Batch System Deployment on a Production Terascale Cluster

Karl W. Schulz, Kent Milfeld, Chona S. Guiang, Avijit Purkayastha,
Tommy Minyard, John R. Boisseau, and John Casu

Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX 78758

{karl, milfeld, chona, avijit, minyard, boisseau}@tacc.utexas.edu, casuj@cray.com

Abstract: On multi-user HPC clusters, the batch system is a key component for aggregating compute nodes into a single, sharable computing resource. The batch system becomes the "nerve center" for coordinating the use of resources and controlling the state of the system in a way that must be "fair" to its users. Large, multi-user clusters need batch utilities that are robust, reliable, flexible, and easy to use and administer. In this paper we present our experiences with the configuration and deployment of a terascale cluster of 600 processors, with particular attention given to the integration of the LSF HPC batch system software by Platform Computing. To begin, we review the cluster design and present our requirements for a production batch environment supporting a community of hundreds of users. Next, we outline the configuration and extensions made to the LSF batch system and operating environment to meet our design criteria, including the development of job monitoring and job filtering applications, authentication modifications to manage compute node access, and integration of the system with internal accounting applications. Initial scalability results using LSF for MPI applications are presented and compared against modified versions of the LSF application suite. The modified version incurred substantially lower overhead and provided good scalability on MPI applications up to 600 processors. Implementation of software updates as RPM packages, the use of modules for environment management, and the development of tools for monitoring compute-node software states have helped to insure a consistent, system-wide environment for user jobs across node failures and system reboots.

1 Introduction

Today's terascale HPC clusters harness hundreds or even thousands of nodes, usually based on commodity processors and open source software. Sometimes these clusters run large-scale parallel applications that consume the whole machine, while others are used as compute farms for serial processing. To manage these extensive systems effectively, an efficient batch system is required to aggregate all processing elements into a single, sharable computing resource. In addition, cluster management utilities are necessary to ensure a consistent operating state across the system and aid in performing system maintenance. In this paper we present our experiences configuring a production-level batch system on a 600-processor Linux cluster (*Lonestar*). To begin, we present the basic functionality required of any batch system and then focus on specific implementation details using the LSF HPC batch system by Platform Computing. Additional tools are also presented which aid in integrating LSF into a production framework along with scalability results for launching large parallel applications. Finally, we explain how the use of the RPM Package Manager (RPMs) and modules helps to ensure a consistent user environment with minimal administration overhead.

Lonestar comprises 300 dual-processor Xeon server nodes grouped by function into three components: compute nodes, I/O nodes, and front-end nodes. The majority of the nodes are configured as compute resources while a small subset are configured to provide parallel file I/O using the Parallel Virtual File System (PVFS). Two nodes are configured as front-end nodes for handling login sessions. Table 1 summarizes the Lonestar hardware and basic software stack.

Table 1: Lonestar Hardware/Software Overview

Hardware Configuration	
Compute Nodes (282)	Dell PowerEdge 1750, 3.06 GHz FSB 533 MHz 2GB Memory (266 MHz, dual channel) 36 GB SCSI Disk
I/O Nodes (16)	Dell PowerEdge 2650 3.06 GHz FSB 533MHz (4GB Mem.)
Front-End Nodes (2)	Dell PowerEdge 2650 3.06 GHz FSB 533MHz (4GB Mem.)
High Speed Interconnect	Myrinet 2000 (2Gb/s) CLOS topology
NFS/Management Network	Dell PowerConnect 5224 / 3248 (GigE switch hierarchy)
Disk RAIDs (16)	Power Vault 220S, 14x72GB SCSI disk/enclosure
Base Software Stack	
Compilers	Intel (7.1), gcc (2.9.6 & 3.2.3), PGI (5.1.3)
Debugging	TotalView 6.3.1
Hardware Performance	PAPI 3.0
Profiling, Monitoring	Ompitrace 1.1/Paraver 3.1, Tau (pdtools 3.0)
Batch	LSF HPC 5.1
Cluster Management	Cray Rx 3.1
Parallel I/O	ROMIO/PVFS 3.5.1

2 Basic Elements of a Production Batch System

The core functionalities of all cluster batch systems are essentially the same, regardless of the size or specific configuration of the computing hardware. In this section we briefly outline some of the elements required; subsequent sections will present implementation-specific details using LSF.

