SLURM: Simple Linux Utility for Resource Management*

Andy B. Yoo, Morris A. Jette, and Mark Grondona

Lawrence Livermore National Laboratory
Livermore, CA 94551
{ayoo,jette1,grondona1}@llnl.gov

Abstract. A new cluster resource management system called Simple Linux Utility Resource Management (SLURM) is described in this paper. SLURM, initially developed for large Linux clusters at the Lawrence Livermore National Laboratory (LLNL), is a simple cluster manager that can scale to thousands of processors. SLURM is designed to be flexible and fault-tolerant and can be ported to other clusters of different size and architecture with minimal effort. We are certain that SLURM will benefit both users and system architects by providing them with a simple, robust, and highly scalable parallel job execution environment for their cluster system.

1 Introduction

Linux clusters, often constructed by using commodity off-the-shelf (COTS) components, have become increasingly popular as a computing platform for parallel computation in recent years, mainly due to their ability to deliver a high performance-cost ratio. Researchers have built and used small to medium size clusters for various applications [3, 16]. The continuous decrease in the price of the COTS parts in conjunction with the good scalability of the cluster architecture has now made it feasible to economically build large-scale clusters with thousands of processors [18, 19].

D. Feitelson, L. Rudolph, W. Schwiegelshohn (Eds.): JSSPP 2003, LNCS 2862, pp. 44–60, 2003.

An essential component that is needed to harness such a computer is a resource management system. A resource management system (or resource manager) performs such crucial tasks as scheduling user jobs, monitoring machine and job status, launching user applications, and managing machine configuration, An ideal resource manager should be simple, efficient, scalable, fault-tolerant, and portable.

Unfortunately there are no open-source resource management systems currently available which satisfy these requirements. A survey [12] has revealed that many existing resource managers have poor scalability and fault-tolerance rendering them unsuitable for large clusters having thousands of processors [14, 11]. While some proprietary cluster managers are suitable for large clusters, they are typically designed for particular computer systems and/or interconnects [21, 14, 11]. Proprietary systems can also be expensive and unavailable in source-code form. Furthermore, proprietary cluster management functionality is usually provided as a part of a specific job scheduling system package. This mandates the use of the given scheduler just to manage a cluster, even though the scheduler does not necessarily meet the need of organization that hosts the cluster. Clear separation of the cluster management functionality from scheduling policy is desired.

This observation led us to set out to design a simple, highly scalable, and portable resource management system. The result of this effort is Simple Linux Utility Resource Management (SLURM[1]). SLURM was developed with the following design goals:

- *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 [9].
- *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 *plug-in* mechanism, which permits a variety of different infrastructures to be easily supported. The SLURM configuration file specifies which set of plug-in modules should be used.
- *Interconnect independence*: SLURM supports UDP/IP based communication as well as the Quadrics Elan3 and Myrinet interconnects. Adding support for other interconnects is straightforward and utilizes the plug-in mechanism described above.
- *Scalability*: SLURM is designed for scalability to clusters of thousands of nodes. Jobs may specify their resource requirements in a variety of ways including requirements options and ranges, potentially permitting faster initiation than otherwise possible.

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

- *Robustness*: 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. 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 due to hardware of software problems, only the scheduling of those tardy nodes will be affected.
- *Secure*: SLURM employs crypto technology to authenticate users to services and services to each other with a variety of options available through the plug-in mechanism. 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 using 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.

The main contribution of our work is that we have provided a readily available tool that anybody can use to efficiently manage clusters of different size and architecture. SLURM is highly scalable[2]. The SLURM can be easily ported to any cluster system with minimal effort with its plug-in capability and can be used with any meta-batch scheduler or a Grid resource broker [7] with its well-defined interfaces.

The rest of the paper is organized as follows. Section 2 describes the architecture of SLURM in detail. Section 3 discusses the services provided by SLURM followed by performance study of SLURM in Section 4. Brief survey of existing cluster management systems is presented in Section 5. Concluding remarks and future development plan of SLURM is given in Section 6.

2 SLURM Architecture

As a cluster resource manager, SLURM has three key functions. First, it allocates exclusive and/or non-exclusive access to resources to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work on the set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work. Users and system administrators interact with SLURM using simple commands.

Figure 1 depicts the key components of SLURM. 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, which can run anywhere in the cluster.

