Job Management Requirements for NAS Parallel Systems and Clusters

William Saphir[1], Leigh Ann Tanner[1], Bernard Traversat[1]

NAS Technical Report NAS-95-006 February 95

NAS Scientific Computing Branch
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

Abstract

A job management system is a critical component of a production supercomputing environment, permitting oversubscribed resources to be shared fairly and efficiently. Job management systems that were originally designed for traditional vector supercomputers are not appropriate for the distributed-memory parallel supercomputers that are becoming increasingly important in the high performance computing industry. Newer job management systems offer new functionality but do not solve fundamental problems. We address some of the main issues in resource allocation and job scheduling we have encountered on two parallel computers — a 160-node IBM SP2 and a cluster of 20 high performance workstations located at the Numerical Aerodynamic Simulation facility. We describe the requirements for resource allocation and job management that are necessary to provide a production supercomputing environment on these machines, prioritizing according to difficulty and importance, and advocating a return to fundamental issues.

1.0 Introduction

Supercomputer centers have historically used batch queuing systems such as NQS [Kin86] to manage oversubscribed resources and schedule computer time fairly.

1. Computer Sciences Corporation, NASA Contract NAS 2-12961, Moffett Field, CA 94035-1000

For distributed-memory parallel computers, which are becoming increasingly important for supercomputing, resource scheduling is significantly more difficult than it is on single processor or multiprocessor shared memory machines. This is because parallel applications have more complex scheduling requirements — resource requirements (e.g. varying the number and type of nodes, different numbers of processors per node) are more complex; choice of timesharing method directly affects scheduling strategy; system software, particularly on parallel computers formed from a network of workstations, is often oriented towards scheduling single-node serial applications.

The Numerical Aerodynamic Simulation (NAS) supercomputer facility, located at NASA Ames Research Center, is actively trying to bring parallel supercomputers into its mainstream production environment. Part of that effort involves finding a robust job management system (JMS). Several job management systems claim to manage parallel jobs, but the experience so far at NAS is that these are not suitable for the NAS workload environment. These packages include NQS [Kin86], DQS [Duk94], DJM [DJM93], Condor [Lit88], LoadLeveler [IBM94] and LSF [Zho93,Pla94]. Section 3.0 explains some of the fundamental reasons why job management systems oriented towards serial jobs are not appropriate for parallel systems, and addresses important issues in resource management and scheduling. Section 4.0 presents a list of requirements for a job management system targeted towards the NAS parallel systems.

We avoid a discussion of what is arguably the most difficult issue in the real world of networks of workstations — politics. We leave it to management to decide how resource availability is determined in the first place and limit JMS requirements to what we believe is technically feasible in the relatively short term. Our requirements also do not address issues such as how fast interactive response must be, how much warning must be given before allowing a workstation to pull out of a pool, etc.

2.0 The NAS Environment

The Numerical Aerodynamic Simulation division is a pathfinder in high performance computing for NASA. This paper focuses on job management requirements for two parallel systems at NAS — a 160-node IBM SP2 and a cluster of 20 high performance workstations (SGI Power Challenge, HP9000 and IBM RS6000/590). The SP2 and the cluster are each viewed as a single parallel computer. The SP2 is a dedicated supercomputer in the traditional sense. The cluster is intended to model an ad-hoc supercomputer formed from idle workstations sitting on employees desks or dedicated workstations assembled as compute servers.

The machines are devoted primarily to computational aeroscience applications, usually in the areas of Computational Fluid Dynamics (CFD) and Structural Mechanics. These are almost always parallel applications, running on as few as 2 nodes to as many as the entire machine. By parallel application, we mean a multiple-instruction-multiple-data (MIMD) program that is designed to run on several distributed-memory processors simultaneously. For convenience we will include one-node serial jobs as a special case. A parallel job is a sequence of operations requested by a user, of which a parallel application is the main part. In the following we use "job" and "application" interchangeably. The applications may be written in a message passing style (such as MPI or PVM) or a data parallel style (using HPF).

Both computers must be able to support a wide range of job sizes, where size is determined primarily by amount of time, number of nodes, but also by memory and disk I/O use. During normal working hours ("prime time"), short debugging jobs should have fast turn around, while allowing larger jobs to run, if possible. During "non-prime time" hours, large jobs should have priority over smaller jobs. In addition, users may request exclusive access ("dedicated time") to a computer for benchmarking, or for running a grand challenge problem.