One fundamental batch system requirement is the ability to create and configure multiple job queues. Batch queues provide an orderly environment for managing the large number of jobs present on a multi-user terascale cluster; they also provide a framework for a scheduler to efficiently assign jobs to free resources. Typically, these queues are defined with a variety of limits for maximum run times, memory usage, and processor counts; they are often assigned different priority levels as well. It is also desirable to provide interactive queues, as well as queues for scheduling background jobs, to facilitate short, interactive debugging and development sessions. Another important feature of batch queues on terascale clusters is the ability to accommodate “big-science” runs. On a multi-user system with several hundred processors such as Lonestar, the majority of simulations will request 32-256 processors. However, there are times when a researcher needs to use most or all of the processors, and a separate “hero” queue can support these large runs on a per-request basis.

A batch system for a multi-user terascale cluster should enable users and administrators to monitor and manage resources effectively. From the user perspective, this includes allocation of resources for a user’s project, simple monitoring and manipulation of individual jobs, and collection of resource usage statistics (e.g., memory usage, CPU usage, and elapsed wall-clock time per job). System administrators must be able to monitor the state of the cluster, enforce allocation limits for each user, perform accounting, conduct local and system-wide maintenance, and configure the workload management parameters to maximize job throughput. For job scheduling, it is often necessary to impose additional constraints on how available resources should be shared. Common scheduling paradigms include: *first in, first out (FIFO)* scheduling, in which jobs are simply scheduled in the order in which they are submitted; *political* scheduling, which enables some users to have more priority than others; and *fairshare* scheduling, in which the scheduler ensures users have equal access over time, helping to smooth out spikes in demand from specific users who would otherwise dominate usage. In addition, the notion of *back-fill* can be combined with any of the scheduling paradigms to allow smaller jobs to run while waiting for enough resources to become available for larger jobs. The back-fill of smaller jobs helps maximize the overall resource utilization of the cluster.

An additional feature that is highly desirable in a HPC batch system is the ability to reserve specific compute resources in advance. These reservations can then be used to match a simulation to the availability of externally coupled resources (such as real-time instrumentation or a separate visualization system), to guarantee a "hero-class" run, or to schedule maintenance for the machine. Another desired component is the ability to run site-specific scripts at multiple stages of a batch-job life cycle. In particular, it is often necessary to run individual scripts at the job submission, job execution, and post job-execution stages.

3 Deployment and Configuration of LSF HPC Batch System

The preceding section outlined key capabilities of a HPC batch environment required for a large, multi-user production computing cluster. In this section, we show how these capabilities were implemented using LSF HPC 5.1 and additional tools we developed. Note that these details are presented from the perspective of a site employing LSF for the first time, but whose staff have experience using other HPC batch systems including PBS, NQS, and LoadLeveler. As such, we provide limited information on items that are easily configurable and instead expand on possible pitfalls for staff deploying the batch system. We also make recommendations for configuring a production HPC environment.

3.1 Queue Definitions and Fairshare Scheduling

Our queue definitions reflect a "keep it simple" strategy by defining only three main queues for regular users: *normal, high,* and *development.* The normal and high queues accommodate the bulk of all user jobs and currently have a 48-hour run limit and a 256-processor limit. The main difference between these two queues is that the *high* queue has an increased scheduling priority and a corresponding increase in the accounting burn rate. The *development* queue was created for immediate, interactive access to a smaller number of compute nodes for development and debugging purposes. On Lonestar, 16 nodes (32 processors) are reserved to support jobs submitted to the development queue; any single interactive job may use up to 16 processors for 30 minutes of runtime. All of the remaining compute nodes are pooled together in a single group that is allocable by the remaining queues. To accommodate runs requiring more than 256 processors (up to the entire system), a *hero* queue accepts large parallel jobs from specific users who are approved by site staff on the basis of requests made in advance. Finally, we define a *systest* queue that can schedule jobs on all available compute resources. This queue is accessible only by staff and provides a mechanism for validating system configuration after hardware or software modifications.

