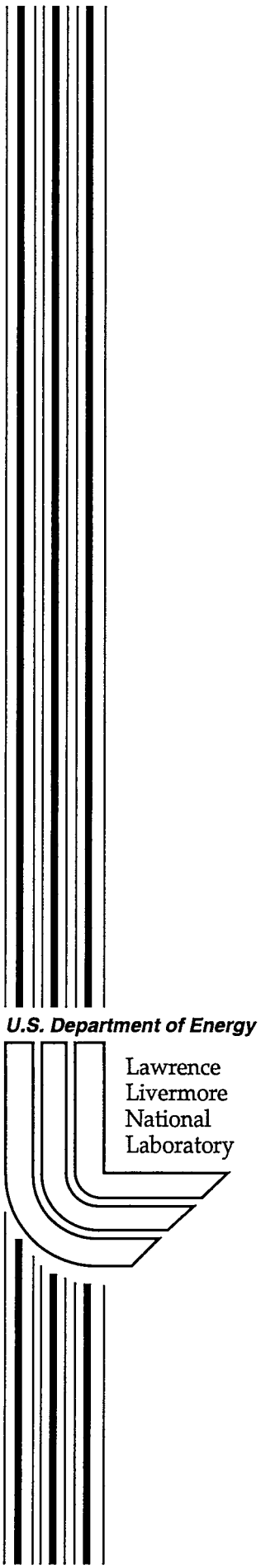
UCRL-MA-147996

SLURM: Simple Linux Utility for Resource Management

M. Jette, C. Dunlap, J. Garlick, and M. Grondona

April 24, 2002



DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

SLURM: Simple Linux Utility for Resource Management

Moe Jette, Chris Dunlap, Jim Garlick, Mark Grondona
{jette,cdunlap,garlick,grondona}@llnl.gov

April 24, 2002

Abstract

Simple Linux Utility for Resource Management (SLURM) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters of thousands of nodes. Components include machine status, partition management, job management, and scheduling modules. The design also includes a scalable, general-purpose communication infrastructure. Development will take place in four phases: Phase I results in a solid infrastructure; Phase II produces a functional but limited interactive job initiation capability without use of the interconnect/switch; Phase III provides switch support and documentation; Phase IV provides job status, fault-tolerance, and job queuing and control through Livermore's Distributed Production Control System (DPCS), a meta-batch and resource management system.

1 Overview

SLURM[1] (Simple Linux Utility for Resource Management) is a resource management system suitable for use on Linux clusters, large and small. After surveying[1] resource managers available for Linux and finding none that were simple, highly scalable, and portable to different cluster architectures and interconnects, the authors set out to design a new system.

The result is a resource management system with the following general characteristics:

- *Simplicity*: SLURM is simple enough to allow motivated end users to understand its source code and add functionality. The authors will avoid the temptation to add features unless they are of general appeal.
- *Open Source*: SLURM is available to everyone and will remain free; its source code is distributed under the GNU General Public License.
- *Portability*: SLURM is written in the C language, with a GNU autoconf configuration engine. While initially written for Linux, other UNIX-like operating systems should be easy porting targets.
- *Interconnect independence*: Initially, SLURM supports UDP/IP based communication and the Quadrics Elan3 interconnect. Adding support for other interconnects is straightforward. Users select the supported interconnects at compile time via GNU autoconf.
- *Scalability*: SLURM is designed for scalability to clusters of thousands of nodes. Prototypes of SLURM components thus far developed indicate that the controller for a cluster with 16k nodes will occupy less than 1 MB of memory and performance will be excellent.[2]
- *Fault tolerance*: SLURM can handle a variety of failure modes without terminating workloads, including crashes of the node running the SLURM controller.
- *Secure*: SLURM employs crypto technology to authenticate users to services and services to services. A Kerberos v5 infrastructure can be utilized if available. SLURM does not assume that its networks are physically secure, but does assume that the entire cluster is within a single administrative domain with a common user base across the entire cluster.
- *System administrator friendly*: SLURM is configured with a few simple configuration files and minimizes distributed state. Its interfaces are usable by scripts and its behavior is highly deterministic.

[1] A tip of the hat to Matt Groening and creators of *Futurama*, where Slurm is the highly addictive soda-like beverage made from worm excrement.

[2] It is anticipated that a Linux cluster of size > 1000 nodes will be available for testing before the initial public release.

1.1 What is SLURM?

As a cluster resource manager, SLURM has three key functions. First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work.

Users interact with SLURM through three command line utilities: `srun` for submitting a job for execution and optionally controlling it interactively; `scancel` for early termination of a job; and `squeue` for monitoring job queues and basic system state.

System administrators perform privileged operations through an additional command line utility: `scontrol`.

External schedulers and meta-batch systems can submit jobs to SLURM and order its queues through an application programming interface (API).

Compute nodes simply run a `slurmd` daemon (similar to a remote shell daemon) to export control to SLURM. The central controller daemon, `slurmctld`, maintains the global state and directs operations.