A job management system is typically composed of three main parts: a *User Server* for submitting, deleting and inquiring about the status of jobs, a *Job Scheduler* for scheduling and queueing jobs, and a *Resource Manager* for allocating, monitoring and enforcing resource allocation and policies. In the NAS environment,

The User Server should:

- provide one entry point (a virtual queue) to which all jobs are submitted.
- provide information about all queued and running jobs.

The Job Scheduler should:

- schedule jobs according to a predetermined scheduling policy.
- be highly configurable, to accommodate arbitrarily complex and changing scheduling rules (including dynamic and preemptive resource allocation).
- be able to sustain hardware or system failures - no jobs get lost (restart or rerun jobs).

The Resource Manager should:

- operate in a heterogeneous multi-computer environment. A single Resource Manager should span all the systems.
- be able to enforce resource allocation and scheduling policies.
- collect complete job's accounting information.

In the rest of this paper, we do not distinguish between the User Server, the Job Scheduler, and the Resource Manager, but combine them all under the label Job

Management System. We realize that some functionality may reside in the operating system (for instance checkpointing) and some in the Job Management System. Some parts may be implemented by the vendor. If so, parts that interact with other piece of software (OS and parallel programming tools), should have a public API.

3.0 Serial vs. Parallel Jobs

All job management systems we have encountered were originally designed for serial jobs, that is, jobs that run on a single processor. We have found two major problems with these systems.

The first we call the scheduling problem. By scheduling we mean the assignment of specific resources to a parallel application. Scheduling is done both by the JMS (i.e., by starting a process on a particular CPU at a particular time) and the operating system (i.e. timesharing several processes on one CPU). For serial applications, JMS and OS scheduling are compatible. For parallel applications they are in general not compatible because of load balance and synchronization issues that may lead to serious performance degradation.

The second problem is the "parallel aware" problem. The most obvious distinguishing feature of a parallel application is that there are multiple processes, yet most job management systems are aware of and keep track of only one process (master process). We explain in section 3.2 why this leads to chaos.

3.1 Time sharing, space sharing, and scheduling

One of the great advances in the development of computers was the concept of timesharing through preemptive multitasking. Several programs can run "simultaneously" by using the processor in alternating time slices. Timesharing makes multi-user interactive use possible and increases processor utilization with only a small performance penalty, as long as all programs fit in memory. Job management systems targeted at serial jobs can make use of operating system-provided timesharing to run more than one job "simultaneously" or can run jobs sequentially when resource requirements are large (say, for memory).

Unfortunately, parallel jobs are significantly more complicated to schedule to achieve efficient use of computational resources. One can argue that the most efficient use of a parallel computer, in terms of raw computational power, is to treat it as a collection of individual single-processor computers each running serial jobs. At NAS, however, we are interested in running jobs that cannot be run on a single node because of their large resource requirements. For instance, a large memory job cannot fit on a single node, or a CPU intensive job will take too long to run.

Today's most promising parallel computers, including the two previously described, are essentially a collection of high performance workstations, each running an independent and full version of the Unix operating system, and each with independent timesharing. While at first glance it appears that parallel jobs can be timeshared automatically by the operating system (many JMSs make this assumption) in practice timesharing does not work for applications in the NAS workload. The two issues are load balancing and synchronization.

Currently most applications at NAS are statically load balanced, assuming that each processor is equally fast so that the work should be divided evenly among them. It is almost impossible, in a timesharing environment, to maintain exact load balance among nodes. Statically balanced applications run only as fast as the slowest node, so that even a slight imbalance reduces parallel efficiency drastically. A single unbalanced node can completely ruin the performance of an entire application. In the near future, we expect to see more dynamically balanced applications, which we discuss in section 3.3.

An obvious "solution" to the load balance problem is to schedule parallel jobs on top of one another so that all nodes are slowed down equally. From a practical point of view the loads are never quite balanced, and for many applications this scheme wouldn't work even if there were perfect load balance. The reason is that uncoordinated timesharing causes synchronization and communication delays in tightly coupled applications. A rare exception is the CM-5 system, which implements coordinated timesharing across the nodes in a partition and does it quite well [Thi92].

Among the statically balanced applications are a very important class of tightly synchronized communication-intensive codes. At NAS these are often CFD solvers using implicit methods (which have global data dependencies) such as POVERFLOW [Rya93]. These applications form a large part of the NAS workload, and tightly synchronized applications are common in other fields.