To ensure equitable access for all users to Lonestar's computing resources, we use a global fairshare scheduling option within LSF. Each user is first assigned a base number of "shares" of the machine. Then, the fairshare mechanism constantly updates a dynamic priority for each user based on the shares assigned and on the history of resources consumed by running jobs. Factors in this calculation include the number of job slots reserved, the run time of all running jobs, and the cumulative CPU time, adjusted so that recently used CPU time is weighted more heavily than CPU time used in the distant past. All of these factors are combined into a dynamic fairshare priority (normalized between 0 and 1) for each user, and this is used as an additional factor in making scheduling decisions.

On Lonestar, all users of the system are assigned an equivalent unit share (although the mechanism to alter shares for favored users or groups is straightforward). However, there is one issue that arises in configuring fairshare that affects the successful recovery of held jobs. The mechanism for configuring equal fairshare suggested in the LSF documentation adds a user to the share partition as jobs are submitted, but this approach can cause problems when queued jobs are placed on hold (e.g., for system maintenance reasons). If enough time elapses during a system maintenance interval, all queued jobs placed on hold will no longer have a fairshare partition associated with the queued job owners (owing to the time-decay of the share). Therefore, they will not be scheduled to run when the maintenance time is completed. Consequently, we recommend configuring equal fairshare to add users to the fairshare policy at the job *scheduling* stage, not at the job submission stage (configuration details are provided in Appendix A.1). With this approach, all held jobs will resume without incident as resources become available after system maintenance.

3.2 Job Monitoring

LSF provides a capable set of utilities for submitting, stopping, deleting, and resuming batch jobs. However, the ability to display a concise and informative summary of all currently queued jobs is necessary both to aid users in managing their runs and to monitor the overall state of the system. While the main job display utility provided within LSF (*bjobs*) provides detailed information for individual jobs, its multi-line format can be cumbersome for viewing a large number of queued jobs, since all compute resources assigned to each job are displayed. Fortunately, LSF HPC provides an application programming interface (API) for developing custom utilities to query the state of the batch system directly. With this API, we developed a job-status utility with an interface similar to the *showq* function available in the Maui scheduling software [1]. This utility gives a concise overview of all running, idle, and blocked jobs on the system, as well as the submission time, maximum runtime, and number of processors requested for each job. In addition, this utility summarizes any upcoming advanced reservations, such as regularly scheduled maintenance. By default, the *showq* utility shows reservations within a one-week window, but all outstanding advanced reservations can be shown using an additional command line argument. Sample output from the *showq* utility is presented in Figure 1. The display shows six running jobs consuming 49% of the available compute resources, two idle jobs waiting for enough processors to execute, and two blocked jobs that are in a held state. The output of Figure 1 also shows that there is one advanced reservation scheduled to use all available compute processors between the hours of 9:00AM-5:30PM on Tuesday, March 23. (This particular reservation was made to drain the queues for system maintenance.) The *showq* output allows users to quickly determine the current load on the system and derive approximate times when their jobs will start, based on the maximum remaining time shown for each running job.

```
ACTIVE JOBS--------------------
JOBID     JOBNAME    USERNAME      STATE  PROC  REMAINING       STARTTIME

14694      equillda     user1    Running    16   18:54:07  Tue Feb  3 17:32:41
14701             V     user2    Running    16    7:02:41  Tue Feb  3 17:41:15
14707             V     user3    Running    16   19:11:02  Tue Feb  3 17:49:36
14708         jet08     user4    Running    32    0:38:36  Tue Feb  3 18:17:10
14713           rti     user5    Running    64    3:58:25  Tue Feb  3 20:36:59
14714           cyl     user6    Running   128   23:16:36  Tue Feb  3 21:55:10

       6 Active jobs     272 of 556 Processors Active (48.92%)

IDLE JOBS----------------------
JOBID     JOBNAME    USERNAME      STATE  PROC    WCLIMIT       QUEUETIME

14716        bigjob     user7       Idle   512    0:15:00  Tue Feb  3 22:18:57
14719      smalljob     user7       Idle   256    0:15:00  Tue Feb  3 22:35:31

       2 Idle jobs

BLOCKED JOBS-------------------
JOBID     JOBNAME    USERNAME      STATE  PROC    WCLIMIT       QUEUETIME

14717         hello     user7       Held    16    0:15:00  Tue Feb  3 22:19:07
14718         hello     user7       Held    32    0:15:00  Tue Feb  3 22:19:15

       4 Blocked jobs

     Total Jobs: 12    Active Jobs: 6     Idle Jobs: 2     Blocked Jobs: 4

ADVANCED RESERVATIONS----------
RESV ID   PROC                    RESERVATION WINDOW
karl#79    556    Tue Mar 23 09:00:00 2004 - Tue Mar 23 17:30:00 2004
```