1.2 What SLURM is Not

SLURM is not a sophisticated batch system. Its default scheduler implements First-In First-Out (FIFO) with backfill and is not intended to directly implement complex site policy. SLURM does however provide a sufficiently sophisticated API for an external scheduler or meta-batch system to order its queues based upon site policy.

SLURM clusters are space shared, and parallel jobs are always assigned whole nodes. Multiple jobs may be allocated the same node(s) if the administrator has configured nodes for shared access and/or the job has requested shared resources for improved responsiveness. SLURM does not support gang scheduling (time-slicing of parallel jobs). However, the explicit preemption and later resumption of a job under the direction of an external scheduler may be supported in the future. At present, an external scheduler may terminate jobs, hold jobs, and requeue them in any desired order order via the API.

SLURM is not a meta-batch system like Globus or DPCS (Distributed Production Control System). SLURM supports resource management across a single cluster.

SLURM is not a comprehensive cluster administration or monitoring package. While SLURM knows the state of its compute nodes, it makes no attempt to put this information to use in other ways, such as with a general purpose event logging mechanism or a back-end database for recording historical state. It is expected that SLURM will be deployed in a cluster with other tools performing these functions.

1.3 Architecture

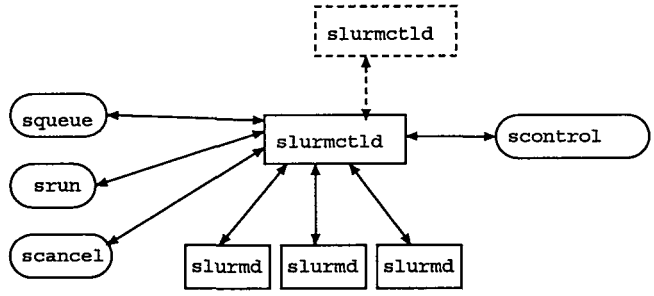


Figure 1: SLURM Architecture

As depicted in Figure 1, SLURM consists of a `slurmd` daemon running on each compute node, a central `slurmctld` daemon running on a management node (with optional failover twin), and a four command line utilities: `srun`, `scancel`, `squeue`, and `scontrol`, which can run anywhere in the cluster. A scalable communications library ties these components together.

Figure 2 exposes the subsystems that are implemented within the `slurmd` and `slurmctld` daemons. These subsystems are explained in more detail below.

1.3.1 Slurmd

The `slurmd` running on each compute node can be compared to a remote shell daemon: it waits for work, executes the work, returns status, then waits for more work. It also asynchronously exchanges node and job status with the controller. It never communicates with other compute nodes and the only job information it has at any given time pertains to the currently executing job.

`slurmd` reads its configuration from a file: `/etc/slurmd.conf` and has three major components:

- *Machine and Job Status Service*: Respond to controller requests for machine and job state information, and send asynchronous reports of some state changes (e.g. `slurmd` startup, job termination) to the controller. Job status includes CPU and real-memory consumption information for all processes including user processes, system daemons, and the kernel.
- *Process Manager*: Start, monitor, signal, and clean up after a set of processes belonging to a parallel job, as dictated by the controller. Starting a process may include executing a prolog script, setting process limits, setting real and effective user id, setting environment variables, setting current working directory, allocating interconnect resources, setting core paths, and managing process groups. Terminating a process may include terminating all members of a process group and executing an epilog script.

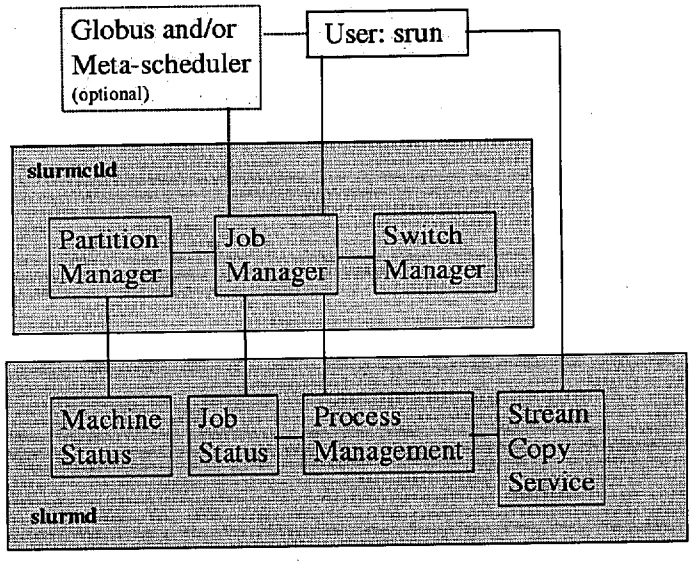


Figure 2: SLURM Architecture - Subsystems

- *Stream Copy Service*: Allow stderr/stdout/stdin and core files to be copied in and out of the spool directory for a job. In the case of batch jobs, this will happen during startup and cleanup only and will occur between the controller and `slurmd`'s. In the case of interactive jobs, stdout/stderr may additionally be "carbon copied" to the `srun` command during job execution.

1.3.2 Controller

Most SLURM state information exists in the controller, `slurmctld`. When `slurmctld` starts, it reads its configuration from a file: `/etc/slurmctld.conf`. It also can read additional state information from a checkpoint file left by from a previous `slurmctld`. `slurmctld` runs in either master or standby mode, depending on the state of its failover twin, if any.