[2] It was observed that it took less than five seconds for SLURM to launch a 1900-task job over 950 nodes on recently installed cluster at Lawrence Livermore National Laboratory.

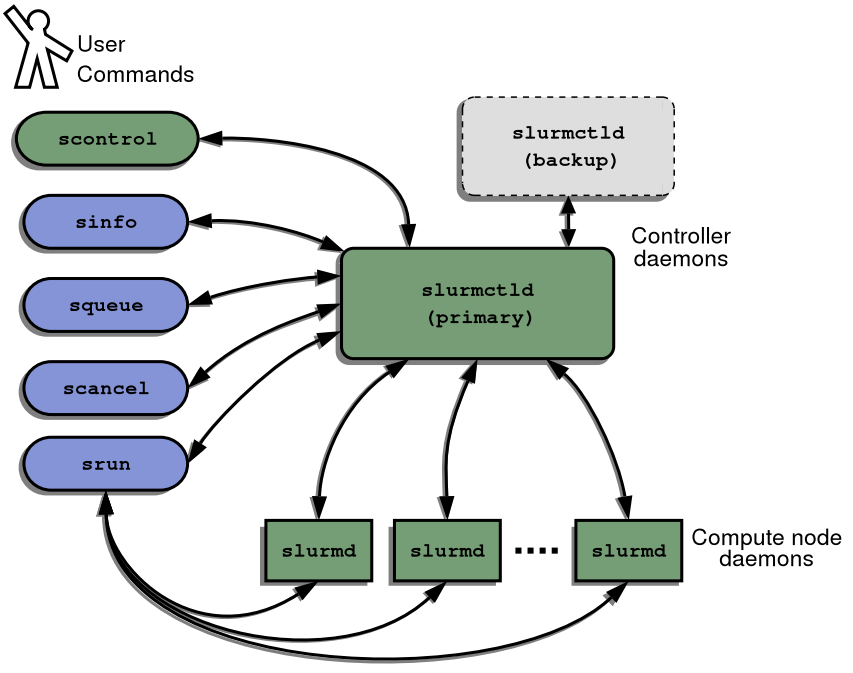

Fig. 1. SLURM Architecture

The entities managed by these SLURM daemons include nodes, the compute resource in SLURM and partitions, which group nodes into logical disjoint sets. The entities also include jobs, or allocations of resources assigned to a user for a specified amount of time, and job steps, which are sets of tasks within a job. Each job is allocated nodes within a single partition. 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 which utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.

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

2.1 SLURM Local Daemon (Slurmd)

The `slurmd` is a multi-threaded daemon running on each compute node. It reads the common SLURM configuration file and recovers any previously saved state information, notifies the controller that it is active, waits for work, executes the work, returns status, and waits for more work. Since it initiates jobs for other users, it must run with root privilege. The only job information it has at any given time pertains to its currently executing jobs. The `slurmd` performs five major tasks.

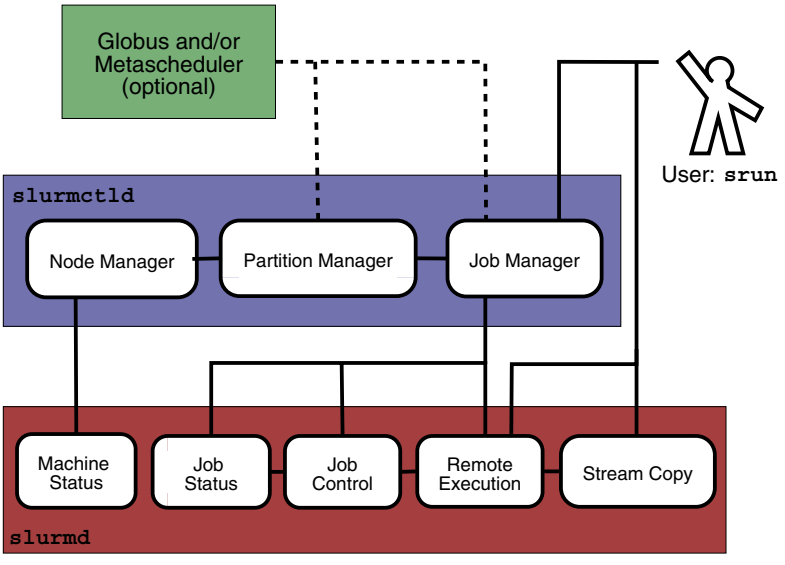

Fig. 2. SLURM Architecture - Subsystems

- *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, monitor, 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 user id, establishing environment variables, setting working directory, allocating interconnect resources, setting core file paths, initializing the Stream Copy Service, 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 file or files, a `srun` process, or /dev/null. Job output may be saved into local files or sent back to the `srun` command. Regardless of the location of stdout or stderr, 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 SLURM Central Daemon (Slurmctld)