The reason these applications cannot be efficiently timeshared is the accumulation of communication delays created by the uncoordinated scheduling of processes across the nodes. In tightly synchronized applications, information flows between nodes as the calculation progresses. Even when there is only nearest-neighbor communication, information flows from neighbor to neighbor eventually reaching all nodes. Every time that information flow is disrupted the entire application slows down. To take a concrete example, imagine that data flows from node 0 to node 1 to node 2. In a timeshared environment, node 1 may be running, ready to receive data from node 0, but node 0 may be running another application, so node 1 waits. When node 1 receives the data, it processes it, at half speed because of timesharing. When node 1 is ready to send data to node 2, but node 2 may not be running, so node 1 will have to wait. So node 1 is delayed by more than just the timesharing going on that node.

For a related but more familiar example, imagine a large household with as many bathrooms as children. What usually happens when leaving on a vacation is that a child delays departure by needing to use the restroom. When that child is done, the next child decides its time to go, and so on, so that the net effect is the same as if there were only one bathroom. Similarly, in the limit of a large number of timeshared processes, each one is completely serialized and N processors are no faster than 1. Of course the whole process is stochastic, and dependent on many factors, but the net effect is always that it is extremely inefficient to independently timeshare tightly synchronized parallel processes.

One solution to this problem is to coordinate timesharing across the nodes of a parallel application, so that all processes of a given application run at the same time. This is called *gang-scheduling* or *co-scheduling*. Gang-scheduling requires operating system support, and a scheme for handling communication in progress (i.e. no messages should be lost when processes are swapped). Although not necessary, it is easier to implement gang scheduling if nodes are divided into fixed-size partitions, though jobs then do not have flexibility in how many nodes they run on. Since all nodes in a partition run the same number of processes, the scheduler does not have to deal with unbalanced scheduling. Gang-scheduling in fixed-size partitions is an effective way to deal with the timesharing problem and has been successfully implemented on the CM-5 [Thi92]. It allows a scheduler to use the same techniques used for scheduling serial jobs. It can be combined with space sharing, discussed below, to provide very flexible resource allocation. Unfortunately, gang-scheduling is available on neither the SP2 nor the NAS cluster.

The alternative to time sharing is *space sharing*. With space sharing, a parallel application gets exclusive access to the nodes on which it runs. There is no time sharing, or if there is timesharing, the competing processes (such as Unix daemons) are mostly inactive. The nodes of a parallel computer are then divided among several parallel processes. Space sharing allows jobs of arbitrary size, is compatible with running a full version of Unix, requires no special hardware support and little software support, but makes scheduling more difficult for the JMS. The two primary difficulties are the tiling problem, which has to do with maximizing utilization, and the large job problem, which is that large jobs take most or all of the computational resources of a machine, preventing any other jobs from running.

3.2 Understanding "parallel"

Most job management systems were originally designed for serial jobs. While they may be able to treat "number of nodes" as just another resource, they don't really understand the *concept* of a parallel job. Typically the JMS launches a starter process, or master process, that is responsible for starting the rest of the parallel application (i.e. worker processes). With this approach, the JMS knows nothing about the worker processes, except perhaps indirectly, through something like an indication of load average. Some of the problems with this approach are:

- If the master process dies unexpectedly, worker processes may not be killed when a job finishes (normally or abnormally). Such "orphaned processes" cause severe performance problems at a minimum and in some cases (e.g. the SP2) render the machine unusable to others. This is a critical problem at NAS on both the SP2 and the cluster, and its importance cannot be overstated.
- There is no way to enforce certain resource limits (e.g. CPU time) or enforce restrictions on foreign jobs (those not created under supervision of the JMS) since the JMS has no way of knowing which processes it is responsible for.
- Accounting information is recorded only for the master process.
- It is impossible for the job manager to kill a job directly. A parallel application must be killed by or with the cooperation of the master process. When this fails (as it often does) the result is usually orphaned jobs.
- Some systems may require specific per-process initialization (e.g. switch adapter mode on the SP2).

In a parallel processing environment, the job management system must either start or be aware of all processes in a parallel application. There must be a well-defined interface between the parallel programming environment and the job management system.

3.3 Dynamic Resources and Scheduling

The parallel applications described so far are statically load balanced and request a fixed amount of resources for the lifetime of the job. These restrictions can be relaxed, introducing dynamicism into the picture.

