Scheduling options for the Cray T3E

Allen B. Downey *

June 18, 1997

Abstract

I describe several problems associated with the scheduler running on the Cray T3E at SDSC, review the goals of job scheduling for parallel scientific applications, and propose possible implementations for a scheduler that is more appropriate for supercomputing environments.

1 Introduction

This is a white paper SDSC is preparing to evaluate the existing scheduling system for the Cray T3E, and to consider possible improvements and alternatives. This version of the paper is a draft and should not be distributed. The opinions I express are mainly my own, although I am trying to describe the goals and requirements of SDSC. Comments are welcome; my addresses are at the bottom of this page.

1.1 Outline

Section 2 defines some of the terms that will be used in this paper. Section 3 describes the current scheduler; Section 4 describes PScheD, a daemon provided by Cray to assist the default scheduler and provides additional scheduling features (like gang scheduling). Section 5 outlines the goals an effective scheduler should achieve, and presents an abstract design that addresses those goals. Section 6 presents two possible implementations of the abstract scheduler. Section 8 summarizes the pros and cons of the two options.

2 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,

*San Diego Supercomputer Center, P.O. Box 85608, San Diego, CA 92186. email: downey@sdsc.edu, http://www.sdsc.edu/~downey

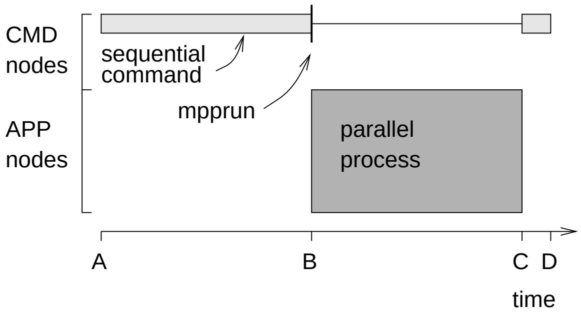

Figure 1: A chart of the execution of a simple script with several sequential commands and one parallel process.

or remove temporary files. Sequential commands run on the command nodes (CMD nodes). Some commands are parallel; these create parallel processes, which run on the application nodes (APP nodes). Accounting data from the first three months of production operation indicate that roughly 25% of jobs execute more than one parallel process.

(Parallel) process: A parallel process 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 process 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 process. Cray documentation sometimes refers to a parallel process as an "application."

Thread: Threads are the subprocesses that make up a parallel process. One thread runs on each APP node. Since each node handles its own thread scheduling, it is not necessarily true that all of a process's threads will be active simultaneously. Gang scheduling enforces this requirement.

Figure 1 shows the execution of a job with a single parallel process. 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 process 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 parallel process completes, the job executes another sequential command and then completes.

3 Scheduling with NQS

The basic scheduler provided by Cray is based on NQS. Users submit jobs using the `qsub` command, which requires them to specify the resource requirement of the job 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` indicates the user's estimate of the run time of the job.

After a job is submitted, NQS decides when the job will begin execution. 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 used 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 component or when it is running a parallel process. 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 process (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 process.

- This configuration does not support jobs with multiple parallel processes that are not the same size. The user is forced to declare the size of the largest parallel process and, 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 that arrives in 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. These long delays greatly decrease users' satisfaction with the system, because they give the impression that the system is "unfair."

In the specific case of the T3E, the scheduler gives priority to large jobs (96 or 128 nodes), because otherwise fragmentation might prevent these jobs from running. If these large jobs arrive frequently enough, though, then smaller jobs (64 nodes and less) sit in queue indefinitely. At the moment, we are addressing this problem with the following ad hoc mechanism:

- After each preventive maintenance shutdown, 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 will run after the next shutdown.

In effect, we are running 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 (by submitting only large jobs) in a way that will further reduce the efficient use of the system.

4 Scheduling with 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 for small or short jobs).

PScheD is a daemon that runs periodically, examines the state of a "domain" of nodes, and makes scheduling decisions that it effects by issuing protected commands.

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:

> 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.