Most SLURM state information is maintained by the controller, `slurmctld`. The `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. It can also read additional state information from a checkpoint file generated by a previous execution of `slurmctld`. Full controller state information is written to disk periodically with incremental changes written to disk immediately for fault-tolerance. The `slurmctld` runs in either master or standby mode, depending on the state of its fail-over twin, if any. The `slurmctld` need not execute with root privilege. The `slurmctld` consists of three major components:

- *Node Manager*: Monitors the state of each node in the cluster. It polls `slurmd`'s for status periodically and receives state change notifications from `slurmd` 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 upon node and partition states and configurations. Requests to initiate jobs come from the Job Manager. The `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 clean-up and performs another scheduling cycle as described above.

3 SLURM Operation and Services

3.1 Command Line Utilities

The command line utilities are the user interface to SLURM functionality. They offer users access to remote execution and job control. They also permit administrators to dynamically change the system configuration. These commands all use SLURM APIs which 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 draining 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. A 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` which initiated it and, if the `srun` persists, managing it. Jobs may be submitted for batch execution, 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, and respond to signals from the user. The `srun` may also be instructed to allocate a set of resources and spawn a shell with access to those resources. `srun` has a total of 13 parameters to control where and when the job is initiated.

3.2 Plug-ins

In order to make the use of different infrastructures possible, SLURM uses a general purpose plug-in mechanism. A SLURM plug-in is a dynamically linked code object which is loaded explicitly at run time by the SLURM libraries. A plug-in provides a customized implementation of a well-defined API connected to tasks such as authentication, interconnect fabric, 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 plug-in must define functions such as: `slurm_auth_activate` to create a credential, `slurm_auth_verify` to verify a credential to approve or deny authentication, `slurm_auth_get_uid` to get the user ID associated with a specific credential, etc. It also must define the data structure used, a plug-in type, a plug-in version number. The available polygons are defined in the configuration file.

3.3 Communications Layer

SLURM presently uses Berkeley sockets for communications. However, we anticipate using the plug-in mechanism to easily permit use of other communications layers. At LLNL we are using an Ethernet 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.

While SLURM is able to manage 1000 nodes without difficulty using sockets and Ethernet, we are reviewing other communication mechanisms which may offer improved scalability. One possible alternative is STORM[8]. 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 does offer the promise of improved performance and scalability.

3.4 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 jobs, 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.

We presently support three authentication mechanisms via plug-ins: `authd` [10], `munged` and `none`. A plug-in can easily be developed for Kerberos or authentication mechanisms as desired. The `munged` implementation is described below. A `munged` daemon running as user *root* on each node confirms the identity of the user making the request using the `getpeername` function and generates a credential. The credential contains a user ID, group ID, time-stamp, lifetime, some pseudo-random information, and any user supplied information. The `munged` uses a private key to generate a Message Authentication Code (MAC) for the credential. The `munged` then uses a public key to symmetrically encrypt the credential including the MAC. SLURM daemons and programs transmit this encrypted credential with communications. The SLURM daemon receiving the message sends the credential to `munged` on that node. The `munged` decrypts the credential using its private key, validates it and returns the user ID and group ID of the user originating the credential. The `munged` prevents replay of a credential on any single node by recording credentials that have already been authenticated. In SLURM's case, the user supplied information includes node identification information to prevent a credential from being used on nodes it is not destined for.

When resources are allocated to a user by the controller, a *job step credential* is generated by combining the user ID, job ID, step ID, the list of resources allocated (nodes), and the credential lifetime. This job step credential is encrypted with a `slurmctld` private key. This credential is returned to the requesting agent (`srun`) along with the allocation response, and must be forwarded to the remote `slurmd`'s upon job step initiation. `slurmd` decrypts this credential with the `slurmctld`'s public key to verify that the user may access resources on the local node. `slurmd` also uses this job step credential to authenticate standard input, output, and error communication streams.

