SLURM: Simple Linux Utility for Resource Management

Morris Jette and Mark Grondona
Lawrence Livermore National Laboratory, USA

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, scheduling, and stream copy modules. This paper presents an overview of the SLURM architecture and functionality.

1 Overview

Simple Linux Utility for Resource Management (SLURM)[1] is a resource management system suitable for use on large and small Linux clusters. 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 resulting design 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 [2].

[1] A tip of the hat to Matt Groening and creators of *Futurama*, where Slurm is the most popular carbonated beverage in the universe.

- `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. SLURM also supports a general purpose "plugin" mechanism, which permits a variety of different infrastructures to be easily supported. The SLURM configuration file specifies which set of plugin modules should be used.
- `Interconnect Independence`: SLURM currently supports UDP/IP-based communication and the Quadrics Elan3 interconnect. Adding support for other interconnects, including topography constraints, is straightforward and utilizes the plugin mechanism described above.
- `Scalability`: SLURM is designed for scalability to clusters of thousands of nodes. The SLURM controller for a cluster with 1000 nodes occupies on the order of 2 MB of memory, and excellent performance has been demonstrated. Jobs may specify their resource requirements in a variety of ways, including requirements options and ranges.
- `Fault Tolerance`: SLURM can handle a variety of failure modes without terminating workloads, including crashes of the node running the SLURM controller. User jobs may be configured to continue execution despite the failure of one or more nodes on which they are executing. The user command controlling a job, `srun`, may detach and reattach from the parallel tasks at any time. Nodes allocated to a job are available for reuse as soon as the job(s) allocated to that node terminate. If some nodes fail to complete job termination in a timely fashion because of hardware or software problems, only the scheduling of those tardy nodes will be affected.
- `Security`: SLURM employs crypto technology to authenticate users to services and services to each other with a variety of options available through the plugin mechanism. SLURM does not assume that its networks are physically secure, but it does assume that the entire cluster is within a single administrative domain with a common user base.
- `System Administrator Friendly`: SLURM utilizes a simple configuration file and minimizes distributed state. Its configuration may be changed at any time without impacting running jobs. Heterogeneous nodes within a cluster may be easily managed. SLURM interfaces are usable by scripts and its behavior is highly deterministic.

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 four command line utilities: `srun` for submitting a job for execution and optionally controlling it interactively, `scancel`

for terminating a pending or running job, `squeue` for monitoring job queues, and `sinfo` for monitoring partition and overall system state. System administrators perform privileged operations through an additional command line utility, `scontrol`.

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

1.2 What SLURM Is Not

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 those functions.

SLURM is not a meta-batch system like Globus [3] or DPCS (Distributed Production Control System) [4]. SLURM supports resource management across a single cluster.

SLURM is not a sophisticated batch system. In fact, it was expressly designed to provide high-performance parallel job management while leaving scheduling decisions to an external entity. Its default scheduler implements First-In First-Out (FIFO). An scheduler entity can establish a job's initial priority through a plugin. An external scheduler may also submit, signal, and terminate jobs as well as reorder the queue of pending jobs via the API.

2 Architecture

As shown 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 fail-over twin), and five command line utilities: `srun`, `scancel`, `sinfo`, `squeue`, and `scontrol`, which can run anywhere in the cluster.

The entities managed by these SLURM daemons include *nodes*, the compute resource in SLURM, *partitions*, which group nodes into logical disjoint sets, *jobs*, or allocations of resources assigned to a user for a specified amount of time, and *job steps*, which are sets of (possibly parallel) tasks within a job. Each job in the priority-ordered queue is allocated nodes within a single partition. Once an allocation request fails, no lower priority jobs for that partition will be considered for a resource allocation. Once a job is assigned a set of nodes, the user is able to initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.

Figure 2 further illustrates the interrelation of these entities as they are managed by SLURM by showing a group of compute nodes split into two partitions. Partition 1 is running one job, with one job step utilizing the full allocation of that

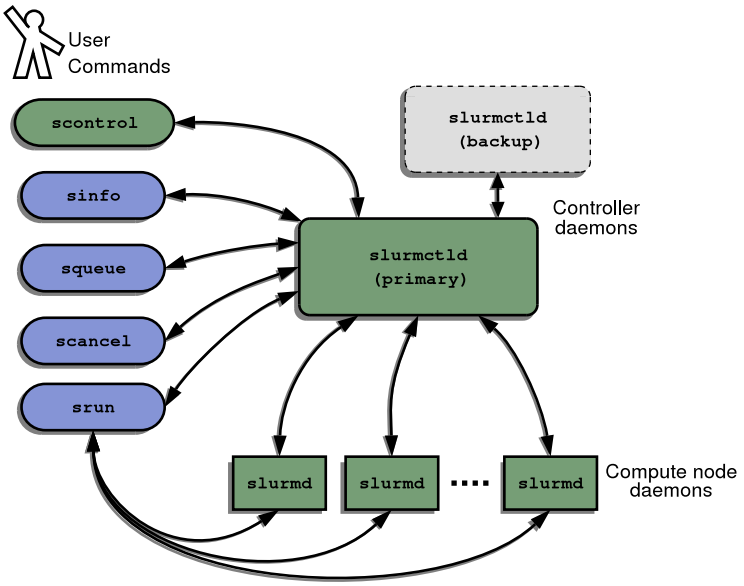

Fig. 1. SLURM architecture

job. The job in Partition 2 has only one job step using half of the original job allocation. That job might initiate additional job steps to utilize the remaining nodes of its allocation.

Figure 3 shows the subsystems that are implemented within the `slurmd` and `slurmctld` daemons. These subsystems are explained in more detail below.