`slurmctld` performs several tasks simultaneously:

- *Partition Manager*: Monitors the state of each node in the cluster. It polls `slurmd`'s for status periodically and receives state change notifications

from `slurmd`'s asynchronously. The partition manager groups nodes into non-overlapping sets called *partitions*. Each partition can have associated with it various job limits and access controls. The partition manager also allocates nodes to jobs based upon node and partition states and configurations. Requests to initiate jobs come from the Job Manager. `scontrol` may be used to administratively alter node and partition configurations.

- *Job Manager*: Accepts user job requests and (if applicable) places pending jobs in a priority ordered queue. By default the job priority will be a simple age based algorithm providing FIFO ordering. An interface is provided for an external scheduler to establish a job's initial priority and API's are available to alter this priority through time for customers wishing a more sophisticated scheduling algorithm. The job manager is awakened on a periodical basis and whenever there is a change in state that might permit a job to begin running, such as job completion, job submission, partition *up* transition, node *up* transition, etc. The job manager then makes a pass through the job queue and starts jobs until a resource allocation fails. When a resource allocation failure occurs, the Job Manager establishes a time when the job might be expected to begin execution. Lower priority jobs in the queue will be allocated resources on that partition only if they will complete prior to the expected initiation time of the higher priority job (backfill scheduling) or utilize resources not required by a higher priority job (e.g. another partition or less capable nodes). After completing the scheduling cycle, the job manager's scheduling thread sleeps. Once a job has been allocated resources, the job manager transfers necessary state information to those nodes and commences its execution. Once executing, the job manager will monitor and record the job's resource consumption (CPU time used, CPU time allocated, and real memory used) in near real-time. When the job manager detects that all nodes associated with a job have completed their work, it initiates cleanup and performs a scheduling cycle as described above.

- *Switch Manager*: Monitors the state of interconnect links and informs the partition manager of any compute nodes whose links have failed. The switch manager can be configured to use Simple Network Monitoring Protocol (SNMP) to obtain link information from SNMP-capable network hardware. The switch manager configuration is optional; without one, SLURM simply ignores link errors.

1.3.3 Command Line Utilities

The command line utilities primarily interact with the controller. The utilities find the host:port of the controller by reading a configuration file: `/etc/slurm.conf`. They authenticate to the controller using a method selected at compile time, initially either Kerberos v5 or ...?

- `scancel`: Cancel a running or a waiting job, subject to authentication. This command can also be used to sent an arbitrary signal to all processes associated with a job on all nodes.
- `scontrol`: Perform privileged administrative commands such as draining a node or partition in preparation for maintenance. It must be run as the user `root`.
- `squeue`: Display the queue of running and waiting jobs. `squeue` can also display a summary of partition and node information.
- `srun`: Submit a job for execution. `srun` may run in either interactive or batch mode. `srun`'s standard input (if it is a file) is copied to the controller, which copies it to `slurmd`'s[3] After job submission, batch `srun` terminates, while interactive `srun` establishes connections with `slurmd`'s to get standard output and error of the tasks in real time, and responds to signals from the user.[4]

1.3.4 Communications Layer

SLURM uses the LLNL developed communications library known as `Mongo`[5]. `Mongo`'s API closely resembles Berkeley sockets. It is built upon the UDP protocol with algorithms providing better performance than TCP, particularly in the event of high network congestion or a high failure rate in message transmission.

Need more details here.

1.3.5 Security

SLURM has a simple security model: Any user of the cluster may submit parallel jobs to execute and cancel his own jobs. Any user may view all SLURM configuration and state information. Only the user `root` may modify SLURM configuration or cancel any job. If permission to modify SLURM configuration without a root account is required, set-uid programs may be used to grant specific permissions to specific users.

`Slurmctld` requires some means of insuring that a request to initiate or cancel a job for some user was in fact initiated by that user....

Need more details here.

Access to some partitions is restricted via a key. This may be used, for example, to provide specific external schedulers with exclusive access to partitions. Individual users will not be permitted to directly submit jobs to such a

[3] `srun` command line options select the stdin handling method such as broadcast to all tasks, or send only to task 0.

[4] From the controller's point of view there is no difference between batch and interactive jobs; an interactive `srun` may detach from its job, leaving it to continue running as though submitted in batch mode. It is also possible for `srun` to attach to a job interactively that was submitted in batch mode, subject to authentication.

[5] Identify location of Mongo documentation here

partition, which would prevent the external scheduler from effectively managing it. This key will be generated by *WHAT* and provided to user `root` upon demand. The external scheduler, which must run as user `root` to submit jobs on the behalf of other users, will submit jobs using this key.

1.4 Example: Executing a Batch Job

A user wishes to run a job in batch mode, in which `srun` will return immediately and the job will execute "in the background" when resources are available.

The job is a two-node run of *mping*, a simple MPI application. The user submits the job:

```
srun --batch --nodes 2 --nprocs 2 1 mping 1 1048576
```

The `srun` command authenticates the user to the controller and submits the job request. As stdin is not a file, it is not copied to the controller. The request includes the `srun` environment, current working directory, and command line option information.