Dynamic load balancing is easily implemented when there are a large number of independent tasks. Typically, these "embarrassingly parallel" applications are implemented with manager and worker processes using a producer-consumer paradigm. If a node becomes slow, for instance due to interference from a time-shared process, the application can shift work to other nodes, avoiding a performance penalty. For dynamically load balanced jobs, gang-scheduling is not necessary. Since these jobs can be easily timeshared, they offer more flexibility to the job management system. Debugging runs for tightly synchronized applications have similar ability to be timeshared because performance is not important. Scheduling for these applications can be performed using a scheme based on load averages. Loadleveler, Condor, and LSF are oriented towards this type of scheduling.

The possibility of being able to change resource allocation dynamically is intriguing, especially in the arena of non-dedicated clusters made up of workstations sitting on user's desks. In a non-dedicated cluster, an allocated workstation may be taken away by its owner, requiring a running application to migrate or reduce its number of processes if no other workstation is available. Reciprocally, as workstations become idle, it may be possible to add additional resources to a running application. Another important use of dynamic resource allocation is for fault-tolerance to allow a parallel job to continue after a node failure.

Any dynamically load-balanced application can handle, in principle, dynamic resource allocation. Statically load-balanced applications can sometimes make use of dynamic resource allocation, but often with a very high cost when a load is rebalanced. Effectively making use of dynamically changing resources requires cooperation between an application and the JMS, including asynchronous notification when resources become available or unavailable. The right way to do this is currently an active area of research. For example, there is work underway to define an interface between MPI and a JMS [Gro95] to support dynamic process management. The Piranha [Car95] adaptive parallelism model defines a *feeder()* and *retreat()* interface to grab and release new resources asynchronously. At a minimum, we anticipate a need for the application to be able to:

- request specific resources, such as number of nodes, amount of time they will be used for, amount of memory, type of architecture, type of timesharing (co-scheduled or independent), what type of network is needed, etc.
- acquire resources asynchronously, through a non-blocking request and asynchronous notification when some/all resources are available.
- specify whether or not it can release resources, and provide a mechanism for the JMS to take them preemptively or cooperatively.

The most recent version of PVM [PVM94] has internal support for interacting with a job manager (requests for adding/deleting a host and spawning tasks). While this is a step in the right direction, it is not general enough to be of use for NAS. For instance, there is no support for requesting specific resources or acquiring or releasing them asynchronously. Furthermore, this functionality is currently supported only by Condor [Lit88], which is tightly integrated with PVM and therefore not appropriate for other JMSs.

4.0 Requirements

This section lists requirements for a job management system for the NAS parallel systems. The primary purpose of the list is to organize and prioritize NAS requirements. A secondary purpose is to provide a checklist for evaluating and comparing existing job management systems. In some cases the requirements will need operating system or application support. Our list of requirements addresses the issues described in section 3 and tries to take into account three main factors:

- **System environment:** The system environment is an overriding constraint. Does the JMS have complete control over all processes on the system? Are resources dedicated or shared? Do interactive users log into the nodes? If interactive users are allowed, do they get priority? We believe that the job management problem can be made arbitrarily difficult by choosing a complicated environment. Instead of trying to solve all problems at once, we choose to address first a simpler environment for covering the basic needs. While several currently available packages try to address complicated environments, they were found inadequate because they don't address simple problems such as the "parallel aware" problem. A goal of this paper is to focus attention on basic requirements for parallel systems.

- **Usefulness:** Next important is the usefulness of a given requirement. Some requirements are absolutely necessary. The lack of those makes a system difficult to manage, and potentially useless. Lack of robust protection against runaway orphan processes on the SP2 is an example. Other requirements we anticipate will become necessary to support more complex parallel computing environments (e.g. networks of non-dedicated workstations). Other features are interesting, but their usefulness is uncertain (job migration between different architectures).
- **Difficulty to implement:** The last factor we take into consideration is whether a requirement is easy or difficult to implement. There is little to gain by requiring something immediately that is difficult to implement. We would rather that difficult features (for instance checkpointing, job migration, gang scheduling) be carefully and correctly implemented.

We decompose the requirements into three phases, taking into account the three factors just described. **Phase I** requirements are absolute requirements for a JMS to be useful to NAS. All requirements included in Phase I exist or can be implemented without any great advances. They assume dedicated, static resources with space sharing but no timesharing, either co-scheduled or independent. **Phase II** requirements relax these assumptions, assuming non-dedicated resources over which the JMS has a large amount of control. **Phase III** requirements relax the assumptions further, fully integrating all types of parallel applications and traditional timesharing.

4.1 Phase I Requirements

4.1.1 System Model

In phase I, we target a dedicated parallel system where the job manager has complete control. Individual nodes are dedicated to a single parallel application, so that timesharing is not an issue. The job manager schedules jobs using space sharing.

