Lachesis: a job scheduler for the Cray T3E

Allen B. Downey

Colby College, Waterville, ME 04901,
downey@colby.edu,
http://www.sdsc.edu/~downey

Abstract. This paper presents the design and implementation of Lachesis, a job scheduler for the Cray T3E. Lachesis was developed at the San Diego Supercomputer Center (SDSC) in an attempt to correct some problems with the scheduling system Cray provides with the T3E.

1 Introduction

The Cray T3E is a distributed-memory multiprocessor consisting of DEC Alpha 21164 processors connected by a bidirectional 3-dimensional torus. The T3E at SDSC has 272 processors, of which 240 are application nodes reserved for parallel applications, 28 are command nodes, which execute sequential commands, and 4 are OS nodes, which provide operating system services like process management and I/O.

Each processor has 128 MB of memory, and is capable of peak performance of 600 MFLOPS. The application (APP) nodes at SDSC are configured with no swap space, because the scheduler does currently allow timesharing between parallel applications. There are, however, 9 GB of swap space for the command (CMD) processors, which do timeshare.

Although the T3E at SDSC can run parallel applications on up to 240 processors, the vast majority run on smaller partitions. These partitions are allocated dynamically by the scheduling system, and must be made up of logically contiguous processors. The contiguity requirement is only one-dimensional, though; each processor has a unique one-dimensional logical address. Thus, the virtual arrangement of processors (linear) is not the same as the virtual topology of the processors on the network (the 3D torus), and the virtual topology does not necessarily reflect the physical arrangement (which can be discontiguous). Unlike on the Cray T3D, partitions on the T3E are not required to be powers of two; jobs can run on any cluster size from 2 to 240.

In the context of this machine, *job scheduling* refers to the following decisions:

Queueing: choosing when each job or application should begin execution.
Preemption/timesharing: deciding when one job should be interrupted (or migrated) to allow another to run.
Allocation: choosing which set of processors to allocate to each parallel application.

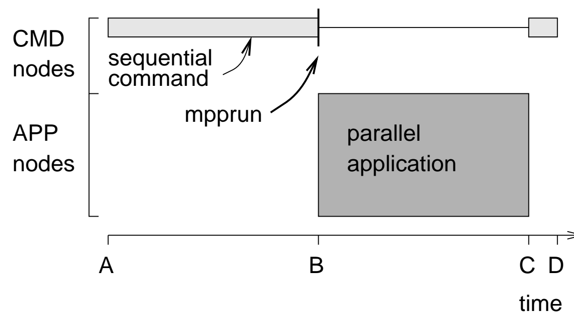

Fig. 1. A chart of the execution of a simple script with several sequential commands and one parallel application.

Section 2 describes the existing scheduler and the problems we encountered at SDSC. Section 3 outlines the goals we would like the scheduler to address; Section 4 proposes an abstract design that might achieve these goals. Section 5 describes two implementations of this design that we considered. Section 6 describes the current status of the project.

1.1 Definitions

Job: A job is a unit of work submitted by a user to the batch system. In most cases, a job executes a single script that contains a sequence of commands (possibly, although not commonly, iterative). Some of these commands are sequential; for example, they might move data between disk and tape or process temporary files. Sequential commands run on the command nodes (CMD nodes). Some commands are parallel; these create parallel applications, which run on the application nodes (APP nodes).

(Parallel) application: A parallel application is a set of sequential processes running concurrently on a set of APP nodes, usually communicating with each other during the execution of a common task. The most common way to spawn a parallel application is to execute the command `mpprun` on a CMD node. This has the effect of invoking the Global Resource Manager (GRM) to allocate a set of APP nodes, load the named executable on each of the allocated nodes, and begin execution of the application.

Process: A job is made up of a set of processes, including both sequential processes and the processes that make up parallel applications. Accounting information is generally collected on a per-process basis, and later aggregated into per-job reports.

