain time, since the FEP is no longer connected.

In the third case, the FEP might receive another signal that causes termination, such as SIGTERM or SIGHUP, usually because of a user kill command. If so, the FEP will exit, and the system will send a SIGTERM to each parallel job thread, regardless of the actual signal received. This is done since most terminating signals are not appropriate to send on to the parallel job, because they denote that something went wrong with the execution of the FEP, not the parallel job. If it is decided in the future that any other signals should be passed on to the parallel threads, they would need to be handled in the same way as SIG-

INT. Again, if the parallel threads choose to ignore the signal, they will eventually be killed because the FEP has terminated.

Finally, the FEP might terminate without warning. This would happen if the FEP receives a SIGKILL signal, which cannot be caught, or an unknown signal that is not handled. In this case, the Host I/O Manager will see that its socket connection is closed to the FEP. Since it did not receive a termination message from the FEP, it will assume the worst and instruct the system to send SIGKILL to each parallel job thread.

3.4 Host I/O Manager

The Host I/O Manager runs on the host machine. All input comes through it to be routed to the correct node. All output comes through it to be routed to the correct FEP. The Host I/O Manager maintains three server sockets. One is for FEP connections, one is for Node I/O Manager connections, and the last is for a single connection to the Host Job Manager.

When the Host I/O Manager starts up, it is merely listening for connection requests on its three server sockets. The Host Job Manager and each Node I/O Manager will connect when they start up. It expects the Host I/O Manager and at least one Node I/O Manager to be connected before any FEP will try to connect, because otherwise, no job should have been accepted in the first place.

When a new parallel job is accepted, the first thing that happens to the Host I/O Manager is that a new FEP makes a connection request. Once the connection is accepted, the FEP sends the PJID it has been given by the Host Job Manager. The Host I/O Manager associates the PJID with the connection to the FEP in its data structures, and then reports the PJID to the Host Job Manager to confirm that such a job has been accepted. If the Host Job Manager can confirm the job, it will begin job execution; otherwise, it will report the

error to the Host I/O Manager, which will then remove the FEP data it recorded, and disconnect from the FEP.

Once a Node I/O Manager executes a job thread, it informs the Host I/O Manager, reporting the thread's PJID and rank. The Host I/O Manager checks to see that it has an FEP connected with that PJID. Then it adds an entry under that PJID, associating the given thread rank with the Node I/O Manager connection that the information came from. All future input directed to that thread rank will need to be sent over this same Node I/O Manager connection.

When a job thread produces output on either standard output or standard error, the Node I/O Manager passes it along to the Host I/O Manager, along with the job PJID and thread rank. The Host I/O Manager then finds the FEP connection associated with that PJID in its data, and sends the output along to the FEP.

When input is received from the user, the FEP sends it on to the Host I/O Manager, along with the job PJID and thread rank it is intended for. The Host I/O Manager checks its data to see if a thread of that rank is recorded for the given PJID. If not, an invalid rank error is reported to the FEP. If there is such a thread, the input is passed along to the Node I/O Manager on the connection associated with that rank.

Note that no matter how many job threads (from different jobs with different PJIDs) are running on a node, there is only one connection between the Node I/O Manager and the Host I/O Manager. Since the PJID is distinct between each thread on a node, this is used to identify the correct thread from the Host I/O Manager.

In the current implementation, the Host I/O Manager knows nothing about how many threads there are in a job. It accepts those which it is informed about by Node I/O Managers. Their rank numbers need not be contiguous or complete. When it informs the FEP of

invalid thread rank, it really just means that if there is such a thread, it hasn't been reported.

When a job thread terminates, the Node I/O Manager informs the Host I/O Manager, which then removes the association it had for that thread rank. If the user tries to send input to that thread, the FEP will be told the thread rank is invalid. When all the threads have terminated, the Host I/O Manager is informed by the Host Job Manager, which does know exactly how many threads were executed. Then the Host I/O Manager tells the FEP of the job termination, and disconnects from it. When the last termination message (either from the Node I/O Managers or the Host Job Manager) arrives, the data for that job is completely removed.