2.1 Slurmd

`slurmd` is a multi-threaded daemon running on each compute node and can be compared to a remote shell daemon: it reads the common SLURM configuration file and saved state information, notifies the controller that it is active, waits for work, executes the work, returns status, then waits for more work. Because it initiates jobs for other users, it must run as user *root*. It also asynchronously exchanges node and job status with `slurmctld`. The only job information it has at any given time pertains to its currently executing jobs. `slurmd` has five major components:

- `Machine and Job Status Services`: Respond to controller requests for machine and job state information and send asynchronous reports of some state changes (e.g., `slurmd` startup) to the controller.
- `Remote Execution`: Start, manage, and clean up after a set of processes (typically belonging to a parallel job) as dictated by the `slurmctld` daemon or an `srun` or `scancel` command. Starting a process may include executing a prolog program, setting process limits, setting real and effective uid, estab-

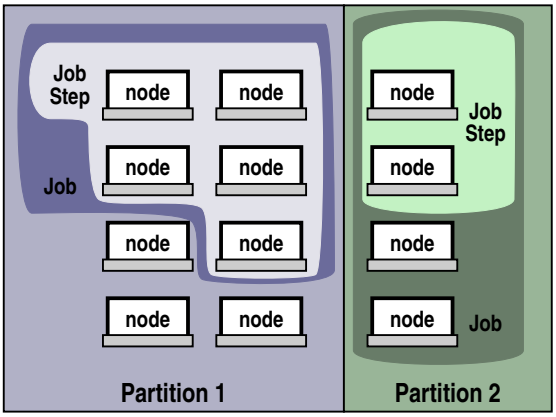

Fig. 2. SLURM entities: nodes, partitions, jobs, and job steps

lishing environment variables, setting working directory, allocating interconnect resources, setting core file paths, initializing stdio, and managing process groups. Terminating a process may include terminating all members of a process group and executing an epilog program.

- `Stream Copy Service`: Allow handling of stderr, stdout, and stdin of remote tasks. Job input may be redirected from a single file or multiple files (one per task), an `srun` process, or /dev/null. Job output may be saved into local files or returned to the `srun` command. Regardless of the location of stdout/err, all job output is locally buffered to avoid blocking local tasks.
- `Job Control`: Allow asynchronous interaction with the Remote Execution environment by propagating signals or explicit job termination requests to any set of locally managed processes.

2.2 Slurmctld

Most SLURM state information exists in `slurmctld`, also known as the controller. `slurmctld` is multi-threaded with independent read and write locks for the various data structures to enhance scalability. When `slurmctld` starts, it reads the SLURM configuration file and any previously saved state information. Full controller state information is written to disk periodically, with incremental changes written to disk immediately for fault tolerance. `slurmctld` runs in either master or standby mode, depending on the state of its fail-over twin, if any. `slurmctld` need not execute as user *root*. In fact, it is recommended that a unique user entry be created for executing `slurmctld` and that user must be identified in the SLURM configuration file as `SlurmUser`. `slurmctld` has three major components:

- `Node Manager`: Monitors the state of each node in the cluster. It polls `slurmds` for status periodically and receives state change notifications from `slurmd`

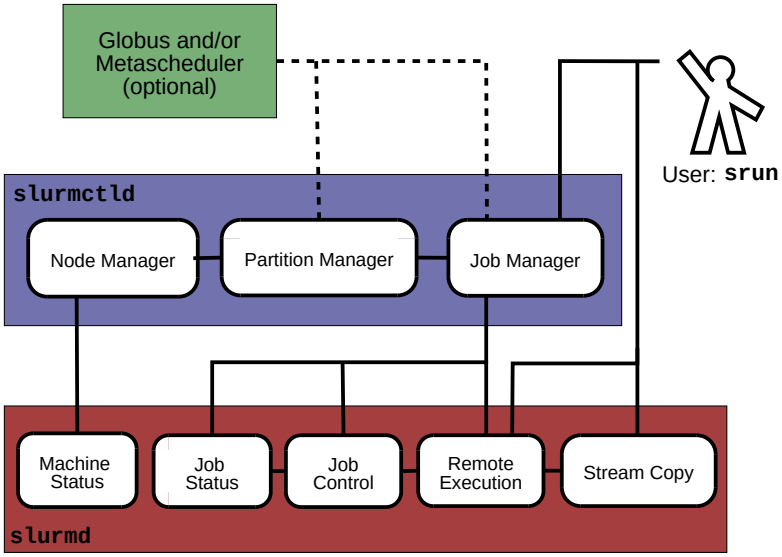

Fig. 3. SLURM architecture - subsystems

daemons asynchronously. It ensures that nodes have the prescribed configuration before being considered available for use.

- `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 on 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 places pending jobs in a priority-ordered queue. The Job Manager is awakened on a periodic 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 priority-ordered job queue. The highest priority jobs for each partition are allocated resources as possible. As soon as an allocation failure occurs for any partition, no lower-priority jobs for that partition are considered for initiation. 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, permitting it to commence execution. When the Job Manager detects that all nodes associated with a job have completed their work, it initiates cleanup and performs another scheduling cycle as described above.

2.3 Command Line Utilities

The command line utilities offer users access to remote execution and job control. They also permit administrators to dynamically change the system configuration. These commands use SLURM APIs that are directly available for more sophisticated applications.

- `scancel`: Cancel a running or a pending job or job step, subject to authentication and authorization. This command can also be used to send an arbitrary signal to all processes on all nodes associated with a job or job step.
- `scontrol`: Perform privileged administrative commands such as bringing down a node or partition in preparation for maintenance. Many `scontrol` functions can only be executed by privileged users.
- `sinfo`: Display a summary of partition and node information. An assortment of filtering and output format options are available.
- `squeue`: Display the queue of running and waiting jobs and/or job steps. A wide assortment of filtering, sorting, and output format options are available.
- `srun`: Allocate resources, submit jobs to the SLURM queue, and initiate parallel tasks (job steps). Every set of executing parallel tasks has an associated `srun` that initiated it and, if the `srun` persists, manages it. Jobs may be submitted for later execution (e.g., batch), in which case `srun` terminates after job submission. Jobs may also be submitted for interactive execution, where `srun` keeps running to shepherd the running job. In this case, `srun` negotiates connections with remote `slurmd`s for job initiation and to get stdout and stderr, forward stdin,[2] and respond to signals from the user. `srun` may also be instructed to allocate a set of resources and spawn a shell with access to those resources.