The controller consults the partition manager to test whether the job will will ever be able to run. If the user has requested a non-existent partition, more nodes than are configured in the partition, a non-existent constraint, etc., the partition manager returns an error and the request is discarded. The failure is reported to `srun` which informs the user and exits:

```
srun: request will never run
```

On successful submission, the controller assigns the job a unique *slurm id*, adds it to the job queue and returns the slurm id to `srun`, which reports this to user and exits, returning success to the user's shell:

```
srun: tinymem-42
```

The controller awakens the job manager which tries to run jobs starting at the head of the job queue. It finds *tinymem-42* and makes a successful request to the partition manager to allocate two nodes from the *tinymem* partition: *dev6* and *dev7*.

The job manager sends a copy of the environment, current working directory, command path, command arguments, interconnect info, etc. to the `slurmd`'s running on *dev6* and *dev7*. The `slurmd`'s establish the environment and execute the command as the submitting user. Stdout and stderr are redirected to files on the compute nodes:

```
/var/spool/slurm/tinymem-42/stdout.[mpirank]
/var/spool/slurm/tinymem-42/stderr.[mpirank]
```

The job manager continues trying to initiate jobs until it cannot, then sleeps. Meanwhile, on *dev6*, `/var/spool/slurm/tinymem-42/stdout.0` accumulates the application's output:

```
1 pinged   0:         1 bytes      5.38 uSec     0.19 MB/s
1 pinged   0:         2 bytes      5.32 uSec     0.38 MB/s
1 pinged   0:         4 bytes      5.27 uSec     0.76 MB/s
1 pinged   0:         8 bytes      5.39 uSec     1.48 MB/s
...
1 pinged   0:   1048576 bytes   4682.97 uSec   223.91 MB/s
```

When all tasks complete, the `slurmd`'s on the two compute nodes notify the job manager, which changes the job status to *stage_out* and begins cleanup. It retrieves the stdout/stderr spool files from the stream copy service of each `slurmd` and merges them into a single report, which it stores in the user's `srun` directory. It directs the slurmd daemon on each of the nodes formerly assigned to the job to execute the epilog commands (if any). Finally, the job manager releases resources and changes the job's state to *complete*. The records of a job's existence will eventually be purged.

1.5 Example: Executing an Interactive Job

A user wishes to run the same job in interactive mode, in which `srun` will block while the job executes and stdout/stderr of the job will be copied onto stdout/stderr of `srun`.

The user submits the job, this time requesting an interactive run:

```
srun --nodes 2 --nprocs 2 1 mping 1 1048576
```

The `srun` command authenticates the user to the controller and executes the same steps described in the batch example, except `srun` does not terminate upon successful submission. Instead, it receives a list of hostnames and a *nonce* used to authenticate to the `slurmd`'s. After the job startup, `srun` has no further interaction with the controller.

`srun` then opens connections to the `slurmd`'s and copies stdout/stderr of each task to stdout/stderr of `srun`. These streams are also spooled on the node and will be collected, merged, and stored in the user's `srun` directory on completion.

The user sees the output of task 0 on stdout of `srun`:

```
1 pinged   0:         1 bytes      5.38 uSec     0.19 MB/s
1 pinged   0:         2 bytes      5.32 uSec     0.38 MB/s
1 pinged   0:         4 bytes      5.27 uSec     0.76 MB/s
1 pinged   0:         8 bytes      5.39 uSec     1.48 MB/s
...
1 pinged   0:   1048576 bytes   4682.97 uSec   223.91 MB/s
```

When the job terminates, `srun` receives an EOF on each stream and closes them, gets the job exit status from `slurmd`'s, and terminates.

If a signal is received by `srun` while the job is executing (for example, a SIGINT resulting from a Control-C), it is sent to each `slurmd` which terminates the individual tasks and reports this to the job status manager, which cleans up the job.

2 Controller Design

The controller will be modular and multi-threaded. Independent read and write locks will be provided for the various data structures for scalability. The controller state will be saved to disk immediately upon change for fault tolerance. The controller will include the following subsystems:

- *Partition management*: Monitor and record the state of each node in the cluster. Group these nodes into disjoint sets called *partitions* with various job limits and access controls. The partition manager also allocates nodes to jobs based upon node and partition states and configurations.
- *Job management*: Accept, initiate, monitor, delete and otherwise manage the state of all jobs in the system. This includes prioritizing pending work.
- *Switch management*: Perform any interconnect-related monitoring and control needed to run a parallel job.

Each of these subsystems is described in detail below.