A user should be able to run a parallel job by specifying the number of nodes (fixed throughout the life of the application), a maximum amount of time, and possibly other resources (e.g., a certain amount of memory, access to a fast file system, etc.). Jobs will be scheduled in a "fair" way determined by the system administrator.

4.1.2 Definitions and assumptions

1. The principal resources are dedicated (cpu, memory, and possibly local disks). If a user gets a node, he/she gets complete control of a node and can assume that nothing else is competing for that node's resources (except possibly for a small amount of OS activity). On most systems, access to the interconnect network and remote file systems will remain available to other users. This is the only simple way to make sure that the load is balanced and that communication-intensive jobs can run efficiently without gang-scheduling. Furthermore, this makes it considerably easier for the JMS to kill a job and clean up afterwards (e.g. every process not owned by root may be fair game to kill).

The definition of a job is therefore node-based — the JMS does not need to be aware of individual processes. When there is gang-scheduling or timesharing (see phases II and III), the job definition must be process based, which is more difficult for a JMS to handle. A process-based definition is possible for phase I, but not required.

2. Resources are static. The size of a parallel job remains fixed for the life of a job. This avoids the complicated area of interaction between the resource manager and the application.

3. The JMS has complete control over resource allocation. When dealing with a large number of users, it becomes critical to be able to enforce a predetermined scheduling policy (usually by preventing, killing or suspending foreign jobs), so that each user can get a fair chance to run his or her jobs. A serious weakness of current batch systems is their inability to enforce such a policy.

4. The job may be based on any standard parallel programming model. At NAS, we support HPF, MPI and PVM. Certain restrictions below don't allow unrestricted use of all PVM features. Note that implementations of MPI, PVM, and HPF must provide hooks for a JMS or this requirement will be impossible to fulfill. Some current implementations (e.g. MPI and HPF on workstation clusters) do not provide enough information for a JMS to be able to clean up cleanly in a process-based environment.

5. There is no system-supported fault tolerance, user level checkpointing, or job migration. If a node dies, the application running on it dies and we assume there is no way to recover except to kill the rest of the job cleanly. All of these are extremely difficult issues, especially for parallel jobs, which is why we omit them from phase I despite their importance. However, if requested by the user, a job can be restarted from the beginning.

6. Computational resources are essentially homogeneous. A user application should be able to assume that equally partitioning work will result in a balanced load. Special resources may exist on certain nodes if requested by a user. This applies as well to machines that appear to be homogeneous, but may not be, such as an SP2 (with different types of nodes).

4.1.3 Scheduling policy

A JMS should be able to implement arbitrarily complex scheduling rules. Fair and efficient scheduling (including site-specific policies and efficient tiling) require sophisticated rules that can change frequently and can't be easily captured by a monolithic set of options. The tiling problem and large job problem mentioned in section 3.2 are also difficult and have no unique solutions.

At a very minimum, we distinguish between three types of jobs — interactive, batch and foreign. The JMS should be able to distinguish between these types of jobs and schedule them appropriately.

- **Interactive Jobs**

Interactive jobs are those that require fast turn-around (e.g. for debugging) with their input/output connected to a terminal. Interactive jobs would usually have small resource requirements, so they can run immediately or in a short period of time.

- **Batch Jobs**

Batch jobs require a larger resource allocation. They will usually run overnight, so that results can be retrieved the next day. Their output is sent to a file rather than the terminal.

- **Foreign Jobs**

These jobs are created outside the JMS. The JMS should be able to track, control and possibly kill foreign jobs according to the current system administration policy. Since the definition of a job in phase I is node based rather than process based. The JMS needs only to be able to distinguish between jobs belonging to the "owner" of a node and all others.

Even these fairly simple categories can involve complicated rules for which a built-in menu of options is unlikely to be sufficient. For instance, guaranteeing interactive availability in a space-sharing environment while maintaining high utilization is a complex task for which we do not have a perfect solution and for which we would not want a JMS to impose a solution. However, we require the JMS scheduler to be highly configurable, to accommodate arbitrarily complex and changing scheduling policies.

This is not to say that the JMS should do nothing. Ideally we envision a scriptable scheduler interface, where the JMS should provide to the script information such as:

- Requested resources and available resources (see section 4.1.5).

- job type (interactive, batch, short debugging, etc.).
- job, user and group priority.
- Time of day.
- Current jobs running, current jobs queued, recent history of resource use by current users.
- Load average (in phases II and III).