Figure 1 shows the execution of a job with a single parallel application. The job starts at time A and runs a sequential command, possibly moving data from tertiary storage, until time B. During this interval, it runs only on the command node; no APP nodes have been allocated yet. Just before time B, the

job executes `mpprun`, which allocates a set of APP nodes and creates the parallel application that runs from time B to time C. During this interval, the command node is idle, and may begin (or continue) execution of another job. After the application completes, the job executes another sequential command and then completes.

2 Existing schedulers for the T3E

The scheduler that is shipped with the T3E has two components: NQS, which handles job scheduling, and GRM, which handles process scheduling. When a job arrives, NQS (Network Queueing System) determines when it will begin execution (and whether it might be interrupted or killed). As the job runs and spawns processes, the GRM (Global Resource Manager) decides where to run each process. Cray also provides an optional scheduling daemon, called PScheD, that provides additional features like gang scheduling. The next two sections describe these scheduling components.

2.1 NQS and GRM

Under NQS users submit jobs using the `qsub` command and specify their resource requirements by choosing the appropriate queue. For example, if a job requires 25 nodes, but does not run for very long, it should be submitted to `q32s`, where 32 is the smallest cluster size greater than the required 25, and `s` (short) indicates the user's estimate of the run time of the job, where the exact definition of "short" varies from site to site. Users have the option of providing additional information about their jobs, either on the command line or as pragmas in their scripts, but few users take advantage of this capability.

NQS holds jobs in queue until it sees that there are enough resources available to run the job (on the T3E, the only resource is processors; on many shared-memory machines, NQS also monitors the availability of memory). Once NQS releases the job, it runs as in Figure 1. Since NQS does not release the job until there are enough idle nodes to run the job, it is guaranteed that when the job executes `mpprun`, it will be able to allocate enough APP nodes from the GRM.

There are several problems with this approach:

- NQS does not communicate directly with the GRM; thus, it often does not know the number of APP nodes that are available. In the example, NQS would reserve 32 nodes for the job, even though it uses only 25. If there were a 7-node job in queue, NQS would not release it, although it would be able to run.
- NQS does not know when the job is running a sequential command or when it is running a parallel application. Thus, it reserves the requested number of APP nodes from the beginning of the job (time A) until the end (time D), although they are only used from the beginning of the parallel application (time B) to the end (time C). As Figure 1 suggests, the duration of the

sequential commands may be large compared to the duration of the parallel application.
– This configuration does not support jobs with multiple parallel applications, if they are not the same size. The user is forced to declare the size of the largest application; again, the system reserves more APP nodes than are needed.

There is another problem with the default scheduler that is due to the queueing discipline used by NQS. Different queues are given different priorities, such that a job submitted to a high-priority queue may run before a lower-priority job that arrived first. Thus, the queueing discipline is non-FIFO (first-in-first-out). The problem with non-FIFO queues is that they are likely to allow *starvation*; that is, there is a class of jobs that might wait in queue indefinitely while other jobs arrive and run.

In the T3E's default configuration, the scheduler gives priority to large jobs (96 or 128 nodes), because otherwise fragmentation might prevent these jobs from running. But if large jobs arrive frequently enough, smaller jobs (64 nodes and smaller) sit in queue indefinitely. At the moment, SDSC addresses this problem with the following ad hoc mechanism:

– After each preventative maintenance shutdown (roughly weekly), the short queues are restarted first, in order to get the starving short jobs out of the queue.
– Over the course of the week, large jobs take up the majority of the nodes, and many small jobs accumulate in queue. Operators sometimes start these jobs manually, by manipulating the NQS queues; otherwise, they run after the next shutdown.

In effect, SDSC runs the system in a weekly cycle: short jobs early in the week, large jobs later. The problems with this system are (1) it requires constant supervision and intervention, (2) the resulting schedules are neither efficient in their use of resources not satisfactory to users, and (3) the system creates incentives for users to modify their workloads (e.g., by submitting only large jobs) in a way that will further reduce the efficient use of the system.

2.2 NQS and GRM and PScheD

PScheD stands for *political scheduling*, which is scheduling that is based on externally-derived priorities (for example, one group of users may outrank another), as opposed to the internal priorities the system might use to improve performance (for example, by giving priority to small or short jobs).