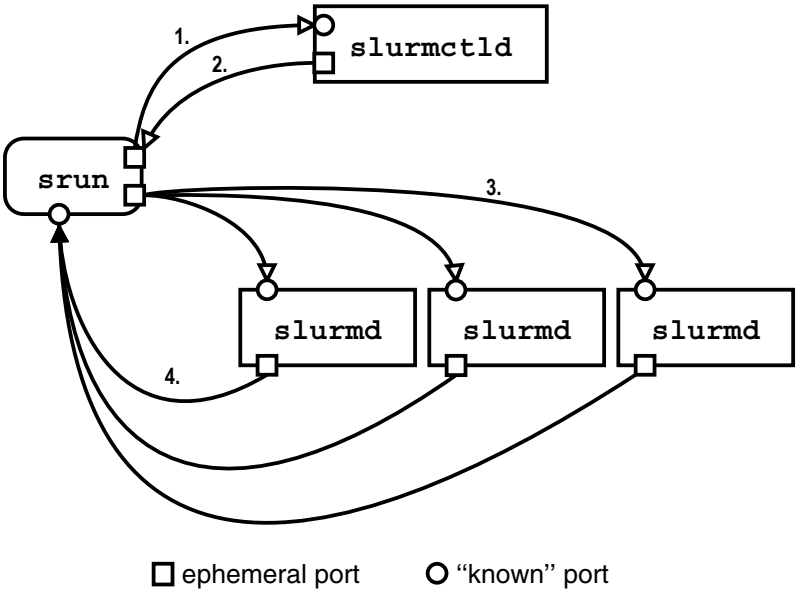

Fig. 3. Job initiation connections overview. 1. The `srun` connects to `slurmctld` requesting resources. 2. `slurmctld` issues a response, with list of nodes and job credential. 3. The `srun` opens a listen port for every task in the job step, then sends a run job step request to `slurmd`. 4. `slurmd`'s initiate job step and connect back to `srun` for stdout/err

3.5 Job Initiation

There are three modes in which jobs may be run by users under SLURM. The first and most simple 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 is *batch* 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 *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 3 gives a high-level depiction of the connections that occur between SLURM components during a general interactive job startup. The `srun` requests a resource allocation and job step initiation from the `slurmctld`, which responds with the job ID, list of allocated nodes, job credential. if the request is granted. The `srun` then initializes listen ports for each task and sends a message 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 and the other initiation modes are described in more detail below.

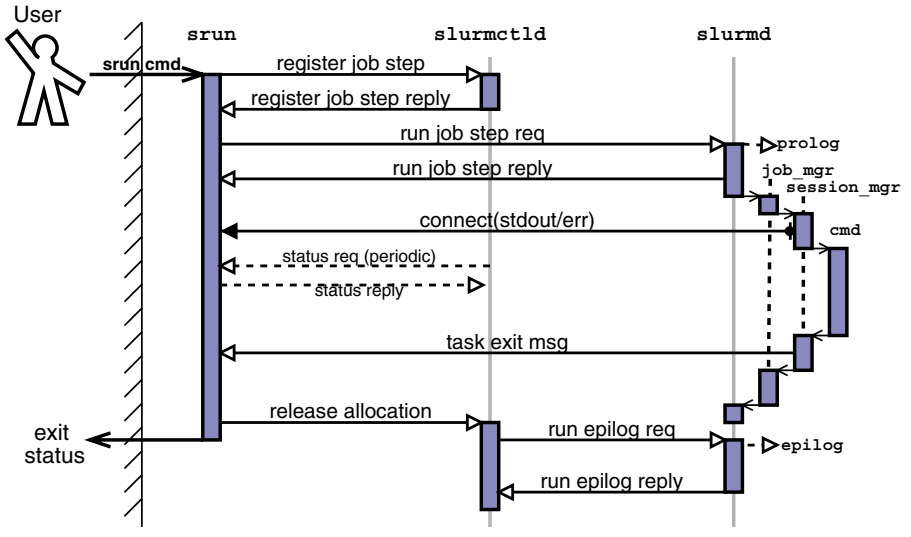

Fig. 4. Interactive job initiation. `srun` simultaneously allocates nodes and a job step from `slurmctld` then sends a run request to all `slurmd`'s in job. Dashed arrows indicate a periodic request that may or may not occur during the lifetime of the job

Interactive Mode Initiation. Interactive job initiation is illustrated in Figure 4. The process begins with a user invoking `srun` in interactive mode. In Figure 4, 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. The `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. The `srun` then sends a message each `slurmd` on the allocated nodes requesting that a job step be initiated. The `slurmd`'s verify that the job is valid using the forwarded job step credential and then respond to `srun`.