We don't want to rule out the possibility of scheduling done entirely by the JMS (without a scriptable interface), but we would be surprised if a sufficiently robust scheduler could be written that could handle all cases. Of course, even a scriptable interface should come with a default scheduler that can handle a lot of interesting cases.

4.1.4 Resources

The JMS should be capable of allocating nodes based on requests for certain resources. In addition to the absolutely essential resources of number of nodes and wall clock time, these include:

- Number of nodes (currently available).
- Wall clock time.
- Node type (compute, I/O node, big memory, multiprocessor node).
- Disk Usage (local disk, system disk, swap space, etc.).
- Network connections (HiPPI, FDDI, ethernet, etc.).
- System specific resources (e.g. switch adapter mode on the SP2).

4.1.5 Resource Limits

The JMS must be able to enforce resource limits. In phase I, we require the JMS to inform the user why a job is rejected or terminated. Before it kills a job for exceeding resource limit, the JMS should send a predefined "warning" signal to the application to give it a chance to abort cleanly. It is, however, the responsibility of the application to catch this signal and abort cleanly.

The JMS should be able to enforce (both at submission time and when a job is running) limits on:

- Number of jobs running (user and group limits).
- Number of nodes.
- Type of nodes (I/O, HiPPI, or multiprocessor).
- Wall clock time (phase I) and CPU time per node or application (phases II and III).
- Disk Usage.
- Dedicated access.

As part of its police duties, a JMS should maintain a clean system. In particular, it should be able to detect orphaned tasks and foreign jobs and clean them up by suspending or killing them. In our experience so far, lack of ability to do this is the single biggest problem we have encountered. In the phase I dedicated-node environment, such detection and cleanup should be quite simple to implement and requires no cooperation from an application.

When appropriate the JMS should support both hard limits and soft limits. A job is always killed when it exceeds a hard limit. When a job exceeds a soft limit, it may be killed, but the JMS can decide not to kill it if the resources it is using are not requested by another process.

4.1.6 Accounting

The JMS should collect the following data about each job:

- User name, group and job name.
- Job submission time.
- Job starting and ending time.
- User and system time per node.
- All resources requested and allocated (e.g. number of nodes).
- Job return status (i.e. completed, killed, suspended).

4.1.7 Programming System Support

The JMS should support and interact with the following parallel programming systems.

- MPI
- PVM
- HPF

In phase I, the JMS needs to be aware at least of a master process, but not necessarily the entire application. For phases II and III, we require that all processes of a parallel application be created by or registered with the JMS. The creation or registration of process will be done via some well-defined interface between the JMS and the application.

4.1.8 Administration

The administration of the JMS should be centralized. All configuration files and log files should be maintained in one location. The administrator should be able to dynamically reconfigure system resources with minimal impact on running jobs. For instance, it should be possible to add a new system to a pool without suspending or killing jobs. During a system shutdown, in Phase I and II running jobs will be lost if they have not been checkpointed by the user. However, if requested by the user, a job may be restarted from the beginning. For Phase III, the JMS should be able to checkpoint jobs (relying on OS checkpointing) for future restart.

Administrator's prologue and epilogue scripts should be able to run before and after each user's job (e.g. for cleaning /tmp, creating additional accounting, or performing security functions).

4.1.9 User Interface

It is important that the JMS provide to users sufficient and clear information about the status of system resources and jobs.

- Status of all system resources (idle, reserved, available, down).
- Job status (queued, running, suspended, killed) - Detailed information should be available to explain to users why a particular job is queued (for example not enough nodes available), or was killed.
- Information about the consumed and remaining resource available to a job.

Users should be able to reliably kill their own job.

Interactive jobs should run with standard input and output connected to a terminal. We have also found it useful to be able to "detach" an interactive job after verifying that it has started correctly [DJM93]. In this case, output must be logged to a file as well. While we are not requiring this for Phase II, it should be available for Phase II.

4.2 Phase II requirements

4.2.1 System Model

Phase II targets a transitionary architecture, somewhere between the dedicated parallel computer model of phase I and the network of workstations in phase III. In phase II, we introduce new requirements that allow for more dynamicism and fault tolerance, but not with the full range of features in phase III.

The main difference in Phase II is that resources are not necessarily dedicated, so that the job manager must track individual processes. While this seems like a small step, it introduces several new requirements and challenges. Some current JMS systems already track jobs on an individual basis, but do not satisfy the more important requirements of phase I.