Figure 1: Example output from developed *showq* utility using LSF API.

3.3 Compute-Node Access Management

In general, users log on to front-end nodes and interact with compute nodes only through batch submission of jobs from the front end. However, the ability to access compute nodes directly enables experienced users to monitor the performance and resource utilization of their applications at a very detailed level on each node (e.g., using *top*). In most cluster configurations, the strategy adopted for login access to the compute nodes is an all-or-nothing approach: access is either granted to all users all the time, or access is denied to all. In our case, we adopted a balance between these approaches by granting login access only to users who have actively scheduled jobs on a specific node. This feature allows users to interactively monitor their job performance, memory usage, and processor utilization via direct access to allocated resources.

To implement this type of managed access, we needed direct integration with the batch scheduler to ascertain which nodes are assigned to individual users. To accomplish this integration, we developed an additional login authentication module that queries the LSF API whenever a user attempts to log on to a compute node via secure shell (*ssh*). This module sits on top of Linux-PAM [2], a pluggable authentication module that provides a flexible mechanism for validating users. This authentication module is executed every time there is an *ssh* login request, and it queries the LSF job scheduler to see if the requesting user has a current job on the given node. If so, access is granted; otherwise, access is denied.

An additional feature of the module is constant access to all compute resources for systems staff. This continuous access is necessary for system administration purposes and diagnosing user problems. Note that we chose to implement this staff access based on the administrator designation within LSF. Hence, any user defined as an LSF administrator (*lsfadmin)* is automatically granted access to all compute nodes at any time. A benefit of this approach is that LSF can be dynamically configured to add or remove administrators by simply updating a configuration file on the front end. Likewise, login access to the entire cluster can be trivially controlled via this single configuration file.

3.4 Pre-Execution Scripts

LSF provides two mechanisms for executing site-specific programs prior to job launch. One executes programs immediately before a job is launched (*pre_exec* script), while the other executes programs when a job is submitted (*esub* script). The former mechanism can be used to set up programming environments. We found the latter useful for managing entry into the batch queues: for checking memory and wall-clock specifications, queue routing, restricting the number of queued entries, command and syntax checking, and evaluating account balances. The *esub* shell script is provided in Appendix A.2.

In the default LSF configuration, jobs that do not specify a wall-clock limit will be assigned the maximum time limit of the queue. However, these jobs often consume only a small percentage of the default maximum time. This makes it difficult for any batch system to schedule jobs to nodes efficiently, which can create large scheduling holes of unused cycles. Consequently, to make jobs more scheduler-friendly, we require users to include an accurate time limit. All jobs without a specified wall-clock time are automatically rejected at submission with a request to specify a reasonable time estimate.

Batch-job redirection at submission is important in heterogeneous hardware systems, and is also useful for setting up several “default” queues. Standard batch submissions go to the default *normal* batch queue, while interactive job submissions are routed to a 16-node *development* queue.

While *fairshare* scheduling helps to ensure fair access to all compute resources, users may still *submit* as many jobs as desired. Historically, we observe that if one user inundates the queues with jobs, this has the effect of causing other users to assume they cannot run a job anytime soon. This queue flooding was removed by using LSF’s API to acquire the total number of queued jobs per user and rejecting any new job that exceeds a threshold number of queued jobs (25, in our case).

We employ parsing scripts at batch submission to check batch-option syntax and specific user commands. For instance, the LSB_HOSTS variable becomes too long for certain batch scripts to handle when the host list contains more than 250 entries. Scripting problems at run-time can be avoided by immediately rejecting large-node jobs that make use of this variable.