PScheD is a daemon that runs periodically, examines the state of a domain (set) of nodes, and makes scheduling decisions [4]. One of the goals of PScheD is to achieve fair scheduling, where Cray defines “fair” to mean that service is allocated to users according to their priorities, independent of the number of jobs the user is running. The PScheD documentation explains [1]:

> With political scheduling, users are allocated a portion of the CPU resource specified by their share. Scheduling *fairly* among users means that users with the same number of shares should be able to consume the same amount of CPU resource, regardless of the number of [jobs] an individual user has active. Scheduling without the political scheduler has a tendency to let users with more [jobs] have a larger share of the machine resources. The political scheduler knows the consumption rate for each user, and users with many active [jobs] consume CPU resources at a higher rate than those with only a few. Therefore, users with many active [jobs] will have their [jobs] positioned lower in the scheduling (run) queue.

This conception of fairness is not an appropriate goal for scheduling at supercomputer centers. In these environments, the resources a researcher needs typically vary over time: during code design and development, a user may consume few node-hours; during production, especially before a deadline, the same user might want the whole machine. Assuming that users' deadlines are not simultaneous, it is desirable to allow a user to dominate the machine at times. When user peaks do coincide, the primary goal of the scheduler will be to avoid starvation, rather than to enforce external priorities.

To find a definition of fairness that is appropriate for supercomputer centers, we divide job scheduling into two separate problems, called "allocation" and "scheduling." The allocation problem is the decision of how many resources should be allocated to each user in a relatively long allocation cycle (typically several months). This decision is made by an Allocation Committee on the basis of the requirements and scientific merit of the project. The scheduling problem is the short-term decision (on the order of minutes or hours) of what job to run and what resources to allocate to it.

The two problems are connected by the liquid currency (node-hours, service units, etc.) that is the unit of allocation. A user's consumption is eventually limited by the allocation, but in the short term there is nothing to prevent a user from allocating a large fraction of the machine.

Even if Cray's notion of fairness were appropriate, their scheduler does not achieve it. Under PScheD, it is possible for an idle user to accumulate such a large priority that when he enters a production phase he is able to dominate the machine for a long time.

A more general problem is that PScheD seems to be based on a scheduling model in which small changes in priority yield gradual changes in quality of service. This model may apply to shared-memory machines with fine-grained time sharing, but on a distributed-memory machine, where time sharing (if it exists) tends to be coarse, with long quanta and significant memory swapping, the tools available to the scheduler are likely to be too blunt for discrimination on the basis of priority to be gradual or subtle.

3 Scheduling goals

Many of the users of parallel computers at supercomputer centers are working on Grand Challenge Problems; the nature of these problems is that they expand to use the available resources. Given a faster computer (or a larger allocation) researchers increase the size of their problems rather than solve the same problem faster.

In this environment, high system utilization is not the most important goal; it is generally not difficult to keep machines busy, provided that at any given time at least a few users are in a production phase. Instead, we conceive the goals of the scheduler from the users' point-of-view. The scheduler should:

- Allow users to run programs up to the limit of their allocations.
- Create the illusion that each user is running on a dedicated system.

The second goal implies that the scheduler should try to minimize queue times while allowing users to allocate large fractions of the machine when necessary. The scheduler should avoid imposing arbitrary restrictions like time limits and cluster size limits.

Of course, these goals are conflicting; for example, if users can allocate the entire machine for indefinite periods, it will be impossible to avoid long queue times. Addressing this conflict has been the focus of a large body of work on job scheduling.