If the FEP catches a terminating signal, it informs the Host I/O Manager before terminating. The Host I/O Manager then instructs the Host Job Manager to terminate the job. Similarly, if the FEP disconnects without warning, the Host I/O Manager instructs the Host Job Manager to kill the job. Either way, the Host I/O Manager records that the FEP is disconnected, to avoid sending it further output, but leaves its data alone otherwise, to be cleaned up by the termination messages that will arrive later, when the job threads are killed.

3.5 Node I/O Manager

A separate Node I/O Manager runs on each node of the parallel machine. It maintains one server socket, intended for a single connection from the local Node Job Manager. It maintains one client socket connection to the Host I/O Manager. For each job thread running on the local node, it has three pipe connections, for standard input, output, and error.

When the Node I/O Manager starts up, it connects to the Host I/O Manager, but sends no data to it. It listens for a connection request from the Node Job Manager, and once this connection is made, it waits for the Node Job Manager to send it a job thread to execute.

Job threads are executed by the Node I/O Manager, so that it can connect the job thread's standard input, output, and error to itself through pipes. The Node I/O Manager receives the executable path, user/group IDs, current directory, command-line arguments, and environment variables from the Node Job Manager. It sets up three new pipes, and forks off another process. The child process rewires its standard input, output, and error, to the correct ends of the pipes, closes the other ends, and executes the new job thread. The Node I/O Manager closes its unused ends of the pipes, and sends the PID of the child process to the Node Job Manager, along with the PJID and thread rank. Then it informs the Host I/O Manager of the new thread with its PJID and rank. It also maintains data structures tracking the association between a thread's PJID and rank, and its three pipe connections.

When threads are running on the node, the Node I/O Manager is waiting for data from the Node Job Manager, the Host I/O Manager, or any of the output or error pipes connected to the various job threads. From the Node I/O Manager it expects more job threads to execute. From the Host I/O Manager, it expects input from the user of one of its threads. From the threads themselves it awaits output or termination.

When input is received from the Host I/O Manager, it is tagged with the PJID of the destination thread. The Node I/O Manager searches its data for the given PJID, and sends the input to the pipe connected to that thread's standard input, as recorded in the data structure.

When output is received from a thread on a certain pipe, the Node I/O Manager searches its data for that pipe and find the associated PJID and rank. It sends these along

with the output, tagged with the type (standard output or standard error), to the Host I/O Manager.

In the current implementation, when an EOF has been received on both the standard output and standard error pipes, the Node I/O Manager informs both the Node Job Manager and the Host I/O Manager that the thread has terminated. The EOF on each pipe really just indicates that those pipes are closed. Instead, the Node I/O Manager should catch SIGCHLD signals to determine when its child processes have terminated.

Chapter 4

Other Design Issues

There are many other issues which have been considered in the design of the Jump-StarT scheduler. Section 4.1 looks at the issue of flexible socket communication in the system through port management. Section 4.2 expresses some aspects of the scheduler's modularity and the resulting advantages. Section 4.3 explains some security issues and solutions. Section 4.4 considers how the system can monitor jobs and provide status information. Finally, Section 4.5 examines the need for global error handling and fault tolerance in the scheduler.

4.1 Socket Port Management

The various modules in the scheduler communicate through many socket connections as mentioned in the previous chapter. Each connection is made with one module acting as a server and one as a client. To connect via sockets, the client has to know both the IP address (or DNS name) of the machine where the server resides, and the port number on which the server is listening for connections. This section explains how the scheduler modules know these two pieces of information for each connection they make.

Socket connections between modules on the same machine can be made by connecting to "localhost" and thus only need the right port number. This covers the connection between each Node I/O Manager and Node Job Manager, and between the Host I/O Manager and Host Job Manager. In each case, the I/O Manager maintains a server socket and the Job Manager connects to it as a client, through localhost.

In the current implementation, the only connections between modules that do not reside on the same machine are connections to the Host I/O and Job Managers. Thus only

the name of the host machine needs to be well-known across all scheduler modules. This machine name is stored in configuration files like *.njmgrrc* for each program. Yet this still doesn't explain how the client knows the correct port number.