Each `slurmd` invokes a job thread to handle the request, which in turn invokes a task thread for each requested task. The task thread connects back to a port opened by `srun` for stdout and stderr. The host and port for this connection is contained in the run request message sent to this machine by `srun`. 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.

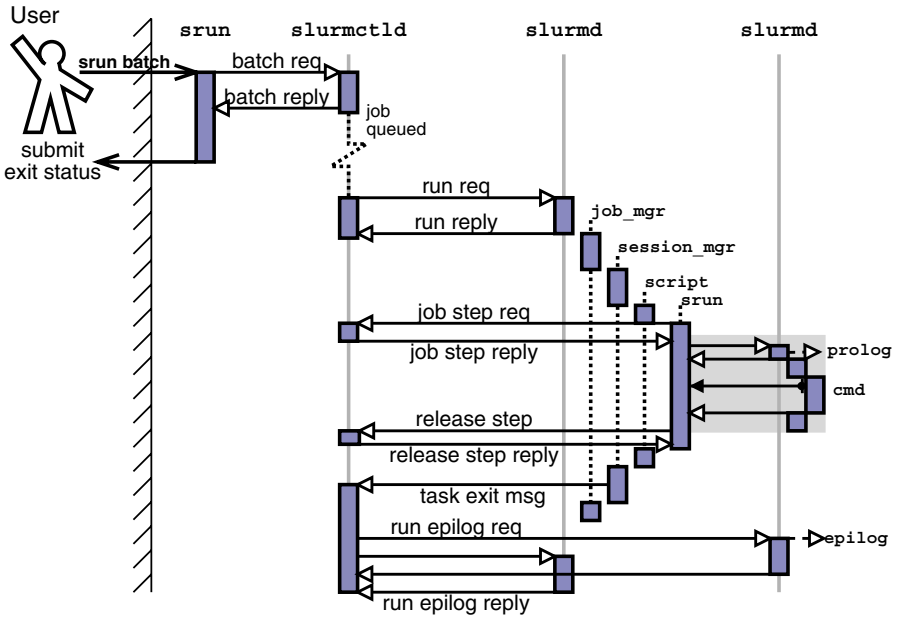

Fig. 5. Queued job initiation. `slurmctld` initiates the user's job as a batch script on one node. Batch script contains an `srun`call which initiates parallel tasks after instantiating job step with controller. The shaded region is a compressed representation and is illustrated in more detail in the interactive diagram (Figure 4)

Once the user process exits, the task thread records the exit status and sends a task exit message back to `srun`. When all local processes terminate, the job thread exits. The `srun` process either waits for all tasks to exit, or attempt to clean up the remaining processes some time after the first task exits. 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.

Batch Mode Initiation. Figure 5 illustrates the initiation of a batch job in SLURM. Once a batch job is submitted, `srun` sends a batch job request to `slurmctld` that contains the input/output location for the job, current working directory, environment, requested number of nodes. The `slurmctld` queues the request in its priority ordered queue.

Once the resources are available and the job has a high enough priority, `slurmctld` allocates the resources to the job and contacts the first node of the