2.4 Plugins

In order to simplify the use of different infrastructures, SLURM uses a general purpose plugin mechanism. A SLURM plugin is a dynamically linked code object that is loaded explicitly at run time by the SLURM libraries. A plugin provides a customized implementation of a well-defined API connected to tasks such as authentication, interconnect fabric, and task scheduling. A common set of functions is defined for use by all of the different infrastructures of a particular variety. For example, the authentication plugin must define functions such as `slurm_auth_create` to create a credential, `slurm_auth_verify` to verify a credential to approve or deny authentication, `slurm_auth_get_uid` to get the uid associated with a specific credential, etc. It also must define the data structure used, a plugin type, a plugin version number, etc. When a SLURM daemon is initiated, it reads the configuration file to determine which of the available plugins should be used. For example *AuthType=auth/authd* says to use the plugin for authd based authentication and *PluginDir=/usr/local/lib* identifies the directory in which to find the plugin.

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

2.5 Communications Layer

SLURM presently uses Berkeley sockets for communications. However, we anticipate using the plugin mechanism to permit use of other communications layers. At LLNL we are using an ethernet network for SLURM communications and the Quadrics Elan switch exclusively for user applications. The SLURM configuration file permits the identification of each node's hostname as well as its name to be used for communications. In the case of a control machine known as *mcri* to be communicated with using the name *emcri* (say to indicate an ethernet communications path), this is represented in the configuration file as *ControlMachine=mcri ControlAddr=emcri*. The name used for communication is the same as the hostname unless otherwise specified.

Internal SLURM functions pack and unpack data structures in machine independent format. We considered the use of XML style messages, but we felt this would adversely impact performance (albeit slightly). If XML support is desired, it is straightforward to perform a translation and use the SLURM APIs.

2.6 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 SLURM configuration and state information. Only privileged users may modify the SLURM configuration, cancel any job, or perform other restricted activities. Privileged users in SLURM include the users *root* and *SlurmUser* (as defined in the SLURM configuration file). If permission to modify SLURM configuration is required by others, set-uid programs may be used to grant specific permissions to specific users.

Communication Authentication. Historically, inter-node authentication has been accomplished via the use of reserved ports and set-uid programs. In this scheme, daemons check the source port of a request to ensure that it is less than a certain value and thus only accessible by *root*. The communications over that connection are then implicitly trusted. Because reserved ports are a limited resource and set-uid programs are a possible security concern, we have employed a credential-based authentication scheme that does not depend on reserved ports. In this design, a SLURM authentication credential is attached to every message and authoritatively verifies the uid and gid of the message originator. Once recipients of SLURM messages verify the validity of the authentication credential, they can use the uid and gid from the credential as the verified identity of the sender.

The actual implementation of the SLURM authentication credential is relegated to an "auth" plugin. We presently have implemented three functional authentication plugins: authd[5], Munge, and none. The "none" authentication type employs a null credential and is only suitable for testing and networks where security is not a concern. Both the authd and Munge implementations employ cryptography to generate a credential for the requesting user that may then be authoritatively verified on any remote nodes. However, authd assumes a secure network

and Munge does not. Other authentication implementations, such as a credential based on Kerberos, should be easy to develop using the auth plugin API.

Job Authentication. When resources are allocated to a user by the controller, a "job step credential" is generated by combining the uid, job id, step id, the list of resources allocated (nodes), and the credential lifetime and signing the result with the `slurmctld` private key. This credential grants the user access to allocated resources and removes the burden from `slurmd` to contact the controller to verify requests to run processes. `slurmd` verifies the signature on the credential against the controller's public key and runs the user's request if the credential is valid. Part of the credential signature is also used to validate stdout, stdin, and stderr connections from `slurmd` to `srun`.

Authorization. Access to partitions may be restricted via a *RootOnly* flag. If this flag is set, job submit or allocation requests to this partition are only accepted if the effective uid originating the request is a privileged user. A privileged user may submit a job as any other user. 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 partition, which would prevent the external scheduler from effectively managing it. Access to partitions may also be restricted to users who are members of specific Unix groups using a *AllowGroups* specification.

2.7 Example: Executing a Batch Job

In this example a user wishes to run a job in batch mode, in which `srun` returns immediately and the job executes in the background when resources are available. The job is a two-node run of script containing *mping*, a simple MPI application. The user submits the job:

```
srun --batch --nodes 2 --nprocs 2 mping.sh
```

The script `mping.sh` contains:

```
#!/bin/sh
srun hostname
srun mping 1 1048576
```

The initial `srun` command authenticates the user to the controller and submits the job request. The request includes the `srun` environment, current working directory, and command line option information. By default, stdout and stderr are sent to files in the current working directory and stdin is copied from `/dev/null`.

The controller consults the Partition Manager to test whether the job will ever be able to run. If the user has requested a non-existent 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, for example:

```
srun: error: Unable to allocate resources: Invalid partition name
```

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

```
srun: jobid 42 submitted
```

The controller awakens the Job Manager, which tries to run jobs starting at the head of the priority ordered job queue. It finds job *42* and makes a successful request to the Partition Manager to allocate two nodes from the default (or requested) partition: *dev6* and *dev7*.

The Job Manager then sends a request to the `slurmd` on the first node in the job *dev6* to execute the script specified on the user's command line.[3] The Job Manager also sends a copy of the environment, current working directory, stdout and stderr location, along with other options. Additional environment variables are appended to the user's environment before it is sent to the remote `slurmd` detailing the job's resources, such as the SLURM job id (*42*) and the allocated nodes (*dev[6-7]*).

The remote `slurmd` establishes the new environment, executes a SLURM prolog program (if one is configured) as user *root*, and executes the job script (or command) as the submitting user. The `srun` within the job script detects that it is running with allocated resources from the presence of a `SLURM_JOBID` environment variable. `srun` connects to `slurmctld` to request a job step to run on all nodes of the current job. `slurmctld` validates the request and replies with a job step credential and switch resources. `srun` then contacts `slurmd`s running on both *dev6* and *dev7*, passing the job step credential, environment, current working directory, command path and arguments, and interconnect information. The `slurmd`s verify the valid job step credential, connect stdout and stderr back to `srun`, establish the environment, and execute the command as the submitting user.