In my opinion, this conception of "fairness" is inappropriate for the environment at SDSC. It is often the case that researchers' needs vary over time: during code design and development, a user may consume few node-hours; during production, and especially before a deadline, the same user might want the whole machine. Assuming that such deadlines are not generally simultaneous, it is desirable to allow a user to dominate the machine for short intervals. 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 SDSC, I think it is best to identify two separate problems, which I call "allocation" and "scheduling." The allocation problem is the decision of how many resources should be allocated to each user in a period of time. This period is long (several months), and the decision is made by the Allocation Committee on the basis of the requirements and scientific merit of the project. The scheduling problem is the decision, on the short term (minutes or hours) of what job to run and (possibly) what resources to allocate to it.

The two problems are connected only in the sense that users are allocated some liquid currency (node-hours, service units, etc.) that they spend when they run jobs. Thus, the number of jobs a user can run is eventually limited by his allocation. But in the short run, there is nothing to prevent a user from allocating a large fraction of the machine.

Even if Cray's definition 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. More generally, PScheD seems to be based on a scheduling model in which small changes in priority yield gradual changes in quality of service. For parallel scheduling, though, the tools available to the scheduler — primarily starting and preempting jobs — are too blunt to achieve the intended gradations. It is unlikely that timesharing will be sufficiently pervasive in a batch domain to make discrimination on the basis of priority gradual or subtle.

5 Scheduling goals

Many of the users of parallel computers at SDSC 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, 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. Thus, the goals of scheduling are:

- 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. Nearly all work on job scheduling has focused on addressing this conflict.

Based on a survey of research in this area and observations of the atmosphere and goals of SDSC, I think that the following principles should guide the design of the T3E scheduler:

- 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 processes, that means that the cost of the job is the number of processors multiplied by the *wall clock time*. For commands running on CMD nodes, though, we may want to charge by CPU time or maybe not at all, since these commands are usually dwarfed by the parallel component.

- Jobs must not starve. I think it is critical that we choose a queueing strategy that makes it impossible for a job to sit in queue indefinitely while later arrivals pass it by. A first-in-first-out queue (FIFO) guarantees that jobs will not starve. 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. I will suggest a form of "lazy time-sharing" that might work solve this problem.

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

An additional requirement for the scheduling system is that it provide a reporting mechanism that lets users check the status of their jobs. Multi-level schedulers sometimes make this difficult, since users might have to run several commands depending on where the job is. It might be a good idea to provide a wrapper around these commands to unify and improve the user interface.

5.1 FIFO queueing with backfilling

The benefits of FIFO queueing are that no jobs starve and that the system is predictable. 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.

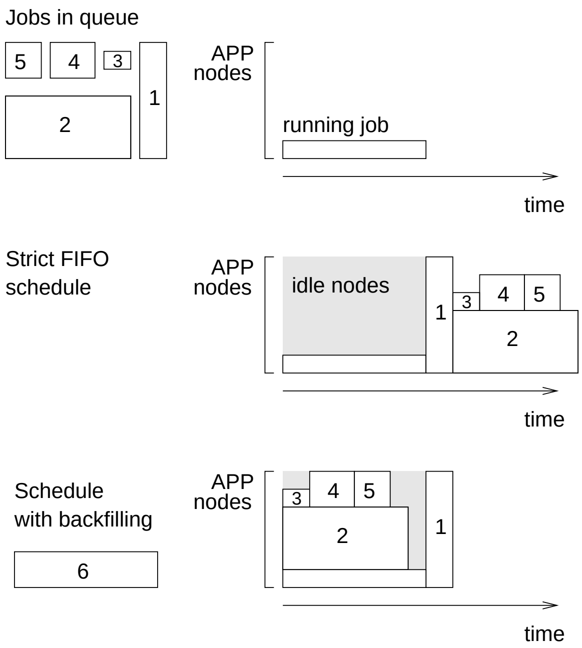

Figure 2: Backfilling can mitigate the cost of FIFO queueing.

The problems with FIFO queueing are that (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 this may not be a good choice.
- 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 (maybe 95%) of completing before the running job. In the occasional event that it exceeds that limit, we would have to delay the waiting job.