The simplest way for clients to know the correct port number is to have the number hard-coded into the scheduler. To change the number, the program would need to be recompiled. This presents a problem if the desired port number happens to be in use when a server tries to set up its socket. Client sockets are assigned port numbers sequentially by the system, much like the PIDs assigned to new Unix processes. If one of these client sockets happens to be assigned to the server port number the scheduler is hard-coded to use, its socket creation will fail. Thus a more satisfactory technique is needed.

To solve this problem, a socket port manager is provided in the scheduler, called the *Master process*. The Master process is the only program in the system with a fixed, well-known port number[1]. The Master process is intended to run on the host, because if its machine crashes, no new jobs can be created until the scheduler is restarted. When each scheduler module starts up, it creates its server sockets on whatever ports it can. It does this by trying one; if the `bind()` call fails, it tries the next one; and so on, until one works. Then the module connects to the Master process on its known port, identifies which type of module it is, and reports each of the port numbers it has bound to server sockets. Node modules also provide their node ID, so they can be distinguished from each other. Then the Master process responds by reporting the port numbers that the module will need for its client connections.

---

1. To further improve reliability, a small, ordered group of known ports could be used here. If the Master process could not bind to the first port it would try the others. When connecting to the Master process, a module would try each port in succession until it connected. Then it would have to verify somehow that it had connected to the Master process and not some other program that had already taken the port.

If any of the servers that the module will try to connect to haven't reported to the Master process yet, the Master process will instead return an error. Then the module sleeps for a while and tries again. In this way, the processes don't have to coordinate to start up, but the necessary start-up order will be imposed by the Master process. As an example, say the Node Job Manager needs to make a client connection to both the Host Job Manager and the Node I/O Manager. If either has not registered with the Master process when the Node Job Manager checks in, it gets an error, waits, and tries again until it succeeds, which will only happen when both have properly registered themselves.

Each scheduler module must now know both the port the Master process is using, and the machine on which the Master process is running. Currently, the port is a compile-time constant and the machine is stored in a configuration file for each module.

4.2 Modularity

The scheduler code attempts to be modular in spite of the C language. This is to help make it understandable to a wide audience, easier to use debugging tools, and easier to update. The scheduler is organized into cooperating programs. Each has its own responsibilities and interacts with the other programs in standard, defined ways.

For example, the Node I/O Manager and Node Job Manager each run on every node of the parallel machine. They could be merged into one program. But each has a very different function. The Node I/O Manager processes all output from the node's job threads, and directs the appropriate input to them. The Node Job Manager is in charge of sharing processor time among the node's job threads. Two practically unrelated aspects of job management are handled by two different node managers. The two managers communicate only when a thread is created and when a thread terminates. Thus there is no great penalty

for separating the managers. Yet by doing so, the code for each is made dramatically more understandable.

The data structures manipulated by each of the scheduler modules are also designed in a modular way. Each moderately complex data structure has a set of associated functions that add, look up, and remove data, thus providing an abstract interface. If the underlying data structure is changed, only these functions need to be updated.

4.3 Security

A parallel machine is an expensive resource is and it needs to operate in a secure manner. A machine's owners do not typically wish to serve the computing needs of just anyone on the Internet that attempts to submit a job. Thus there is an issue of validating the users.

In order to establish a socket connection, the client must know the server's IP address, or at least the DNS name. When a server accepts a connection, it learns the IP address of the client that connected. This provides a simplistic method of validating clients. When a connection is accepted, the server validates the IP address. The server looks at a configuration file which lists IP addresses, hostnames, IP address masks, and NIS maps. Any machine from which parallel job requests are to be allowed should be available on the list in some fashion. The server validates the connected IP address with each entry in the configuration file. If no entries prove the validity of the IP address, the connection is then closed and ignored.

This validation is most useful in the Host Job Manager, for checking the incoming PRUN job request connections. It can also be used by each of the Host Managers, to ensure that the accepted connections are from actual machine nodes as expected.

An additional security concern is that I/O be routed correctly. An FEP should not be able to send input to any job other than its own. This is guaranteed because the Host I/O

Manager maintains an association between each PJID and the one FEP connection associated with that PJID. Input from an FEP is sent only to the job with the associated PJID. If another FEP tries to connect with the same PJID, it will not be accepted. Similarly, output from the threads of a job with a particular PJID will be routed only to the FEP associated with that PJID.