Unless instructed otherwise by the user, stdout and stderr are copied to a file in the current working directory by `srun`:

```
/path/to/cwd/slurm-42.out
```

The user may examine output files at any time if they reside in a globally accessible directory. In this example `slurm-42.out` would contain the output of the job script's two commands (hostname and mping):

```
dev6
dev7
  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
```

[3] Had the user specified an executable file rather than a job script, an `srun` program would be initiated on the first node and `srun` would initiate the executable with the desired task distribution.

```
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 tasks complete execution, `srun` is notified by `slurmd` of each task's exit status. `srun` reports job step completion to the Job Manager and exits. `slurmd` detects when the job script terminates and notifies the Job Manager of its exit status and begins cleanup. The Job Manager directs the `slurmd`s formerly assigned to the job to run the SLURM epilog program (if one is configured) as user *root*. Finally, the Job Manager releases the resources allocated to job *42* and updates the job status to *complete*. The record of a job's existence is eventually purged.

2.8 Example: Executing an Interactive Job

In this example a user wishes to run the same *mping* command in interactive mode, in which `srun` blocks while the job executes and stdout/stderr of the job are copied onto stdout/stderr of `srun`. The user submits the job, this time without the `batch` option:

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

The `srun` command authenticates the user to the controller and makes a request for a resource allocation and job step. The Job Manager responds with a list of nodes, a job step credential, and interconnect resources on successful allocation. If resources are not immediately available, the request terminates or blocks depending on user options.

If the request is successful, `srun` forwards the job run request to the assigned `slurmd`s in the same manner as the `srun` in the batch job script. In this case, the user sees the program output 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 (End Of File) on each stream and closes it, then receives the task exit status from each `slurmd`. The `srun` process notifies `slurmctld` that the job is complete and terminates. The controller contacts all `slurmd`s allocated to the terminating job and issues a request to run the SLURM epilog, then releases the job's resources.

Most signals received by `srun` while the job is executing are transparently forwarded to the remote tasks. SIGINT (generated by Control-C) is a special case and only causes `srun` to report remote task status unless two SIGINTs are received in rapid succession. SIGQUIT (Control-\) is another special case. SIGQUIT forces termination of the running job.

3 Slurmctld Design

`slurmctld` is modular and multi-threaded with independent read and write locks for the various data structures to enhance scalability. The controller includes the following subsystems: Node Manager, Partition Manager, and Job Manager. Each of these subsystems is described in detail below.

3.1 Node Management

The Node Manager monitors the state of nodes. Node information monitored 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 can specify a list of system node names using a numeric range in the SLURM configuration file or in the SLURM tools (e.g., "*NodeName=linux[001-512] CPUs=4 RealMemory=1024 TmpDisk=4096 Weight=4 Feature=Linux*"). These values for CPUs, RealMemory, and TmpDisk are considered to be the minimal node configuration values acceptable for the node to enter into service. The `slurmd` registers whatever resources actually exist on the node, and this is recorded by the Node Manager. Actual node resources are checked on `slurmd` initialization and periodically thereafter. If a node registers with less resources than configured, it is placed in DOWN state and the event logged. Otherwise, the actual resources reported are recorded and possibly used as a basis for scheduling (e.g., if the node has more RealMemory than recorded in the configuration file, the actual node configuration may be used for determining suitability for any application; alternately, the data in the configuration file may be used for possibly improved scheduling performance). Note the node name syntax with numeric range permits even very large heterogeneous clusters to be described in only a few lines. In fact, a smaller number of unique configurations can provide SLURM with greater efficiency in scheduling work.

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 each resource. Pending jobs are assigned the least capable nodes (i.e., lowest weight) that satisfy their requirements. This tends to leave the more capable nodes available for those jobs requiring those capabilities.

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 are acceptable (e.g., "$Feature = FS1|FS2|FS3$" means the job should be allocated nodes that all have the feature "FS1" or they all have feature "FS2," etc.).

Node records are kept in an array with hash table lookup. If nodes are given names containing sequence numbers (e.g., "lx01", "lx02", etc.), the hash table permits specific node records to be located very quickly; therefore, this is our recommended naming convention for larger clusters.

An API is available to view any of this information and to update some node information (e.g., state). APIs designed to return SLURM state information permit the specification of a time stamp. If the requested data has not changed since the time stamp specified by the application, the application's current information need not be updated. The API returns a brief "No Change" response rather than returning relatively verbose state information. Changes in node configurations (e.g., node count, memory, etc.) or the nodes actually in the cluster should be reflected in the SLURM configuration files. SLURM configuration may be updated without disrupting any jobs.

3.2 Partition Management

The Partition Manager identifies groups of nodes to be used for execution of user jobs. One might consider this the actual resource scheduling component. Data associated with a partition includes:

- Name
- RootOnly flag to indicate that only users *root* or `SlurmUser` may allocate resources in this partition (for any user)
- List of associated nodes
- State of partition (UP or DOWN)
- Maximum time limit for any job
- Minimum and 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 no partition is specified in a job request)

It is possible to alter most of this data in real-time in order to affect the scheduling of pending jobs (currently executing jobs would not be affected). This information is confined to the controller machine(s) for better scalability. It is 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.

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 on 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 are never ensured exclusive access to nodes, but SLURM may initiate multiple jobs on the nodes for improved system utilization and responsiveness. In this case, job requests for exclusive node access are not 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 may improve overall system utilization and responsiveness. The proper support of shared resources, including enforcement of limits on these resources, entails a substantial amount of effort, which we are not presently planning to expend. 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 effectively space-share individual nodes. The Partition Manager will allocate nodes to pending jobs on request from the Job Manager.

Submitted jobs can specify desired partition, time limit, node count (minimum and maximum), CPU count (minimum) task count, the need for contiguous node assignment, and an explicit list of nodes to be included and/or excluded in its allocation. Nodes are 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 request may also indicate node configuration constraints such as minimum real memory or CPUs per node, required features, shared access, etc. Overall there are 13 different parameters that may identify resource requirements for a job.

Nodes are selected for possible assignment to a job based on the job'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 lowest *Weight* preferred. Finally, the physical location of the nodes is considered.

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. If so configured, a job's resource requirements would be compared with the (relatively small number of) node configuration records, each of which has an associated bit map. Usable node configuration bitmaps would be ANDed with the selected partitions bit map ANDed with the UP node bit map and possibly ANDed with the IDLE node bit map (this last test depends on the desire to share resources). This method can eliminate tens of thousands of individual node configuration comparisons that would otherwise be required in large heterogeneous clusters.

The actual selection of nodes for allocation to a job is currently tuned for the Quadrics interconnect. This hardware supports hardware message broadcast only if the nodes are contiguous. If a job is not allocated contiguous nodes, a slower software based multi-cast mechanism is used. Jobs will be allocated continuous

nodes to the extent possible (in fact, contiguous node allocation may be specified as a requirement on job submission). If contiguous nodes cannot 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 that 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 builds a list of nodes to satisfy a job's request. It also caches the IP addresses of each node and provides this information to `srun` at job initiation time for improved performance.

The failure of any node to respond to the Partition Manager only affects jobs associated with that node. In fact, a job may indicate it should continue executing even if allocated nodes cease responding. In this case, the job needs 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 is allocated to the failed node, and it will be pinged periodically to determine when it resumes responding. The node may then be returned to service (depending on the `ReturnToService` parameter in the SLURM configuration).

3.3 Configuration

A single configuration file applies to all SLURM daemons and commands. Most of this information is used only by the controller. Only the host and port information is referenced by most commands. A sample configuration file is shown in Table 1.

3.4 Job Manager

There are a multitude of parameters associated with each job, including:

- Job name
- Uid
- Job id
- Working directory
- Partition
- Priority
- Node constraints (processors, memory, features, etc.)

Job records have an associated hash table for rapidly locating specific records. They also have bit maps of requested and/or allocated nodes (as described above).

The core functions supported by the Job Manager include:

- Request resource (job may be queued)
- Reset priority of a job
- 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)