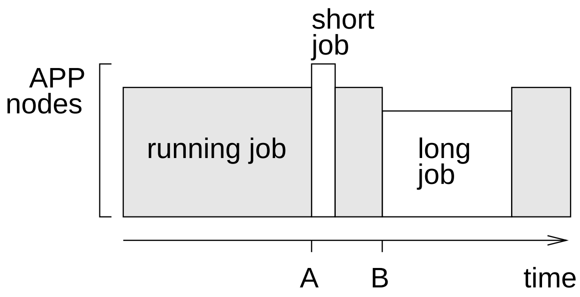

Figure 3: An example of lazy timesharing.

- 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. Applications that do their own checkpointing might like that offer. Contractual backfilling requires a user interface that allows users to identify killable jobs.

With non-deterministic backfilling, it would be helpful to get an estimate 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. In general, the information is not very accurate, but it is still useful. In future systems, I would like to move to an interface in which users explicitly provide two estimates of run time, best guess and 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. Jobs will be backfilled according to their maximum run time; thus, 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.

5.2 Lazy timesharing

The best way to prevent long jobs from imposing queue times on short jobs is to allow preemption. The motivation for preemption 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.

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 would run the new job until its expected remaining lifetime exceeds that of the running job. At that point, we should resume the original job. The new job might migrate to another set of nodes, or continue to timeshare with the original job.

There is consensus in the scheduling research community that gang-scheduling is a necessary feature for timesharing parallel processes with any significant communication. Gang scheduling is available under PScheD, but not under the vanilla UNICOS/mk scheduler. However, the timesharing strategy used by PScheD is very different from the lazy timesharing I am proposing. It is based on fixed quantum lengths that are much shorter than I have in mind (between 10 seconds and one minute, as opposed to minutes or even hours). It may not be possible to modify PScheD to implement lazy timesharing.

Memory constraints will limit out 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.

5.3 Malleable processes

Many of the jobs running at SDSC are malleable, 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 a job once it starts running.

Ideally, users should specify a range of cluster sizes on which their jobs could run, so that the system could choose the size most appropriate for the state of the system. In current systems, though, the user interface requires users to choose a specific cluster size. I would like to provide a more general interface to facilitate malleable scheduling. One of the implementation options I will propose, putting a wrapper around `mpprun`, makes it possible to provide such an interface.

5.4 Abstract design

Based on these requirements, I envision an abstract design based on the goal of separating job-level scheduling from process-level scheduling.

Figure 4 shows a diagram of this design. Jobs are represented by small rectangles; the processes 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 processes. When this happens, the *process scheduler* determines when and on which APP nodes the process begins execution. Thus, after a job executes mpprun, it may be some time before the spawned process begins execution.

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

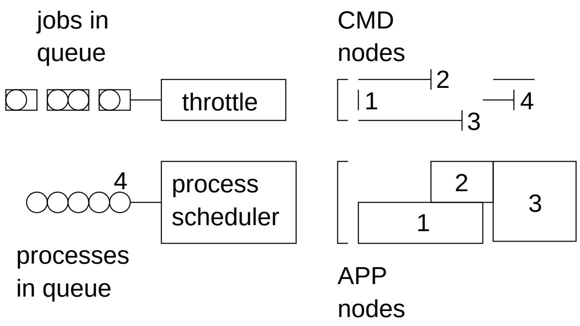

Figure 4: The abstract design of my dream scheduler.

effectively. Also, the sooner a job arrives at its (first) `mpprun`, the sooner the spawned process 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) disk thrashing, and (3) lost state in the event of a crash.

Regarding the first point, accounting information from the past three months suggests that more than 99% of jobs use fewer than 4MB of memory on the CMD nodes. Since each node has 128MB of memory, we could easily 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 process. During this time, the data moved from tertiary storage might be purged. It is not clear whether this is a serious concern in this environment.

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 have to do to help users recover from jobs that are active during a crash. Again, this may not be a serious concern.

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.

Thus, all the 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 process arrives or completes, and may use status commands to keep track of various system state. The process scheduler is responsible for starting new processes, preempting running jobs if there is timesharing, and killing jobs that exceed their time limits. The most appropriate structure for the process scheduler is what I call an *event-driven decision-maker*. That is, the process scheduler should be notified about relevant events and allowed to execute commands to effect its decisions.