2.1 Partition Management

The partition manager will monitor the state of nodes and allocate these resources to jobs selected by the Job Manager. Node information that we intend to monitor includes:

- Count of processors on the node
- Size of real memory on the node
- Size of temporary disk storage
- State of node (RUN, IDLE, DRAINED, etc.)
- Weight (preference in being allocated work)
- Feature (arbitrary description)
- IP address

The SLURM administrator could at a minimum specify a list of system node names using a regular expression (e.g. "NodeName=linux[001-512] CPUs=4 RealMemory=1024 TmpDisk=4096 Weight=4 Feature=Linux"). These values for CPUs, RealMemory, and TmpDisk would be considered the minimal node configuration values which are acceptable for the node to enter into service. The slurmd will register whatever resources actually exist on the node and this will be recorded by the Partition Manager. Resources will be checked on slurmd initialization and periodically thereafter. If a node registers with less resources than configured, it will be placed in DOWN state and the event will be logged. Otherwise the actual resources reported will be used as a basis for scheduling

(e.g. if the node has more RealMemory than recorded in the configuration file). Note the regular expression node name syntax permits even very large heterogeneous clusters to be described in only a few lines. In fact, a smaller number of unique configurations provides SLURM with greater efficiency in scheduling work.

The weight is used to order available nodes in assigning work to them. In a heterogeneous cluster, more capable nodes (e.g. larger memory or faster processors) should be assigned a larger weight. The units are arbitrary and should reflect the relative value of that resource. Pending jobs will be assigned the least capable nodes (i.e. lowest weight) which satisfy their requirements. This will tend to leave the more capable nodes available for those jobs requiring those capabilities.

The feature is an arbitrary string describing the node, such as a particular software package, file system, or processor speed. While the feature does not have a numeric value, one might include a numeric value within the feature name (e.g. "1200MHz" or "16GB_Swap"). If the nodes on the cluster have disjoint features (e.g. different "shared" file systems), one should identify these as features (e.g. "FS1", "FS2", etc.). Programs may then specify that all nodes allocated to it should have the same feature, but that any of the specified features is acceptable (e.g. "Feature=FS1—FS2—FS3").

The partition manager will identify groups of nodes to be used for execution of user jobs. Data to be associated with a partition will include:

- Name
- Access controlled by key granted to user root (to support external schedulers)
- List of associated nodes (may use regular expression)
- State of partition (UP or DOWN)
- Maximum time limit for any job
- Maximum nodes allocated to any single job
- List of groups permitted to use the partition (defaults to ALL)
- Shared access (YES, NO, or FORCE)
- Default partition (if not specified in job request)

It will be possible to alter most of this data in real-time in order to effect the scheduling of pending jobs (currently executing jobs would continue). This information can be confined to the SLURM control machine for better scalability. It would be used by the Job Manager (and possibly an external scheduler), which either exist only on the control machine or communicate only with the control machine. An API to manage this information was developed first, followed by simple command-line tools utilizing the API. APIs designed to return

SLURM state information will permit the specification of a time-stamp. If the requested data has not changed since the time-stamp provided by the application, the application's current information need not be updated. The API will return a brief "no_change" response rather than returning relatively verbose state information.

The nodes in a partition may be designated for exclusive or non-exclusive use by a job. A shared value of "YES" indicates that jobs may share nodes upon request. A shared value of "NO" indicates that jobs are always given exclusive use of allocated nodes. A shared value of "FORCE" indicates that jobs will never be insured exclusive access to nodes, but SLURM will initiate multiple jobs on the nodes for high system utilization and responsiveness. In this case, job requests for exclusive node access will not be honored. Non-exclusive access may negatively impact the performance of parallel jobs or cause them to fail upon exhausting shared resources (e.g. memory or disk space). However, shared resources should improve overall system utilization and responsiveness. The proper support of shared resources, including enforcement of limits on these resources, entails a substantial amount effort which we are not planning to address presently. However, we have designed SLURM so as to not preclude the addition of such a capability at a later time if so desired. Future enhancements could include constraining jobs to a specific CPU count or memory size within a node, which could be used to space-share the node. The partition manager will allocate nodes to pending jobs upon request by the job manager.

Bit maps are used to indicate which nodes are up, idle, associated with each partition, and associated with each unique configuration. This technique permits scheduling decisions to normally be made by performing a small number of tests followed by fast bit map manipulations.

Submitted jobs can specify desired partition, CPU count, node count, task count, task distribution pattern (round-robin or sequential within a node), the need for contiguous nodes assignment, and (optionally) an explicit list of nodes. Nodes will be selected so as to satisfy all job requirements. For example a job requesting four CPUs and four nodes will actually be allocated eight CPUs and four nodes in the case of all nodes having two CPUs each. The submitted job may have an associated key, and by virtue of this can be granted access to specific partitions. The request may also indicate node configuration constraints such as minimum real memory or CPUs per node, required features, etc.

Nodes are selected for possible assignment to a job based upon it's configuration requirements (e.g. partition specification, minimum memory, temporary disk space, features, node list, etc.). The selection is refined by determining which nodes are up and available for use. Groups of nodes are then considered in order of weight, with the nodes having the minimum resources to satisfy the request preferred. Finally the physical location of the nodes is considered.

The actual selection of nodes for allocation to a job is currently tuned for the Quadrics interconnect. This hardware supports hardware message broadcast, but only if the nodes are contiguous. If a job is not allocated contiguous nodes, a slower software based multi-case mechanism will be used. Job's will be allocated continuous nodes to the extent possible (in fact, contiguous node allocation can

be required by a job is so specified at submission time). If contiguous nodes can not be allocated to a job, it will be allocated resources from the minimum number of sets of contiguous nodes possible. If multiple sets of contiguous nodes can be allocated to a job, the one which most closely fits the job's requirements will be used. This technique will leave the largest continuous sets of nodes intact for jobs requiring them.