Significant programming effort was invested in creating a project filter that performs several accounting operations at batch submission. Foremost, the filter determines the total time cost for a potential run and rejects it if the user does not have sufficient allocation remaining for the maximum job time charge.

The time charge must also account for single-processor-per-node runs and the burn factor associated with the queue priority. Users are able to charge time to different projects (allocation accounts). Also, the requested resources for each job (memory, nodes, cpus/node, wall-clock time) are logged and entered into a database daily. The database information is used to profile requests and analyze job characteristics such as accuracy of user-selected time limits. All rejected jobs are logged, and any accounting or filter errors are immediately sent to administrators.

3.5 Post-Execution Scripts

In order to provide a clean scratch file system for each job, it is necessary to clean up files that users leave behind after their job completes. The natural way to implement a disk-purge policy is through a post-execution shell script, and LSF provides such a mechanism on a queue-level basis. However, based on our initial experiences using the post-execution option, we recommend purging through the LSF *eexec* option. The *eexec* script is defined as an external executable that can be configured to run at job completion time. With the standard post-execution script, compute resources assigned to a user's job are released *prior* to running the post-execution script. This allows new jobs to be scheduled before the completion of the post-execution script from a previous job. Such a scenario is undesirable since it can lead to the erroneous removal of user files (a subtle problem that can be difficult to diagnose). In contrast, the *eexec* option does not release assigned compute resources until after the *eexec* script has completed and thus provides a safe mechanism for purging temporary file systems. An example *eexec* script that connects to each compute resource associated with a user's job and removes any temporary files is provided in Appendix A.3.

3.6 Scalability of the Batch System

Although the batch system is a central element for managing a production cluster, it must provide all the necessary batch functionality efficiently by introducing a minimal amount of overhead. This efficiency is critical for maintaining scalability to hundreds or thousands of processors. To ascertain the additional overhead imposed by LSF in launching parallel applications, we conducted a series of MPI runs using from 8 to 564 processors, both with and without the batch system. For the test cases run without LSF, we ran each application "by hand" using a Myrinet version of MPICH (MPICH-GM) and monitored the total wall-clock time to complete the application. (This is equivalent to running "mpirun a.out" interactively.) We repeated these same test cases by submitting each simulation through the LSF batch system (which also uses MPICH-GM), monitoring the total wall-clock time required to return the application once it had been scheduled and successfully completed. These LSF measurements thus include any overhead associated with launching the application and any post-execution overhead. (Note: any idle time spent in the queues waiting for free resources was subtracted from these measurements to ensure accuracy.) The overhead imposed by LSF was obtained by simply subtracting non-LSF times from the LSF times. The results from this comparison are shown in Figure 2 for two different versions of LSF. One set of results is from an early 5.1 HPC distribution. We observed minimal overhead up to 128 processors but more substantial overhead for higher numbers of processors. As an example, the 564-processor case incurred almost 9 minutes of LSF overhead. After reporting these high overhead trends for large parallel jobs to Platform Computing, we investigated the causes with LSF developers and continued benchmarking the LSF overhead to check scalability. Ultimately, we received a modified LSF distribution that performed substantially better. The final modified version's performance (Figure 2) shows the LSF overhead to be consistently less than 30 seconds for the processor range considered.