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

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

6 Implementation options

6.1 PScheD

PScheD is implemented as a daemon that runs periodically, collecting system state and issuing privileged system calls and commands in order to effect its decisions. Some of the features offered by PScheD are:

- Load balancing: PScheD monitors the load on the CMD nodes and migrates jobs from one to another in order to keep the load balanced. It is not clear how necessary this is, or whether it has any great effect on I/O-bound commands like `pftp`.
- Gang scheduling: PScheD makes it possible to run more than one parallel process on a given set of nodes, timesharing among them. As I mentioned earlier, the default configuration for timesharing is not exactly what I have in mind, but I think it could be adapted. Without PScheD, there is currently no support for gang scheduling.
- Packing: In order to reduce fragmentation, PScheD periodically migrates processes from one set of nodes to another. The algorithm it uses is cryptic; I am not satisfied that it is properly taking into account (1) the danger associated with migration (how robust is it?) and (2) the tight relationship between packing and scheduling. It does not appear that PScheD considers the backlog when it makes its packing decisions. This is clearly a problem, since it is pointless to reduce fragmentation unless a waiting job requires it.

PScheD works with NQS, although it does not directly address the problems regarding communication between NQS and the GRM. However, assuming that we can build an appropriate process scheduler with PScheD, then we should be able to use NQS as a simple throttle. The only difficulty I can foresee is that if users specify the attributes of their jobs by choosing an NQS queue, we may have to build a mechanism for making that information available to the process scheduler.

I have a number of concerns about PScheD. The first is that PScheD is heavily invested in the notion of fair-share scheduling, which I think is not well-considered. As a result, the gang scheduling and packing mechanisms are complicated with elaborate priority schemes based on a frightening number of mysterious parameters. I don't know how easy it will be to control PScheD by manipulating these parameters.

PScheD has recently added a new feature to facilitate explicit control by site-provided modules. We could use this feature to get rid of the "fair-share" mechanisms and take explicit control of the scheduling features. However, my impression is that the interfaces between PScheD and the user-provided modules are fairly narrow; that is, PScheD provides little information and gives the modules little control. Thus, I expect that implementing a general processor scheduler will require a fair amount of struggle against and circumvention of PScheD.

My second concern about PScheD is that its future as a Cray product seems uncertain. It is brand-new and not very robust. The people at NERSC have been trying it out on their T3E, and have reported a number of unexpected behaviors. While Cray does seem to be addressing these problems, I am not convinced that Cray is committed to making PScheD a "real" product. It seems to have been built as an afterthought, and I think it is possible that Cray will not the kind of support we would need in order to run a PScheD-based scheduler in a production environment.

My final concern is that PScheD is based on a model of periodic intervention rather than event-driven decision making. For example, one of the bugs reported by NERSC is that PScheD does not know when a process completes. Thus, it may leave processors idle for several minutes until the next "heartbeat." This is not an insurmountable problem, but there may be other problems that will arise because of this awkward paradigm.

6.2 Scheduler daemon and `mpprun` wrapper

A second alternative is to implement something like the abstract design from scratch, using a Scheduler daemon very similar to the PScheD daemon, but based on an event-driven paradigm.

The daemon would work with a wrapper around `mpprun` that would (1) provide a flexible user interface, making it possible to solicit different or additional information from users, and (2) notify the daemon when processes arrive and complete. We could use an eviscerated configuration of NQS as a throttle.

If it is possible to obtain source code for PScheD, I would use a stripped-down version of PScheD as a starting place. In this case, it would be straightforward to implement timesharing. Otherwise, it might be necessary to do without preemption, at least initially.

The Scheduler daemon would most likely be written in C, or possibly Perl. The advantage of C is that it would be easier to maintain a data object describing the state of the system, and easier to issue system calls. The advantages of Perl are that it would be (1) easier to parse status commands in order to derive system state, and (2) easier to maintain the database of past executions, and generate accounting information.