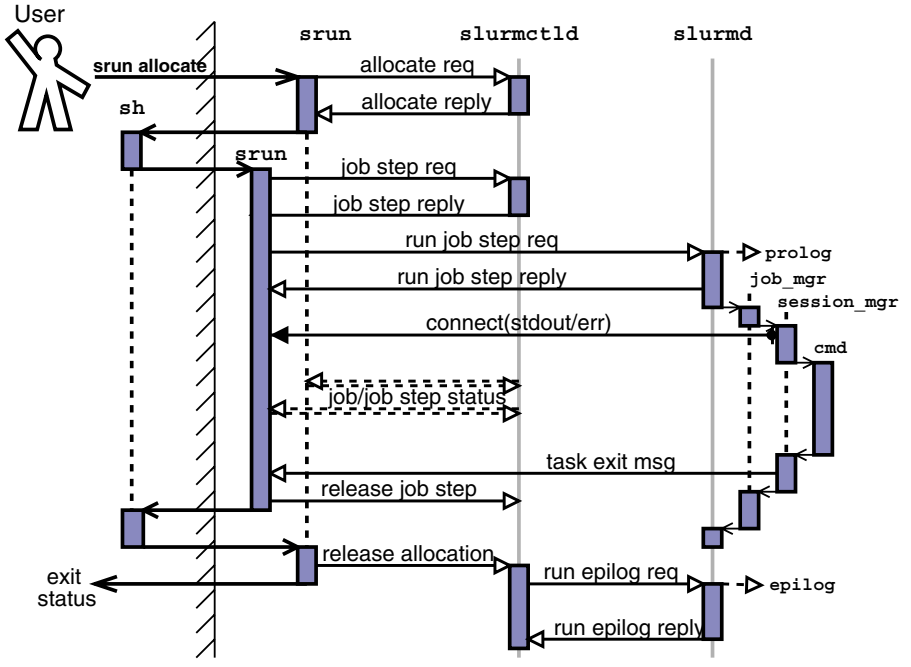

Fig. 6. Job initiation in allocate mode. Resources are allocated and `srun` spawns a shell with access to the resources. When user runs an `srun` from within the shell, the a job step is initiated under the allocation

allocation requesting that the user job be started. In this case, the job may either be another invocation of `srun` or a *job script* which may have multiple invocations of `srun` within it. The `slurmd` on the remote node responds to the run request, initiating the job thread, task thread, and user script. An `srun` executed from within the script detects that it has access to an allocation and initiates a job step on some or all of the nodes within the job.

Once the job step is complete, the `srun` in the job script notifies the `slurmctld` and terminates. The job script continues executing and may initiate further job steps. Once the job script completes, the task thread running the job script collects the exit status and sends a task exit message to the `slurmctld`. The `slurmctld` notes that the job is complete and requests that the job epilog be run on all nodes that were allocated. As the `slurmd`'s respond with successful completion of the epilog, the nodes are returned to the partition.

Allocate Mode Initiation. In allocate mode, the user wishes to allocate a job and interactively run job steps under that allocation. The process of initiation in this mode is illustrated in Figure 6. The invoked `srun` sends an allocate request to `slurmctld`, which, if resources are available, responds with a list of nodes allocated, job id, etc. The `srun` process spawns a shell on the user's terminal

with access to the allocation, then waits for the shell to exit at which time the job is considered complete.

An `srun` initiated within the allocate sub-shell recognizes that it is running under an allocation and therefore already within a job. Provided with no other arguments, `srun` started in this manner initiates a job step on all nodes within the current job. However, the user may select a subset of these nodes implicitly.

An `srun` executed from the sub-shell reads the environment and user options, then notify the controller that it is starting a job step under the current job. The `slurmctld` registers the job step and responds with a job credential. The `srun` then initiates the job step using the same general method as described in the section on interactive job initiation.

When the user exits the allocate sub-shell, the original `srun` receives exit status, notifies `slurmctld` that the job is complete, and exits. The controller runs the epilog on each of the allocated nodes, returning nodes to the partition as they complete the epilog.

4 Related Work

Portable Batch System (PBS). The Portable Batch System (PBS) [20] is a flexible batch queuing and workload management system originally developed by Veridian Systems for NASA. It operates on networked, multi-platform UNIX environments, including heterogeneous clusters of workstations, supercomputers, and massively parallel systems. PBS was developed as a replacement for NQS (Network Queuing System) by many of the same people.

PBS supports sophisticated scheduling logic (via the Maui Scheduler). PBS spawn's daemons on each machine to shepherd the job's tasks. It provides an interface for administrators to easily interface their own scheduling modules. PBS can support long delays in file staging with retry. Host authentication is provided by checking port numbers (low ports numbers are only accessible to user root). Credential service is used for user authentication. It has the job prolog and epilog feature. PBS Supports high priority queue for smaller "interactive" jobs. Signal to daemons causes current log file to be closed, renamed with time-stamp, and a new log file created.

Although the PBS is portable and has a broad user base, it has significant drawbacks. PBS is single threaded and hence exhibits poor performance on large clusters. This is particularly problematic when a compute node in the system fails: PBS tries to contact down node while other activities must wait. PBS also has a weak mechanism for starting and cleaning up parallel jobs.

4.1 Quadrics RMS

Quadrics RMS[21] (Resource Management System) is for Unix systems having Quadrics Elan interconnects. RMS functionality and performance is excellent. Its major limitation is the requirement for a Quadrics interconnect. The proprietary code and cost may also pose difficulties under some circumstances.