Based on a survey of this work and observations of the atmosphere and goals of supercomputer centers, we have adopted the following design goals for the scheduler:

- Jobs must not starve. In terms of user satisfaction, avoiding starvation is more important than any other performance metric. It is also one of the most difficult to measure because we seldom see, in real systems, a set of jobs that literally starve. Instead, users typically kill starving jobs and learn not to submit the sort of job that receives poor service. Thus, it is not enough to say that starvation is not a problem because we don't see it. Rather, it is important to choose a queueing strategy that makes starvation impossible. A first-in-first-out queue (FIFO) can make this guarantee. There are, however, serious problems with FIFO queueing. The next section explains these problems and proposes ways to mitigate them.
- Delays should be proportional to run time. It may be acceptable for a 24-hour job to wait in queue for 12 hours, but it is not acceptable for a 1-hour job to wait that long. During testing phases, fast turnaround time for short jobs is critical. In general, the best way to prevent long-running jobs from interfering with short jobs is with preemption. We are considering a form of *lazy time-sharing* that might solve this problem.
- The system should provide a range of quality-of-service. Users should have the option of requesting higher priority (at some cost) or lower priority (in exchange for a discount). Although this feature is desirable, it may be incompatible with the requirement to avoid starvation. It also complicates the

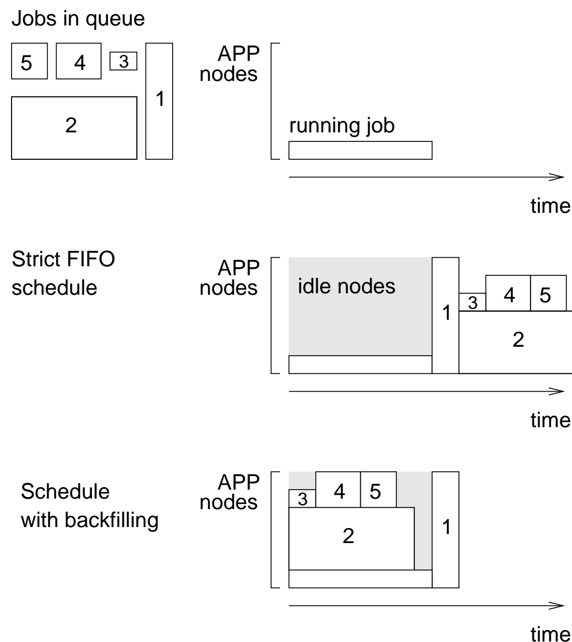

Fig. 2. Backfilling can mitigate the cost of FIFO queueing.

use of accounting to enforce externally-chosen allocations, since a CPU-hour would no longer be a unit of currency.

– Effective accounting is essential in order to control the allocation of the machine, monitor its utilization, and make it possible for users to make informed decisions. The cost of running a job must reflect the resources allocated to the job, rather than the resources used by the job. For parallel application, that means that the relevant measure of a job is the number of processors multiplied by the *wall clock time*. The existing system charges by CPU time, thereby undercharging jobs that perform a lot of I/O or leave processors idle.

The next three sections describe three scheduling features that might achieve these goals: backfilling, lazy timesharing, and support for moldable jobs.

3.1 Backfilling

The benefits of FIFO queueing are the guarantee that no jobs will starve and the ability to make predictions. Predictability is particularly useful in the context of metasystems, in which software agents will need to observe the state of the system (the length of the queue, the jobs already running, etc.) and predict the queue time until a new job can run.

The problems with FIFO queueing are (1) large jobs can impose long queue times on many small jobs, and (2) the system may be underutilized. Figure 2

demonstrates these effects. One way to improve the performance of a FIFO system is to add backfilling. Backfilling allows small jobs to begin execution even if a larger job is waiting, provided that the backfilled jobs do not delay the waiting job.

In the figure, Job 1 is waiting for a large cluster. If the other jobs were forced to wait, many processors would be left idle. We would like to allow the smaller jobs to run, but we can only do so if we can guarantee that they will complete before the running job. There are several variations on this strategy:

- Deterministic backfilling: if the run times of all jobs are known, it is easy to tell when backfilling is safe. In general, though, this is not the case.
- Pessimistic backfilling: we can assume that all jobs will run until their time limits, and backfill accordingly. Observations of other systems indicate that few jobs run until their time limits, so pessimism may not be warranted.
- Non-deterministic backfilling: given the distribution of lifetimes for past jobs, we can make claims about the probability that a job will run for a certain time. We might allow a job to backfill if it has a high probability of completing before the running job. In the occasional event that it exceeds that limit, we would have to delay the waiting job. Although this approach violates the FIFO principle, it can still be shown to avoid starvation, since you can't beat the odds forever.
- Contractual backfilling: we might offer a contract to a waiting job, offering to let it backfill, with the understanding that if it runs longer than the running job it will be killed. For applications that do their own checkpointing, this offer would be attractive. Of course, less robust jobs would decline. Contractual backfilling requires a user interface that allows users to identify killable jobs.