Since the parallel machine runs a Unix operating system on each node, the system inherits security benefits from Unix. One example is memory protection, which is provided to the parallel job threads as processes under each node's Unix system. On the other hand, the system is only as secure as that of its underlying Unix operating systems. Any encryption, for instance, is up to the individual application.

4.4 Monitoring

One of the secondary roles of the scheduler is to provide data to users about the jobs, or record it for monitoring purposes. The most crucial data that it should provide is information on job status. Other data could include the amount of standard I/O bandwidth being used by the parallel jobs, or statistics on machine usage levels at different times of the day.

4.4.1 System and Job Status

The status information can be provided through the implementation of a PSTAT program. Like PRUN, this program would connect to the Host Job Manager, perhaps to the same port. Instead of making a job request, PSTAT would request information about the status of the system or of running jobs.

System status information could include things like the number of nodes that are currently running and their node IDs. This would be useful for locating hardware and software failures. The Master process could contact each scheduler module, requesting

acknowledgment. Any modules that did not respond would be reported by the PSTAT process.

In reporting information on the jobs currently running, PSTAT would behave much like the Unix *ps* command. It could accept the same basic parameters and report on either one user's jobs, a particular job, or all the jobs in the system. Among the data it could report for each job are the process PJID, the executable name, the number of processors currently being used, and the name of the machine where the FEP is running. More information could be provided when a single job is queried, such as the node IDs of the nodes on which the job is running, the CPU time used on each node. In addition, it could report information specific to the scheduling strategy like deadline, job class, or current share of each node's CPU.

The PSTAT program has not yet been implemented.

4.4.2 Usage Statistics

Data could also be collected to monitor the usage of the system or of certain modules. One example would be to record the number of bytes of input and output sent through the Host I/O Manager. This would provide information on whether I/O is a serious bottleneck in the system. Another example is that the scheduler could record of the CPU utilization of each node at five-minute intervals. These data could be used to improve the performance of the scheduler by adjusting the behavior of the scheduling strategy accordingly.

4.5 Error Handling and Fault Tolerance

A parallel machine is an expensive, powerful resource. Parallel jobs typically require significant computational power and time, because otherwise they would be written for a much more common and affordable uniprocessor workstation. Because the jobs often need to run for long periods of time, the machine itself must be very stable and the soft-

ware that is managing the jobs must be stable as well. These facts lead to the consideration of fault tolerance issues in the scheduler. The scheduler should be written with all kinds of failures in mind, and be able to overcome them whenever possible.

Consider what will happen if a single node of the parallel machine is suddenly lost due to a hardware failure or loss of power. All the job threads running on that node are lost. The simplest good behavior would be for the scheduler to terminate all the remaining threads of each job that have lost a thread, on the assumption that the jobs themselves can not recover from such a fault. The scheduler would then report the failure to the users through the Front End Processes of the jobs. Note that any job which did not lose a thread would still be running as if nothing happened. The scheduler would then remove the node from its processor pool. When the node recovers, its scheduler modules would be executed again and the node would be added back into the pool.

A better solution could be much more difficult to implement. Such a solution could involve a way to inform each job about its lost thread. The fault tolerant jobs would have to register with the scheduler as being capable of dealing with node faults. Then, in such a situation, the scheduler would report the lost thread to each of the remaining job threads. The job might request that the scheduler execute the lost thread again on another available node, or the remaining threads could assume the work of the lost thread somehow. The latter approach would require a significant change in the programming model used.

4.5.1 Global Error Handling

A variety of errors can occur in the scheduler modules. With each system call that is made, there is a host of possible error values. Many can be avoided with properly working code. But many more are a result of uncontrollable problems with the operating system. If handled correctly, many errors will have effects on the scheduler as a whole, requiring modifications to data in multiple modules. The error recovery process therefore needs to be

handled by a global error manager, which can communicate with any of the scheduler modules. A convenient place to put such an error manager is in the Master process in the proposed scheduler design. The Master process is the first central place where the presence of each scheduler module is detected. It keeps track of all the ports for the other modules, so it has the necessary data to communicate with any of them.