Maui Scheduler. Maui Scheduler [17] is an advanced reservation HPC batch scheduler for use with SP, O2K, and UNIX/Linux clusters. It is widely used to extend the functionality of PBS and LoadLeveler, which Maui requires to perform the parallel job initiation and management.

Distributed Production Control System (DPCS). The Distributed Production Control System (DPCS) [6] is a scheduler developed at Lawrence Livermore National Laboratory (LLNL). The DPCS provides basic data collection and reporting mechanisms for prject-level, near real-time accounting and resource allocation to customers with established limits per customers' organization budgets, In addition, the DPCS evenly distributes workload across available computers and supports dynamic reconfiguration and graceful degradation of service to prevent overuse of a computer where not authorized.

DPCS supports only a limited number of computer systems: IBM RS/6000 and SP, Linux, Sun Solaris, and Compaq Alpha. Like the Maui Scheduler, DPCS requires an underlying infrastructure for parallel job initiation and management (LoadLeveler, NQS, RMS or SLURM).

LoadLeveler. LoadLeveler [11, 14] is a proprietary batch system and parallel job manager by IBM. LoadLeveler supports few non-IBM systems. Very primitive scheduling software exists and other software is required for reasonable performance, such as Maui Scheduler or DPCS. The LoadLeveler has a simple and very flexible queue and job class structure available operating in "matrix" fashion. The biggest problem of the LoadLeveler is its poor scalability. It typically requires 20 minutes to execute even a trivial 500-node, 8000-task on the IBM SP computers at LLNL.

Load Sharing Facility (LSF). LSF [15] is a proprietary batch system and parallel job manager by Platform Computing. Widely deployed on a wide variety of computer architectures, it has sophisticated scheduling software including fairshare, backfill, consumable resources, an job preemption and very flexible queue structure. It also provides good status information on nodes and LSF daemons. While LSF is quite powerful, it is not open-source and can be costly on larger clusters.

Condor. Condor [5, 13, 1] is a batch system and parallel job manager developed by the University of Wisconsin. Condor was the basis for IBM's LoadLeveler and both share very similar underlying infrastructure. Condor has a very sophisticated checkpoint/restart service that does not rely upon kernel changes, but a variety of library changes (which prevent it from being completely general). The Condor checkpoint/restart service has been integrated into LSF, Codine, and DPCS. Condor is designed to operate across a heterogeneous environment, mostly to harness the compute resources of workstations and PCs. It has an interesting "advertising" service. Servers advertise their available resources and

consumers advertise their requirements for a broker to perform matches. The checkpoint mechanism is used to relocate work on demand (when the "owner" of a desktop machine wants to resume work).

Beowulf Distributed Process Space (BPROC). The Beowulf Distributed Process Space (BPROC) is set of kernel modifications, utilities and libraries which allow a user to start processes on other machines in a Beowulf-style cluster [2]. Remote processes started with this mechanism appear in the process table of the front end machine in a cluster. This allows remote process management using the normal UNIX process control facilities. Signals are transparently forwarded to remote processes and exit status is received using the usual wait() mechanisms. This tight coupling of a cluster's nodes is convenient, but high scalability can be difficult to achieve.

5 Performance Study

We were able to perform some SLURM tests on a 1000 node cluster at LLNL. Some development was still underway at that time and tuning had not been performed. The results for executing simple 'hostname' program on two tasks per node and various node counts is show in Figure 7. We found SLURM performance to be comparable to the Quadrics Resource Management System (RMS) [21] for