For all kinds of backfilling, the system needs estimates from users about the run times of their jobs. At the moment, users provide this information by choosing a queue with the appropriate time limit. This information is often not very accurate, but it is still useful. One of the features we are planning to add to Lachesis is an interface that solicits two estimates of run time, a best guess and a maximum. The best guess is the user's estimate of how long the job will take; the maximum is the time after which the job may be killed. Users have an incentive to provide accurate information, so that their jobs will be scheduled as soon as possible, with minimal chance of being killed.

The EASY scheduler, developed for the IBM SP at the Argonne National Laboratory, uses pessimistic backfilling [5] [7].

3.2 Lazy timesharing

The best way to prevent long jobs from imposing queue times on short jobs is to allow preemption. The motivation for preemption is that the longer a job has run the longer we expect it to run. So a newly-arrived job has a shorter expected remaining lifetime than a running job. It is preferable to impose a short delay on the (long) running job than a long delay on the (probably short) arrival.

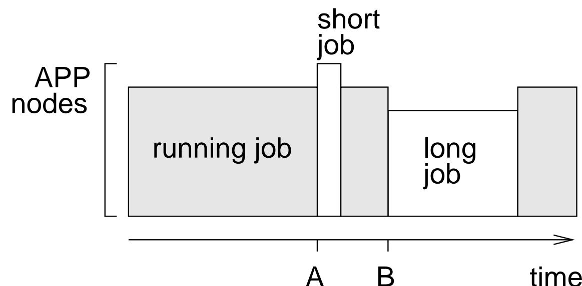

Fig. 3. An example of lazy timesharing.

Figure 3 shows how lazy timesharing might work. At Time A, a job arrives and preempts the running job. Since it turns out to be a short job, as expected, the delay imposed on the running job is not significant, and the turnaround time for the short job is good. The second job, arriving at Time B, turns out to be a long job. In this case, we run the new job until its expected remaining lifetime exceeds that of the original job. At that point, we resume the original job. The new job might migrate to another set of nodes, or continue to timeshare with the original job. Feitelson proposed the name "lazy timesharing" for this kind of preemption [3].

There is consensus in the scheduling research community that gang-scheduling is a necessary feature for timesharing parallel applications with significant communication (although Dusseau et al. argue to the contrary [2]). Gang scheduling is available with PScheD, but not under the vanilla UNICOS/mk scheduler. However, the timesharing strategy used by PScheD is very different from lazy timesharing; it is based on fixed quantum lengths that are shorter than lazy timesharing calls for—on the order of seconds, rather than minutes or hours. It may not be possible to modify PScheD to implement lazy timesharing.

Memory constraints will limit our ability to implement timesharing. Although the T3E memory system implements virtual memory, the T3E at SDSC is currently configured with no swap space. Thus, in order for jobs to timeshare, they must all fit in physical memory. The accounting data for past jobs suggests that there are many jobs (about 60%) that use less than half of the available memory, so some timesharing will be possible. On the other hand, it is not easy to predict before a job begins execution how much memory it will use, and many jobs grow dynamically as they run. Dealing with the allocation of physical memory will significantly complicate the timesharing strategies we can implement (for discussion of this issue, see [6]).