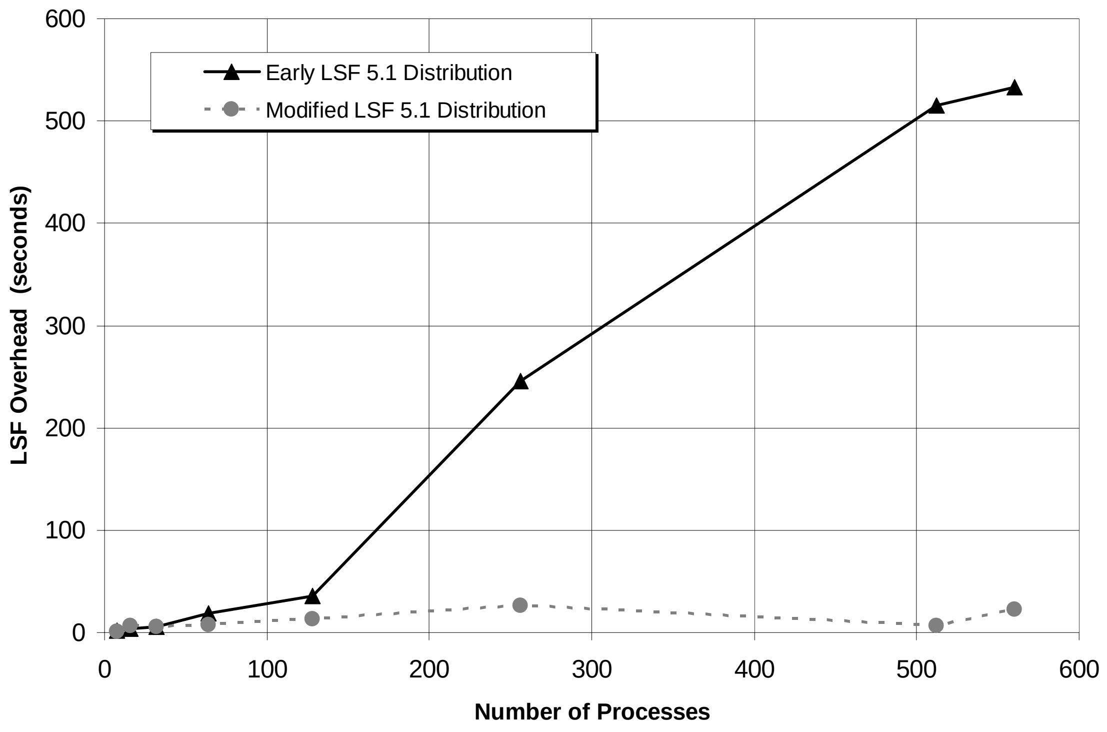

Figure 2: Overhead measurements for MPI jobs executed within the LSF batch system.

4 User Environment: Modules and RPMs

On large systems there is always an influx of new and revised applications that must be available to multiple users in a common location. The installation and maintenance of these applications need to be well organized and integrated into the user environment.

4.1 Modules

We use the Modules package [3] to create user environments at login and specific application environments on demand. Modules provide a convenient way to set paths and environment variables through "modulefiles" that are effectively sourced. Unlike shell scripts, however, Modules commands can be undone. For example "module load dot" might be used to put "." in the user's PATH variable, and "module delete dot" to remove it. With this capability, it is much easier to compare several different versions of an application by simply loading and deleting the version's environment. To accommodate multiple application versions, we have created a hierarchical directory structure to organize installed applications. On Lonestar, the /opt/apps directory contains names of all user applications. Below each application directory, there are subdirectories of the particular application versions. Version-number suffixes are used to indicate the different versions. In each subdirectory, the base application name is linked to a default version. The directory structure is illustrated in Figure 3. This organization keeps the top-level application directory uncluttered and provides a convenient listing of all available packages.

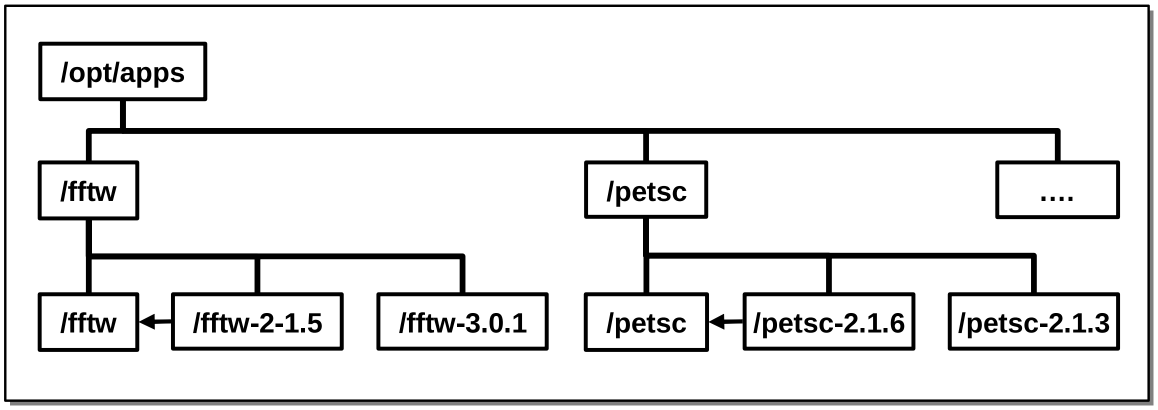

Figure 3: Hierarchical application directory structure.