```
#
# Sample /etc/slurm.conf
# Author: John Doe
# Date: 11/06/2001

ControlMachine=lx0000   ControlAddr=elx0000
BackupController=lx0001 BackupAddr=elx0001

AuthType="auth/authd"
Epilog=/etc/slurm/epilog
FastSchedule=1
FirstJobId=65536
HashBase=10
HeartbeatInterval=60
InactiveLimit=120
JobCredentialPrivateKey=/etc/slurm/private.key
JobCredentialPublicCertificate=/etc/slurm/public.cert
KillWait=30
PluginDir=/usr/lib/slurm
Prioritize=/usr/local/slurm/etc/priority
Prolog=/etc/slurm/prolog
ReturnToService=0
SlurmctldDebug=4
SlurmctldLogFile=/var/tmp/slurmctld.log
SlurmctldPidFile=/var/run/slurmctld.pid
SlurmctldPort=7002
SlurmctldTimeout=120
SlurmdDebug=4
SlurmdLogFile=/var/tmp/slurmd.log
SlurmdPidFile=/var/run/slurmd.pid
SlurmdPort=7003
SlurmdSpoolDir=/var/tmp/slurmd.spool
SlurmdTimeout=120
SlurmUser=slurm
StateSaveLocation=/tmp/slurm.state
TmpFS=/tmp

#
# Node Configurations
#
NodeName=DEFAULT TmpDisk=16384 Procs=16 RealMemory=2048 Weight=16
NodeName=lx[0000-0002] NodeAddr=elx[0000-0002] State=DRAINED
NodeName=lx[0003-8000] NodeAddr=elx[0003-8000]

NodeName=DEFAULT CPUs=32 RealMemory=4096 Weight=40 Feature=1200MHz
NodeName=lx[8001-9999] NodeAddr=elx[8001-9999]
#
# Partition Configurations
#
PartitionName=DEFAULT MaxTime=30 MaxNodes=2 Shared=NO
PartitionName=debug Nodes=lx[0003-0030] State=UP Default=YES
PartitionName=class Nodes=lx[0031-0040] AllowGroups=students,teachers
PartitionName=login Nodes=lx[0000-0002] State=DOWN   # Don't schedule work here