Global error management would require a set of unique error codes to be compiled into all the scheduler modules. Each distinct error type would be given a different number, and the error manager would base its error recovery process on the error type. To handle some errors, the error manager would need to contact the reporting module and request data. Then it would send instructions to the affected modules.

The errors to be reported need to be chosen carefully, to keep the error manager from being flooded by reports while trying to respond to them. Whenever possible, a given error should be detected and reported by only one module.

Global error management has not yet been implemented.

4.5.2 Module and Data Stability

Beyond the detailed error handling of a global error manager, something is needed to recover from the abnormal termination of a module. One such method is to provide a shadow process for each scheduler module. A shadow process lurks in the background while a server module is running. If the module process dies, the shadow process restarts it. Unix provides a convenient way to implement such a mechanism. When a scheduler module starts up, it can fork off a copy of itself to do the real work. The parent process then acts as the shadow process. It does no computation, but executes the `wait()` system call. If the child terminates, the `wait()` function returns and the shadow process is ready to fork again and bring the module back up.

A key element in using the shadow processes successfully is to maintain each module's state in a stable way. The Host I/O Manager records plenty of data, but it all has to do with maintaining socket connections; there is really no state. If the Host I/O Manager died and was restarted by its shadow process, all its clients could reconnect. The Host I/O Manager's data would be rebuilt through all the connections that would be made. Other modules do have state, though. The Node Job Managers, for instance, keep track of the PIDs of each of the node's job threads. If the Node Job Manager died, and this data was lost, the Node Job Manager could no longer manipulate the node's processes through signals and priority settings.

Therefore, for those modules with actual state data, the shadow process would only be of value if the data could be recorded in a stable manner. A possible way implement this stable data is to save it to disk each time it is updated. The data could be saved under two alternating filenames. Then, when the shadow process restarts the module, the module would look at the two file timestamps, and read from the most recently saved file. If any errors are encountered, the other file would be used. If both files are corrupt, the data would be lost and the same problem would exist. But this would be unlikely, and the module would usually recover.

These shadow processes and stable data practices have not yet been implemented.

Chapter 5

Conclusion

5.1 Results

The basic Jump-StarT scheduler developed so far was written in less than 5000 lines of C code. It exhibits most of the behavior described in this document, but the job class model of scheduling has yet to be implemented. The code is scalable to more than 32 processors, but I/O through the single Host I/O Manager would become a major bottleneck if used intensively by applications.

Jump-StarT uses socket connections for all its communication, rather than the fast network layer that will be present in the StarT machines, because its communication is done directly between the host and each node. It appears that this will be sufficient for a scheduling strategy that runs based on schedule patterns supplied by the Host Job Manager, provided that the scheduling rounds are long enough. If communication was occurring between the nodes in a more complex scheduling strategy, there might be significant benefits to using the message passing layer.

In designing the Jump-StarT scheduler, most of the effort went into refining the system's flexibility. The load characteristics and I/O usage of applications in the system as it will ultimately run are still unknown. This meant there were always trade-offs between efficiency and flexibility. It was often unclear which goals were most important in a certain context.

The data structures and abstraction layers were also constantly refined. All the scheduler modules behave similarly in some respects. They act as servers, waiting to process incoming messages. Yet their differences made it difficult to find common ground while they were being written. It was only after they were fully implemented that the commonal-

ity could be seen clearly, and then parametrized functions created to greatly simplify the code.

Some design issues that arose well into the implementation required widespread changes. For instance, deciding to create the Master process for socket port management meant replacing all the socket initialization code in each module.

Often the easiest way to produce a certain effect on a system is not the portable way. The system was designed to rely only on functions that were available on both the AIX and Linux systems. It is amazing how different the "standard" C library can seem under different versions of Unix.

5.2 Summary

It is possible to write a parallel job scheduler that works on top of unmodified operating systems. A common underlying structure can be given to the scheduler to allow experimentation with scheduling strategies. Such strategies can use both centralized and distributed job control, and support a job class model. Standard input, output, and error streams can be provided in the parallel execution environment as an extension of the sequential computing environment.

Appendix A

Implementation Issues

This appendix covers issues in the specific implementation of the scheduler developed in preparation for this thesis. Section A.1 covers some race condition issues in job creation and termination. Section A.2 describes the organization of the code for these executables into various files. Section A.3 covers the structure of the code, and the similarities between the different programs.