4.2 RPMs

Since many applications, compilers, and tools have shared libraries, there is a constant need to access files in /opt. To minimize NFS traffic on the system, we chose to replicate the /opt directory structure on each compute node instead of cross-mounting the file system. However, this replication does require more effort and diligence to maintain across all nodes. All applications in /opt are maintained as RPMs [4] and installed with Cray Rx. Cray Rx is an extension to NPACI Rocks [5], a cluster-management toolkit that provides the infrastructure to maintain and rebuild a node at any time from a single kernel image, RPMs, and configuration commands. Consequently, we integrate all application RPMs into the Cray Rx distribution. In some cases, we build source RPMs that build the package during the install phase. In other cases, we simply incorporate package binaries into a single tarfile (binary RPM).

5 Summary

The LSF HPC batch system was deployed on a production, multi-user terascale Linux cluster with 600 Xeon processors. Recommendations for configuration of the batch system along with descriptions of custom utilities were presented in the context of building a stable, easy to maintain production cluster supporting hundreds of users nationwide. Scalability of the batch system was demonstrated for MPI jobs using up to 564 processors with less than 30 seconds of LSF overhead incurred per job.

6 Acknowledgments

The authors would like to acknowledge the diligent support of Dr. Vadim Elisseev from Platform Computing, who was instrumental in minimizing LSF overhead.

References

[1] http://mauischeduler.sourceforge.net/
[2] Barrett, D., Silverman, R., and Byrnes, R., *Linux Security Cookbook*, O'Reilly & Associates, 2003.
[3] Furlani, J., “Modules: Providing a Flexible User Environment”, Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V), pp. 141-152, San Diego, CA, September 30-October 3, 1991.
[4] Foster-Johnson, E., *Red Hat RPM Guide,* John Wiley & Sons, 2003.
[5] http://rocks.npaci.edu/Rocks/

Appendix

The Appendix contains example scripts and configuration options used with LSF HPC to achieve a working production environment. Note that some of these scripts call external Perl scripts or separate C applications written using the LSF API. While source code for the external applications is not provided here, these scripts do provide an outline of the configuration strategy.

A.1 Global Fairshare Configuration

Global fairshare can be configured with LSF HPC by giving equal shares to all users of the system. However, to keep queued jobs from loosing their fairshare partitions during maintenance and downtime periods, we recommend adding users to the fairshare policy at the job scheduling stage, not at the job submission stage. This can be accomplished via the following definitions in the *lsb.users* and *lsb.hosts* configuration files:

- Pertinent section from *lsb.users* configuration file:

```
Begin UserGroup
GROUP_NAME       GROUP_MEMBER              USER_SHARES
allusers         (all)                     ([default, 1])
End UserGroup
```

- Pertinent section from *lsb.hosts* configuration file:

```
# Enable Equal Fairshare

Begin HostPartition
HPART_NAME = GlobalPartition
HOSTS =  all
USER_SHARES = [allusers, 1]
End HostPartition
```