From a user's point of view, the additional functionality in phase II is the ability to have time shared pools of nodes for debugging (where performance isn't important) or applications that can do dynamic load balancing. In addition, phase II includes limited support for user level checkpointing (with cooperation from the user application), with automatic restart and support for dynamic resource allocation.

4.2.2 Definitions and assumptions

1. In addition to the dedicated resources from phase I, portions of a parallel computer may be set aside for timesharing (independent scheduling).

2. The definition of a parallel job must now be process based, rather than node based, in order to manage multiple jobs on a node. The JMS should be aware of all processes associated with parallel jobs (interactive or batch) as well as foreign jobs. The JMS should be able to do accounting at the process level.

In any case, the JMS should be aware of any jobs created under its supervision, so that it can kill an entire parallel application (not just a master process) if something goes wrong (such as a node dies), or if a job exceeds its allocated resources. Alternatively, if an application declares that it is fault-tolerant, the JMS needs to take special actions to automatically restart the job from the last checkpoint or from the start.

4.2.3 Dynamic Resource Allocation

The JMS should support dynamic resource allocation (e.g. changes in the number of nodes of a parallel job). For instance, it should be possible for a job to request additional resources asynchronously, to release some of its current resources, and for the JMS to preemptively take resources from a job. Obviously this requires application level support, and an interface between the parallel programming system and the JMS (as described in section 3.3). While few applications are currently structured to be able to handle dynamicism, allowing it introduces considerable flexibility in scheduling algorithms to maximize resource utilization and throughput.

4.2.4 User Level Checkpointing

The JMS should provide support for checkpoint/restart at the user level. An application will have the opportunity to checkpoint its state periodically. The JMS should have a well-defined interface to facilitate checkpoint-restart.

4.2.5 Heterogeneous Environment

The JMS should be able to operate in a heterogeneous environment. In particular, it should be able to allocate and schedule resources on machines from different vendors using a single point of contact. Complex scheduling cost analysis and trade-off between various systems are left for Phase III.

4.2.6 Job Inter-dependency

Users should be able to express dependencies between a set of jobs. The JMS scheduler must be aware of these dependencies while scheduling the jobs. Types of dependencies are:

- Job State (queued, running, etc.).
- Job Return Status (Success, Failure).
- Job Submission time.

4.2.7 Resource requests and limits, accounting

With timesharing, the following resources become important. The JMS should be able to allocate and limit resources based on:

- CPU time (per node or total).
- Memory use.
- Swap space.
- Disk usage.

In phase I, only wall clock time was relevant on dedicated nodes. CPU time was either redundant or misleading. An application waiting on communication would not use any CPU time but would still prevent any other job from using the dedicated resources. In a timesharing environment, wall clock time has a lesser meaning for the purposes of accounting, and CPU time is a better measure.

Also with the introduction of timesharing, memory use becomes more important. It is crucial for most applications (unless performance and efficiency are of no importance) to avoid swapping. Some applications may be able to use swapping without performance hit. Avoiding swapping is more important when there is timesharing because one large process can ruin performance for all the others. A JMS should be able to schedule jobs so that they do not exceed available memory and should be able to kill off jobs that exceed their memory request.

In a timesharing environment, it is expected that the primary method of load balancing used by the JMS will be to schedule according to load average. The JMS must therefore have some mechanism to determine load average. In certain environments, it might be acceptable not to measure load average directly but only to keep track of the number of processes on a node.

4.3 Phase III Requirements

4.3.1 System Model

Phase III targets a fully production system, allowing production parallel applications to run in an environment that includes workstations sitting on people's desks. It requires the solution of difficult problems in computer science before it can be implemented and certainly requires support at the operating system and communication library level. Many of the operating system issues are being addressed by the NOW project at Berkeley [Pat94].

The major developments required for phase III are gang-scheduling and automatic checkpointing, restart, and migration.

4.3.2 Definitions and Assumptions

1. Gang scheduling is available. A fixed size partition may be setup to easily implement gang scheduling.

Gang scheduling is necessary to allow efficient execution of communication-intensive applications, combined with reasonable response and throughput. We have in mind the CM-5 model. It may turn out that gang-scheduling is possible or efficient only on dedicated parallel computers unless a large time slice is used. The use of a smaller time slice will require a low latency network. It is not yet clear whether it can be useful on networks of non-dedicated workstations.

2. There is robust support for checkpointing, restart from checkpoint, and migration (probably implemented as checkpoint-restart). This is crucial to provide fault tolerance and to return resources back to interactive workstation users.