An alternate form of timesharing, checkpointing, may eliminate the problems associated with memory scheduling. Checkpointing differs from standard context-switching in that all of the memory associated with a process is written to disk, along with its system state. Of course, the overhead is potentially much greater, but the mechanism has the potential to be more robust, reducing the possibility that one process can interfere with another, and thereby improving

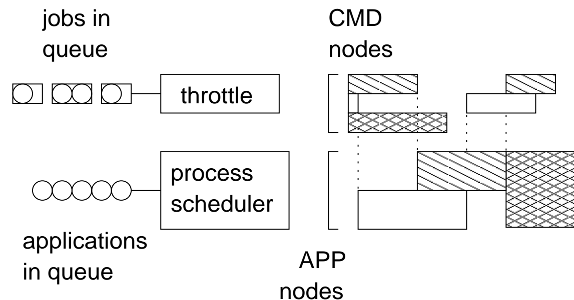

Fig. 4. The abstract design of a two-level job scheduler.

the illusion of a dedicated machine. Because we expect context switches to be infrequent, the overhead of checkpointing may not be prohibitive. Checkpointing was not available in UNICOS/mk when we designed Lachesis, but became available with Version 2.0.

3.3 Support for moldable applications

Many of the applications running at SDSC are moldable, meaning that they can be configured to run on a range of cluster sizes. In general, this configuration happens once; it is not possible to reconfigure an application once it starts running. We distinguish moldable applications from malleable ones, which can change size dynamically as they execute. Most of the applications running on MPPs are written in SPMD style, which seldom supports dynamic reconfiguration.

Ideally, users should specify the range of cluster sizes on which their applications can run, so that the system can choose the size most appropriate for the current load. In current systems, though, the user interface requires users to choose a specific cluster size. One of the features Lachesis will provide is an interface that allows users to specify a range of cluster sizes (or a set of particular values).

4 Abstract design

The following is the abstract design of a scheduler that meets the criteria discussed above. This design is based on the goal of separating job-level scheduling from process-level scheduling. In the next section, we will discuss the particular implementation of Lachesis.

Figure 4 shows a diagram of the design. Jobs are represented by small rectangles; the processes (sequential and parallel) that make up jobs are represented by circles. When a new job arrives, the *throttle* controls when and on which CMD node the job begins execution.

Once the job begins execution, it may execute `mpprun` one or more times, spawning parallel applications. When this happens, the *process scheduler* determines when and on which APP nodes the application runs. Thus, after a job

executes `mpprun`, it may be some time before the spawned application begins execution. In the figure, the first two applications start right away; the third waits in queue until the second completes.

The primary goal of the throttle is to allow as many jobs as possible to begin execution, probably several on each CMD node. Since many of the housekeeping commands that make up a job are I/O bound, multiple jobs may timeshare effectively. Also, the sooner a job arrives at its (first) `mpprun`, the sooner the spawned application becomes visible to the process scheduler. The only limitations on the number of jobs running simultaneously are (1) available memory on the CMD nodes, (2) the possibility of thrashing at the level of tertiary-secondary storage, and (3) lost state in the event of a crash.

Regarding the first point, accounting information from the first three months of operation (August through October 1997) suggests that more than 99% of jobs use fewer than 4 MB of memory on the CMD nodes. Since each node has 128 MB of memory, we can fit 32 to 64 jobs on each node.

Regarding the second point, it is possible that if many jobs start simultaneously and move data from tertiary storage onto disk, then there might be a long delay between the execution of `mpprun` and the beginning of a parallel application. During this time, the data moved from tertiary storage might be purged. It is not clear whether this is a serious concern at SDSC.

The last consideration is that it may be wise to limit the number of jobs running simultaneously in order to reduce the amount of work operators and users have to do to restart jobs that are active during a crash. As users gain experience with the T3E, they tend to write scripts that are robust, either by taking advantage of new checkpoint-restart mechanisms, or by using their own application-level checkpointing. Thus, it may not be necessary to worry about the number of jobs running simultaneously.

Since it is not clear that there are any immediate limitations on the number of jobs running simultaneously, an initial implementation of the throttle is likely to be trivial—it should allow all jobs to begin execution immediately.