A.2 *esub*: Job Submission Script

When a user submits a new job to the batch system the *esub* script is executed immediately. We use *esub* to check for simple job syntax errors, route interactive jobs to a development queue, force users to specify a wall–time limit, enforce maximum per-user job limits, and enforce accounting limits.

```
exec 1>&2
. $LSB_SUB_PARM_FILE  >& /dev/null

TACC_FILTER_DIR=/opt/lsf/5.1/linux2.4-glibc2.2-x86/etc/TACC_utils
MAX_QUEUED_JOBS=25

#-----------------------------------------------------------------
# Syntax Error Checking: Perl script looks for correct inclusion of
# pam –g 1 gmpirun_wrapper, etc.
#-----------------------------------------------------------------

status=`parse.pl $LSB_SUB_PARM_FILE`
if [ -n "$status" ]; then
   echo -------------------------------------------------------------
   echo $status | fold -w 72 -s 2>&1
   echo -------------------------------------------------------------
   exit $LSB_SUB_ABORT_VALUE
fi

#-----------------------------------------------------
# All interactive jobs are forced into development queue
#-----------------------------------------------------

if [ "$LSB_SUB_INTERACTIVE" = Y -a "$LSB_SUB_QUEUE" != development ]; then
   echo "Your job is being routed to the development queue..."
   echo 'LSB_SUB_QUEUE="development"' >> $LSB_SUB_MODIFY_FILE
```

```
fi

#--------------------------------
# Require User-specified wall time
#--------------------------------

if [ ! -n "$LSB_SUB_RLIMIT_RUN" -a  "$LSB_SUB_INTERACTIVE" != Y ]; then
   echo ------------------------------------------------------------
   echo "   ERROR: You must specify a walltime limit in batch jobs."
   echo "   Syntax: -W[[hours]:][minutes]"
   echo "           e.g., for 1 and 1/2 hours, use:"
   echo "   #BSUB -W 1:30"
   echo ------------------------------------------------------------
   exit $LSB_SUB_ABORT_VALUE
fi

#--------------------------
# Enforce maximum Job Limit
#--------------------------

NUM_JOBS=`$TACC_FILTER_DIR/lsf_query_user $USER | awk '{print $8}'`
if [ $NUM_JOBS -ge $MAX_QUEUED_JOBS ];then
   echo ------------------------------------------------------------
   echo "ERROR: Maximum number of queued jobs exceeded (limit = $MAX_QUEUD_JOBS)"
   echo "       Current number of queued jobs for <$USER> = $NUM_JOBS"
   echo ------------------------------------------------------------
   exit $LSB_SUB_ABORT_VALUE
fi

#-------------------------------------------------------------
# Accounting: determine if user has enough time, log requests, …
#-------------------------------------------------------------

 FILTER_RETURN=`/opt/lsf/acct/bin/lsf_project_filter $LSB_SUB_PARM_FILE \
       $LSB_SUB_MODIFY_FILE`
 FILTER_STATUS=$?

if [ $FILTER_STATUS != 0 ] ; then
    echo "ERROR: Job rejected by TACC Project Filter."
    echo "       lsf_project_filter: err=$FILTER_RETURN."
    exit $LSB_SUB_ABORT_VALUE
fi
```

A.3 *eexec*: Post Execution Script

The *eexec* script is configured to execute immediately after a user's job script completes, just prior to releasing the resources associated with the job. We use the *eexec* script to remove temporary files created by the user in the /tmp or /scratch file systems on each node owned by the job. Execution of *eexec* is controlled by the LS_EXEC_T environment variable (this is in contrast to the on-line documentation for LSF HPC 5.1 which erroneously states that *LS_EEXEC_T* is the environment variable that indicates job startup or completion entry points). Note that certain key temporary files belonging to mpich-gm and LSF accounting processes are not removed at this stage of the job. Also, the LSB_MCPU_HOSTS variable is used to obtain host names.

```
#!/bin/sh

[ "$LS_EXEC_T" = "START" ] && exit 0
if [ "$LSFUSER" != "root" ]; then
   for i in `echo $LSB_MCPU_HOSTS`; do
      if [ "$i" != "2" -a "$i" != "1" ]; then
         ssh $i "find /tmp -user $LSFUSER -a ! -name gmpi_\* -a ! -name \
               .\*.acct -exec rm -rf {} \; ; \
               find /tmp -user $LSFUSER -a ! -name gmpi_\* -a ! -name \
               .\*.acct -exec echo "Cannot remove " {} \;" >> $LS_SUBCWD/$LSB_OUTPUTFILE

         ssh $i "find /scratch -user $LSFUSER -a ! -name gmpi_\* -a ! -name \
               .\*.acct -exec rm -rf {} \; ; \
               find /scratch -user $LSFUSER -a ! -name gmpi_\* -a ! -name \
               .\*.acct -exec echo "Cannot remove " {} \; ">> $LS_SUBCWD/$LSB_OUTPUTFILE
      fi
   done
fi
```