The partition manager will build a list of nodes to satisfy a job's request, including the distribution of tasks to nodes (recognizing the number of CPUs on each node). It will also cache the IP addresses of each node and provide this information to srun at job initiation time for improved performance.

The failure of any node to respond to the partition manager will only effect jobs associated with that node. In fact, jobs may indicate they should continue executing even if nodes allocated to it cease responding. In this case, the job will need to provide for its own fault-tolerance. All other jobs and nodes in the cluster will continue to operate after a node failure. No additional work will be allocated to the failed node and it will be pinged periodically to determine when it has been restored to serviced.

A sample configuration file follows.

```
#
# Sample /etc/SLURM.conf
# Author: John Doe
# Date: 11/06/2001
#
ControlMachine=lx0001
BackupController=lx0002
#
# Node Configurations
#
NodeName=DEFAULT TmpDisk=16384
NodeName=lx[0001-0002] State=DRAINED
NodeName=lx[0003-8000] CPUs=16 RealMemory=2048 Weight=16
NodeName=lx[8001-9999] CPUs=32 RealMemory=4096 Weight=40 Feature=1200MHz
#
# Partition Configurations
#
PartitionName=DEFAULT MaxTime=30 MaxNodes=2
PartitionName=login Nodes=lx[0001-0002] State=DOWN   # Don't schedule work here
PartitionName=debug Nodes=lx[0003-0030] State=UP    Default=YES
PartitionName=class Nodes=lx[0031-0040] AllowGroups=students,teachers
PartitionName=batch Nodes=lx[0041-9999] MaxTime=UNLIMITED MaxNodes=4096 Key=YES
```

2.2 Job Manager

The core functions to be supported by the job manager include:

- Queue job request

- Reset priority of jobs (for external scheduler to order queue)
- Reserve/allocate nodes for a future job
- Initiate job
- Will job run query (test if "Initiate job" request would succeed)
- Status job (including node list, memory and CPU use data)
- Signal job (send arbitrary signal to all processes associated with a job)
- Terminate job (remove all processes)
- Preempt/resume job (future)
- Checkpoint/restart job (future)
- Change node count of running job (could fail if insufficient resources are available, future)

We need more detail here. None of this has been even prototyped yet.

We propose that SLURM implement a very simple scheduling algorithm, namely FIFO with backfill. Backfill scheduling means that a lower priority job (i.e. newer) can be scheduled before another job only if doing so does not delay the expected initiation time of the higher priority job. Backfill scheduling tends to improve both system utilization and responsiveness by initiating smaller node count and shorter time limit jobs more rapidly. An attempt will be made to schedule pending jobs on a periodic basis and whenever any change in job, partition, or node state might permit the scheduling of a job. All nodes allocated to a job will remain so until *all* processes associated with that job terminate. If a node allocated to a job fails, the job may either continue execution or terminate depending upon its configuration.

We are well aware this algorithm will not satisfy the needs of many customers and provide the means for establishing other scheduling algorithms. Before a newly arrived job is placed into the queue, it is assigned a priority that may be established by an administrator defined program. SLURM APIs permit an external entity to alter the priorities of jobs at any time to re-order the queue as desired. The Maui Scheduler[6] is one example of an external scheduler suitable for use with SLURM.

Another scheduler that we plan to offer with SLURM is DPCS[7]. DPCS has flexible scheduling algorithms that suit our needs well and provides the scalability required for this application. Most of the resource accounting and some of the job management functions presently within DPCS would be moved into the proposed SLURM Job Management component. DPCS will require some modification to operate within this new, richer environment. The DPCS Central Manager requires porting to Linux.

[6] http://mauischeduler.sourceforge.net/

[7] http://www.llnl.gov/icc/lc/dpcs/dpcs_overview.html

The DPCS writes job accounting records to Unix files. Presently, these are moved to a machine with the Sybase database. This data can be accessed via command-line and web interfaces with Kerberos authentication and authorization. We are not contemplating making this database software available through SLURM, but might consider writing this data to an open source database if so desired.

System specific scripts can be executed prior to the initiation of a user job and after the termination of a user job (e.g. prolog and epilog). These scripts are executed as user `root` and can be used to establish an appropriate environment for the user (e.g. permit logins, disable logins, terminate "orphan" processes, etc.). An API for all functions would be developed initially, followed by a command-line tool utilizing the API.

The job manager will collect resource consumption information (CPU time used, CPU time allocated, and real memory used) associated with a job from the slurmd daemons. When a job approaches its time limit (as defined by wall-clock execution time) or an imminent system shutdown has been scheduled, the job will be terminated. The actual termination process is to notify slurmd daemons on nodes allocated to the job of the termination request along with a time period in which to complete the termination. The slurmd job termination procedure, including job signaling, is described in the slrumd section.

If for some reason, there are non-killable processes associated with the job, nodes associated with those processes will be drained and the other nodes relinquished for other uses.