The more interesting work happens at the level of process scheduling. The process scheduler needs to keep track of the state of the system; thus, it must be notified when a parallel application arrives or completes, and may use status commands to keep track of various system information. The process scheduler is responsible for starting new processes, preempting running applications (if there is timesharing), and killing applications that exceed their time limits. The most appropriate structure for the process scheduler is an *event-driven decision-maker*. That is, the process scheduler should be notified about relevant events and allowed to execute commands to realize its decisions.

If the process scheduler is doing probablistic backfilling, it will need to maintain a database of accounting information from past applications, so that it can predict the resource requirements of new applications. It also needs to generate accounting data of its own and respond to user queries about the state of the system.

A basic principle of this design is that in order to do intelligent scheduling, the scheduler must be aware of the applications that are waiting for service and their attributes. The only way to expose these applications is to get jobs running as quickly as possible until they execute `mpprun`, and then schedule the spawned applications.

5 Implementation options

We considered two implementation options, one based on PScheD, the other based on a wrapper around the existing `mpprun` combined with a scheduling daemon we would write from scratch. We chose not to work with PScheD; the following section explains why.

5.1 PScheD

As discussed in Section 2.2, PScheD implements a scheduling strategy very different from what we want. It does, however, provide a set of hooks where sites can install scheduling modules that implement alternate strategies. Thus, it might be possible to take advantage of the infrastructure provided by PScheD and build our scheduler on top of it.

The primary advantage of a PScheD-based scheduler is the possibility of gang scheduling. ScheD makes it possible to run more than one parallel applications on a given set of nodes, timesharing among them. Without PScheD, the T3E provides no support for gang scheduling; that is, parallel applications can share processors, but there is no guarantee that the processes that make up an application will be scheduled at the same time.

A fundamental problem with PScheD is that it does not address the problems regarding communication between NQS and the GRM. For example, one of the scheduling modules tries to reduce processor fragmentation by migrating applications from one set of nodes to another. It appears that PScheD does not consider the backlog of queued jobs when it makes its packing decisions. But this information is certainly relevant; as a trivial example, it is pointless to reduce fragmentation unless a waiting job requires it.

A second problem is that the interfaces between PScheD and the scheduling modules are narrow—PScheD provides little information and gives the modules little control. In our preliminary designs we realized that we would have to do a lot of work to get the information the scheduler needs into the appropriate modules. We also got the sense that we were not using PScheD in a way that was intended, and feared that we would waste too much effort cutting against the grain.

A final difficulty with the PScheD scheduling modules is that they were not available in the version of UNICOS/mk that was available when we started implementing Lachesis. Although we knew they would be available soon, we decided not to wait.

5.2 Scheduler daemon and `mpprun` wrapper

The design we implemented is based on a scheduler daemon similar to the PScheD daemon, but based on an event-driven paradigm. The daemon works with a wrapper around `mpprun` that notifies the daemon when applications arrive and complete. The entire system is called Lachesis, after the Fate in Greek mythology that "schedules" the threads of men's lives. The scheduling daemon is called lachd; the wrapper around `mpprun` is called `mppfun`.

In order to activate Lachesis, we start lachd and replace `mpprun` with `mppfun`. Then, when a user executes `mpprun`, he actually executes our wrapper, which communicates with lachd through a socket. The wrapper collects information about the application, including the user's name, the location of the executable, and any user-provided information like the cluster size and expected duration, and sends this information to lachd.

When lachd receives a request, it either allocates processors immediately or adds the request to its queue. Eventually, lachd starts the job by sending a message to the `mpprun` wrapper, telling it which processors (and how many) it can allocate. The wrapper then executes the "real" `mpprun`, spawning the parallel application.

When the application completes, the wrapper collects the completion code and sends it to lachd. If an application exceeds its time limit, the `mpprun` wrapper kills it (not, as might be expected, the daemon). Since the wrapper is executed by the owner of the parallel application, it does not need any special permission to kill (or migrate) the application. Furthermore, since lachd does not execute any protected commands, it does not need to run with any special permissions. Thus, at least abstractly, running Lachesis does not introduce any security problems. In the next section, though, we will discuss some implementation problems that undermine what would otherwise be a pleasantly secure system.