Another possible implementation for the Scheduler daemon is to add a Tcl script to the NQE lightweight server (LWS). The LWS already runs on each

NQE host (including the T3E), running periodically to collect and disseminate information about the state of the system. The LWS could be augmented to issue scheduling commands to NQS and the GRM. An advantage of this approach is that we would get leverage from the existing NQE distributed system; a disadvantage is that the scheduler would be implemented in Tcl, which is not a natural choice for this task, nor a language that is well-known at SDSC.

One advantage of implementing the scheduler as a stand-alone daemon is that it would be possible to comply with an emerging standard API for parallel scheduling. The name of the standard, confusingly, is PSCHED; it is being developed at NAS with the intention of making it possible to write a scheduler that could port to several parallel platforms. It might be possible for a carefully-designed T3E scheduler to run on another architecture, reducing the lag between the arrival of a new machine and the implementation of an effective scheduler.

In addition, the existence of a scheduler API standard will make it possible to test a new scheduler on a trace-driven system simulator, and then install the same scheduler in a production environment. This will make it much easier to evaluate new scheduling research and transfer it into real systems.

7 Minimal effort solutions

Rather than replace the existing scheduler, we might decide to keep what we currently have and make incremental changes that address the most pressing problems. This section lists those problems and the minimal-effort solutions we could implement.

7.1 Starvation

Victor Hazlewood and Peter Ashford have experimented with a script that runs periodically, parses the output of the `jstat` command, and effects scheduling decisions by manipulating NQS queues. For example, if the script sees that small jobs are starving, it turns off the large job queues until the small jobs ran. In effect, the script does automatically what operators have been doing explicitly.

One problem with this approach is that it tends to complicate the structure of the NQS queues. The solutions we have envisioned involve large numbers of queue complexes and pipe queues that might be difficult to maintain. Also, the queue model that is presented to the user is complicated. If a given job is not running, it is not always easy to figure out why, without knowing a lot of details about the queues.

Another problem is that this solution is a dead end — it does not lend itself to future improvements like backfilling and timesharing.

7.2 Accurate reservations

As described above, the NQS-GRM interaction sometimes reserves more nodes than a job uses, and for too long. A possible quick fix would be to configure NQS to oversubscribe the system by some amount, and rely on the GRM to queue excess demand when necessary.

7.3 Accurate accounting

Cray System Accounting (`csa`) maintains information about each process in a job, but generally reports information on a per-job basis. Thus, summary statistics like the total CPU time for a set of parallel processes are easy to generate, but per-process statistics, like the duration and cluster size of individual processes, are harder to come by.

If we allow users to execute more than one parallel process per job, and I think we should, then we should modify the accounting system to charge parallel processes by wall clock time and cluster size. The most likely implementation is a script that parses the output of `csajrep` (per-process information) and generates per-job summaries. This is much harder than generating the statistics we are currently using.

8 Summary

The good things about PScheD are

- It provides hooks for site-specific modifications. Using these hooks, I think we can implement an effective scheduler.
- It already provides packing and timesharing, which would otherwise be difficult to implement.

Possible problems with PScheD are

- We will not be using PScheD the way it was intended. Thus, we will leave large chunks of functionality unused (like the fair-share system), while trying to sneak different functionality in through what are supposed to be small modifications. The resulting system may not be robust.
- NERSC has reported some problems with the PScheD implementation. Nevertheless, there is no reason to think that Cray will not be able to fix them.
- It is not clear what future PScheD has as a Cray product. I have inquired with the authors (Richard Lagerstrom and Stephan Gipp at Cray), but I haven't heard back.

Some of the good things about implementing our own scheduler are

- By designing from scratch, we can build a robust, extensible system and a framework for future schedulers on the T3E and other parallel machines.
- If we make such a scheduler available to other T3E sites, we will further the goals of NPACI, and possibly establish several "firsts" in parallel scheduling. For example, there are currently no schedulers in production environments that support malleable jobs; this could be the first.
- We can build the scheduler to meet the PSCHED API specifications (or possibly define them). Such an implementation will give enormous leverage to future research in parallel scheduling.

Potential problems are

- This option requires the greatest implementation effort, although the result may, in the long run, be easier to maintain.
- Adding timesharing to this system will require us to redo much of what PScheD has already done. If we implement timesharing, it would be very useful to obtain PScheD source code.