2.3 Switch Manager

The switch manager would be responsible for allocating switch channels and assigning them to user jobs. The switch channels would be de-allocated upon job termination. The slurmd (on each compute node) would provide user jobs with the authentication required for switch use as directed by the switch manager. Switch health monitoring tools will also be implemented in phase two. It may be desirable for the SLURM daemons to use the switch directly for communications, particularly for the movement of a large executable and/or standard input file. This option will be investigated in phase three.

2.4 Fault Tolerance

A backup `slurmctld`, if one is configured, will periodically ping the primary `slurmctld`. Should the primary `slurmctld` cease responding, the backer will load state information from the last `slurmctld` state save, and assume control. All `slurmd` daemons will be notify of the new controller location and be requested to upload current state information to it. When the primary `slurmctld` is returned to service, it will tell the backup `slurmctld` to save state and terminate. The primary will then load state, assume control, and notify `slurmd` daemons.

SLURM utilities and the APIs will read the /etc/slrumd.conf files and initially try to contact the primary `slurmctld`. Should that attempt fail, an attempt will be made to contact the backup `slurmctld` before terminating.

3 Slurmd

Slurmd is a multi-threaded daemon for managing user job and monitoring system state. Upon initiation it will read the /etc/slurmd.conf file, capture system state, and await requests from the SLURM control daemon (slurmctld).

It's most common action will be to report system state upon request. Upon slurmd startup and periodically thereafter, it will gather the processor count, real memory size, and temporary disk space for the node. Should those values change, the controller will be notified. Another thread will be created to capture CPU, real-memory and virtual-memory consumption from the process table entries. Differences in resource utilization values from process table snapshot to snapshot will be accumulated. Slurmd will insure these accumulated values are not decremented if resource consumption for a user happens to decrease from snapshot to snapshot, which would simply reflect the termination of one or more processes. Both the real and virtual memory high-water marks will be recorded and the integral of memory consumption (e.g. megabyte-hours). Resource consumption will be grouped by user ID and SLURM job ID (if any). Data will be collected for system users (root, ftp, ntp, etc.) as well as customer accounts. The intent is to capture all resource use including kernel, idle and down time. Upon request, the accumulated values will be uploaded to the controller and cleared. When all processes associated with a SLURM job have terminated, slurmd with notify the controller. Since multiple parallel job executions may occur from within a single SLURM job, slurmd will not execute the epilog until requested to do so by the controller upon termination of all processes associated with the SLURM job.

Slurmd will accept requests from the SLURM control daemon to initiate and terminate user jobs. The initiate job request will contain: real and effective user IDs, environment variables, working directory, task numbers, Kerberos credential (?), interconnect specifications and authorization, core paths, process limits (?), SLURM job ID, command to execute, and it arguments. Slurmd will execute the prolog script (if any), reset its session ID, and then initiate the job as requested. It will record to disk the SLURM job ID, session ID, process ID associated with each task, and user associated with the job. In the event of slurmd failure, this information will be recovered from disk in order to identify a specific job. This job identity will be used in communications with the SLURM controller.

The job termination request will contain the SLURM job ID and a delay period. Jobs will have an API made available to register with slurmd exactly which process(s) should be send what signals with how much lead time prior to termination. Slurmd will send requested signal (or SIGNXCPU if none specified) to the identified process(es) associated with the SLURM job (or all processes

We can get this job registration and signalling code from DPCS. –MJ

associated with that session ID or process tree by default), sleep for the delay specified, and send SIGKILL to all of the job's processes. If the processes do not terminate, SIGKILL will be sent again. If the processes still do not terminate slurmd will notify the slurmctld, which will log the event and set node's state to DRAINED. After all processes terminate, slurmd will execute the epilog program (if any).

4 Command Line Utilities

4.1 scancel

`scancel` will prematurely terminate a queued or running job. If the job is in the queue, it will just be removed. If the job is running, it will be signaled and terminated as described in the slurmd section of this document. It will identify the job(s) to be terminated through input specification of: SLURM job ID or user name. If no job specification is supplied, the user will be asked for one. If the user name is supplied, all jobs associated with that user will be terminated. `scancel` can only be executed by the job's owner or user root.

4.2 scontrol

`scontrol` is a tool meant for SLURM administration by user root. It provides the following capabilities:

- Reconfigure - Cause slurmctld to re-read its configuration file.
- Show build parameters - Display the values of parameters that SLURM was built with such as locations of files and values of timers. This can either display the value of specific parameters or all parameters.
- Show job state - Display the state information of a particular job or all jobs in the system.
- Show node state - Display the state and configuration information of a particular node, a set of nodes (using regular expressions to identify their names, or all nodes.
- Show partition state - Display the state and configuration information of a particular partition or all partitions.
- Update job state - Update the state information of a particular job in the system. Note that not all state information can be changed in this fashion (e.g. the nodes allocated to a job).
- Update node state - Update the state of a particular node. Note that not all state information can be changed in this fashion (e.g. the amount of memory configured on a node). In some cases, you may need to modify the SLURM configuration file and cause it to be re-read using the "Reconfigure" command described above.