4.3.3 Scheduling

The JMS may need to understand and interact with the gang-scheduler to be able to schedule jobs appropriately. For example, gang-scheduling is not necessary for dynamically balanced jobs.

To make effective use of checkpoint-restart, a JMS will have to reschedule or migrate an application when a node or other resource becomes unavailable.

4.3.4 OS Level Checkpointing

OS level checkpointing of serial jobs is a difficult task. To our knowledge, very few systems support robust OS checkpointing. Checkpointing of parallel jobs is even more complicated both because there is always network activity (what should be done with messages in transit?) and because it can be quite difficult to capture a well-defined "state" in a loosely synchronous parallel application. OS level checkpointing clearly involves cooperation between the operating system, the communication library, and probably the application itself. While OS level checkpointing is quite difficult, we have found it to be an important element of a production supercomputing environment of the type found on a "traditional" supercomputer such as the Cray C90.

Checkpointing is essential for implementing fault tolerance and for migrating jobs from machines that must be used for some other purpose (e.g., a workstation owner reclaims her/his workstation or there is a higher priority job).

4.3.5 Job Migration

In a non-dedicated cluster environment, it should be possible to migrate a job from a workstation which has to be returned to its owner. To do this robustly requires OS level checkpointing, support from the communication layer, and may require moving data files used by the job. In an heterogeneous environment, it should be possible to migrate a job through a combination of user-level checkpointing and restart. Of course this will make sense only for certain types of applications.

In order to maximize throughput, the JMS scheduler should be able to migrate jobs as new resource become available. For instance, removing a large memory job from a busy system into an empty one.

5.0 Conclusions

A robust job management system is an essential component of a production parallel supercomputer. Current job management systems, designed originally for serial jobs or without awareness of the special requirements for parallel applications, do not work well in the NAS environment. By providing a detailed list of requirements for a job management systems, we hope to focus attention on the basic issues and make sure that these are covered before more adventurous projects are attempted. We can wait for some time until difficult problems are solved, but we have immediate needs for a JMS that will solve the simple ones. Phase I requirements are a minimum to have a usable dedicated system, and Phase II will be necessary to start supporting a production workload. Phase III requirements are important for a full production system, comparable to the production environment found on traditional vector supercomputers at NAS.

We are working with the designers of the Portable Batch System [PBS95] developed at NAS and Livermore National Lab to make sure our requirements are addressed in their design.

6.0 References

[Car95] "Adaptive Parallelism and Piranha", N. Carriero, E. Freeman, D. Gelernter and D. Kaminsky, IEEE Computer, January 1995.

[DJM93] "Distributed Job Manager Administration Guide" AHPCRC, *Minnesota Supercomputer Center*, 1993.

[Duk94] "Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System Version 3.0," D. Duke, T. Green, J. Pasko, *Supercomputer Computations Research Institute*, Florida State University, March, 1994.

[Gro95] "Dynamic Process Management in an MPI Setting", W. Gropp and E. Lusk, draft report, ANL, 1995.

[IBM94] "IBM Loadleveler Administration and Installation Guide", Document Number SH26-7220-02, *IBM Kingston Research Center*, January 1994.

[Kin86] "The Network Queuing System," B.A. Kinsbury, *Cosmic Software*, NASA Ames Research Center, 1986.

[Lit88] "Condor: A Hunter of Idle Workstations", M. Litzkow, M. Livny and M. Mutka, *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, June 1988.

[Pat94] "GLUnix: A New Approach to Operating Systems for Networks of Workstations." D. Patterson and T. Anderson, *Proceedings of the First Workshop on Networks of Workstations*, San Jose, October 1994.

[Pla94] "LSF: Load Sharing Facility Administrator's Guide", *Platform Computing Corporation*, December 1994.

[PBS95] "Portable Batch System: External Reference Specification", Revision 1.4, NAS, NASA Ames Research Center, January 1995.

[PVM94] "PVM3 Users Guide and Reference Manual," Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, Oak Ridge National Lab TM-12187, September, 1994.

[Rya93] "Parallel Computation of 3-D Navier-Stokes Flowfields for Supersonic Vehicles", J.S. Ryan and S.K. Weeratunga, *AIAA 93-0064*, Jan. 1993.

[Thi92] "Connection Machine CM-5 Technical Summary", *Thinking Machines Corporation*, November 1992.

[Zho93] "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," S. Zhou, X. Zheng, J. Wang, and P. Delisle, *Software- Practice and Experience*, Vol. 23, December 1993.