An advantage of putting a wrapper around `mpprun` is that is it is easy to extend the interface to solicit different or additional information from users. In the current implementation, we have not modified the interface, but soon we will make it possible for users to specify a range (or set) of possible cluster sizes, rather than a single choice, and solicit more information about the expected run times of jobs.

We are using an eviscerated configuration of NQS as a throttle. Since users provide information for each application (when they execute `mpprun`), they no longer have to provide information about jobs when they are submitted. Thus, we do not need a queue for each cluster size and duration. Instead, we have only a few queues, specifying the desired level of service. Within each queue, NQS starts jobs in FIFO order. Some queues have higher priority than others, with correspondingly higher pricing.

5.3 Implementation difficulties

We ran into several problems building Lachesis, some of which we have still not addressed to our satisfaction. The first is that it is possible on the T3E to start a

parallel application without using `mpprun`. This capability is new, and we did not realize when we designed Lachesis that so many people were using it. Of course, putting a wrapper around `mpprun` is not as effective if users can circumvent it. So, one user-visible consequence of the new scheduler is that we require users to use `mpprun`.

Unfortately, in the current version of Lachesis, we have no way to enforce this restriction. Thus, it is possible for users to run *rogue jobs* that never notify Lachesis, and that occupy nodes that Lachesis think are idle. For this reason (and others) Lachesis performs periodic *reality checks* that allow it to detect and monitor rogue jobs.

Because of the reality check mechanism, Lachesis has the ability to start on the fly—that is, while there are already jobs in the system. When Lachesis starts, it collects information about running jobs and monitors them until they complete. Meanwhile, it starts scheduling arriving jobs on the idle processors. The ability to start on the fly has made it possible to test Lachesis in a production environment with a minimal impact on users.

We encountered one other difficulty that is a result of the division of labor between Lachesis and `mppfun`. Although lachd chooses which processors to allocate to an application, `mppfun` actually starts the application. Thus, we need a mechanism whereby `mppfun` specifies where each application runs. In a typical T3E installation, this capability is reserved for system administrators. Users do not have control over the placement of their applications. In order to give users this capability, we had to give all users a special permission bit called DIAG. At the moment, this permbit is not used for anything else, so it does not create a security problem to give it to everyone, but in the future there may be other diagnostic activities we would like to protect, and in that case we would have to create a new permbit.

6 Project status

A simple version of the Lachesis daemon and the `mpprun` wrapper ran in production at SDSC from August 1997 to January 1998. During that time, Lachesis did not actually schedule the machine; it only observed and logged system activity. A benefit of the prototype system is that it provides more complete information about parallel applications than is available from the Cray Accouting System. We are currently using that information in our design of a non-deterministic backfilling strategy.

Since January 1998, we have been testing Lachesis version 1.0, which implements a variation of the pessimistic backfilling strategy decribed in Section 3.1. At the time of this writing we do not have enough information to evaluate the performance of the new system.

Along with the development of Lachesis, we have also built a simulator that reads events from the Lachesis logs and simulates the state of the system. The same scheduling module that plugs into the Lachesis daemon also plugs into

the simulator, allowing us to test new modules for correctness and to evaluate various scheduling strategies.

Acknowledgements

The primary implementors of Lachesis are Allen Downey and Victor Hazlewood at SDSC. We would also like to thank Larry Diegel from SDSC and Peter Ashford from Cray for all their help.

References

1. Cray Research, Inc. *UNICOS/mk Resource Administration, SG2602*, 1997.
2. Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 25–36, May 1996.
3. Dror G. Feitelson. Job scheduling in multiprogrammed parallel system. Technical Report RC 19790 (87657), T. J. Watson Research Lab, I.B.M., August 1997. Second revision.
4. Richard N. Lagerstrom and Stephan K. Gipp. PScheD: Political scheduling on the Cray T3E. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 1291*, pages 117–138, 1997.
5. David Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 949*, pages 295–303, 1995.
6. Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 57–67, May 1996.
7. Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY – LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing, Springer-Verlag LNCS Vol 1162*, pages 41–47, 1996.