#
PartitionName=DEFAULT MaxTime=UNLIMITED RootOnly=YES
PartitionName=batch Nodes=lx[0041-9999] MaxNodes=4096
```

Table 1. Sample SLURM config file

- Change node count of running job (could fail if insufficient resources are available)

Jobs are placed in a priority-ordered queue and allocated nodes as selected by the Partition Manager. SLURM implements a very simple default scheduling algorithm, namely FIFO. An attempt is 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.

We are aware that this scheduling algorithm does not satisfy the needs of many customers, and we provide the means for establishing other scheduling algorithms. Before a newly arrived job is placed into the queue, an external scheduler plugin assigns its initial priority. A plugin function is also called at the start of each

scheduling cycle to modify job or system state as desired. SLURM APIs permit an external entity to alter the priorities of jobs at any time and re-order the queue as desired. The Maui Scheduler [6, 7] is one example of an external scheduler suitable for use with SLURM.

LLNL uses DPCS [4] as SLURM's external scheduler. DPCS is a meta-scheduler with flexible scheduling algorithms that suit our needs well. It also provides the scalability required for this application. DPCS maintains pending job state internally and only transfers the jobs to SLURM (or another underlying resources manager) only when they are to begin execution. By not transferring jobs to a particular resources manager earlier, jobs are assured of being initiated on the first resource satisfying their requirements, whether a Linux cluster with SLURM or an IBM SP with LoadLeveler (assuming a highly flexible application). This mode of operation may also be suitable for computational grid schedulers.

In a future release, 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. Presently, only the wall-clock run time of a job is monitored. When a job approaches its time limit (as defined by wall-clock execution time) or an imminent system shutdown has been scheduled, the job is terminated. The actual termination process is to notify `slurmd` daemons on nodes allocated to the job of the termination request. The `slurmd` job termination procedure, including job signaling, is described in Section 4.

One may think of a job as described above as an allocation of resources rather than a collection of parallel tasks. The job script executes `srun` commands to initiate the parallel tasks or "job steps. " The job may include multiple job steps, executing sequentially and/or concurrently either on separate or overlapping nodes. Job steps have associated with them specific nodes (some or all of those associated with the job), tasks, and a task distribution (cyclic or block) over the nodes.

The management of job steps is considered a component of the Job Manager. Supported job step functions include:

- Register job step
- Get job step information
- Run job step request
- Signal job step

Job step information includes a list of nodes (entire set or subset of those allocated to the job) and a credential used to bind communications between the tasks across the interconnect. The `slurmctld` constructs this credential and sends it to the `srun` initiating the job step.

3.5 Fault Tolerance

SLURM supports system level fault tolerance through the use of a secondary or "backup" controller. The backup controller, if one is configured, periodically pings the primary controller. Should the primary controller cease responding, the backup loads state information from the last state save and assumes control. When the

primary controller is returned to service, it tells the backup controller to save state and terminate. The primary then loads state and assumes control.

SLURM utilities and API users read the configuration file and initially try to contact the primary controller. Should that attempt fail, an attempt is made to contact the backup controller before returning an error.

SLURM attempts to minimize the amount of time a node is unavailable for work. Nodes assigned to jobs are returned to the partition as soon as they successfully clean up user processes and run the system epilog. In this manner, those nodes that fail to successfully run the system epilog, or those with unkillable user processes, are held out of the partition while the remaining nodes are quickly returned to service.

SLURM considers neither the crash of a compute node nor termination of `srun` as a critical event for a job. Users may specify on a per-job basis whether the crash of a compute node should result in the premature termination of their job. Similarly, if the host on which `srun` is running crashes, the job continues execution and no output is lost.

4 Slurmd Design

The `slurmd` daemon is a multi-threaded daemon for managing user jobs and monitoring system state. Upon initiation it reads the configuration file, recovers any saved state, captures system state, attempts an initial connection to the SLURM controller, and awaits requests. It services requests for system state, accounting information, job initiation, job state, job termination, and job attachment. On the local node it offers an API to translate local process ids into SLURM job id.

The most common action of `slurmd` is to report system state on request. Upon `slurmd` startup and periodically thereafter, it gathers the processor count, real memory size, and temporary disk space for the node. Should those values change, the controller is notified. In a future release of SLURM, `slurmd` will also capture CPU and real-memory and virtual-memory consumption from the process table entries for uploading to `slurmctld`.

`slurmd` accepts requests from `srun` and `slurmctld` to initiate and terminate user jobs. The initiate job request contains such information as real uid, effective uid, environment variables, working directory, task numbers, job step credential, interconnect specifications and authorization, core paths, SLURM job id, and the command line to execute. System-specific programs can be executed on each allocated node prior to the initiation of a user job and after the termination of a user job (e.g., *Prolog* and *Epilog* in the configuration file). These programs 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.). `slurmd` executes the prolog program, resets its session id, and then initiates the job as requested. It records 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 is recovered from disk in order to identify active jobs.

When `slurmd` receives a job termination request from the SLURM controller, it sends SIGTERM to all running tasks in the job, waits for *KillWait* seconds (as specified in the configuration file), then sends SIGKILL. If the processes do not terminate `slurmd` notifies `slurmctld`, which logs the event and sets the node's state to DRAINED. After all processes have terminated, `slurmd` executes the configured epilog program, if any.

5 Command Line Utilities