- Update partition state - Update the state of a partition node. Note that not all state information can be changed in this fashion (e.g. the default partition). In some cases, you may need to modify the SLURM configuration file and cause it to be re-read using the "Reconfigure" command described above.

4.3 squeue

squeue will report the state of SLURM jobs. It can filter these jobs input specification of job state (RUN, PENDING, etc.), job ID, user name, and job name. If no specification is supplied, the state of all jobs will be reported.

squeue can also report the state of SLURM partitions and nodes. By default, it will report a summary of partition state with node counts and a summary of the configuration of those nodes (e.g. "PartitionName=batch Nodes=lx[1000-9999] RealMemory=2048-4096 IdleNodes=1234 ...").

4.4 srun

Mark, this is for you

5 Infrastructure: Communications Library

Optimal communications performance will depend upon hierarchical communications patterned after DPCS and GangLL work. The SLURM control machine will generate a list of nodes for each communication. The message will then be sent to one of the nodes. The daemon on that node receiving the message will divide the node list into two or more new lists of similar size and retransmit the message to one node on each list. Figure 3 shows the communications for a fan-out of two. Acknowledgments will optionally be sent for the messages to confirm receipt with a third message to commit the action. Our design permits the control machine to delegate one or more compute machine daemons as responsible for fault-tolerance, collection of acknowledgment messages, and the commit decision. This design minimizes the control machine overhead for performance reasons. This design also offers excellent scalability and fault tolerance.[8]

[8] Arguments to the communications request include:

- Request ID
- Request (command or acknowledgment or commit)
- List of nodes to be effected
- Fan-out (count)
- Commit of request to be required (Yes or No or Delegate node receiving message)
- Acknowledgment requested to node (name of node or NULL)
- Acknowledgment requested to port (number)

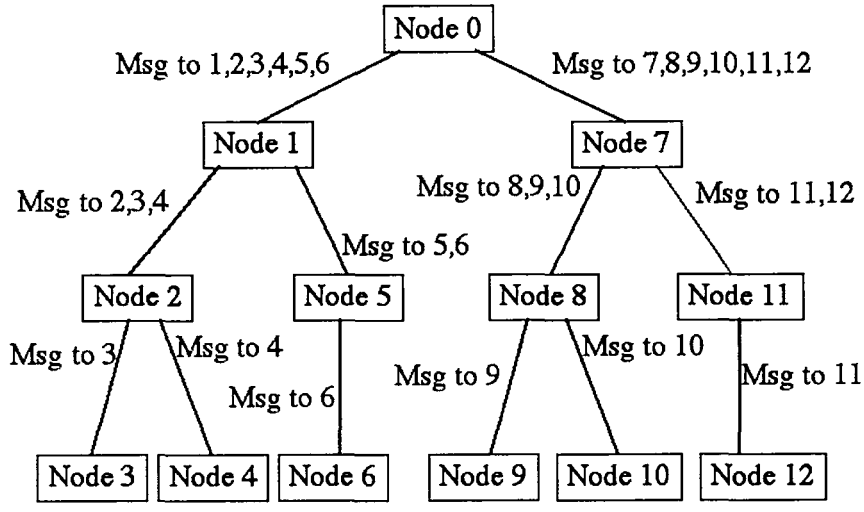


Figure 3: Sample communications with fanout = 2

Security will be provided by the use of reserved ports, which must be opened by root-level processes. SLURM daemons will open these ports and all user requests will be processed through those daemons.

5.1 Infrastructure: Other

The state of slurmctld will be written periodically to disk for fault tolerance. Daemons will be initiated via `inittab` using the `respawn` option to insure their continuous execution. If the control machine itself becomes inoperative, its functions can easily be moved in an automated fashion to another computer. In fact, the computer designated as alternative control machine can easily be relocated as the workload on the compute nodes changes. The communications library design is very important in providing this flexibility.

A single machine will serve as a centralized cluster manager and database. We do not anticipate user applications executing on this machine.

The syslog tools will be used for logging purposes and take advantage of the severity level parameter.

6 Development Plan

The design calls for a four-phase development process. Phase one will develop infrastructure: the communications layer, node status information collection and management. There will be no development of a scheduler in phase one.

Phase two will provide basic job management functionality: basic job and partition management plus simple scheduling, but without use of an interconnect.

Phase three will add Quadrics Elan3 switch support and overall documentation.

Phase four rounds out SLURM with job accounting, fault-tolerance, and full integration with DPCS (Distributed Production Control System).

A Glossary

DCE Distributed Computing Environment

DFS Distributed File System (part of DCE)

DPCS Distributed Production Control System, a meta-batch system and resource manager developed by LLNL

GangLL Gang Scheduling version of LoadLeveler, a joint development project with IBM and LLNL

Globus Grid scheduling infrastructure

Kerberos Authentication mechanism

LoadLeveler IBM's parallel job management system

LLNL Lawrence Livermore National Laboratory

NQS Network Queuing System (a batch system)

OSCAR Open Source Cluster Application Resource

References

[1] Moe Jette et al. Survey of batch/resource management-related system software. Technical report, LLNL, 2002.