A.1 Preventing Race Conditions

When a job is created in the scheduler, the Host Job Manager reports the new job's PJID to the requesting PRUN process. The PRUN process then executes an FEP, which connects to the Host I/O Manager. Meanwhile, the Host Job Manager goes about executing the job on the chosen nodes. Immediately after executing a job thread, the Node I/O Manager reports the new thread to the Host I/O Manager. If the Host I/O Manager gets this report before the FEP is connected, it will think that the new thread is invalid, because it has not yet found out about that job PJID at all.

There are probably other ways to handle this problem, but one is to ensure that the job threads are not executed until the FEP has connected to the Host I/O Manager. This is easily done by making the Host Job Manager wait until it knows the FEP is connected before going about executing the job. To know this, it must receive a message from either the FEP itself or the Host I/O Manager.

Similar challenges arise when a job terminates. When a job terminates normally, each of its threads terminate one by one. As each thread terminates, its parent process, the local

Node I/O Manager, finds out and reports the termination to both the Host I/O Manager and the Node Job Manager. The Node Job Manager passes it on to the Host Job Manager.

In the current implementation, the Host I/O Manager does not have first-hand knowledge of how many threads are in a job. All it knows is how many threads have set up I/O through the Node I/O Managers. Even when all the threads it has seen have disconnected it can't be sure there aren't more threads which haven't even connected yet. So the Host Job Manager sends the Host I/O Manager a message when all the threads have terminated. Only when the Host I/O Manager has received all the individual thread termination messages and the job termination message does it completely remove the job from its data.

Under abnormal termination, the user either forces termination or something goes wrong with the FEP that necessitates job termination. In this case, the Host I/O Manager notifies the Host Job Manager which notifies all concerned Node Job Managers, which effect the termination of the individual job threads.

There is now an issue about data structures getting cleaned up correctly, when they can be receiving termination messages from both directions. To solve this problem, the system does not believe that the threads are terminated until it receives the actual termination messages that originate from the Node I/O Managers. The Node I/O Managers are not informed about the forced termination; therefore, they go ahead and send termination messages as if the termination was normal. The system uses these to know when to clean up data, so it always happens in the same way.

A.2 Organization of Files

There are currently seven distinct executables in the scheduler. These are the Master process, the PRUN process, the Front End Process, the Host I/O Manager, the Host Job Manager, the Node I/O Manager, and the Node Job Manager. Each of the scheduler

executables has a corresponding C file which contains the `main()` function for the program. These files are *master.c*, *prun.c*, *fep.c*, *hiomgr.c*, *hjmgr.c*, *niomgr.c*, and *njmgr.c*. Some of these files have associated header files, only for the purpose of cleaning up the C file itself. In other words, all of the functions in these files are helper functions, and are not called by any other programs.

All the functions used by more than one program are found in the scheduler library. This consists of one file called *schedlib.c*, which contains no `main()` function. There are several header files associated with the scheduler library: *msg.h*, *master.h*, *parmutil.h*, and *sockutil.h*. These contain function declarations for sets of functions found in *schedlib.c*, grouped by category. The *msg.h* functions allow a program to send and receive the standard scheduler messages. The *master.h* functions allow a program to communicate with the Master process. The *parmutil.h* functions are utilities for dealing with command line parameters and environment variables. The *sockutil.h* functions are utilities that simplify the creation of sockets. Thus a program that uses the standard messages but doesn't deal with command-line parameters would include *msg.h*, but not *parmutil.h*.

A.3 Code Structure and Similarities

The program code for the scheduler modules is very similar in structure. The PRUN process is the most unique, because all it does is find out whether or not the job is accepted, and then it either terminates or executes the Front End Process. The rest of the programs act as servers, processing incoming messages, and behave much alike.

The main modules all start up in the same way. First, each program sets up a global buffer. Whenever the buffer is going to be written into, the program calls an `increaseBuffer()` function which checks to make sure the buffer will be big enough for the data and increases it if not.