5.1 scancel

`scancel` terminates queued jobs or signals running jobs or job steps. The default signal is SIGKILL, which indicates a request to terminate the specified job or job step. `scancel` identifies the job(s) to be signaled through user specification of the SLURM job id, job step id, user name, partition name, and/or job state. If a job id is supplied, all job steps associated with the job are affected as well as the job and its resource allocation. If a job step id is supplied, only that job step is affected. `scancel` can only be executed by the job's owner or a privileged user.

5.2 scontrol

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

- `Shutdown`: Cause `slurmctld` and `slurmd` to save state and terminate.
- `Reconfigure`: Cause `slurmctld` and `slurmd` to reread the configuration file.
- `Ping`: Display the status of primary and backup `slurmctld` daemons.
- `Show Configuration Parameters`: Display the values of general SLURM configuration parameters such as locations of files and values of timers.
- `Show Job State`: Display the state information of a particular job or all jobs in the system.
- `Show Job Step State`: Display the state information of a particular job step or all job steps in the system.
- `Show Node State`: Display the state and configuration information of a particular node, a set of nodes (using numeric ranges syntax 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 reread 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 reread using the "Reconfigure" command described above.

5.3 squeue

`squeue` reports the state of SLURM jobs. It can filter these jobs input specification of job state (RUN, PENDING, etc.), job id, user name, job name, etc. If no specification is supplied, the state of all pending and running jobs is reported. `squeue` also has a variety of sorting and output options.

5.4 sinfo

`sinfo` reports the state of SLURM partitions and nodes. By default, it reports a summary of partition state with node counts and a summary of the configuration of those nodes. A variety of sorting and output formatting options exist.

5.5 srun

`srun` is the user interface to accessing resources managed by SLURM. Users may utilize `srun` to allocate resources, submit batch jobs, run jobs interactively, attach to currently running jobs, or launch a set of parallel tasks (job step) for a running job. `srun` supports a full range of options to specify job constraints and characteristics, for example minimum real memory, temporary disk space, and CPUs per node, as well as time limits, stdin/stdout/stderr handling, signal handling, and working directory for job. The full range of options is detailed in Table 2.

The `srun` utility can run in four different modes: *interactive*, in which the `srun` process remains resident in the user's session, manages stdout/stderr/stdin, and forwards signals to the remote tasks; *batch*, in which `srun` submits a job script to the SLURM queue for later execution; *allocate*, in which `srun` requests resources from the SLURM controller and spawns a shell with access to those resources; *attach*, in which `srun` attaches to a currently running job and displays stdout/stderr in real time from the remote tasks.

6 Job Initiation Design

There are three modes in which jobs may be run by users under SLURM. The first and most simple mode is *interactive* mode, in which stdout and stderr are displayed on the user's terminal in real time, and stdin and signals may be forwarded from the terminal transparently to the remote tasks. The second mode is *batch* or *queued* mode, in which the job is queued until the request for resources can be satisfied, at which time the job is run by SLURM as the submitting user. In the third mode,

Option	Arg Type	Description
attach	string	attach `srun` to a running job
allocate	boolean	allocate nodes only
batch	boolean	submit a batch script to job queue
cddir	string	working directory of remote processes
constraint	string	arbitrary feature constraints
contiguous	boolean	allocate contiguous nodes only
cpus-per-task	number	number of CPUs needed per process
distribution	string	distribution method for processes (block|cyclic)
error	string	location of stderr redirection
exclude	string	do not allocate from a specific set of hosts
immediate	boolean	exit if resources are not immediately available
input	string	location of stdin redirection
join	string	join existing `srun` to collect output of a running job
job-name	string	name of job
label	boolean	prepend task number to lines of stdout/err
mem	number	minimum amount of real memory per node
mincpus	number	minimum number of CPUs per node
no-kill	boolean	don't kill job if allocated nodes fail
nodelist	string	request a specific set of hosts
nodes	numbers	minimum and maximum number of nodes on which to run
ntasks	number	number of tasks to run
output	string	location of stdout redirection
overcommit	boolean	allow more than 1 process per CPU
partition	string	partition name in which to run
share	boolean	allow nodes to be shared with other jobs
threads	number	run with this number of communication threads
time	number	wall-clock time limit in minutes
tmp	number	minimum amount of temporary disk space
verbose	boolean	verbose operation
version	boolean	print `srun` version and exit
wait	number	seconds to wait after first task end before killing job

Table 2. List of `srun` user options

allocate mode, a job is allocated to the requesting user, under which the user may manually run job steps via a script or in a sub-shell spawned by `srun`.

Figure 4 shows a high-level depiction of the connections that occur between SLURM components during a general interactive job startup. `srun` requests a resource allocation and job step initiation from the `slurmctld`, which responds with the job id, list of allocated nodes, job step credential, etc. if the request is granted, `srun` then initializes a listen port for stdio connections and connects to the `slurmd`s on the allocated nodes requesting that the remote processes be initiated. The `slurmd`s begin execution of the tasks and connect back to `srun` for stdout and stderr. This process is described in more detail below. Details of the batch and allocate modes of operation are not presented due to space constraints.

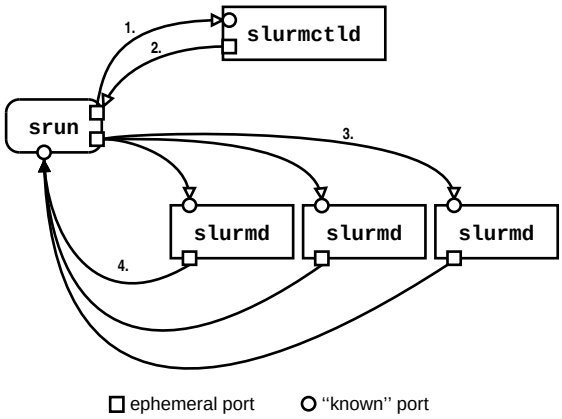

Fig. 4. Job initiation connections overview. 1. `srun` connects to `slurmctld` requesting resources. 2. `slurmctld` issues a response, with list of nodes and job step credential. 3. `srun` opens a listen port for job IO connections, then sends a run job step request to `slurmd`. 4. `slurmd`initiates job step and connects back to `srun` for stdout/err

6.1 Interactive Job Initiation

Interactive job initiation is shown in Figure 5. The process begins with a user invoking `srun` in interactive mode. In Figure 5, the user has requested an interactive run of the executable "`cmd`" in the default partition.