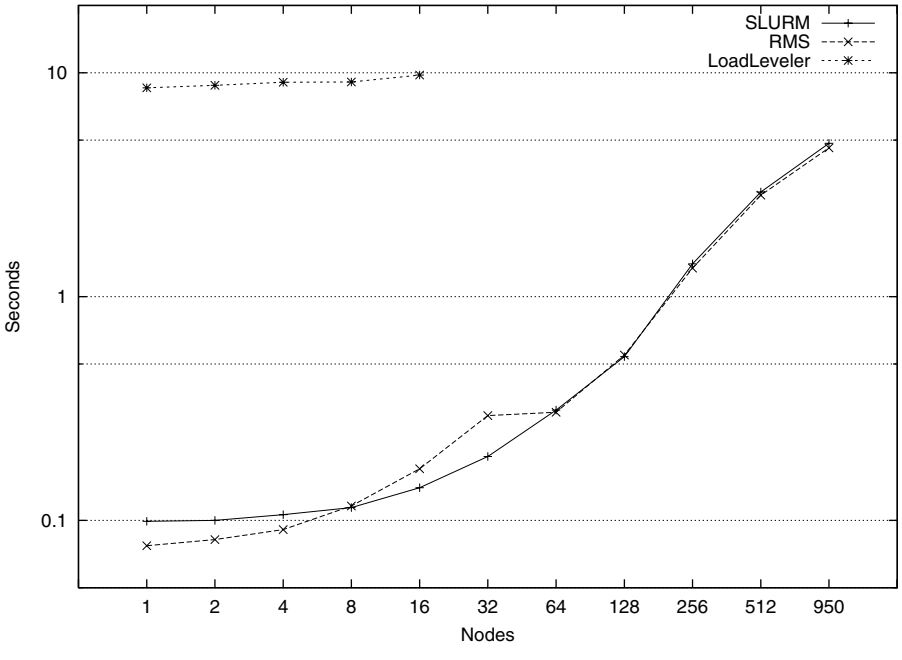

Fig. 7. Time to execute /bin/hostname with various node counts

all job sizes and about 80 times faster than IBM LoadLeveler [14, 11] at tested job sizes.

6 Conclusion and Future Plans

We have presented in this paper an overview of SLURM, a simple, highly scalable, robust, and portable cluster resource management system. The contribution of this work is that we have provided a immediately-available and open-source tool that virtually anybody can use to efficiently manage clusters of different sizes and architecture.

Looking ahead, we anticipate adding support for additional operating systems. We anticipate adding a job preempt/resume capability, which will provide an external scheduler the infrastructure required to perform gang scheduling, and a checkpoint/restart capability. We also plan to use the SLURM for IBM's Blue Gene/L platform [4] by incorporating a capability to manage jobs on a three-dimensional torus machine into the SLURM.

Acknowledgments

Additional programmers responsible for the development of SLURM include: Chris Dunlap, Joey Ekstrom, Jim Garlick, Kevin Tew and Jay Windley.

References

[1] J. Basney, M. Livny, and T. Tannenbaum. High Throughput Computing with Condor. *HPCU news*, 1(2), June 1997. 57
[2] Beowulf Distributed Process Space. `http://bproc.sourceforge.net`. 58
[3] Beowulf Project. `http://www.beowulf.org`. 44
[4] Blue Gene/L. `http://cmg-rr.llnl.gov/asci/platforms/bluegenel`. 59
[5] Condor. `http://www.cs.wisc.edu/condor`. 57
[6] Distributed Production Control System. `http://www.llnl.gov/icc/lc/dpcs_overview.html`. 57
[7] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999. 46
[8] E. Frachtenberg, F. Petrini, et al. Storm: Lightning-fast resource management. In *Proceedings of SuperComputing*, 2002. 50
[9] GNU General Public License. `http://www.gnu.org/licenses/gpl.html`. 45
[10] A. home page. http://www.theether.org/authd/. 51
[11] IBM Corporation. *LoadLeveler's User Guide, Release 2.1*. 45, 57, 59
[12] M. Jette, C. Dunlap, J. Garlick, and M. Grondona. Survey of Batch/Resource Management-Related System Software. Technical Report N/A, Lawrence Livermore National Laboratory, 2002. 45
[13] M. Litzknow, M. Livny, and M. Mutka. Condor - a hunter for idle workstations. In *Proc. International Conference on Distributed Computing Systems*, June 1988. 57

[14] Load Leveler. http://www-1.ibm.com/servers/eservers/pseries/library/sp_books/loadleveler.html. 45, 57, 59
[15] Load Sharing Facility. http://www.platform.com. 57
[16] Loki – Commodity Parallel Processing. http://loki-www.lanl.org. 44
[17] Maui Scheduler. mauischeduler.sourceforge.net. 57
[18] Multiprogrammatic Capability Cluster. http://www.llnl.gov/linux/mcr. 44
[19] Parallel Capacity Resource. http://www.llnl.gov/linux/pcr. 44
[20] Portable Batch System. http://www.openpbs.org. 56
[21] Quadircs Resource Management System. http://www.quadrics.com/website/pdf/rms.pdf. 45, 56, 58