Next, each program sets up all its initial network connections. Those programs that have server sockets all use the same function, `prepareRendezvousSocket()`, to create the socket, bind it to a port, and begin listening for connection requests. Each program calls the function in a loop, iterating over different port numbers until one is found that is not in use. Once all the server sockets are prepared, the program sends the port numbers to the Master process. Most of the programs also expect to receive one or more port numbers back from the Master process, which they need for their client socket connections.

If the required port numbers are not available yet, the Master sends an error, and the program waits for a time and tries again. When the program gets a successful response from the Master process, it goes on to set up its client connections, if any. For this, each program uses the function `prepareClientSocket()`.

Each of these programs then uses the C Library function `select()` to wait for messages on all of their sockets at once. To prepare for the function call, each program adds all its server and client sockets to a file descriptor set. The `select()` function will return if any of the rendezvous sockets have connection requests waiting to be accepted, or if any of the connection sockets have data waiting to be read.

From there, each program determines which type of event caused the select to return. If it was a connection request, the `accept()` function is called and the resulting connection socket is added to the file descriptor set. If it was data available, then helper functions are called to process the message based on who sent it. For instance, the Host Job Manager has three such processing functions: `processPrunMsg()`, `processHioMsg()`, and `processNjMsg()`. They handle messages from a PRUN process, the Host I/O Manager, and a Node Job Manager, respectively.

Since these programs are all supposed to run forever as daemons, they loop indefinitely on this cycle of calling `select()` and processing a message. Of the programs mentioned, only the FEP ever terminates without an error, when its job has terminated. So the structure of all these programs is very much the same.

The mechanisms for communication between modules are also very standardized. Functions named `sendMsg()`, `sendExistingMsg()`, and `recvMsg()` in the scheduler library are used in all modules, so that reading and writing to socket streams can be robust without being difficult to use. These message functions all use a common message header and the length of variable data following the header is defined in the header. The header is converted by the send functions to network order, and back to host order by the receive function. The send functions ensure that the entire message is sent, unlike the underlying `write()` library function they use.

References

[1] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "Scheduling on the Tera MTA". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 19-44. Springer-Verlag, 1995.

[2] J. M. Barton and N. Bitar, "A scalable multi-discipline, multiple-processor scheduling framework for IRIX". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 45-69. Springer-Verlag, 1995.

[3] G. A. Boughton, "Arctic Routing Chip". In *Parallel Computer Routing and Communication: Proceedings of the First International Workshop, PCRCW '94*, K. Bolding and L. Snyder (eds.), Lecture Notes in Computer Science, Vol. 853, pp. 310-317. Springer-Verlag, 1994.

[4] Chris Brown, *UNIX Distributed Programming*. New York: Prentice Hall, 1994.

[5] D. Chiou, B. Ang, Arvind, et al., "StarT-NG: Delivering Seamless Parallel Computing". In *Parallel Processing EURO-PAR '95*, S. Haridi, K. Ali, and P. Magnusson (eds.), Lecture Notes in Computer Science, Vol. 966, pp. 101-116. Springer-Verlag, 1995.

[6] D. G. Feitelson and L. Rudolph, "Distributed hierarchichal control for parallel processing". *Computer* **23(5)**, pp. 65-77, May 1990.

[7] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306-318, Dec 1992.

[8] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[9] D. G. Feitelson and L. Rudolph, "Coscheduling based on runtime identification of activity working sets". *Intl. J. Parallel Programming* **23(2)**, pp. 135-160, Apr 1995.

[10] D. G. Feitelson and L. Rudolph, "Parallel Job Scheduling: Issues and Approaches". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 1-18. Springer-Verlag, 1995.

[11] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "Implementation of gang-scheduling on workstation cluster". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science (to appear). Springer-Verlag, 1996.

[12] D. Lifka, "The ANL/IBM SP scheduling system". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 295-303. Springer-Verlag, 1995.

[13] J. Pruyne and M. Livny, "Parallel processing on dynamic resources with CARMI". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 259-278. Springer-Verlag, 1995.

[14] P. G. Sobalvarro and W. E. Weihl, "Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, pp. 106-126. Springer-Verlag, 1995.

[15] J. Torrellas, A. Tucker, and A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors". *J. Parallel & Distributed Comput.* **24(2)**, pp. 139-151, Feb 1995.