After processing command line options, `srun` sends a message to `slurmctld` requesting a resource allocation and a job step initiation. This message simultaneously requests an allocation (or job) and a job step. `srun` waits for a reply from `slurmctld`, which may not come instantly if the user has requested that `srun` block until resources are available. When resources are available for the user's job, `slurmctld` replies with a job step credential, list of nodes that were allocated, cpus per node, and so on. `srun` then sends a message each `slurmd` on the allocated nodes requesting that a job step be initiated. The `slurmd` daemons verify that the job is valid using the forwarded job step credential and then respond to `srun`.

Each `slurmd` invokes a job manager process to handle the request, which in turn invokes a session manager process that initializes the session for the job step. An IO thread is created in the job manager that connects all tasks' IO back to a port opened by `srun` for stdout and stderr. Once stdout and stderr have successfully been connected, the task thread takes the necessary steps to initiate the user's executable on the node, initializing environment, current working directory, and interconnect resources if needed.

Each `slurmd` forks a copy of itself that is responsible for the job step on this node. This local job manager process then creates an IO thread that initializes stdout, stdin, and stderr streams for each local task and connects these streams to the remote `srun`. Meanwhile, the job manager forks a session manager process

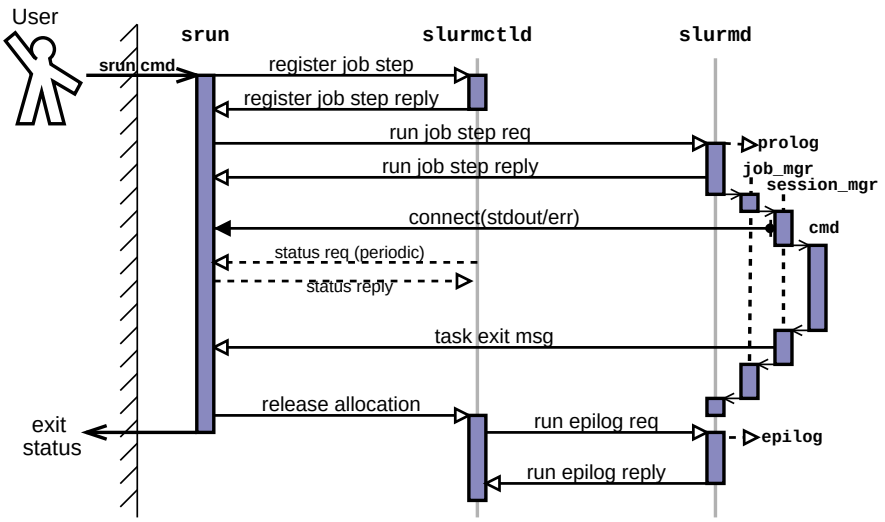

Fig. 5. Interactive job initiation. `srun` simultaneously allocates nodes and a job step from `slurmctld` then sends a run request to all `slurmd`s in job. Dashed arrows indicate a periodic request that may or may not occur during the lifetime of the job

that initializes the session becomes the requesting user and invokes the user's processes.

As user processes exit, their exit codes are collected, aggregated when possible, and sent back to `srun` in the form of a task exit message. Once all tasks have exited, the session manager exits, and the job manager process waits for the IO thread to complete, then exits. The `srun` process either waits for all tasks to exit, or attempts to clean up the remaining processes some time after the first task exits (based on user option). Regardless, once all tasks are finished, `srun` sends a message to the `slurmctld` releasing the allocated nodes, then exits with an appropriate exit status.

When the `slurmctld` receives notification that `srun` no longer needs the allocated nodes, it issues a request for the epilog to be run on each of the `slurmd`s in the allocation. As `slurmd`s report that the epilog ran successfully, the nodes are returned to the partition.

7 Results

We were able to perform some SLURM tests on a 1000-node cluster in November 2002. Some development was still underway at that time and tuning had not been performed. The results for executing the program */bin/hostname* on two tasks per node and various node counts are shown in Figure 6. We found SLURM performance to be comparable to the Quadrics Resource Management System (RMS)

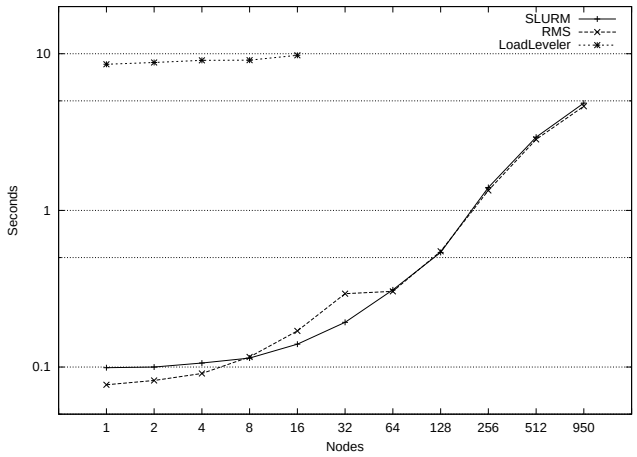

Fig. 6. Time to execute /bin/hostname with various node counts

[8] for all job sizes and about 80 times faster than IBM LoadLeveler[9] at tested job sizes.

8 Future Plans

SLURM begin production use on LLNL Linux clusters in March 2003 and is available from our web site[10].

While SLURM is able to manage 1000 nodes without difficulty using sockets and Ethernet, we are reviewing other communication mechanisms that may offer improved scalability. One possible alternative is STORM [11]. STORM uses the cluster interconnect and Network Interface Cards to provide high-speed communications, including a broadcast capability. STORM only supports the Quadrics Elan interconnnect at present, but it does offer the promise of improved performance and scalability.

Looking ahead, we anticipate adding support for additional interconnects (InfiniBand and the IBM Blue Gene [12] system[4]). We anticipate adding a job preempt/resume capability to the next release of SLURM. This will provide an external scheduler the infrastructure required to perform gang scheduling. We also anticipate adding a checkpoint/restart capability at some time in the future ,and we plan to support changing the node count associated with running jobs (as needed for MPI2). Recording resource use by each parallel job is planned for a future release.

[4] Blue Gene has a different interconnect than any supported by SLURM and a 3-D topography with restrictive allocation constraints.

9 Acknowledgments

SLURM is jointly developed by LLNL and Linux NetworX. Contributors to SLURM development include:

- Jay Windley of Linux NetworX for his development of the plugin mechanism and work on the security components
- Joey Ekstrom for his work developing the user tools
- Kevin Tew for his work developing the communications infrastructure
- Jim Garlick for his development of the Quadrics Elan interface and technical guidance
- Gregg Hommes, Bob Wood, and Phil Eckert for their help designing the SLURM APIs
- Mark Seager and Greg Tomaschke for their support of this project
- Chris Dunlap for technical guidance
- David Jackson of Linux NetworX for technical guidance
- Fabrizio Petrini of Los Alamos National Laboratory for his work to integrate SLURM with STORM communications

References

1. Jette, M.A., et al.: Survey of Batch/Resource Management Related System Software. Lawrence Livermore National Laboratory (2002) (unpublished).
2. The GNU Project: The GNU Public License. http://www.gnu.org/licenses/licenses.html (2003)
3. The Globus Project: The Globus Project. http://www.globus.org (2003)
4. Lawrence Livermore National Laboratory: Distributed Production Control System (DPCS). http://www.llnl.gov/icc/lc/dpcs/dpcs_overview.html (2002)
5. Chun, B.: Authd. http://www.theether.org/authd/ (2002)
6. Jackson, D., Snell, Q., Clement, M.: Core Algorithms of the Maui Scheduler. In: Job Scheduling Stategies for Parallel Processing. Volume 2221., 7th International Workshop, JSSP 2001, Cambridge, MA (2001) 87–102
7. Maui Scheduler: Maui Scheduler. http://mauischeduler.sourceforge.net (2003)
8. Quadrics Ltd.: Resource Management (RMS). http://www.quadrics.com/ (2003)
9. IBM: LoadLeveler – Efficient job scheduling and management. http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/loadleveler.html (2003)
10. Lawrence Livermore National Laboratory: SLURM: Simple Linux Utility for Resource Management. http://www.llnl.gov/linux/slurm/ (2003)
11. Frachtenberg, E., et al.: STORM: Lightning-Fast Resource Management. In: Proceedings of SuperComputing 2002, Baltimore, MD (2002) Available from http://www.cs.huji.ac.il/∼etcs/papers/sc02.pdf.
12. Lawrence Livermore National Laboratory: Blue/GeneL. http://www.llnl.gov/asci/platforms/bluegenel (2003)